Representing Logics and Logic Translations

Florian Rabe

submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Submitted: 30.05.2008
Defended: 19.06.2008
Approved: 19.06.2008

Jacobs University Bremen
School of Engineering and Science

Dissertation Committee

Prof. Dr. Michael Kohlhase, Jacobs University Bremen (Supervisor)
Prof. Dr. Herbert Jaeger, Jacobs University Bremen
PD Dr. Till Mossakowski, German Research Center for Artificial Intelligence, Bremen
Prof. Dr. Frank Pfenning, Carnegie Mellon University, Pittsburgh

I hereby declare that this thesis has been written independently except where sources and collaborations are acknowledged and has not been submitted at another university for the conferral of a degree.

Parts of this thesis are based on or closely related to previously published material or material that is prepared for publication at the time of this writing. These are [RK08], [AR08], [KMR08], [GMdP+07], [Rab07], [KLR07], and [Rab06]. [Aga08] and [Soj08] are theses supervised by the author that are related to this work. Parts of this work are the result of collaborations with other researchers: for Sect. 4 with Steve Awodey and for Sect. 6 and 7 with Michael Kohlhase. In each case the precise connection to this work is detailed in the relevant passages of the text.

Bremen, 17.11.2008, Florian Rabe

Abstract

Logic is the study of formal languages for propositions and truth. Logics are used both as a foundation of mathematics and as specification languages in mathematics and computer science. Since logic is intricately intertwined with the nature of mathematics, the question how to represent logics in our minds is a constant challenge to our understanding. And only when it is understood can we begin to answer the corresponding question about logic translations. At the same time logics are used to a large extent in computer science to reason about both mathematics and software systems. This brings up the question how logics and their translations can be represented in a computer system. In this text we try to give answers to these fundamental research questions.

Throughout the 20^{th} century several answers have already been provided. Most of them can be grouped into two kinds, which can be denoted by set/model theory and type/proof theory. These two classes represent different research fields with conflicting philosophical and mathematical backgrounds, which makes their conceptualizations ontologically different. However, both fields have developed very sophisticated and strong solutions.

Therefore, we base our investigation on the desire to reconcile these two views. Our focus is on extending the existing notions of logic and logic translation in a way that retains their nature and the accumulated knowledge about them while leading them into a new direction. Our goal is to enrich both research fields by applying them more cogently to and making them more accessible and understandable to one another.

We choose institutions as a set/model theoretical and dependent type theory as a type/proof theoretical representative of the two kinds. We observe that both have complementary advantages that reflect their different backgrounds and devise our own answer by combining them in a way that consequently exploits their respective strengths. Mathematically, our main results can be summarized as follows. We define logics by extending institutions with notions of proof categories and proof theoretical truth that are very much parallel to the model categories and satisfaction relation of institutions. Then we give a concrete simple logic for dependent type theory using models inspired by Kripke models for intuitionistic logic. Finally, we show how to use this logic to define or encode logics and logic translations.

While this answers the question how to represent logics and logic translations in our minds, it is not adequate for the specific constraints and use cases of software systems. Therefore, in a second investigation, we explore how to make our representations more concrete and robust enough to permit a mechanized treatment on a large scale.

We choose OMDoc as a scalable, web-compatible representation language for mathematical knowledge. Because we find that its applicability to logical knowledge is limited, we revise it taking into account the characteristic requirements of logic. While keeping the motivation behind and flavor of OMDoc, we employ a different methodology more suitable to logic, thus effectively reinventing it. Mathematically our main results can be summarized as follows. We provide a formal abstract syntax for modular theory development and a formal semantics for it. Our module system treats logical frameworks, logics, and logical theories as well as the translations between them uniformly as theories and theory morphisms that are related via the "is meta-language for" relation. And it consequently separates logic-independent and logic-specific language constructs, which lets us derive a logic-independent flattening theorem and constitutes the basis for logic-independent logical knowledge management services.

Taking these two results together, we obtain a triangle of different mathematical communities, research objectives, and philosophies consisting of set/model theory, proof/type theory, and mathematical knowledge management. Our main contribution is to integrate its corners into a coherent framework centered around logic that capitalizes on their comparative advantages.

I am most grateful to Michael for having the overview to find the right research topic for me and for making it possible for me to pursue it. I am especially indebted to Frank, Michael, and Till whose insights, advice, and guidance have been as different as they have been compelling and fruitful.

I have benefited a lot from discussions and collaborations with Steve Awodey and numerous other researchers, in particular Carsten Schürmann, Geoff Sutcliffe, and Valeria de Paiva, and the members of the KWARC group at Jacobs University Bremen.

Finally, I would like to thank Jacobs University, the German Academic Exchange Service, and the German National Merit Foundation who have provided me with financial independence for the past three years.

Contents

Part I

Introduction and Preliminaries

Chapter 1

Introduction

In this section, we first look at the history and the fundamental concepts of mathematical logic in Sect. 1.1. We see that there are many logics and not *the* logic, an observation that gives rise to the concept of logic translations. Furthermore, we see that the choice of logic is deeply connected to the nature of mathematics. Therefore, the representation of logics and logic translations requires a general abstract formalism called a logical framework. The question how to choose the logical framework is the central question of this text. While formal logic is technically the language of mathematics, actual mathematical discourse usually uses less formal languages. Therefore, we also look at semi-formal languages in Sect. 1.2.

After these introductions, we state the motivation of this text and describe our intended contribution to the topic in Sect. 1.3. Finally, we outline the following sections in Sect. 1.4.

1.1 Formal Languages for Mathematics

1.1.1 Logic(s)

Logic is the field of mathematics concerned with the study of the concepts **proposition** and **truth** and the reasoning about them. Intuitively, propositions are statements about mathematical objects, and propositions may be true or false.

It is not possible to define mathematical objects first and propositions and truth later — their definitions must be intertwined: To develop the mathematical objects, it is frequently necessary to know the truth of certain propositions about previously developed objects. The simplest example is the object $\{x \mid P(x)\}$ containing all objects x such that the proposition P is true about x. Thus, logic is fundamental to mathematics. It is so fundamental that it is hard to give a formal definition of proposition and truth. In fact even the notion of definition itself is unclear in the absence of the notion of truth.

When understanding logic, we must distinguish between logic and logics. We will consider **a logic** to be a way to formalize the notions of proposition and truth. Thus, logics are specific choices how to understand and study these notions. Then logic is the field of mathematics concerned with logics. A specific logic that is picked as a starting point of mathematics is called a **foundation** of mathematics.

Logic has been studied in most ancient high cultures, most influentially in Greece, and has been a discipline of philosophy ever since. However, modern logic in the sense of mathematics and computer science goes back to only the late 19^{th} century. Frege's Begriffsschrift ([Fre79]) from 1879, which introduces a formal language for propositions, is generally seen as the beginning of modern logic. Further seminal works of the 19^{th} century are by Peirce, who introduced notations and terminology that are still in use today ([Pei85]), and Peano, who gave an axiomatization of arithmetic ([Pea89]).

In the following we will summarize the modern history of logic by relating three pivotal discoveries of mathematics and their consequences: the Grundlagenkrise starting with the 20^{th} century, Gödel's incompleteness theorems from 1930, and the development of electronic computer systems in the second half of the 20^{th} century. We will cite the original papers and refer to [vH67] for reprints and English translations of historical papers. We also recommend [Zal08] as a source of historical and introductory articles and further references.

Grundlagenkrise The Grundlagenkrise was caused by the discovery of paradoxa, i.e., contradictions, in what is called **naive set theory** in retrospect. Naive set theory was the implicitly assumed foundation of mathematics at the time, Cantor's Grundlagen ([Can83]) from 1883 being the most influential contribution. The best known paradoxon was found by Russell in 1901 ([Rus01]). Peano had noticed a similar one in 1897.

Roughly, Russell's paradoxon arises from unlimited set comprehension. That leads to a contradiction because it permits to form the set of all sets that do not contain themselves. Intuitively, a contradiction in a logic means that something is both true and not true. That typically makes everything true, by which truth becomes vacuous. Since mathematics is a strictly hierarchical science with every new concept resting on the preceding ones, a contradiction in mathematics, unless it can be remedied somehow, is tantamount to total destruction. Therefore, the freeness from contradictions, called **consistency**, is crucial for a foundation.

In response to this, mathematicians have developed several — sometimes alternative, sometimes complementary — foundations that can replace naive set theory. This happened over several decades as an evolutionary creative process. But it did not culminate in a commonly accepted solution. Rather, it led to profound and sometimes fierce debates on what mathematics is. The personal quarrel between Hilbert and Brouwer, which was partially fuelled by these debates, is an almost tragic example. From this evolution emerged two major classes of foundations: **axiomatic set theory** and **type theory**, which we will describe in the following. (The clear separation between set and type theory is partially drawn here for instructive purposes: The historical and mathematical boundaries are not as sharp.)

The basic idea of **axiomatic set theory** is that there is a universe of sets, and any mathematical object ever introduced is a set. The sets are related via the binary relations of equality and membership. For example $m \in M$ is used to say that the set m is a member of the set M. Depending on context, M is regarded as a property of m or as a structuring concept.

To talk about sets, equality, and membership, propositions are used. The basic propositions are of the form $m = m'$ and $m \in M$. Composed propositions are built up from the basic ones. Typically, (at least) first-order logic (FOL) is used as the language of composed propositions: FOL uses propositions such as $F \wedge G$ and $\forall x.F(x)$ denoting "F and G are true" and "for all (sets) x, F is true about x".

Then a limited collection of propositions (the **axioms**) is chosen as fundamental truths. These are chosen very carefully to prevent contradictions and to obtain a minimal set of axioms. Based on the axioms, **proofs** are used to single out the true propositions. A proof consists of a sequence of steps that derive one true proposition from other true propositions starting with the axioms. In this way the whole of mathematics is developed, and for every proposition, truth is defined by whether it has a proof.

The languages for proofs can be informal or formal. Mathematicians tend to favor informal proofs that use natural — albeit highly standardized and precise — language. Formal language is used when advantageous for brevity, disambiguation, or intuition. Looking at an arbitrary mathematical textbook, a — maybe surprising — prevalence of informal language is likely to become apparent.

Type theory mainly differs from set theory in that it employs a stratification of the mathematical universe. In the simplest type theories, the basic concepts are *term* and type. Intuitively, terms represent mathematical objects, and types represent properties and structuring concepts. And for a term m and a type M, the propositions (in the context of type theory often called

judgments) $m = m$, $M = M$, and $m : M$ are used, where the first two state equality and the third is used to say that m has type M. Thus, the cases $m : m$ and $M : M$ leading to the paradoxon of naive set theory are excluded by construction. Often a term may only have one type, which is in contrast to set theory, where a set may be a member of arbitrary many other sets. In that case there is a characteristic contrast between the universe of sets in set theory and the typed terms of type theory.

The proofs of type theory are conceptually similar to those of set theory. However, type theory tends to favor a more restricted language of propositions and a completely formal language of proofs. Furthermore, it favors constructive and algorithmic reasoning over the assumption of axioms. Then the consistency of set theory corresponds to the correctness and termination of these algorithms.

Both set theory and type theory have led to **numerous specific foundations** of mathematics. Zermelo-Fraenkel set theory, based on [Zer08, Fra22], is most commonly in use today. Other variants are von Neumann-Bernays-Gödel set theory, based on [vN25, Ber37, Göd40], which is important for category theory, and Tarski-Groethendieck set theory, based on [Tar38, Bou64]. The first type theory was Russells's ramified theory of types ([Rus08]). And in their Principia ([WR13]), Whitehead and Russell gave one of the most influential foundations of mathematics. Church's simple theory of types, also called higher-order logic, ([Chu40]) is the most-used type theory today. Important other type theories are typically organized in the lambda cube ([Bar91]) and include dependent type theory ([ML74, HHP93]), System F ([Gir71, Rey74]), and the calculus of constructions ([CH88]). Most of these foundations have further variants, such as Zermelo-Fraenkel set theory with or without the Axiom of Choice or type theory with or without product types.

Besides the set/type theory distinction, there are **other dimensions** along which foundations can be distinguished. We will only briefly mention the question of the philosophical nature of mathematics. In **platonism** (going back to Plato, with various defenders in the 20^{th} century, e.g., K. Gödel), formal mathematical objects are only devised as representations of abstract platonic objects to assist in reasoning. And within the non-platonistic view, four important schools can be singled out. In **formalism** (main proponent D. Hilbert, see, e.g., [Hil26]), the formal mathematical objects themselves are the objects of interest. Thus, mathematical reasoning can be reduced to purely mechanical procedures. In **logicism** (main proponent G. Frege, see [Fre84]), all of mathematics is reduced to logic, i.e., all axioms are logical truths without mathematical intuition. In **intuitionism** (main proponent L. Brouwer, see [Bro07]), truth depends on a mathematician's experience of it. Thus, mathematical proofs are only supplements of mental constructions. Intuitionism rejects, e.g., an argument that derives "F" from "not not F" because the absence of the truth of "not F" does not yield the truth of "F". And **predicativism** (main proponents H. Weyl, S. Feferman, see [Wey18, Fef05]) emphasizes the need that a mathematical definition may not depend on the defined object itself. Predicativism rejects, e.g., the definition of a closure of a set as the intersection of all supersets with a certain property because the closure is among the intersected sets.

The correlation of these philosophical views to each other and to the set/type theory distinction is complex. Most mathematicians tend to think platonistically, and platonists tend to favor set theory over type theory. Computer-supported work tends to be formalistic. But formalism is not strongly correlated with either set or type theory. Type theories are usually predicative and often intuitionistic whereas set theories are usually neither. Similarly, researchers favoring type theory tend to favor intuitionism and predicativism. The importance of logicism has faded in general. A good overview is given in [Hor08].

Gödel's Incompleteness Results Hilbert's formalistic program, set forth in his second problem ([Hil00]) and various texts from the 1920s, e.g., [Hil26], called for the reduction of all mathematics to a set of axioms and a consistency proof for these axioms using only finitary means. Since proofs are built up from the axioms, such a reduction would yield all true propositions by

systematically searching all proofs. In 1930, Gödel established two negative results ([Göd31]), which as von Neumann recognized first showed that the goal of Hilbert's program is unreachable.

The first one roughly says that no foundation of mathematics can be found that defines the truth of all propositions in an algorithmic way. The second one says that no foundation can prove its own consistency. Gödel worked in the Principia, which were the foundation mainly in use at the time, but the results extend to all foundations beyond a certain level of expressivity.

This is the major reason why no foundation has won the endorsement of mathematicians as a whole and why there will not be a final answer which foundation of mathematics is the best. Since no perfect foundation exists, the personal preferences and the characteristics of a problem lead to different choices of foundation.

All foundations have been used independently and with varying degrees of depth to develop the central fields of mathematics such as analysis and algebra. And in every foundation, it is possible to introduce other logics as derived notions. These provide interfaces between mathematical theories that are important for the hierarchic development of mathematics. Most importantly, FOL can be defined in almost any foundation and suffices for large portions of analysis and algebra.

In particular, within a foundation, a second one may be defined. In fact, FOL alone is enough to define set theory. Thus, mathematics as developed in one foundation can often be transferred to another one. And far up in the mathematical hierarchy, it is often not so important which foundation is used because the development relies on interface logics that can be introduced within several foundations. For example the real numbers in higher-order logic are the same as those in Zermelo-Fraenkel set theory. However, proving such inter-definabilities is a difficult problem: It is either done outside of mathematics appealing only to intuition, or two foundations and a translation between them are formalized in a third foundation.

Due to the inter-definability of foundations, the choice of foundation is, to a certain extent, a matter of a researcher's personal preference. The majority of mathematicians uses set theory. It can indeed be argued that the set-theoretical universe is more appropriate to the intuition of mathematics. However, set theory only quieted but never extinguished its well-founded criticism due to the danger of inconsistencies and the non-intuitionistic and impredicative reasoning. In particular, computer science has continued the development of type theory because its rigorous use of formal syntax and its focus on algorithmic definitions make it very appropriate as a foundation used in a computer system. For example, functional programming languages can be defined directly within a type theoretical foundation leading to a profound analogy between proofs and programs.

Computers Modern hardware and software systems are huge and need to employ a significant degree of structuring. This structuring is both vertical, i.e., using a hierarchy of system layers, and horizontal, i.e., using inter-connected and inter-dependent modules. Both the development and the deployment of the components may be distributed over persons, points in time, locations, and cultures. Thus, computer systems have created an enormous demand of interface languages, which describe mathematical objects and their properties. Furthermore, there is a great demand to check automatically the correctness of (i) the interface structure of a computer system and (ii) its separate components with respect to the interfaces. Therefore, dedicated logics are used to define propositions for a specific context and then study their truth.

The computational tractability of a logic tends to react very sensitively to its expressivity. In simple logics, the set of true propositions is decidable. However, even for the simplest logic, the set of true formulas is NP-complete (called the SAT problem). But in more complex logics, this set is only recursively enumerable or not even that. Thus, the choice of logic for a specific purpose is subject to non-trivial trade-offs. FOL is important here because it represents a compromise between expressivity and algorithmic treatment. But together with the extreme diversity among the nature of computer systems, this has led to what Goguen and Burstall

called a "population explosion among the logical systems" at the beginning of their seminal paper introducing institutions ([GB92]).

1.1.2 Logical Frameworks

The population explosion has led to attempts to standardize and structure logics. This starts with the question what the definition of **a logic** should be. However, if a definition is attempted, systematically looking for a logic not covered is likely to yield an interesting logic. To be useful, a definition must make assumptions about logics that will invariably exclude others.

For example, already the seemingly broad description of Sect. 1.1.1 conveys too narrow a view: We distinguished the language of proposition and the method to define truth. But in some logics, e.g., some dependent type theories (e.g., [ORS92] or [CAB+86]), the property of being a proposition itself depends on the truth of other propositions. Another caveat is that logics disagree strongly on what it means to be true: Modal logics (going back to [Lew18]) use qualified truth such as necessarily true and sometimes true. Intuitionistic logics (going back to [Bro07]) reject the assumption that a proposition is either true or false. Many-valued (such as in [Łuk20]) and fuzzy (based on [Zad65]) logics replace the distinction between true and false with degrees of truth. Linear logic ([Gir87]) sees the truth of assumptions as a resource that is consumed when it is used. And paraconsistent logics (see [Pri02] for an overview) admit propositions that are both true and false, which is a contradiction in most logics.

A **logical framework** provides a specific definition of what a logic is, and the mathematical infrastructure to define logics and reason within them and about them. A typical characteristic of mathematical developments in logical frameworks is that they consist of two parts: a first part defining a logic in the logical framework and a second one defining a theory within that logic. Another use of logical frameworks is to encode logics that have already been defined in no or in a different framework. Then **adequacy** means that the encoding yields indeed the right logic.

Several logical frameworks have been suggested and used. But just like there is no universally accepted foundation, there is no universally accepted definition what a logic is. For example in 2005, the World Congress on Universal Logic held a contest about what a logic is [Béz05].

Two classes of logical frameworks can be distinguished: those built on type theoretical and those built on set theoretical foundations. The former tend to favor a proof theoretical view and the latter a model theoretical view of truth, although neither is a priori restricted to one view. This tendency increases the distance between research on logics: The two views are sometimes in conflict with each other in a way that is not only due to mathematics but also due to the mathematical philosophies and preferences of their proponents.

The **proof theoretical** view sees both propositions and proofs as formal mathematical objects (called terms in the context of type theory). Every proof term is typed by the proposition it proves. Logics are defined by rules that construct the propositions and the proof terms. And the truth of a proposition is defined by whether a proof term exists for it. The first such framework was Automath ([dB70]). And the most important frameworks in use today are Isabelle ([Pau94]) based on higher-order logic ([Chu40]), LF ([HHP93, PS99]) based on dependent type theory ([ML74]), and Coq ([BC04]) based on the calculus of constructions ([CH88]). An overview over type theoretical logical frameworks is given in [Pfe01].

A proof theoretical framework in a type theoretical foundation typically uses the **Curry-Howard correspondence** ([CF58, How80]): It regards **propositions as types** and **proofs as terms**. Then the type of a proof is the proposition it proves. The Curry-Howard correspondence has led to numerous deep correspondences between type theories and logics (e.g., [See84, BdP00]). More generally, types can be seen as judgments and terms as evidence: Then the typing relation $m : M$ expresses that m is evidence for the judgment M ([ML96]).

The **model theoretical** view sees propositions as mathematical objects that do not have meaning per se. This is because the propositions may contain symbols that are place-holders for

mathematical objects. Only after assigning concrete mathematical objects to these symbols, the proposition is meaningful. Such an assignment is called an **interpretation** (function), **model**, or **structure**. To define a logic, one gives a set of propositions and a collection of interpretations. Then the truth of a proposition is defined relative to a fixed interpretation. And a proposition is **valid** iff it is true under all interpretations.

The model theoretical view goes back to Tarski's definition of truth ([Tar33, TV56]). Influential steps were Robinson's models for algebra ([Rob50]) and Kripke's models for modal logic ([Kri63]). An overview over model theoretical logics can be found in ([BF85]).

Model theoretical logical frameworks are often called **general logics**. The most important ones are the frameworks of general logics ([Mes89]) and **institutions** ([GB92]). Both abstract from the specific syntax of propositions and the specific structure of models, which yields a high degree of generality. They are closely related, but [Mes89] also covers proof theory. Another proposal to add proof theory to institutions was given in [MGDT05].

Categorical logic ([LS86]) extends the Curry-Howard correspondence to categories so that propositions and proofs correspond to objects and morphisms. Thus, it combines proof and model theoretic aspects. A recent application of this approach as a logical framework is given in [GMdP+07].

1.1.3 Logical Reasoning

Logical reasoning can be organized in a four-levelled hierarchy.

1.1.3.1 Intra-Theory Reasoning

Logic is not only the study of the truth of propositions. It is also the study of consequences between propositions. Intuitively, consequence is a relation between a set of propositions Δ and a proposition F. This relation is one of the central questions of logic and studied in an area we call intra-theory reasoning. Notation and terminology differ between the proof theoretical and the model theoretical view.

Model-theoretically, consequence is written as $\Delta \models F$. Δ is called a **theory**, the propositions in Δ are called the (non-logical) **axioms**, and F is called a **theorem** of the theory. $\Delta \models F$ is defined to mean: F is true under all interpretations that make all axioms in Δ true.

Proof-theoretically, consequence is written as $\Delta \vdash F$. Δ is called a **context**, the propositions in Δ are called **hypotheses**, and F is said to be true in context Δ. The crucial concept of proof theoretical consequence is that of a **rule** (also called inference rule or proof rule). In the simplest case, a rule is a relation between propositions. Then it is written as

$$\frac{F_1 \quad \dots \quad F_n}{F}$$

for propositions F_i and F. The F_i are called the **hypotheses** and F the **conclusion** of the rule. The intuition is that if all F_i are true, then F is true as well. If a rule has no hypotheses, its conclusion is always true. Such rules are called (logical) **axioms**. From the axioms, the proof-theoretically true propositions are obtained by finitely many rule applications. Precisely, a **proof** is a tree in which the nodes are labelled with propositions and every node is the conclusion of a rule in which its children are the hypotheses. If the proof tree has the root F, it is a proof of F.

Usually, logics have both a proof theory and a model theory. Then **soundness** means that proof theoretical consequence implies model theoretical consequence. And **completeness** expresses the converse implication. Often a specific model or proof theory is used to define a logic. But there can be several model or proof theories inducing the same consequence relation. If logics are sound and complete, the two notions of consequence coincide. This is essential for

logical reasoning because proof theoretical consequence is an algorithmic notion that can be automated. Then the consequence relation can be determined systematically.

Thus, the consequence problem is theoretically quite well-understood. However, it is a very hard theoretical problem to prove the completeness for a pair of proof and model theory. And implementations of the proof theoretical consequence pose a hard practical problem as well. The naive automation of the proof theoretical consequence enumerates all proofs by exhaustively applying rules until a proof of the desired theorem is found. In general, for almost all interesting logics, the consequence relation is at most enumerable and the enumeration intractable. Therefore, much research is concerned with devising efficient algorithms that yield useful automated provers or disprovers. Another important goal is to find logics that have better decidability or tractability properties such as description logics (see [BCM+03]). Furthermore, semi-automated provers try to only automate some parts keeping humans in the loop for the crucial proof steps.

Several competing or complementary automated and semi-automated theorem provers have been implemented for various logics. We refrain from giving an elaborate overview and mention only some examples.

Among the automated provers, most success has been achieved for first-order logic where provers compete in the annual CASC ([PSS02]) competition. The provers Vampire ([RV02]), Spass ([WBH+02]), and E ([Sch01]), and the model finder Paradox ([CS03]) are among the strongest systems. Further example provers are FaCT ([Hor98]) for modal logic and TPS ([ABI+96]) and Leo-II ([BPTF07]) for higher-order logic.

For the formalization of mathematics, automated provers are typically too weak and need interactive human support. But semi-automated provers have been used to formalize large portions of mathematics. The largest library is available in the Mizar system, which uses Tarski-Grothendieck set theory ([Tar38, Bou64]) as a foundation. Further large libraries exist in the Isabelle/HOL ([NPW02]) and the Coq ([BC04]) systems, which are based on type theory. One of the biggest successes in this area was the formalization of the Jordan curve theorem ([Hal05b]); the Flyspeck project ([Hal03]) formalizes the proof of the Kepler conjecture ([Hal05a]). An overview over systems specifically used for the formalization of mathematics is given in [Wie03].

An important aspect of logical reasoning is the use of proof tactics and strategies. These can be heuristics that implementations employ in order to cut down the search space by identifying presumably futile proof directions. They can also be small programs that apply certain combinations of rules such as for induction. This yields high-level languages in which humans can write proofs in a way that is closer to natural language proofs written by mathematicians. The most advanced tactic languages are the ones in Mizar ([TB85]) and the Isar language for Isabelle ([Nip02]). Coq ([BC04]), PVS ([ORS92]), HOL ([Gor88]), and HOL Light ([Har96]), and ΩMEGA ([BCF+97]) are also notable.

1.1.3.2 Inter-Theory Reasoning

Inter-theory reasoning studies relationships between theories. Since the inter-theory level is the most relevant level for this text, we give a careful overview over the relevant concepts and the related work.

Theory Graphs Sophisticated mathematical reasoning usually involves several related but different mathematical contexts, and it is desirable to exploit these relationships by moving theorems between contexts. The first applications of this technique in mathematics are found in the works by Bourbaki ([Bou68, Bou74]), which tried to prove every theorem in the context with the smallest possible set of axioms. We will follow the "little theories approach" proposed in [FGT92] in which separate contexts are represented by separate **theories**. Structural relationships between contexts are represented as **theory morphisms**, which serve as conduits for passing information (e.g., definitions and theorems) between theories (see [Far00]).

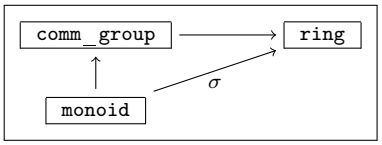


Figure 1.1: A Theory of Rings

We will use a portion of the elementary algebraic hierarchy as a running example. This defines the theory of monoids, extends it to the theory of commutative groups and then combines these two into the theory of rings. Fig. 1.1 shows the hierarchy in a graph whose nodes are theories, and whose edges are the extension, combination, and interpretation operations. To understand it in more detail, we first need to come to grips with the notion of a theory morphism.

Concrete theories are usually defined by a set of (possibly typed or defined) **symbols** (the signature) and a set of **axioms** describing the properties of the symbols. A **signature morphism** σ from a theory S to a theory T translates or interprets the symbols of S in T. If we have entailment relations for the formulas of S and T, a signature morphism is particularly interesting if the following holds: All theorems of S become theorems of T — after translation via σ. Taking this as a guiding intuition, a signature morphism σ between two theories S and T is called a **theory morphism** iff the σ-images of all S-axioms are T-theorems.

In the algebraic hierarchy, the theory of commutative groups is a subtheory of that of rings, since their signatures and axiom sets are subsets. Therefore, to save space we only need to represent those symbols and axioms in the `ring` node that are not already in the `comm_group` node if we know that the theory `ring` is created by extending `comm_group`. We say that `ring` imports from `comm_group`. Similarly, `comm_group` imports from `monoid`. The relation between `ring` and `monoid` is more difficult: Here we need a morphism σ that gives a specific translation from `monoid` to `ring`. In general, such a morphism from S to T means that some symbols imported from S are identified with objects that are already defined over B, and only the remaining symbols of S are imported. In our example, the carrier set of `monoid` must be translated to the carrier set already imported from `comm_group`.

If we view the nodes in graphs like the one in Fig. 1.1 as partial theory specifications and the edges as importing relations labelled with morphisms, we arrive at a **theory graph** that serves as a compact specification of a collection of mathematical theories and their relations. More precisely, every node T in such a theory graph induces a theory $\overline{T}$ as follows: T itself contributes its symbols and axioms to the theory $\overline{T}$. And every import from S to T with a morphism σ contributes the symbols and axioms of $\overline{S}$ translated via σ. Thus, the content of T and the nodes below T determine the theory $\overline{T}$. Clearly, a theory graph must be acyclic.

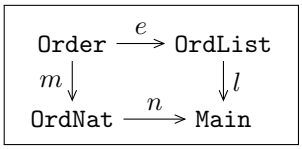


Figure 1.2: Instantiation

Theory graphs can handle high-level theory constructions like **parametric polymorphism** and instantiation. For example, in Fig. 1.2, `OrdList` is a theory of ordered lists, which is parametric in a theory `Order` for orderings via the import e. Assume we have established a theory morphism m from `Order` to the theory `OrdNat` of ordered natural numbers that expresses that `OrdNat` models the specification `Order`. Then the instantiation of `OrdList` with `OrdNat` arises as the category theoretical pushout of the diagram (see Def. 2.17).

In general, if the node T in a theory graph imports from the node S via a morphism σ, the pushout semantics induces a theory morphism from S to T by construction. We call these theory morphisms **definitional** because all axioms of S are imported into T and thus hold by definition. We also allow **postulated** theory morphisms or **views**, where the theory morphism property has to be established by proving $\sigma(F)$ in T for all S-axioms F. Then we call the $\sigma(F)$ **proof obligations**.

In software engineering, theories and theory graphs have been studied with two applications in mind. The first is in **(algebraic) specification** where theories are used to specify the behavior of programs and software components, and theory morphisms are used to enable reuse. The second application of theory graphs is in **module systems**, where modules are used to encapsulate program functionality into meaningful units and hide implementational details.

Functors are used to flexibly export module functionality for reuse in other situations.

Theory Graphs and Module Systems The parallelism in ideas (Module specifications correspond to theories and implementations to theory morphisms.) can be made use of for software development, if we structure modular program specifications and implementations correspondingly. More precisely, a module T can be regarded as a theory, and the fact that T implements a specification S can be expressed as a theory morphism $S \to T$. This approach naturally leads to a regime of specification and implementation codevelopment, where initial, declarative specifications are refined to take operational issues into account. Implementations are adapted to changing specifications, and verification conditions and their proofs have to be adapted as programming errors are found and fixed. This has been studied extensively, and numerous systems have been developed.

Properties of Module Systems To compare existing module systems with ours, we will introduce some terminology first that describes module systems on an abstract level.

Modules declare named symbols that may be typed or defined, for example sorts, constants, operations, or predicates. Sometimes they may also declare axioms, which can be named or unnamed. In addition, various system-specific concepts may be declared such as notations or proof rules. Other names used instead of module in some contexts are theory, signature, specification, (type) class, (module) type, and locale.

Inheritance means that one module T can **import** another module S. An important distinction is whether the imports are **named** or **unnamed**. In the former case, the name of the import is available to refer (i) to the imported module as a whole, or (ii) to the imported symbols via qualified names. With unnamed imports, all imported symbols must reside in the same flat name space. Unnamed imports are conceptually easier but make multiple inheritance difficult: For example, in Fig. 1.1, `ring` imports from `monoid` twice. Some systems permit to **rename** symbols of S upon an import from S. In systems with named imports, this is just syntactic sugar; but in systems with unnamed imports, renaming is necessary for multiple inheritance, i.e., to disambiguate two imports of the same module.

Instantiation means that when importing S into T, some names declared in S may be mapped to expressions of T. Module systems differ as to what kind of mappings are allowed. Some systems only allow the map of S-symbols to T-symbols. This has the advantage that it is easier to check whether a map is well-typed. Other systems allow to map symbols to composed expressions. And systems with named imports, can permit the map of an import to a structure (see below). Another difference regarding instantiation is which symbols or imports may be instantiated: We speak of **free** instantiation if arbitrary symbols or imports can be instantiated. Free instantiations must **explicitly** associate some names of S with expression of T. And we speak of **interfaced** instantiation if the declarations of S are divided into two blocks, and only the declarations in the first block — the interface — are available for instantiations. Interfaced instantiations are often **implicit**: The order of declarations in the interface of S must correspond to the order of provided T-expressions. Furthermore, instantiations may be **total** or **partial**: Total instantiations provide expressions for all symbols or imports in (the interface of) S. Finally, some systems restrict inheritance to axioms; in such systems, imports must carry instantiations for all symbols; we speak of **axiom-inheritance**.

A further distinction regards the relation between the imports and the other declarations. We speak of **separated** imports if all imports must be given at the beginning of the module; otherwise, we call them **interspersed** imports. Separated imports are conceptually easier, but less expressive: At the beginning of a module, less syntactic material is available to form expressions that can be used in instantiations.

Structures of the module (type) S over the module T provide values or implementations for the symbols declared in the module S in terms of the syntactic material of T. In particular,

every import from S into T yields a structure of type S over T, which is particularly interesting if the imports are named. Contrary to imports, **views** from S to T are structures that are explicitly declared on toplevel. Views are similar to imports in that they carry an instantiation that provides T-expressions for the symbols of S. Views are independent of S and T and change neither module. Finally, **grounded** structures are similar to views, but instead of T the global environment is used. For example, the programs implementing a module S are often grounded structures of type S. Other name used instead of structure are interpretation, link, view, instance, and (module) term/value/expression.

Functors are operations that take structures as arguments and return other structures. If modules are regarded as types and structures as values, then functors correspond to functions. A module system is **higher-order** if functors may take other functors as arguments.

Hiding permits to omit some symbols of S when importing from S. That solves the problem that S may contain auxiliary symbols that are only needed locally. A standard example used for CASL ([CoF04]) is given in Fig. 1.3 (in simplified syntax, actually CASL's hiding is a structuring operation on specifications). This describes the specification of function that sorts a list of elements of type A according to some ordering on A. The predicates `is_sorted` and `is_permutation_of` are defined by axioms and should be hidden because an implementation should only implement the sorting operation. Giving a formal semantics to hiding in the context of formal specification has proved much harder than for instantiation. One crucial difficulty is that implementations must be required to satisfy all axioms — including those about the hidden operations — even though they do not provide implementations for the hidden operations. A special form of hiding is given by declaring **private** symbols.

```
specification sorter [A: Order] {
  symbols:
    sort : List(A) -> List(A)
    hidden is_sorted : List(A) -> bool
    hidden is_permutation_of : List(A) -> List(A) -> bool
  axioms:
    forall L : List(A). (is_sorted(sort(L)) and is_permutation_of(L,sort(L)))
    [axiomatic definitions of is_sorted and is_permutation]
}
```

Figure 1.3: Hiding

Finally, some module systems are not specific to a certain logic or programming language; rather, they are defined within a logical framework and thus parametrized by the underlying logic. We call such module systems **generic** and further distinguish whether the logical framework is based on set/model theory or type/proof theory.

We say that a module system is **conservative** if every module is equivalent to a self-contained module, i.e., one that does not inherit from other modules. Such modules are also called flat, and the process of transforming a module into its flat equivalent is called **flattening**. Language features that typically prevent flattening are higher-order functors (as in Coq) and complex hiding (as in ASL).

Important Module Systems Now we are ready to describe some important specific module systems. We will loosely group the module systems into set theory-based systems, i.e., algebraic specification languages, type theory-based systems, and programming languages. Among the algebraic specification languages, we consider OBJ ([GWM+93]), ASL ([SW83, ST88]), CASL ([CoF04, MML07]), and development graphs ([AHMS99, MAH06]). The type theories with module systems we consider are IMPS ([FGT93]), PVS ([ORS92, OS97]), Isabelle ([Pau94]), Coq ([BC04]), and Nuprl ([CAB+86]). Finally, among the programming languages, we use SML ([MTHM97]) and Java ([GJJ96]) as examples.

OBJ refers to a family of languages based on variants of sorted first-order logic. It was

originally developed in the 1970s based on the Clear programming language and pioneered many ideas of modular specifications, in particular initial semantics. Virtually all module systems in use today are influenced by OBJ. The most important variant is OBJ3; Maude ([CELM96]) is a rewriting logic system based on OBJ. OBJ permits named, separated imports between modules. Instantiations are interfaced, implicit, and total. And the interface of a module may only declare imports, which can be instantiated with views.

ASL is a generic module system over an arbitrary institution ([GB92]). It uses only axiom-inheritance with renaming. The possible instantiations are abstract and given by the signature morphisms of the underlying institution. ASL introduced hiding in specifications by using instantiations in the direction opposite to the inheritance direction; thus, hiding could be treated as dual to instantiation.

The common algebraic specification language (**CASL**) was initiated in 1994 in an attempt to unify and standardize existing specification languages. As such, it was strongly influenced by other languages such as OBJ and ASL. The CASL logics are centered around partial subsorted first-order logic. Specific logics are obtained by specializing (e.g., total functions, no subsorting) or extending (e.g., modal logic or higher-order logic). In HetCASL ([Mos05]), CASL is extended to heterogeneous specifications (i.e., different logics in the same specification). CASL modules are called specifications, the imports are unnamed and interspersed. Multiple inheritance is supported by renaming, and disambiguation is also possible via parameter ascription. Instantiations are interfaced, explicit, and total; they map symbols to symbols. Views are possible and can be used in instantiations. The support for hiding is similar to ASL.

The **development graph** language is an extension of ASL specifically designed for the management of change. The modules are called theories, the imports definitional links, and the views theorem links. Theory morphisms are composable sequences of links. Definitional links are like in ASL, and theorem links represent proof obligations. Local links are used to decompose links into smaller components. The Maya system ([AHMS02]) implements development graphs for first-order logic. It is a notable implementation because — contrary to most other module systems — it does not flatten the specification while reading it in. Thus, the modular information, in particular the decompositions, is available in the internal data structures. And it turns out that this is much more robust against changes in the underlying modules and provides a good basis for a management of change. The motivation for local links is the realization that for large theory graphs proving proof obligations becomes a redundant exercise, which can be avoided: To establish a link $\mu : S \to T$, it is possible to concentrate on proof obligations induced by the local axioms of S. If possible, the axioms S imports from say S' are discharged by finding a theory morphism from S' to T. Thus, local links are introduced that only import or induce proof obligations for the local axioms of the source theory. Then a theory level calculus is used to prove theorem links by decomposing them into local ones. For example, in Fig. 1.2, a local import could be used for l; then a theorem link from `OrdList` to `Main` is postulated, and decomposed. The decomposition uses l to prove the local axioms of `OrdList` and $n \circ m$ to prove the axioms imported into `OrdList` via e.

IMPS was the first theorem proving system that systematically exploited the "little theories approach" of separating theories into small modules and moving theorems along theory morphisms. It was initiated in 1990 and is built around a custom variant of higher-order logic. The imports are unnamed and separated and do not carry instantiations; there is no renaming. Modules can be related via views, which map symbols to symbols.

PVS is an interactive theorem prover for a variant of classical higher-order logic with a rich undecidable type system. PVS offers unnamed, interspersed imports. Instantiations are interfaced, total, and implicit; they map symbols to terms. There is no renaming, but multiple inheritance is supported through a complex overloading resolution algorithm. Hiding is supported by export declarations that determine which names become available upon import.

Isabelle is an interactive theorem prover based on simple type theory ([Chu40]) with a structured high-level proof language. It provides two generic module systems. Originally, only

axiomatic type classes were used as modules. They permit only inheritance via unnamed, separated imports without instantiations. Type class ascriptions to type variables and overloading resolution are used to access the symbols of a type class. Independently, later locales were introduced as modules in [KWP99] and gradually extended. The current release offers unnamed, separated imports between locales; renaming is possible. There are no instantiations, but symbols from different locales can be identified by renaming them to the same name. Interestingly, neither module system has no notion of instantiation or structure. Instead of instantiating a locale, a locale predicate is exported to the toplevel that abstracts from all symbols and assumptions of the locale; every theorem proved in the locale is relativized by the locale predicate and exported to the toplevel. Thus, instantiation is reduced to β-reduction. In [HW06], the developers propose a redesign of the Isabelle module system, which unifies axiomatic type classes and locales and introduces structures (called interpretations).

Coq is an interactive theorem prover based on the calculus of constructions ([CH88]). It provides a higher-order module system, which is not conservative. Being higher-order it contains the analogues of all other module systems we describe.

Nuprl is an interactive theorem prover based on a very rich undecidable type theory. It does not provide a module system per se. But its type theory is so expressive that a higher-order module system can be defined in it as done in [CH00]. Dependent sum and product types are used to model modules and structures, and inheritance is reduced to subtyping.

SML provides two interconnected module systems, one for specifications and one for implementations. The specification level module system has signatures as modules. Imports are interspersed and can be named (called structure declarations) or unnamed (called inclusions); there is no renaming. Both kinds of imports carry free, explicit, and partial instantiations that map symbols to symbols or structures to structures. The implementation level module system has functors as modules. Imports between functors (called structure definitions) are named and interspersed. And the instantiations are interfaced, explicit, and total; they map symbols to symbols and structures to structures. Hiding is supported by structural subtyping.

The modules of the **Java** module system are called classes. There are two kinds of imports. Firstly, unnamed, separated imports without renaming are called class inheritance. Secondly, named, interspersed imports are called object instantiation, and the resulting structures are called objects. Instantiations are interfaced, implicit, and total, but a class may provide multiple interfaces (the constructors). They map symbols to expressions or objects to objects. Since the constructors may execute imperative code, the expressions passed to the constructor do not have to correspond to symbols or objects declared in the class. Views are possible, but the domains are restricted to so-called interfaces. Functors are subsumed by the concept of methods. Hiding is supported via private declarations.

1.1.3.3 Inter-Logic Reasoning

Inter-logic reasoning is very similar to inter-theory reasoning. Here, the motivation is to move results from one logic to another one via a translation between the logics. Most importantly, borrowing ([CM97]) means to use a translation from a logic L to a logic L' in order to use automated reasoning systems for L' to solve problems in L. Another application is the modular composition of logics.

The **systematic study** of logic translations requires a logical frameworks: Only then do the source and target logic conform to the same definition of what a logic is. Otherwise, the notion of a meaningful translation would not be definable.

Among the **model theoretical** frameworks, translations have been studied very successfully in the framework of institutions ([GB92]). HetCASL and the Hets system ([MML07]) generalize CASL to a logical framework in which translations between logics can be used within one specification. We speak of heterogeneous specifications ([Mos05]).

There have also been some results about how to use institutions to develop module systems for logics, e.g., [Tar96, MTP97].

Among the **proof theoretical** frameworks, the Logosphere project ([PSK+03]) uses LF as the basis of a formal digital library in which logics, logic translations, and theorem are stored. The only translation realized so far is one from HOL to Nuprl ([Chu40, CAB+86, NSM01, SS04]).

However, so far the majority of representations and implementations of logic translations has been realized outside of logical frameworks. Such **ad hoc translations** can be developed as separate mathematical results. While all such results make some sort of truth-preservation statement, the exact form of the translation and the meaning of the truth-preservation vary from case to case due to the lack of a logical framework.

For example, [McL06], gives a translation of Isabelle/HOL ([NPW02]) expressions to HOL Light ([Har96]) expressions that preserves truth via a meta-mathematical result about the translation. On the other hand, in [OS06] a translation in the opposite direction is given, which only uses the HOL Light proof as an oracle to construct a completely new Isabelle/HOL proof. Translations with the purpose of borrowing are given in [JM93] to translate parts of DTT into simple type theory, in [BPTF07] to translate parts of HOL ([Chu40]) into FOL, in Scunak [Bro06] to translate parts of DTT into FOL, in [Urb03] to translate Mizar ([TB85]) into FOL, and in [HS00] to translate ML to FOL.

1.1.3.4 Inter-Framework Reasoning

Inter-framework reasoning would extend the principle of inter-theory and inter-logic translations to logical frameworks. Of course, a formal truth-preservation condition cannot be given because that would require an even higher framework in which logical frameworks are defined. But it is interesting to explore if some results can be moved between logical frameworks if we take a broader view on *results*. For example, some applications that are not results in a strict mathematical sense such as databases and management of change may well be independent of the logical framework. This has not been explored so far.

1.2 Semi-Formal Languages for Mathematics

1.2.1 Mathematical Knowledge in Traditional Form

Mathematics is one of the oldest areas of human knowledge and forms the basis of most modern sciences. It provides them not only with modeling tools like statistical analysis or differential equations, but also with a knowledge representation regime based on rigorous language that can (in principle) be formalized in logic. However, mathematical knowledge is far too vast to be understood by one person. It has been estimated that the total amount of published mathematics doubles every ten – fifteen years ([Odl95]). Indeed, Zentralblatt Math ([ZBM31]) maintains a database of 2.3 million reviews for articles from 3500 journals from 1868 to 2007. If we take into account the "hard sciences", as they are represented, e.g., in the Cornell ePrint archive ([ArX94]), then we can see a similar trend: The collection is just short of half a million papers with over 80 000 submitted in 2007 alone.

Thus, the question of supporting the management and dissemination of mathematical knowledge is becoming ever more pressing. But it remains difficult. The advent of the internet, which has made more mathematics accessible than ever before does little to help in this respect. It only makes mathematics accessible in the sense that it can be viewed or downloaded, but it does not help with the organization of mathematical knowledge into a form that can be understood by machines.

Currently, the best way for this organization of mathematical knowledge is to have humans read mathematical documents or communicate to peers to build a cognitive representation of the contents in their minds, find mappings from these to the problem at hand, transform the

organized knowledge via these mappings so that they can be applied, and then finally document and communicate the results. Throughout this cycle, the documentation and communication uses natural — i.e., informal — language with interspersed formulas. This mind-based process is well-suited to doing mathematics "in the small" where human creativity is needed to create new mathematical insights. But it leads to increasing specialization, academic isolation, and missed opportunities for knowledge transfer.

The sheer volume of mathematical knowledge precludes this approach to organize mathematics "in the large". The mathematical community has sometimes joined forces on prestige projects such as the classification of finite simple groups ([Sol95]), but collaboration in mathematics is largely small-scale. Thus, it is necessary to transcend the confines of a single mind and make use of computer support. As computer programs still lack any real understanding of mathematics, human mathematicians must make structures in mathematical knowledge sufficiently explicit so that we can make use of the strengths of the computer: its ability to systematically manage extremely large data sets.

This approach has been pioneered in an area, where computer support is desired for another reason: the field of formal methods in software engineering (FMSE). In the verification of software systems, a sound logical foundation and the incorruptibility of computers are combined to obtain reliable statements about their safety-critical properties. Such computer-aided proofs rely on large amounts of formal knowledge about the mathematics of programming language constructs, data structures, and program fragments, and the productivity of FMSE in practice is restricted by the effectivity of managing this knowledge.

Even though mathematical knowledge can vary greatly in its presentation as well as its level of formality and rigor, there is a level of deep semantic structure that is common to all forms of mathematics and that must be represented to capture the essence of the knowledge. However, the large-scale structure of mathematical knowledge is much less apparent than that of formulas. Experienced mathematicians are nonetheless aware of it, and use it for navigating in and communicating mathematical knowledge. Much of this structure can be found in networks of **mathematical theories**: groups of mathematical statements, e.g., those in a monograph "Introduction to Group Theory" or a chapter in a textbook. The relations among such theories are described in the text, sometimes supported by mathematical statements called representation theorems. We can observe that mathematical texts can only be understood with respect to a particular mathematical context given by a theory which the reader can usually infer from the document. The context can be stated explicitly (e.g., by the title of a book) or implicitly (e.g., by the fact that an email comes from a person that we know works on finite groups).

If we make the structure of the context as explicit as the structure of the mathematical objects, then mathematical software systems are able to provide novel services that rely on this structure such as semantics-based searching and navigation or object classification. Over the last years this problem has been studied in the emerging field of mathematical knowledge management (MKM).

1.2.2 Mathematical Knowledge on the Web

The challenge in putting mathematics on the World Wide Web is to capture both notation and meaning in such a way that documents can utilize the highly-evolved notational forms of written and printed mathematics, and the potential for interconnectivity in electronic media. The W3C recommendation for mathematics on the web is the MathML language ([ABC+03]). It provides two sublanguages: the first — **presentation MathML** — allows to specify high-quality notations for mathematical formulas, the second — **content MathML** — is geared towards specifying the meaning.

Content MathML represents the meaning of mathematical formulas in terms of **OpenMath** objects ([BCC+04]), i.e., as tree-like expressions built up from constants and variables via function applications and bindings. In the upcoming OpenMath 3 and MathML 3 stan-

dards, content representations will be isomorphic. The meaning of constants (called "symbols" in MathML and OpenMath) is given by reference to **content dictionaries** (CDs), i.e. machine-readable and web-accessible documents that give the meaning of symbols. CDs provide a simple form of meaning to mathematical objects for the communication over the WWW: The meaning of variables is local, and that of function application and binding is well-understood and specified in the OpenMath standard.

The notion of "meaning" in the OpenMath/MathML approach is strictly structural as there is no requirements for CDs to be machine-understandable. Two mathematical objects are considered equal, iff the encoded OpenMath objects are; in particular two symbols are equal, iff their names and the URIs of their CDs are. This gives us a level of communication safety over traditional mathematics: It can no longer be the case that the author writes $\mathbb{N}$ for the set of natural numbers with 0, and the reader understands the set of natural number without 0, as the two notions of "natural numbers" are represented by different symbols (probably from different CDs). Thus, the service offered by the OpenMath/MathML approach is one of disambiguation, and any machine support for dealing with OpenMath objects can only be based on this.

The standardized XML-based universal syntax for mathematical formulas depends on a notion of content dictionaries as a context representation. CDs can be seen as a very simple representation of mathematical background knowledge that enables formula disambiguation and communication of mathematical objects at a web scale. But the lack of machine-understandable intra-CD knowledge structure and inter-CD relations preclude higher-level machine support.

The OMDoc format ([Koh06]) represents mathematical knowledge at three levels: **objects**, i.e., content representations of mathematical objects and propositions about them, **statements**, e.g., symbols, axioms, definitions, theorems, and proofs, and **theories**, i.e., a representation of theory graphs. For communication and archival, OMDoc provides a basic, content-oriented infrastructure for documents as well. Mathematical objects are represented as OpenMath objects, statements, theories, and documents have a custom XML-based syntax. The use of XML as a syntactical basis and Uniform Resource Identifiers (URIs) throughout make the format web-scalable. This, together with the fact that all formal mathematical elements of the language can be augmented or even replaced by natural language text fragments distinguishes the format from the purely formal approaches mentioned in Sect. 1.1. From the OpenMath/MathML languages, OMDoc is distinguished by its rich infrastructure for inter-theory relations.

1.3 Motivation

1.3.1 Combining Model and Proof Theory

Objective In Sect. 1.1.1, we described two groups of foundations for mathematics, the set theoretical and the type theoretical ones. And we described two views on logics, the model theoretical and the proof theoretical one. We mentioned that mathematicians and computer scientists who favor set theory also tend to favor model theory, and similarly researchers favoring type theory tend to favor proof theory. We hold that the distance between these mathematical directions is unnatural and undesirable, and we attempt to contribute to mathematics and computer science by fostering a closer relationship between them.

Using the words of an unnamed reviewer of a related paper of ours, we can state our first motivation by saying we are "trying to bridge the gap between two strongly conflicting cultures and a[t]titudes in logic and computer science: model theory and proof/type theory", which is worthwhile because "attempts to bridge these two cultures are rare and rather timid". Thus, we want to provide a logical framework that combines model and proof theory. It is our goal not only to provide a framework for the study of logics that connects the two views, but also to bring together research and researchers from both views.

Requirements Our design choices for this framework are based on the following concrete requirements. The framework should

(R1) subsume existing popular model- and proof-theoretic frameworks,

(R2) support logical reasoning on the intra-theory, inter-theory, and inter-logic level,

(R3) not commit to any particular syntax for mathematical objects, propositions, and proofs,

(R4) provide an implementable and scalable interface infrastructure for logics.

The motivation for **(R1)** is obvious: Any framework not compatible with popular predecessors is doomed from the start. **(R2)** expresses that we want to exploit the levels of logical reasoning. As seen in Sect. 1.1.3, there are various systems providing strong support on the intra-theory and the inter-theory level. However, the support on the higher levels is small. The number of logic translations conducted within proof theoretical logical frameworks is restricted to a single translation in Logosphere. And although the model theoretical system Hets supports various translations, currently, these translations are hard-coded. Furthermore, while many formal languages support module systems, translations that preserve the modular structure are much less understood. Therefore, a strong support for module systems would make our framework more attractive for users as an interface language. We will get back to this in Sect. 1.3.2.2. **(R3)** is necessary because users are always unwilling to leave established notations behind. Therefore, an abstraction layer above the actual syntax is necessary. However, for usability in practice, straightforward encodings of specific syntaxes must be possible. Finally, **(R4)** expresses our commitment to contribute to practical applications that go beyond toy examples and — borrowing Frank Pfenning's words — help researchers get work done.

Among the model theoretical frameworks, **(R1)** makes a strong statement in favor of institutions ([GB92]). The institution-independent research has reached textbook strength ([Dia08]); and the semantics of various specification languages are based on or related to institutions such as OBJ ([GWM+93]), ASL ([ST88]), ASL+ ([SST92]), CASL ([CoF04]), Maude ([CELM96]), and development graphs ([AHMS99]). Furthermore, institutions are very strong with respect to **(R2)** and **(R3)**: Institutions provide a very strong framework for logic-independent high-level specification as shown by languages such as ASL, which are fully parametric in the underlying base institution.

Institutions provide abstractions for propositions, truth, and model theory. The propositions and models are described by objects of set theory, and truth is a relation between them. However, the framework does not cover proof theory. Furthermore, the strength of institutions — their abstraction — entails the weakness that they do not formalize the structure of mathematical objects at all. Various attempts have been made to remedy this. Charters and parchments were added by the developers of institutions ([GB86]) to provide a more concrete framework. These were used in [MTP97] to represent logics in a universal logic. A different approach to make institutions more concrete was undertaken in [Mos99]. Proof theoretical structure for institutions was already suggested in [GB92] following the Lambek-Scott approach ([LS86]) of proofs from one sentence to another one. In [FS88] and [Mes89] *entailment systems* were introduced to capture the proof theoretic consequence relation, i.e., proofs from a set of sentences to a single sentence. In [MGDT05] *proof theoretic institutions* were introduced that use categories of proofs from a set of sentences to another one.

However, these approaches suffered from — besides some details — the simple fact that they were undertaken by researchers leaning towards set and model theory. Only few attempts were undertaken to combine institutions with the equally numerous work already existing in type and proof theory. A notable exception is the work in [HST94] and [Tar96], which suggests the use of LF, the Edinburgh Logical Framework ([HHP93]), as a universal logic, in which other logics are specified via institution translations. However, a deep and long-lived attempt has so far been missing.

To fully achieve **(R1)**, we have to look at the type and proof theoretic frameworks. We have already mentioned in Sect. 1.1.2 that these can be organized in the λ-cube, and we gave Isabelle ([Pau94]), LF ([HHP93]), and Coq ([BC04]) as example frameworks. Our framework of choice is LF. LF is a variant of dependent type theory related to Martin-Löf type theory ([ML74]). It includes simple type theory ([Chu40]), which forms the base of Isabelle, and is included by the calculus of constructions ([CH88]), which forms the base of Coq.

LF was designed with the specific goal to be a logical framework. Therefore, LF is a very simple language that provides exactly those constructs needed to specify formal systems following the judgments-as-types approach ([Pfe01, ML96]). In particular, all mathematical objects, propositions, and proofs are uniformly represented as LF-terms and judgments about them as LF-types. This permits very elegant representations of syntax and proof theory of logics.

Thus, LF is distinguished from Isabelle and Coq, which tend to put more emphasis on providing a foundation of mathematics. Coq is usually not seen as a logical framework even though it could be used as one; and while Isabelle is a logical framework, it is mainly used in the instantiations Isabelle/HOL for higher-order logic and Isablelle/ZF for Zermelo-Fraenkel set theory. This is reflected in the literature about and the work conducted in the systems: There is a large body of work on the study of logics in LF (e.g., [HHP93], [Pfe00], [HST94], [AHMP92], [AHMP98], [NSM01]), whereas Isabelle and Coq are distributed with large libraries of formalized mathematics ([Isa08], [Coq08]) and provide strong support for semi-automated proof search (see ([Nip02]) for Isabelle and [BC04] for Coq).

A weakness of LF is that dependent types make the system more complicated. For example, the model theory even for simple type theory as in Isabelle is already complex. Thus, a satisfactory connection between LF and model theory in general is so far missing. Furthermore, systems for LF (e.g., Twelf [PS99]) tend to focus on intra-theory reasoning. While some type theoretic systems provide module systems as described in Sect. 1.1.3, the problem of high-level reasoning is much more complicated and less well-understood in type theoretical frameworks because they focus on concrete syntax instead of abstracting from it like the model theory-based frameworks.

As to the implementations required by **(R4)**, the type theoretical frameworks are much stronger than the model theoretical ones. The goal behind implementations is the automation of logical reasoning, which naturally favors proof theoretical approaches. Only on the higher levels of logical reasoning can model theoretical approaches show significant advantages over proof theoretical ones (e.g., [AHMS02], [MML07]). The scalability requirement of **(R4)** is the hardest one with the least precedent in existing systems. We will focus on it separately in Sect. 1.3.2.

Course of Action Thus, exempting scalability, we have a somewhat complementary picture with model and proof theoretical frameworks exhibiting different strengths. And from that, we draw the following more precise objective: We attempt to combine the set/model theoretical framework of institutions and the type/proof theoretical framework of dependent type theory.

To that end, we (i) extend the concept of institutions with an abstract notion of proof theory. Thus, we lay the ground to integrate type/proof theory into set/model theory. We call the notion we obtain in this way "logic". Conversely, we (ii) give a model theoretical semantics and then a logic for for dependent type theory (DTT). Thus, we lay the ground to integrate set/model theory into type/proof theory. Finally, we (iii) combine the two results by building a universal logic based on DTT, which yields our logical framework.

Limitations When designing a logical framework, the decision what not to integrate is as important as the decision what to integrate. Allowing for too many features makes the framework too complex to implement or reason about or too abstract to be meaningful. The main limitations of our framework are the following.

- Institutions favor two-valued logics. This does not exclude intuitionistic logics, which

admit truth value-based interpretations even if negation is not interpreted classically. But it limits the applicability to fuzzy, linear, or paraconsistent logics even though these are not excluded a priori.

- LF judgments obey the rules of contraction, weakening, and (whenever well-formed) exchange. This limits applications to substructural logics because their encodings become less natural. Future work could replace LF with LLF ([CP02]) or CLF ([WCPW02]), which were designed to overcome this drawback.

- Our notion of logic translation uses compositional maps between syntactical expressions, which is a very reasonable abstraction. However, this excludes some interesting cases, in particular, those proof translations where proof rules are mapped to admissible rules (as opposed to derivable rules) such as cut elimination translations. While non-compositional translations can be captured naturally in both model and proof theoretical frameworks, a combination of the two is more complicated. Therefore, we defer such translations to future work. For example, using the recently matured Delphin ([PS08]) instead of LF seems promising.

1.3.2 Logical Knowledge Management

There are two directions within MKM: They correspond to the two possible bracketings of MKM. Firstly, M(KM) applies mathematical methods to knowledge management problems in all areas of science. This involves techniques such as formal languages and standardized interfaces. For example, CML ([MRR03]) is a markup language for chemistry. Secondly, (MK)M develops and applies knowledge management methods particularly suited for mathematics. This involves the representation, storage, distribution, analysis, and display of mathematical content.

By creating the acronym LKM in analogy to MKM, we can succinctly express our second core motivation. Like MKM, its meaning is twofold.

1.3.2.1 Logical (Knowledge Management)

No L(KM) in Practice From a practical point of view, the characteristic problem of logical or formal methods is that their cost is high. To apply logical methods, all problems and their solutions have to be stated fully formally. Only then can machines be used to find or verify solutions. In practice, this is almost never done. The only area outside mathematics to which it can realistically be applied at all is information technology, on which we will focus.

Requirements for information processing systems are stated in semi-formal or even fully informal languages. And the vast majority of implementations is carried out in informal programming languages, by which we mean languages with formal syntax but mainly informal semantics like Java and C++. We will not go into the details here of the numerous attempts at giving a formal semantics for these languages, such as [ABB+05]. And even programming languages with a high degree of formality in their semantics like SML, which admit a very strong static safety analysis, are rarely used in practice outside theoretical computer science.

As a tell-tale example, we can consider the "Common Criteria for Information Technology Security Evaluation" ([Com98]), an international standard (ISO/IEC 15408) for computer security. It defines seven "Evaluation Assurance Levels". But only the highest of these levels (And even the second-highest is rarely used.) evaluates a formally verified design. And any Common Criteria evaluation is only used in a situation with very high security requirements to begin with.

Two Ways towards L(KM) in Practice Therefore, a crucial direction of past and future research in L(KM) is how to make logical methods more applicable in practice. We will focus on two promising ways how to do this.

Firstly, using inter-theory and inter-logic reasoning makes it possible to abstract from the implementation details and concentrate the power of logic on the higher-level structure. This is incidentally the most important area of verification because errors in higher levels are typically far more costly to fix. This can be seen as a **top-down** approach: Formal and highly structured specifications are used at the higher levels of design and implementation and applied to abstractions of the actual verification targets (e.g., documents or programs). Then logical methods can be pushed down as far as time and cost constraints admit. We have provided an overview over the numerous languages used for this purpose in Sect. 1.1.3.2.

And secondly, incorporating informal representations into logical reasoning can provide a way to apply logical methods without the cost of full formalizations. This can be seen as a **bottom-up** approach: As much formal markup of the actual verification target is introduced as the external constraints admit, and applications exploit whatever formal structure is present. This approach has two important pragmatic advantages. Firstly, its appeal to users is higher because it can provide them with immediate small but low-cost benefits such as dependency checking, search, and management of change. And secondly, the simpler, less formal languages are significantly easier to learn, which means that implementation tasks can be assigned to less qualified researchers whose time is less valuable. This approach is far less explored than the former. OMDoc ([Koh06]) was a pioneer in this area by combining formal and informal representations of mathematical knowledge.

Meeting in the Middle with Mmt We contribute to L(KM) by providing the framework Mmt, which connects the top-down and the bottom-up approach.

The design of OMDoc suffers from a bias towards the bottom-up approach. The semantic markup it provides itself has no clear semantics because the specification — despite its length — is incomplete or vague in crucial aspects. In particular, this applies to the concepts of multiple inheritance, which is at the core of the inter-theory reasoning, and semantic reference, which is at the core of large-scale integration. Furthermore, in an attempt to keep OMDoc as appealing to users as possible, many primitive notions were introduced, which if taken together make the current OMDoc 1.2 specification a formidable read. This severely slowed down the implementation of OMDoc applications, which are necessary to make the bulky XML representation transparent to the user. A pointed way to phrase it would be that OMDoc currently combines the human-unfriendliness of XML with the machine-unfriendliness of informal language.

A crucial design choice in the ongoing development of its successor OMDoc 2 was to introduce a distinction between strict and pragmatic OMDoc. In analogy to the upcoming MathML 3 specification ([ABC+03]), pragmatic OMDoc provides user-friendly apparently primitive notions that are in fact defined by a possibly partial translation to strict OMDoc. And strict OMDoc provides a lean markup language with a fully formal semantics while retaining an expressivity high enough to naturally represent a wide variety of object languages. Mmt introduces the language and its semantics that is the formal counterpart of the core of strict OMDoc 2.

1.3.2.2 (Logical Knowledge) Management

(LK)M in Practice Mathematical theories have been studied by mathematicians and logicians in the search of a rigorous foundation for mathematical practice. They have been formalized as collections of symbol declarations, i.e., names for mathematical objects that are particular to the theory, and axioms, i.e., logical formulas which state the laws governing the objects described by the theory. A key research question was to determine conditions for the consistency of mathematical theories.

It is one of the critical observations of logic that theories can be extended without endangering consistency, if the added formulas can be proved from the axioms or provide definitions of newly introduced symbols. This leads to the principle of **conservative extension** that are safe

for mathematical theories and that permit to narrow down possible sources for inconsistencies to small sets of axioms. Even though this has theoretically been known to mathematicians for almost a century, it has only been an explicit object of formal study and exploited by mathematical software systems in the last decades.

And as seen in Sect. 1.1.3, (LK)M has had notable success in applying formal methods to mathematics. However, (LK)M applications tend to have a major restriction, namely a lack of **scalability**. A simple reason for this is that (LK)M applications implementing logics or even logical frameworks are often only understood by relatively small numbers of people because of the high degree of abstraction. Thus, resources to implement services for logical frameworks are very limited.

In Sect. 1.3.1, we listed scalability as a requirement. And in Sect. 1.1.3.4, we indicated that there may be possibilities to support logical reasoning on the inter-framework level. These aspects combine to our goal of a generic scalable module system, for which we see applications in areas such as content representation, web compatibility, documentation, and management of change.

Content Representation Most (LK)M systems hold all their knowledge either in complex data structures in main memory or in ASCII files on the local file system. And knowledge is only accessible if in the former form. A central process is the loading of a file, which involves reading it from disk, parsing it, and building the internal data structures for the knowledge represented in the file. This does not scale to the situations when holding all accessible information in main memory is inefficient and when file system paths are not expressive enough to organize theories meaningfully.

On the other hand, operations such as storage, search, structural analysis, retrieval of document fragments, aggregation of documents, or editing and viewing document parts could in principle be delegated to other applications. But in most cases, these tasks require the use of the non-scaling primary application because no other application is able to understand the syntax used in an input file. A simple solution for this problem is to use XML encodings, in which both syntax and semantic structure are easily parsable.

While some languages offer XML encodings such as Isabelle, Twelf, and Mizar, these are typically not the best-developed component. For example, Isabelle — one of the most advanced (LK)M applications — only offers a not officially documented partial encoding of the core language. And most existing XML-encodings such as the one for Coq ([BC04]) are one-way encodings, i.e., the system can output but not input the XML documents. Only the encodings of Mizar ([TB85, Urb03]) and Matita can be processed in a more sophisticated way.

Another problem is that there are no scaling interface languages that can faithfully represent the content structure on the inter-theory or even on the inter-logic level. Both mathematically and pragmatically, the key problem here is to find an interface language that is so simple that it is easy to translate out of it, but yet so expressive that it is easy to translate into it.

Web Compatibility One major hurdle for the web scalability of (LK)M applications is standards compliance. This means the rigorous use of URIs as identifiers, of XML documents as interface formats, and of OpenMath ([BCC+04]) or MathML ([ABC+03]) for the representation of mathematical objects. For example, all symbols of the main syntax (e.g., λ in a system implementing lambda calculus) should be declared in OpenMath content dictionaries. Instead, applications that provide XML encodings favor custom XML elements that introduce additional complexity into the interfaces (e.g., an element `<lambda>`). The possibility to use URIs as identifiers — a triviality in MKM — is not supported by most applications. As a consequence, standard techniques for the referencing of document fragments fail.

This is aggravated by the fact that (LK)M systems are almost exclusively applied by highly trained users, and the documentation is targeted at them. Therefore, it is almost impossible

for developers of knowledge management services to apply their techniques to logical languages. On the other hand, the scarcity of resources often precludes developers of logical systems to provide such services themselves.

Documentation The inline attribution of documentation implemented in languages such as Javadoc ([Jav04]) is a matter of course in software engineering. However, most (LK)M systems only provide the possibility to mark certain parts or lines as comments. Moreover, even though most comments speak about mathematical objects, the comments cannot natively represent them. This makes it hard to write content in a way that can be used to extract documentation and interfaces. Exceptions are Isabelle ([Pau94]), which features a LaTeX authoring component, and Mizar ([TB85]), which generates journal articles from Mizar articles.

Management of Change In multi-user development situations, concurrent versioning systems such as CVS and SVN are indispensable. However, both compare text files line by line, which is not optimal for logical knowledge represented in text files. There we often have a distinction between semantically relevant and irrelevant changes. For example, changes such as whitespacing, comments, α,β,η-conversion, redundant bracketing, and the order of declarations are or can be semantically irrelevant. And such changes are often not locally constrained by the line structure. Furthermore, semantically relevant changes typically have far-reaching consequences that may not only affect other parts of the file but also completely different files.

Intelligent versioning systems can in principle detect such changes and determine their semantic relevance. If they are provided with dependency relations between document parts (or means to infer them), they can derive which changes to propagate to which document parts. However, such systems are very costly to design and implement, which makes them infeasible for single systems, especially if they are maintained by small research groups.

Generic (LK)M Services It is an important observation that the services discussed above do not or only partially depend on the specific logical system. Thus, they can be implemented generically provided that appropriate representation languages are available. This was a major motivation behind the development of OMDoc, which offers the most advanced generic representation language applicable to mathematical knowledge. Furthermore, for a variety of services, prototype implementations are already available.

However, OMDoc is not appropriate for logical knowledge: Due to their high cost, logical methods are only used when correctness is paramount, and OMDoc is not formally rigorous enough to be trustworthy. Furthermore, it lacks adequate support for the representation of the logics themselves. Thus, translations between logics could be represented even less. This led to a central motivation of the work presented here.

Bridging the Gap with Mmt We want to contribute to (LK)M by bridging the gap between logical knowledge and generic knowledge management services. Our system Mmt remedies the shortcomings of OMDoc 1.2 by providing a simple fully formal language with a formal semantics for knowledge representations. It unifies the higher levels of logical reasoning by using the same primitive notion — that of a theory — to represent logical frameworks, logics, and theories. Similarly, it only uses one primitive notion — that of a theory morphisms — for inter-framework, inter-logic, and inter-theory translations. Thus, it can be called a meta-meta-logical framework in the sense that an Mmt specification contains (or refers to) theories for the used logic and logical framework.

Furthermore, Mmt offers a module system that combines the simplicity and the expressivity requirements mentioned above. In particular, Mmt permits natural representations of all of the module systems reviewed in Sect. 1.1.3.2 except for the higher-order systems and the complex

hiding constructs. Furthermore, the same module system can be used on all levels of logical reasoning.

1.4 Outline

Content In Part II, we focus on our first objective, the combination of model and proof theory. As already laid out in Sect. 1.3.1, we pick the logical frameworks of institutions and dependent type theory and combine them. First, we extend the concept of institutions with an abstract notion of proof theory in Sect. 3 to obtain our definition of logic. In Sect. 4, we give a model theoretic semantic for dependent type theory (DTT) and use it to define a logic for DTT. And in Sect. 5, we combine the two results and use the logic for DTT as a logical framework. This reflects the objectives described at the end of Sect. 1.3.1.

In Part. III, we focus on our second objective, logical knowledge management. In Sect. 6, we develop MMT as a module system for logical knowledge and as a meta-meta-logical framework. In Sect. 7, we show how the logical framework obtained in Part. II is represented in MMT. And in Sect. 8, we demonstrate the scalability of MMT by sketching the design of future applications.

We will begin with an introduction to and overview over the needed prerequisites in Sect. 2.

Dependencies We briefly review the semantic dependency between the sections in order to help readers who are only interested in parts of this text.

Sect. 2 reviews known concepts. Readers familiar with the basic definitions in institutions and dependent type theory can skip it; however, they should have a look at Not. 2.1, 2.2, 2.19, 2.21, and 2.23. Furthermore, the examples introduced in Sect. 2.2 will serve as running examples throughout Part II and Sect. 7.

Part II relies heavily on institutions as introduced in Sect. 2 and is self-contained with respect to dependent type theory. Sect. 4 is self-contained except for Sect. 4.10 and can be read independently. Readers with no background in category theory may try to skip Sect. 4.4 to 4.9.

Within Part III, Sect. 7 and 8 are quite independent. Sect. 6 and 8 do not rely on Part II. And for Sect. 8, basic knowledge about XML is assumed. Readers with no background in type systems may want to skip Sect. 6.3.

Chapter 2

Preliminaries

It is in the nature of this work to form a connection between rather remote fields of mathematics, namely set/model theory and type/proof theory. Furthermore, it is in the nature of logical frameworks to be very abstract mathematically and demand a great deal of patience and persistence from learners. In particular, both our starting points, the logical frameworks of institutions and DTT, are typically not fully understood before completing a one-semester course on them, and almost no student does so for both of them. Therefore, we will begin by giving a gentle bottom-up introduction to both of them in Sect. 2.2, which only assumes the basic notions of set theory.

Then Sect. 2.3 will list all necessary definitions in a technically precise way. Those definitions form the official basis of the following. Before that we briefly list some general meta-level conventions and notations in Sect. 2.1.

2.1 Basic Concepts

The following notations are used throughout this text.

Notation 2.1. We use the following basic notations on the meta-level.

- $\mathbb{N}$ and $\mathbb{Z}$ denote the natural and integer numbers. 0 is a natural number.
- For $m, n \in \mathbb{Z}$, $a_m, \ldots, a_n$ denotes a finite list or sequence of expressions. If $m > n$, it denotes the empty sequence.
- Sometimes we write $f(-)$ instead of f to stress that f is an unapplied function.
- If f is a function with domain D and $A \subseteq D$, we write $f(A)$ for the set $\{f(a) \mid a \in A\}$.
- If $f(a)$ is a function or functor, we write $f(a)(b)$ for the curried application of $f(a)$ to b.
- We use the meta-variable _ for arbitrary irrelevant expressions.

Notation 2.2. We frequently reserve letters as meta-variables for certain concepts. Within the scope of such a reservation, the reserved letters denote instances of the respective concept unless mentioned otherwise.

Notation 2.3. We use the following abbreviations throughout this text. They will be introduced again when they are used, and we only collect them here for convenient lookup.

FOL	first-order logic
ML	modal logic
MLTT	Martin-Löf type theory
DTT	dependent type theory
LF	the Edinburgh logical framework
MKM	Mathematical Knowledge Management

2.2 Institutions and Dependent Type Theory, bottom-up

The semantic structure of this section is multi-dimensional. In the present linearization of the dependency ordering between the sections, we follow the levels of logical reasoning: We describe the intra-theory level in Sect. 2.2.1, the inter-theory level in Sect. 2.2.2, and the inter-logic level in Sect. 2.2.3.

Each section has five subsections organized as follows. First we give an example for the syntax, then an example for the model theory. Then we abstract from the examples to obtain the model theoretical framework of institutions. Then we give an example for the proof theory, and then we abstract from the syntax and the proof theory to obtain the proof theoretical framework of dependent type theory. The structure is given in the table below, which we will reproduce in several places to ease navigation.

We use theories for monoids and groups within FOL as examples on the intra-theory level. On the inter-theory level, we translate from monoids to groups. Then we introduce modal logic in Sect. 2.2.2.6 so that we can use a translation from ML to FOL as an example on the inter-logic level.

2.2.1 The Intra-Theory Level

	Intra-Theory	Inter-Theory	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.1.1 Syntax

Formulas The motivation of FOL formulas is to provide formal equivalents for the natural language making up the core of reasoning. Thus, FOL introduces, for example, the symbols $\wedge$ and $\forall$ for "and" and "for all". The formulas are certain strings that are built up inductively from atomic formulas. Since the induction is independent of the specific choice of atomic formulas, we give the induction steps first:

- Nullary connectives: tt (truth) and ff (falsity) are formulas.
- Unary connective: If F is a formula, then so is $\neg F$ (negation).
- Binary connectives: If F and G are formulas, then so are $(F \wedge G)$ (conjunction), $(F \vee G)$ (disjunction), and $(F \Rightarrow G)$ (implication).
- Quantifiers: If F is a formula and x is a variable, then so are $\forall x F$ and $\exists x F$.

Furthermore, we use $F \equiv G$ (equivalence) as an abbreviation of $(F \Rightarrow G) \wedge (G \Rightarrow F)$. Here variables are a fixed collection of objects that serve as names. The precise choice for this collection is irrelevant as long as there are infinitely many and as long as none of them is a symbol used anywhere else. For example, $\{x_n \mid n \in \mathbb{N}\}$ can serve as the set of variables. In $\forall x F$

and $\exists x F$, every occurrence of x in F is called **bound**, variable occurrences in a formula that are not bound are called **free**. A formula without free variables is called **closed**. The closed formulas are the **propositions** (also called sentences).

Signatures It is a crucial feature that FOL does not commit to any specific choice of elementary mathematical objects or propositions. Rather, FOL is a family of languages that is indexed by a collection of so-called **signatures**. A signature is a triple $(\Sigma_p, \Sigma_f, arit)$ where

- Σ_f is a set of names (**function symbols**),
- Σ_p is a set of names (**predicate symbols**),
- Σ_f and Σ_p are disjoint,
- $arit$ assigns to every function of predicate symbol a natural number (its **arity**).

The function symbols are place-holders for mathematical objects. They are used to build up the **terms**, which represent mathematical objects. And the predicate symbols represent elementary propositions about mathematical objects. They are used to build up the **atomic formulas**. Furthermore, we use **equality** as a fixed predicate symbol. Then the terms and atomic formulas are defined as follows:

- Every variable is a Σ-term.
- If $f \in \Sigma_f$ and $arit(f) = n$ and if $t_1, \ldots, t_n$ are Σ-terms, then $f(t_1, \ldots, t_n)$ is a Σ-term.
- If t_1 and t_2 are Σ-terms, than $t_1 \doteq t_2$ is an atomic Σ-formula.
- If $p \in \Sigma_p$ and $arit(p) = n$ and if $t_1, \ldots, t_n$ are Σ-terms, then $p(t_1, \ldots, t_n)$ is an atomic Σ-formula.

For example, a signature Σ^M for monoids is given by

- $\Sigma_f^M = \{comp, unit\}$,
- $\Sigma_p^M = \{invertible\}$,
- $arit^M : comp \mapsto 2, unit \mapsto 0, invertible \mapsto 1$.

The intended meaning is that *comp* is a binary operation on mathematical objects, *unit* a constant object, and *invertible* a proposition about one object. A signature Σ^G for groups is obtained by adding a unary function symbol *inv* for the inverse of an object.

Theories Theories are pairs (Σ, Δ) of a signature Σ and a set Δ of Σ-sentences. The elements of Δ are called the axioms of the theory. For example, for the signature Σ^M, a reasonable choice Δ^M is the set containing the sentences

- $\forall x_1 \forall x_2 \forall x_3\ comp(x_1, comp(x_2, x_3)) \doteq comp(comp(x_1, x_2), x_3)$,
- $\forall x_1\ (comp(x_1, unit) \doteq x_1 \wedge comp(unit, x_1) \doteq x_1)$,
- $\forall x_1\ (invertible(x_1) \equiv \exists x_2 (comp(x_1, x_2) \doteq unit \wedge comp(x_2, x_1) \doteq unit))$.

Then we obtain a theory $Monoid = (\Sigma^M, \Delta^M)$. If we add the axiom

$$\forall x_1\ comp(x_1, inv(x_1)) \doteq unit,$$

we obtain a theory *Group* for groups.

	Intra-Theory	Inter-Theory	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.1.2 Model Theory

Models The model theoretical view on logic interprets the terms as mathematical objects. Formulas are treated as propositions about the objects, their semantics is that they are true or false. This is relative to a fixed but arbitrary interpretation I for the function and predicate symbols. In the context of institutions, it is common to call the interpretations *models*. A model I for a signature $\Sigma = (\Sigma_f, \Sigma_p, arit)$ provides interpretations for the symbols as follows:

- a set $[\![\iota]\!]^I$ of mathematical objects,
- for every $f \in \Sigma_f$ with $arit(f) = n$, a mapping $[\![f]\!]^I : ([\![\iota]\!]^I)^n \to [\![\iota]\!]^I$ (where A^n denotes the n-fold cartesian power of A with $A^0 = \{\varnothing\}$),
- for every $p \in \Sigma_p$ with $arit(f) = n$, a relation $[\![p]\!]^I \subseteq ([\![\iota]\!]^I)^n$.

For example, the integers are expressed as a model $I = Int^M$ of the signature $\Sigma^M$by:

- $[\![\iota]\!]^I = \mathbb{Z}$,
- $[\![comp]\!]^I(a, b) = a + b$,
- $[\![unit]\!]^I = 0$,
- $[\![invertible]\!]^I = \mathbb{Z}$.

Int^M is extended to a model Int^G of the signature Σ^G by adding the interpretation $[\![inv]\!]^I(a) = -a$.

Satisfaction Clearly, every Σ-model I extends to a function from all closed terms to $[\![\iota]\!]^I$. But I does not provide a way to interpret variables. Thus, assignments α into I are introduced: They are mappings from the set of variables to $[\![\iota]\!]^I$. Then we can define an interpretation function $[\![-]\!]^I_\alpha$ from terms to $[\![\iota]\!]^I$ as follows:

- $[\![x]\!]^I_\alpha = \alpha(x)$ for a variable x,
- $[\![f(t_1, \ldots, t_n)]\!]^I_\alpha = [\![f]\!]^I([\![t_1]\!]^I_\alpha, \ldots, [\![t_1]\!]^I_\alpha)$ for $f \in \Sigma_f$ (where the empty tuple in case $n = 0$ is $\varnothing$),

Furthermore, we can define the truth of a formula F in I under α, written $I, \alpha \models F$.

- $I, \alpha \models_\Sigma p(t_1, \ldots, t_n)$ iff $([\![t_1]\!]^I_\alpha, \ldots, [\![t_n]\!]^I_\alpha) \in [\![p]\!]^I$ for $p \in \Sigma_p$,
- $I, \alpha \models_\Sigma F \wedge G$ iff $I, \alpha \models_\Sigma F$ and $I, \alpha \models_\Sigma G$,
- $I, \alpha \models_\Sigma F \vee G$ iff $I, \alpha \models_\Sigma F$ or $I, \alpha \models_\Sigma G$,
- $I, \alpha \models_\Sigma F \Rightarrow G$ iff $I, \alpha \not\models_\Sigma F$ or $I, \alpha \models_\Sigma G$,
- $I, \alpha \models_\Sigma \forall x F$ iff $I, \alpha^x_a \models_\Sigma F$ for all $a \in [\![\iota]\!]^I$,
- $I, \alpha \models_\Sigma \forall x F$ iff $I, \alpha^x_a \models_\Sigma F$ for some $a \in [\![\iota]\!]^I$.

Here α^x_a denotes the assignment that is as α except that it maps x to a. If $I, \alpha \models_\Sigma F$, we say that F holds in I or that I satisfies F under the assignment α. $I, \alpha \not\models_\Sigma F$ denotes the opposite. Finally, we say $I \models_\Sigma F$ if F holds under all assignments.

Then we can check whether Int^M satisfies all axioms of the theory *Monoid*. And this is of course the case. Thus, we say that Int^M is a model of the theory *Monoid*. As another example, we can see that even $Int^M \models_{\Sigma^M} \forall x\ invertible(x)$.

	Intra-Theory	Inter-Theory	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

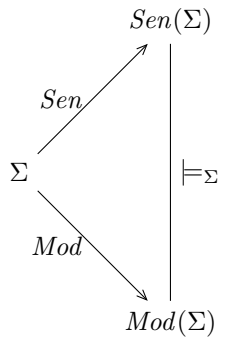
2.2.1.3 Institutions

Institutions By abstracting from the model theory for FOL, we reach a preliminary definition of an institution as follows. An **institution** is a tuple $(Sig, Sen, Mod, \models)$ such that

- Sig is a class, the class of **signatures**,
- $Sen : Sig \to \mathcal{SET}$ is a mapping assigning to every signature a set, the set of **sentences**,
- $Mod : Sig \to \mathcal{CLASS}$ is a mapping assigning to every signature a class, the class of **models**,
- $\models$ is a family of relations $\models_\Sigma$ for every signature Σ such that $\models_\Sigma \subseteq Mod(\Sigma) \times Sen(\Sigma)$ is a relation, the **satisfaction relation** between Σ-models and Σ-sentences.

Here, "class" is a concept of axiomatic set theory used for a large collection of sets. In particular, we can have the class of all sets (but not the set of all sets). The class of all sets is denote $\mathcal{SET}$. And the collection of all classes is denoted $\mathcal{CLASS}$.

This can be visualized as follows



Then it is obvious how to turn FOL into an institution $\mathcal{FOL}$ using the above material:

- $Sig^{\mathcal{FOL}}$ is the class of FOL-signatures,
- $Sen^{\mathcal{FOL}}$ assigns to a signature the set of closed formulas,
- $Mod^{\mathcal{FOL}}$ assigns to a signature the class of models,
- $\models^{\mathcal{FOL}}$ is the satisfaction relation of FOL.

Note that $Sen^{\mathcal{FOL}}(\Sigma)$ only contains the Σ-sentences and not all Σ-formulas. Thus $\models^{\mathcal{FOL}}_{\Sigma}$ can be a relation between models and sentences without referring to assignments.

There is a second way to turn FOL into an institution. This institution is called the institution $\mathcal{FOL}^{Th}$ of FOL-theories:

- *Sig* is the class of FOL-theories,
- *Sen* assigns to a theory (Σ, Δ) the set of closed Σ-formulas,
- *Mod* assigns to a theory (Σ, Δ) the class of Σ-models I such that $I \models_\Sigma F$ for all $F \in \Delta$,
- $\models$ is the satisfaction relation of FOL.

The construction of the **institution of theories** is possible for every institution.

Truth and Consequence Finally, for an arbitrary institution, the **validity** of a proposition can be defined: The Σ-formula F is valid if it is satisfied in all Σ-models. This is denoted by $\models_\Sigma F$, and we say that F is a Σ-theorem. To distinguish the model-dependent and the model-independent notions of truth, we will use the names satisfaction and validity, respectively.

Furthermore, we can define **consequence** between propositions: The Σ-formula F is a consequence of the set of Σ-formulas Δ, if all models satisfying all formulas in Δ also satisfy F. This is written $\Delta \models F$, and we say that F is a (Σ, Δ)-**theorem**. In particular, validity is the special case of consequence for $\Delta = \varnothing$.

	Intra-Theory	Inter-Theory	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.1.4 Proof Theory

Judgments and Proofs The central concept of proof theory is that of **judgments**: They act as the connection between propositions and proofs. In the simplest case a judgment asserts the truth of a proposition. For a proposition F, this could be written as $\vdash F$ or simply as F. To reason about FOL, a more general form of judgments is needed, namely:

$$\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash F$$

where the x_i are variables and the F_i and F are formulas with free variable occurrences of at most the variables $x_1, \ldots, x_m$. This judgment is usually written as $F_1, \ldots, F_n \vdash F$, i.e., the variables are left implicit. Its intuition is the following: If arbitrary fixed terms are substituted for $x_1, \ldots, x_m$, then the truth of the assumptions $F_1, \ldots, F_n$ has as a consequence the truth of F.

Proofs are special objects that provide evidence for a judgment. Every proof proves exactly one judgment, and a judgment may or not have a proof. If p is a proof of J, we write $p : J$.

Rules and Calculi A (inference) **rule** is a relation between judgments. It is written as

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

for judgments J_i and J. The J_i are called the **hypotheses** and J the **conclusion** of the rule. Intuitively, a rule acts as a proof constructor: It returns a new proof of J if applied to proofs p_i

for each J_i. Thus, the hypotheses can be seen as input types and the conclusion as the output type of the rule. By giving the rule a name, say R, it becomes possible to refer to the proof of the conclusion as $R(p_1, \ldots, p_n)$ — this is how proofs become mathematical objects. If a rule has no hypotheses, its conclusion is always true. Such rules are called (logical) **axioms**. Additional (non-logical) axioms may be given as a theory. Clearly, all proofs must start with axioms.

A particular choice of rules is called a **calculus**, a deductive system, or an inference system. More generally, a calculus C assigns to every theory (Σ, Δ) a set of rules. There are various calculi for FOL (some of which use very different judgments), which have been shown to make the same propositions true. We will use the **natural deduction** calculus to have a grounded definition of $C(\Sigma, \Delta)$, but we refer to [Gen34] and [ML96] for the actual rules and only mention some example rules.

The rule

$$\frac{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n, F \vdash G}{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash F \Rightarrow G}$$

gives implication its meaning by saying what it means to derive an implication: If for arbitrary fixed $x_1, \ldots, x_m$ the judgment that G is a consequence of $F_1, \ldots, F_m, F$ has a proof, then the judgment that $F \Rightarrow G$ is a consequence of $F_1, \ldots, F_n$ should also have a proof. Here F_i, F, and G are arbitrary fixed Σ-formulas.

Similarly, the rule

$$\frac{\text{for all } x_1, \ldots, x_m, x : \quad F_1, \ldots, F_n \vdash F}{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash \forall x F}$$

gives the universal quantifier its meaning: If there is a proof that establishes that F is a consequence of $F_1, \ldots, F_m$ for arbitrary fixed $x_1, \ldots, x_m, x$, then there should also be a proof that established that $\forall x F$ is a consequence of $F_1, \ldots, F_n$.

These proof rules are also called **introduction rules** because the conclusion contains a symbol ($\Rightarrow$ or $\forall$) that does not occur in the hypotheses. Introduction rules are used to derive theorems. Dually, **elimination rules** are used to apply previously derived theorems. The elimination rule for implication is

$$\frac{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash F \Rightarrow G \qquad \text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash F}{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash G}$$

which is also called modus ponens. It says that $F \Rightarrow G$ and F together can be used to derive G.

The elimination rule for universal quantification is

$$\frac{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash \forall x F}{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash F[x/t]}$$

It says that $\forall x F$ can be used to derive F if an arbitrary term t is substituted for the variable x of F.

As an example for a rule dealing with equality, we give the reflexivity rule:

$$\frac{}{\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \vdash s \doteq s}$$

	Intra-Theory	Inter-Theory	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.1.5 Dependent Type Theory

Abstracting from the proof theory for FOL, we obtain the principal design features of the proof theoretical logical framework LF. (A rigid definition will be part of Sect. 4.) Being a type theory, its objects are terms s and types S related via the typing relation $s:S$.

The most important operation of LF is the function type construction: For two types S and S', the type $S \to S'$ is the type of functions from S to S'. Terms of type $S \to S'$ are functions that take an input of type S and return an output of type S'. In particular, function terms are obtained by lambda abstraction: If t is a term of type S' with a free variable x of type S, then $\lambda_{x:S}\, t$ is the function of type $S \to S'$: It returns $t[x/s]$ for every input s, the substitution of s for x in t.

LF Languages Similar to FOL, LF is a family of languages. A specific LF language is obtained by declaring type symbols and term symbols. In general, there is no need for strict definitions of what a logic is. Logics, signatures, and theories are all represented as LF-languages. We will explain this by representing our example FOL-theories in LF.

To represent the logic FOL, we declare two special type symbols ι for mathematical objects and o for formulas. Then formulas are represented by declarations of symbols that return formulas. This is straightforward for the connectives and equality:

$$\begin{array}{llll} tt:o & \wedge:o \to o \to o & \Rightarrow:o \to o \to o & \doteq:\iota \to \iota \to o \\ ff:o & \vee:o \to o \to o & \equiv:o \to o \to o & \end{array}$$

Here $\to$ is right-associative, i.e., $o \to o \to o$ is the type of functions taking two formulas as arguments and returning a formula.

The declarations for the quantifiers are more subtle:

$$\begin{array}{l} \forall:(\iota \to o) \to o \\ \exists:(\iota \to o) \to o \end{array}$$

Intuitively, a formula F in a free variable x can be seen as a function with input type ι and output type o: Given a term t as input, F returns $F[x/t]$. Thus, quantification is an operation with input type $\iota \to o$ and output type o. This method of representing quantifiers is called **higher-order abstract syntax** and is a principal feature of type theories. Its advantage is that binding, α-renaming, and substitution of FOL can conveniently be defined using their LF counterparts.

Then a specific FOL-signature is represented as follows:

- a function symbol f with arity n: a term symbol declaration $f : \underbrace{\iota \to \ldots \to \iota}_{n} \to \iota$,
- a predicate symbol p with arity n: a term symbol declaration symbol $p : \underbrace{\iota \to \ldots \to \iota}_{n} \to o$.

For example, the signature for monoids is given by the declarations

$$\begin{array}{ll} \iota:\texttt{type} & comp:\iota \to \iota \to \iota \\ o:\texttt{type} & unit:\iota \\ & invertible:\iota \to o \end{array}$$

Judgments as Types and Proofs as Terms Following the Curry-Howard correspondence ([CF58, How80]), LF represents all judgments as types. To represent a proof theory in LF, appropriate type constructors have to be declared for the desired judgments. For example, for FOL, we need a judgment for the truth of propositions. This is done by declaring a type:

$$true:o \to \texttt{type}$$

true is called a **type family**: *true* is not a type itself, only after applying *true* to a formula, it returns a type. For example, *true* tt is a type. Thus, types may depend on terms, hence we speak of **dependent types**. All proofs are represented as terms. If a proof p proves the judgment J, then LF represents this as $p : J$. For example, the terms of type *true* F are the proofs of the formula F.

With this intuition, we can already represent theories in LF: We add to the representation of the signature, constants that represent the axioms. For example, the representation of the theory *Monoid* of monoids is obtained by adding the declarations:

$$\begin{array}{l} assoc : true\big(\forall\lambda_{x_1:\iota}\,\forall\lambda_{x_3:\iota}\,\forall\lambda_{x_3:\iota}\,(comp(x_1, comp(x_2, x_3)) \doteq comp(comp(x_1, x_2), x_3))\big) \\ neutral : true\big(\forall\lambda_{x_1:\iota}\,(comp(x_1, unit) \doteq x_1 \wedge comp(unit, x_1) \doteq x_1)\big) \\ def\_invertible : true\big(\\ \quad \forall\lambda_{x_1:\iota}\,(invertible(x_1) \equiv \exists\lambda_{x_2:\iota}\,(comp(x_1, x_2) \doteq unit \wedge comp(x_2, x_1) \doteq unit)) \\ \big) \end{array}$$

where we have used infix notation for the formula constructors. An LF-language *Group* for groups is obtained similarly.

Thus, we are able to represent terms, propositions, signatures, and axioms. However, we are still missing the proof rules. Currently, the only proofs are the axioms. To represent a calculus, we must first represent every judgment as a type. For example, for FOL, we represent the judgment

$$\text{for all } x_1, \ldots, x_m : \quad F_1, \ldots, F_n \;\vdash\; F$$

as the type $true\ F_1 \to \ldots \to true\ F_n \to true\ F$

Now we can represent every proof rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

with name R as a declaration $R\colon J_1 \to \ldots \to J_n \to J$. This matches the intuition of proof rules as proof constructors: R takes proofs of $J_1, \ldots, J_n$ as input and returns a new proof of J as output. To represent a logic, we need one such declaration for every proof rule.

In fact, when representing a rule, we can even omit those assumptions that are the same in all judgments. Thus, we can represent the natural deduction rules for the implication of FOL as follows:

$$\begin{array}{l} \Rightarrow Intro : (true\ F \to true\ G) \to true(F \Rightarrow G) \\ \Rightarrow Elim : (true\ (F \Rightarrow G) \to true\ F) \to true\ G \end{array}$$

Intuitively, $\Rightarrow Intro$ says that if there is a function, say p, that returns a proof of G if provided with a proof of F as input, then there is also a proof of $F \Rightarrow G$, namely $\Rightarrow Intro(p)$.

It is not as easy to represent the rules for the universal quantifier. The hypothesis of the introduction rule says: For an arbitrary x of type ι, there is a proof of type $true\ F(x)$. This is an example of a **dependent function type**: The return type $true\ F(x)$ depends on the input value x. In LF, such a dependent function type is written as $\Pi_{x:\iota}\,(true\ F(x))$.

Then the rules are represented as

$$\begin{array}{l} \forall Intro : (\Pi_{x:\iota}\, true\ F(x)) \to true\ \forall\lambda_{x:\iota}\, F \\ \forall Elim : true\ \forall\lambda_{x:\iota}\, F \to (\Pi_{x:\iota}\, true\ F(x)) \end{array}$$

For a simpler example, consider the reflexivity rule:

$$refl\colon \Pi_{x:\iota}\, true(x \doteq x)$$

Here the type $\Pi_{x:\iota}\, true(x \doteq x)$ takes an argument x of type ι, and its return type is the type of proofs that x is equal to itself.

Truth and Consequence Finally, we can define proof theoretical truth and consequence as follows. The judgment J is **true** if there is a term of type J. And the judgment J is a **consequence** of the judgments $J_1, \ldots, J_n$ if there is a term of the type $J_1 \to \ldots \to J_n \to J$. In particular, in FOL, the formula F is true if there is a term of type *true* F.

2.2.2 The Inter-Theory Level

	Intra-Theory	**Inter-Theory**	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.2.1 Syntax

Signature Morphisms Signature and theory translations are called morphisms. Assume two FOL-signatures Σ and Σ'. We want to define the notion of a signature translation σ from Σ to Σ'. The simplest definition is the following: A signature morphism $\sigma : \Sigma \to \Sigma'$ is a mapping that maps Σ-symbols to Σ'-symbols in a way that respects the arity. For example, we have a signature morphism $\sigma^{MG} : \Sigma^M \to \Sigma^G$, which is simply the identity for all symbols.

Signatures and signature morphisms have the following properties:

- There is a class of objects, the signatures.
- For every two objects Σ and Σ', there is a class of morphisms from Σ to Σ', the signature morphisms.
- For every object Σ, there is an identity morphism $id_\Sigma : \Sigma \to \Sigma$, which is the identity on all symbols.
- For two morphisms $\sigma : \Sigma \to \Sigma'$ and $\sigma' : \Sigma' \to \Sigma''$, there is a composition morphism $\sigma' \circ \sigma : \Sigma \to \Sigma''$, which arises by composing the maps of σ and σ'.
- The identity morphism behaves like a neutral element for composition.
- The composition is associative.

Category theory ([Lan98]) is the field of mathematics studying precisely such structures. And we can summarize the above properties as: Signatures and signature morphisms form a **category**.

An intuition for categories is best obtained by thinking of an abstraction from sets. $\mathcal{SET}$ is the category where the objects are the sets, and the morphisms from A to B are the mappings from A to B. Identity and composition are obvious. Thus, category theory talks about sets and mappings as atomic objects without referring to their specific internal structure. This abstraction has proved extremely valuable because lots of other concepts, such as signatures, naturally admit a category structure.

Sentence Translation A signature morphism σ can be extended to a translation $Sen(\sigma)$ from Σ-sentences to Σ'-sentences in a straightforward way. Here we overload the symbol Sen: Firstly, it represents the mapping of signatures Σ to the set of sentences $Sen(\Sigma)$; and secondly, it represents the mapping of signature morphisms σ to sentence translations $Sen(\sigma)$.

$Sen(\sigma)$ maps from $Sen(\Sigma)$ to $Sen(\Sigma')$: It maps every Σ-sentences F to itself except that every function or predicate symbol in F is replaced with its image under σ. It is easy to see that $Sen(\sigma)(F)$ is indeed a Σ'-sentence.

Then Sen has the following properties:

- It maps signatures Σ to sets.
- If $\sigma : \Sigma \to \Sigma'$, then $Sen(\sigma)$ is a mapping from $Sen(\Sigma)$ to $Sen(\Sigma')$.
- $Sen(id_\Sigma)$ is the identity mapping of the set $Sen(\Sigma)$.
- The mapping $Sen(\sigma' \circ \sigma)$ is equal to applying first $Sen(\sigma)$ and then $Sen(\sigma')$.

In other words, *Sen* preserves the structure, and the identity and composition. In category theory, we can summarize this by saying that *Sen* is a **functor** from *Sig* to $\mathcal{SET}$.

	Intra-Theory	**Inter-Theory**	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.2.2 Model Theory

Model Reduction Consider the signature morphism σ^{MG} from the previous section. Intuitively, this morphism expresses the fact that every group is a monoid. For example, the group Int^G of the integers can also be considered as a monoid, namely Int^M. We can define the following model reduction. Assume a signature morphism $\sigma : \Sigma \to \Sigma'$ and a Σ'-model I'. Then we can define a Σ-model $I := Mod(\sigma)(I')$ as follows: $[\![s]\!]^I := [\![\sigma(s)]\!]^{I'}$ for every function or predicate symbol s of Σ. Then, our example can be made precise as $Int^M = Mod(\sigma^{MG})(Int^G)$.

Intuitively, if U is the collection of all mathematical objects, then Σ-models can be considered as mappings from Σ (which is essentially a set of symbols) to U. Since signature morphisms $\sigma : \Sigma \to \Sigma'$ are mappings from the symbols of Σ to the symbols of Σ', we can understand model reduction as composition as follows:

$$Mod(\sigma)(I') := I' \circ \sigma$$

$$\Sigma \xrightarrow{\sigma} \Sigma' \xrightarrow{I'} U$$

In general $Mod(\sigma)$ has the following properties:

- For $\sigma : \Sigma \to \Sigma'$, $Mod(\sigma)$ is a mapping from $Mod(\Sigma')$ to $Mod(\Sigma)$.
- $Mod(id_\Sigma)$ is the identity mapping of the class $Mod(\Sigma)$.
- The mapping $Mod(\sigma' \circ \sigma)$ is equal to applying first $Mod(\sigma')$ and then $Mod(\sigma)$.

Note that *Mod* is structurally very similar to *Sen*, but not quite: $Mod(\sigma)$ is a mapping in the reversed direction, from $Mod(\Sigma')$ to $Mod(\Sigma)$. In the language of category theory, this is stated as saying that *Mod* is a **contravariant functor** from *Sig* to $\mathcal{CLASS}$. The fact that $Sen(\sigma)$ and $Mod(\sigma)$ go in opposite direction is generally called the **adjunction between syntax and semantics**.

Contravariant functors are somewhat more complicated. In order to avoid them, we can define $\mathcal{CLASS}^{op}$ to be just like $\mathcal{CLASS}$ but with all morphisms flipped; then *Mod* is a normal functor from *Sig* to $\mathcal{CLASS}^{op}$.

Model Morphisms Assume the signature Σ^G and its model Int^G. Another example of a model is Rat^G, which is defined like Int^G, except that we use the rational number instead of the integers. Int^G can be seen as a submodel of Rat^G. More precisely, we have:

- There is a map from $[\![\iota]\!]^{Int^G}$ to $[\![\iota]\!]^{Rat^G}$, namely the inclusion i.
- The unit and addition of Rat^G agree with those of Int^G.
- If an element of $[\![\iota]\!]^{Int^G}$ is invertible, it is also invertible if considered as an element of $[\![\iota]\!]^{Rat^G}$.

This can be generalized to obtain the notion of model morphisms as follows. Assume a FOL-signature Σ and two Σ-models I and I'. Then a mapping $\varphi : [\![\iota]\!]^I \to [\![\iota]\!]^{I'}$ is called a model morphism from I to I' if

- for all function symbols f in Σ and all $x_1, \ldots, x_n \in [\![\iota]\!]^I$ where $n = arit(f)$:
$$\varphi\big([\![f]\!]^I(x_1, \ldots, x_n)\big) = [\![f]\!]^{I'}\big(\varphi(x_1), \ldots, \varphi(x_n)\big),$$
- for all predicate symbols p in Σ and all $x_1, \ldots, x_n \in [\![\iota]\!]^I$ where $n = arit(p)$:
$$\text{if } (x_1, \ldots, x_n) \in [\![p]\!]^I, \text{ then } \big(\varphi(x_1), \ldots, \varphi(x_n)\big) \in [\![p]\!]^{I'}.$$

Model morphisms admit an identity and a composition in the obvious way. And thus the Σ-models and their model morphisms form a category.

The model reduction $Mod(\sigma)$ can also reduce model morphisms. For example, take the above model morphism $i : Int^G \to Rat^G$. Then i is also a model morphism from $Mod(\sigma^{MG})(Int^G) = Int^M$ to $Mod(\sigma^{MG})(Rat^G) = Rat^M$. And indeed it is possible to put for arbitrary model morphisms $Mod(\sigma)(\varphi) := \varphi$. We omit this (non-trivial) step. Then $Mod(\sigma)$ becomes a functor from $Mod(\Sigma')$ to $Mod(\Sigma)$.

Now category theory shows that the categories themselves admit the structure of a category where the morphisms between two categories are given by the functors. $\mathcal{CAT}$ is the category of categories. Then Mod becomes a contravariant functor from Sig to $\mathcal{CAT}$.

Satisfaction Condition Translations are only interesting if truth is preserved along them. To see what this means, take the signature morphism σ^{MG}, the Σ^G-model Int^G, and some Σ^M-formula F. Clearly, we cannot check the satisfaction of F in Int^G because they live over different signatures. But there are two ways to use σ^{MG} to permit such a check: We can translate F to Σ^G via $Sen(\sigma^{MG})$ and check satisfaction over Σ^G; or we can reduce Int^G to Σ^M via $Mod(\sigma^{MG})$ and check satisfaction over Σ^M. A straightforward induction over F shows that both possibilities yield the same result.

In general, we can formulate this as the following condition on satisfaction: For all signature morphisms $\sigma : \Sigma \to \Sigma'$, all Σ'-models I' and all Σ-sentences F, we have

$$Mod(\sigma)(I') \models_\Sigma F \quad \text{iff} \quad I' \models_{\Sigma'} Sen(\sigma)(F)$$

The involved expressions are visualized in the following diagram, where the nodes on the right side are elements of the respective set or category on the left side.

$$\begin{array}{ccc} Sen(\Sigma) & \xrightarrow{Sen(\sigma)} & Sen(\Sigma') \\ \models_\Sigma \big| & & \big| \models_{\Sigma'} \\ Mod(\Sigma) & \xleftarrow{Mod(\sigma)} & Mod(\Sigma') \end{array} \qquad \begin{array}{ccc} F & \longmapsto & Sen(\sigma)(F) \\ \big| & & \big| \\ Mod(\sigma)(I') & \longleftarrow\!\shortmid & I' \end{array}$$

Theory Morphisms Just like we can translate between signatures, we can also translate between theories. Model-theoretically, a signature morphism $\sigma : \Sigma \to \Sigma'$ is a **theory morphism** from (Σ, Δ) to (Σ', Δ') if the following holds: If $F \in \Delta$, then $Sen(\sigma)(F)$ is a theorem of (Σ', Δ').

This means that σ maps every axiom and therefore every theorem of (Σ, Δ) to a theorem of (Σ', Δ'). Thus, theory morphisms are theorem-preserving translations between theories. For example, the observation that every theorem F over monoids is also a theorem over groups is made precise by saying that σ^{MG} is a theory morphism from *Monoid* to *Group*.

	Intra-Theory	**Inter-Theory**	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.2.3 Institutions

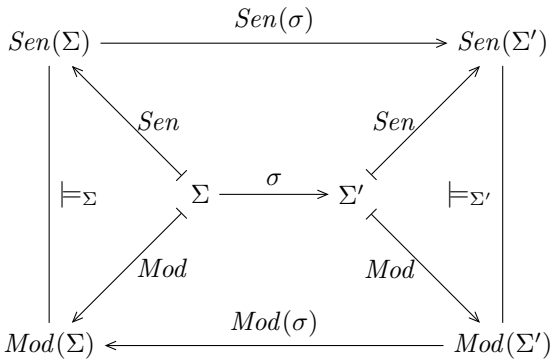
Again, we can abstract from the above notions and reach the final definition of an institution. An *institution* is a tuple $(Sig, Sen, Mod, \models)$ such that

- Sig is a category, the class of signatures and signature morphisms,
- $Sen : Sig \to \mathcal{SET}$ is a functor assigning to every signature its sentences and to every signature morphism its sentence translation,
- $Mod : Sig \to \mathcal{CAT}^{op}$ is a functor assigning to every signature its model category and to every signature morphism its model reduction functor,
- $\models$ is a family of relations $\models_\Sigma$ for every signature Σ such that $\models_\Sigma$ is a relation between the Σ-models in $Mod(\Sigma)$ and the Σ-sentences in $Sen(\Sigma)$,

such that the satisfaction condition holds: For all signature morphisms $\sigma : \Sigma \to \Sigma'$, all Σ'-models I' and all Σ-sentences F, we have

$$Mod(\sigma)(I') \models_\Sigma F \quad \text{iff} \quad I' \models_{\Sigma'} Sen(\sigma)(F).$$

The involved expressions can be visualized as follows



Then we can summarize the properties of the syntax and the model theory of FOL by saying that FOL forms an institution $\mathcal{FOL} = (Sig^{\mathcal{FOL}}, Sen^{\mathcal{FOL}}, Mod^{\mathcal{FOL}}, \models^{\mathcal{FOL}})$.

Adjunction between Syntax and Semantics For an arbitrary institution, the theories and the theory morphisms form a category as well. Just like in Sect. 2.2.1.3, we can obtain the institution of theories. Assume an institution $\mathbb{I} = (Sig, Sen, Mod, \models)$. Then we obtain an institution $\mathbb{I}^{Th} = (Th, Sen^{Th}, Mod^{Th}, \models^{Th})$ by putting

- Th is the category of $\mathbb{I}$-theories,
- Sen^{Th} acts like Sen but after dropping the set of axioms from a theory,
- $Mod^{Th}(\Sigma, \Delta)$ is the class of (Σ, Δ)-models of $\mathbb{I}$, and $Mod^{Th}(\sigma) = Mod(\sigma)$,
- $\models^{Th}$ is the appropriate restriction of $\models$.

The fact that $\mathbb{I}^{Th}$ is an institution is not totally trivial. To show well-definedness, it must be shown that for every theory morphism $\sigma : (\Sigma, \Delta) \to (\Sigma', \Delta')$ and every $I' \in Mod^{Th}(\Sigma', \Delta')$, we have $Mod^{Th}(\sigma)(I') \in Mod^{Th}(\Sigma, \Delta)$.

This follows from the properties of theory morphisms and the satisfaction condition of $\mathbb{I}$. Intuitively, it means the following: Theory morphisms from (Σ, Δ) to (Σ', Δ') translate (Σ, Δ)-theorems to (Σ', Δ')-theorems and reduce (Σ', Δ')-models to (Σ, Δ)-models. This is a manifestation of one of the guiding principles of institutions, namely that of an adjointness between syntax (theories) and semantics (models).

	Intra-Theory	**Inter-Theory**	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.2.4 Proof Theory

Derived Proof Rules If there is a proof, i.e., a finite sequence of rule applications, leading from the judgments $J_1, \ldots, J_n$ to the judgment J, this proof is called a derived proof rule. The name is justified because adding the proof rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

is always possible without changing the derivable judgments.

For example, for FOL, the rule

$$\frac{\text{for all } x_1, \ldots, x_m : \ F_1, \ldots, F_n, F, F' \vdash G}{\text{for all } x_1, \ldots, x_m : \ F_1, \ldots, F_n \vdash F \Rightarrow (F' \Rightarrow G)}$$

is a derivable rule of $C(\Sigma, \Delta)$: It is derived by composing two applications of the implication introduction rule.

Theory Morphisms Similarly, to the extension of σ to $Sen(\sigma)$, a FOL-signature morphism $\sigma : \Sigma \to \Sigma'$ extends to a translation from Σ-judgments to Σ'-judgments: For a judgment

$$J \quad = \quad \text{for all } x_1, \ldots, x_m : \ F_1, \ldots, F_n \vdash F,$$

put

$$C(\sigma)(J) \quad := \quad \text{for all } x_1, \ldots, x_m : \ C(\sigma)(F_1), \ldots, C(\sigma)(F_n) \vdash C(\sigma)(F)$$

where the application of $C(\sigma)$ to formulas is defined as for $Sen(\sigma)$

Then proof-theoretically, a signature morphism $\sigma : \Sigma \to \Sigma'$ is called a **theory morphism** from (Σ, Δ) to (Σ', Δ') if the following holds: For all proof rules

$$\frac{J_1 \quad \dots \quad J_n}{J}$$

in $C(\Sigma, \Delta)$,

$$\frac{C(\sigma)(J_1) \quad \dots \quad C(\sigma)(J_n)}{C(\sigma)(J)}$$

is a derived proof rule of $C(\Sigma', \Delta')$.

In particular, this means that σ maps every axiom of (Σ, Δ) to a theorem of (Σ', Δ'). And that implies that σ maps every theorem of (Σ, Δ) to a consequence of (Σ', Δ'). Thus, theory morphisms are theorem-preserving translations between theories. For example, the observation that every theorem of F over monoids is also a theorem of $Sen(\sigma)(F)$ over groups is made precise by saying that σ^{MG} is a theory morphism from *Monoid* to *Group*. This definition is remarkably similar to the model theoretical definition.

	Intra-Theory	**Inter-Theory**	Inter-Logic
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.2.5 in Dependent Type Theory

LF Translations Using LF, we are able to state the definition of proof theoretical theory morphisms in a more precise way, namely by using translations between LF-languages. If two LF-languages Σ and Σ' are given, an LF-translation from Σ to Σ' maps every symbol of Σ to a term of Σ'.

LF-translations may not map symbols arbitrarily. Rather they must preserve types in the following sense. For every Σ-term symbol c of type S, $\sigma(c)$ must be a Σ'-term of type $\overline{\sigma}(S)$. Here $\overline{\sigma}(S)$ is the extension of σ to S: $\overline{\sigma}$ replaces every Σ-symbol c in S with $\sigma(c)$. Similarly, every Σ-type symbol must be mapped to a Σ'-type.

Signature Morphisms For example, for a translation between two LF languages representing FOL theories, the special type symbols ι and o are mapped to themselves. Similarly, the connectives, the equality symbol, and the quantifiers are mapped to themselves. A function symbol $f : \iota \to \dots \to \iota \to \iota$ must be mapped to any term of type

$$\overline{\sigma}(\iota \to \dots \to \iota \to \iota) = \iota \to \dots \to \iota \to \iota,$$

and similarly for predicate symbols. Then the signature morphism σ^{MG} from *Monoid* to *Group* can be expressed in LF as the LF translation that maps *comp*, *unit*, and *invertible* to themselves.

However, in LF, we can do something more general. Consider the symbol *invertible* of the signature M for monoids. Usually this symbol is not part of the theory of monoids, and we might wish to eliminate it, thus obtaining the signature M'. Then there is an obvious signature inclusion morphism from M' to M. However, it is intuitively clear that there is also a translation in the opposite direction. This translation can also be formulated as an LF translation $\sigma^{MM'}$. The type of *invertible* is $\iota \to o$, thus $\sigma^{MM'}(invertible)$ must have type $\overline{\sigma}(\iota \to o) = \iota \to o$. And we can put

$$\sigma^{MM'}(invertible) := \lambda_{x_1:\iota} F_{inv}(x_1)$$

where $F_{inv}(x_1)$ abbreviates the predicate that states that x_1 has an inverse, i.e.,

$$F_{inv}(x_1) := \exists \lambda_{x_2:\iota}\, (comp(x_1, x_2) \doteq unit \wedge comp(x_2, x_1) \doteq unit).$$

Theory Morphisms Just like the notions of signature and theory are unified as LF languages, the notions of signature and theory morphism are unified as LF translations. Via the Curry-Howard correspondence, just like proof rules

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

are represented as declarations of constants of type $J_1 \to \ldots \to J_n \to J$, the corresponding derived proof rules are represented as composed terms of that type.

Since proof theoretical theory morphisms translate proof rules to derived proof rules, they can be represented directly as LF translations. Such LF translations map constants (i.e., rules) to terms (i.e., derived rules) of the appropriate types.

For example, for FOL, the truth judgment *true* and all rules are translated to themselves. And in order for a specific signature morphism such as $\sigma^{MM'}$ to become a theory morphism, all axioms (i.e., rules without hypotheses) of σ^{M} must be translated to theorems (i.e., derived rules without hypotheses) of $\sigma^{M'}$. This is trivial for all axioms except for *def_invertible*. It must be mapped to a proof of type

$$\overline{\sigma}\left(true\big(\forall \lambda_{x_1:\iota}\, invertible(x_1) \equiv F_{inv}(x_1)\big)\right) \;=\; true\big(\forall \lambda_{x_1:\iota}\, F_{inv}(x_1) \equiv F_{inv}(x_1)\big)$$

where F_{inv} is as above. And such a map is possible because in FOL there is a derived proof rule that proves that every formula is equivalent to itself.

2.2.2.6 Modal Logic

In order to give an example of a logic translation in the next section, we will sketch the definition of a fragment of propositional modal logic (ML). This can also serve as an exercise for the reader because it defines a new logic using the two logical frameworks developed in the preceding sections.

Syntax: Signatures and Sentences The signature category $Sig^{\mathcal{ML}}$ is $\mathcal{SET}$. For a signature Σ, i.e., a set, the elements $p \in \Sigma$ serve as the propositional constants. The Σ-sentences are built up using the unary connectives $\neg$ and $\Box$, and the binary connective $\Rightarrow$ as in the following LF language:

$$\begin{array}{l} o\text{:}\mathtt{type} \\ \neg\text{:}o \to o \\ \Rightarrow\text{:}o \to o \to o \\ \Box\text{:}o \to o \\ p\text{:}o \qquad \text{for all } p \in \Sigma \end{array}$$

A signature morphism, i.e., a mapping $\sigma : \Sigma \to \Sigma'$ induces an LF translation σ' by mapping every $p \in \Sigma$ to $\sigma(p)$ and the remaining symbols to themselves. The sentence translation $Sen^{\mathcal{ML}}(\sigma)$ is the restriction of $\overline{\sigma'}$ to the terms of type o.

For example, assume an ML-signature morphism $\sigma_e : \{p, q\} \to \{a, b\}$ mapping both p and q to a. Then $Sen^{\mathcal{ML}}(\sigma_e)$ maps the formula $\Box(p \Rightarrow q)$ to $\Box(a \Rightarrow a)$.

Model Theory: Models and Satisfaction Σ-models are Kripke-models ([Kri63]), i.e., triples (W, acc, I) where

- W is a non-empty set,

- acc is a binary relation on W,
- $I \subseteq \Sigma \times W$ is a world-wise interpretation of the propositional variables.

Intuitively, W is a set of worlds and acc is an accessibility relation between worlds, i.e., $acc(w, w')$ can be understood as a wormhole from w to w'. If acc is an ordering, the elements of W can also be seen as points in time with $acc(w, w')$ expressing that w' is in a future of w. All formulas are interpreted in all worlds, but the interpretations in the worlds are not independent. In particular, I gives the interpretation of the propositional constants: $(p, w) \in I$ expresses that p holds in world w.

For simplicity, we omit model morphisms, i.e., the identity morphisms are the only model morphisms. Model reduction is defined by

$$Mod^{\mathcal{ML}}(\sigma)(W', acc', I') = (W', acc', \{(p, w) \mid (\sigma(p), w) \in I'\}).$$

For example, $K'_e := (W', acc', I')$ where $W' = \{w_1, w_2, w_3\}$, $acc' = \{(w_1, w_2), (w_1, w_3)\}$, and $I' = \{(a, w_1), (b, w_2)\}$ is a Σ'-model. Via σ_e, it is reduced to the Σ-model $K_e := (W, acc, I)$ with $W := W'$, $acc := acc'$, and $I = \{(p, w_1), (q, w_1)\}$.

World-dependent satisfaction is defined by

- $W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} p$ iff $(p, w) \in I$ for $p \in \Sigma$,
- $W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} \neg F$ iff $W, acc, I, w \not\models_{\Sigma} F$,
- $W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} F \Rightarrow G$ iff $W, acc, I, w \not\models_{\Sigma} F$ or $W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} G$,
- $W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} \Box F$ iff $W, acc, I, w' \models^{\mathcal{ML}}_{\Sigma} F$ for all w' such that $(w, w') \in acc$,

Thus, $\Box$ quantifies over all worlds accessible from the current world, or over the future of the current point. The satisfaction in a model is defined by

$$W, acc, I \models^{\mathcal{ML}}_{\Sigma} F \quad \text{iff} \quad W, acc, I, w \models^{\mathcal{ML}}_{\Sigma} F \text{ for all } w \in W.$$

For example, with K_e from above, we have

$$K_e, w_1 \models^{\mathcal{ML}}_{\{p,q\}} \Box(p \Rightarrow q)$$

because $p \Rightarrow q$ holds both in w_2 and in w_3.

We show the satisfaction condition by proving by induction on F that for all $w \in W'$

$$Mod^{\mathcal{ML}}(\sigma)(W', acc', I', w) \models^{\mathcal{ML}}_{\Sigma} F \quad \text{iff} \quad W', acc', I', w \models^{\mathcal{ML}}_{\Sigma} Sen^{\mathcal{ML}}(\sigma)(F).$$

The only non-trivial case is that of an atomic formula. So assume $\sigma : \Sigma \to \Sigma'$, $F = p \in \Sigma$, and $(W', acc', I') \in Mod^{\mathcal{ML}}(\Sigma')$. Firstly, we translate F along σ and evaluate over Σ': We have $Sen^{\mathcal{ML}}(\sigma)(F) = \sigma(p)$, and the satisfaction holds in w iff $(\sigma(p), w) \in I'$. Secondly, we reduce (W', acc', I') to (W, acc, I) and evaluate over Σ: p holds in world w of the reduced model iff $(p, w) \in I$ where by definition $(p, w) \in I$ iff $(\sigma(p), w) \in I'$. Thus, the two conditions are equivalent for atomic F.

Therefore, the syntax and model theory form an institution

$$\mathcal{ML} := (Sig^{\mathcal{ML}}, Mod^{\mathcal{ML}}, Mod^{\mathcal{ML}}, \models^{\mathcal{ML}}).$$

Proof Theory: Judgments and Proof Rules The proof theory for a signature Σ is given by extending the LF language from above with the following declarations, thus obtaining the signature $\mathcal{C}^{\mathcal{ML}}(\Sigma)$

$$
\begin{array}{l}
true : o \to \texttt{type} \\
A : true\big(F \Rightarrow (\neg F \Rightarrow G)\big) \\
B : true\big((F \Rightarrow G) \Rightarrow ((G \Rightarrow H) \Rightarrow (F \Rightarrow H))\big) \\
C : true\big((\neg F \Rightarrow F) \Rightarrow F\big) \\
MP : true(F \Rightarrow G) \to true\ F \to true\ G \\
Nec : true(F) \to true(\Box F) \\
K : true\big(\Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)\big)
\end{array}
$$

Then the proofs of F are the terms of type $true(F)$. This is the standard axiomatization of the normal modal logic K ([HC96]) based on axioms for propositional logic that are due to Łukasiewicz.

Then every signature morphism $\sigma : \Sigma \to \Sigma'$ extends to an LF translation $\mathcal{C}^{\mathcal{ML}}(\sigma)$ from $\mathcal{C}^{\mathcal{ML}}(\Sigma)$ to $\mathcal{C}^{\mathcal{ML}}(\Sigma')$, by mapping the symbols *true* through K to themselves.

The logical framework of dependent type theory has no analogue to an institution that collects the syntax and the proof theory into a single object. In analogy to institutions, this object could be the tuple $(Sig^{\mathcal{ML}}, Mod^{\mathcal{ML}}, \mathcal{C}^{\mathcal{ML}}, true)$ where $\mathcal{C}^{\mathcal{ML}}$ assigns proof theories to signatures and *true* determines proof theoretical truth. Our logical framework will do something similar.

2.2.3 The Inter-Logic Level

	Intra-Theory	Inter-Theory	**Inter-Logic**
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.3.1 Syntax

Signature Translation Assume an ML-signature Σ. We define a FOL-signature $\Phi(\Sigma)$, which contains:

- a unary predicate symbol $@_p$ for every $p \in \Sigma$,
- a binary predicate symbol Acc.

For example, for the ML-signature $\{p, q\}$, $\Phi(\{p, q\})$ contains unary predicate symbols $@_p$ and $@_q$, and a binary predicate symbol Acc. The intuition is that the universe $[\![\iota]\!]^{J'}$ of $\Phi(\Sigma)$-model J' corresponds to the set W of worlds. Then Acc corresponds to acc, and the truth of $@_p(w)$ corresponds to $(p, w) \in I$.

Φ can be extended to signature morphisms easily. If $\sigma : \Sigma \to \Sigma'$ is an ML-signature morphism, we define a FOL-signature morphism $\Phi(\sigma) : \Phi(\Sigma) \to \Phi(\Sigma')$ as follows: $@_p$ of $\Phi(\Sigma)$ is mapped to $@_{\sigma(p)}$ of $\Phi(\Sigma')$, and Acc is mapped to Acc.

Φ has the structure-preserving properties of a functor, and thus we can summarize: Φ is a functor from $Sig^{\mathcal{ML}}$ to $Sig^{\mathcal{FOL}}$.

Sentence Translation The idea behind $\Phi(\Sigma)$ is that it has enough structure to encode Kripke models. Thus, the model theory of ML can be expressed using the syntax of FOL. This is formalized by giving a family of mappings $\alpha_\Sigma : Sen^{\mathcal{ML}}(\Sigma) \to Sen^{\mathcal{FOL}}(\Phi(\Sigma))$ for every $\Sigma \in$

$Sig^{\mathcal{ML}}$. The translation must be done in two steps. The first step is a compositional translation $\overline{-}^w$ for a variable w:

- $\overline{p}^w := @_p(w)$,
- $\overline{\neg F}^w := \neg \overline{F}^w$,
- $\overline{F \Rightarrow G}^w := \overline{F}^w \Rightarrow \overline{G}^w$,
- $\overline{\Box F}^w := \forall w'(Acc(w,w') \Rightarrow \overline{F}^{w'})$.

Then it is intuitively clear that if the FOL-universe is the set of worlds, $\overline{F}^w$ expresses the truth of F in world w. The second step is only applied once at the toplevel: We put

$$\alpha_\Sigma(F) := \forall w \overline{F}^w.$$

For example, we have

$$\alpha_{\{p,q\}}(\Box(p \Rightarrow q)) \;=\; \forall w\big(\forall w'\big(Acc(w,w') \Rightarrow (@_p(w') \Rightarrow @_q(w'))\big)\big).$$

α has a structure-preserving property that is not so easy to pin down. Take the example ML-signature morphism $\sigma_e : \Sigma_0 \to \Sigma_1$ from above where $\Sigma_0 := \{p,q\}$ and $\Sigma_1 := \{a,b\}$. Assume we want to translate a sentence from Σ_0 to $\Phi(\Sigma_1)$, i.e., we compose the sentence translation along σ_e and the sentence translation along α. Then there are two ways to do this as in the following diagram:

$$\begin{array}{ccc} Sen^{ML}(\Sigma_0) & \xrightarrow{Sen^{ML}(\sigma_e)} & Sen^{ML}(\Sigma_1) \\ \downarrow{\scriptstyle \alpha_{\Sigma_0}} & & \downarrow{\scriptstyle \alpha_{\Sigma_1}} \\ Sen^{\mathcal{FOL}}(\Phi(\Sigma_0)) & \xrightarrow{Sen^{\mathcal{FOL}}(\Phi(\sigma_e))} & Sen^{\mathcal{FOL}}(\Phi(\Sigma_1)) \end{array} \qquad \sigma_e : \begin{cases} p \mapsto a \\ q \mapsto a \end{cases}$$

$$\Phi(\sigma_e) : \begin{cases} @_p \mapsto @_a \\ @_q \mapsto @_a \\ Acc \mapsto Acc \end{cases}$$

And this diagram commutes. For example, for $\Box(p \wedge q)$, the translations yield

$$\begin{array}{ccc} \Box(p \wedge q) & \xmapsto{Sen^{ML}(\sigma_e)} & \Box(a \wedge a) \\ \downarrow{\scriptstyle \alpha_\Sigma} & & \downarrow{\scriptstyle \alpha_{\Sigma'}} \\ \forall w\big(\forall w'\big(Acc(w,w') \Rightarrow (@_p(w') \Rightarrow @_q(w'))\big)\big) & \xmapsto{Sen^{\mathcal{FOL}}(\Phi(\sigma_e))} & \forall w\big(\forall w'\big(Acc(w,w') \Rightarrow (@_a(w') \Rightarrow @_a(w'))\big)\big) \end{array}$$

In general, we have that for all ML-signature morphism $\sigma : \Sigma \to \Sigma'$:

$$Sen^{\mathcal{FOL}}(\Phi(\Sigma)) \circ \alpha_\Sigma = \alpha_{\Sigma'} \circ Sen^{\mathcal{ML}}(\sigma).$$

In the language of category theory, this means that α is a **natural transformation** from the functor $Sen^{\mathcal{ML}}$ to the functor $Sen^{\mathcal{FOL}} \circ \Phi$. These functors are as in the following diagram:

$$\begin{array}{ccc} & & \mathcal{SET} \\ & \overset{Sen^{ML}}{\nearrow} & \uparrow{\scriptstyle Sen^{\mathcal{FOL}}} \\ Sig^{ML} & \xrightarrow{\Phi} & Sig^{\mathcal{FOL}} \end{array}$$

This diagram does not commute, i.e., $Sen^{\mathcal{ML}}(\Sigma) \neq Sen^{\mathcal{FOL}}(\Phi(\Sigma))$. But α_Σ mediates between these two sets.

	Intra-Theory	Inter-Theory	**Inter-Logic**
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

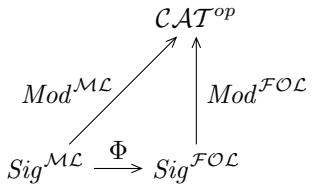
2.2.3.2 Model Theory

Model Reduction The model reduction formalizes the intuition how the FOL-models correspond to the Kripke-models. Assume a $\Phi(\Sigma)$-model J'. We define a Kripke-model $\beta_\Sigma(J') = (W, acc, I)$ as follows:

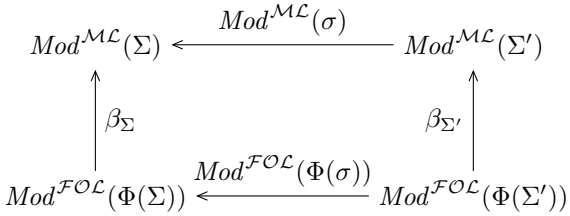
- $W = [\![\iota]\!]^{J'}$,
- $acc = [\![Acc]\!]^{J'}$,
- $I = \{(p, w) \in \Sigma \times W \mid w \in [\![@_p]\!]^{J'}\}$.

For example, assume $\Sigma = \{p, q\}$ and a $\Phi(\Sigma)$-model J' with $[\![\iota]\!]^{J'} = \{w_1, w_2, w_3\}$, $[\![Acc]\!]^{J'} = \{(w_1, w_2), (w_1, w_3)\}$, $[\![@_p]\!]^{J'} = \{w_1\}$, and $[\![@_q]\!]^{J'} = \{w_1\}$. Then $\beta_\Sigma(J') = M_e$ from Sect. 2.2.2.6.

Dually to the naturality of α, β is a natural transformation from $Mod^{\mathcal{ML}}$ to $Mod^{\mathcal{FOL}} \circ \Phi$ as in



That means that for all ML-signature morphisms $\sigma : \Sigma \to \Sigma'$, the following diagram (which is a diagram in $\mathcal{CAT}$) commutes:



Note that the naturality condition should technically be illustrated by a diagram in $\mathcal{CAT}^{op}$ with the same nodes, but all arrows flipped. We choose the illustration in $\mathcal{CAT}$ so that all arrows point in the intuitive direction, i.e., models are reduced from the lower right corner to the upper left corner.

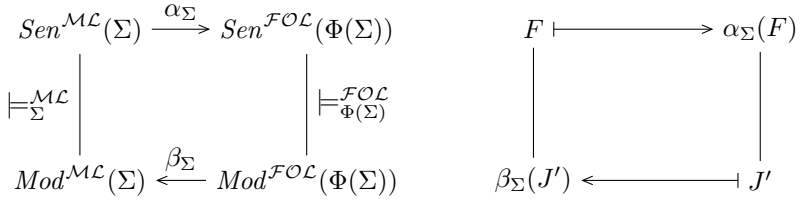
β has another important property: Every β_Σ is surjective on objects, i.e., every Kripke-model of Σ is in the image of β_Σ. This can be seen easily. For example, above we picked J' such that $\beta_\Sigma(J') = M_e$.

Satisfaction Condition Just like inter-theory translations, inter-logic translations should preserve truth. It should not matter whether an ML-sentence F over Σ is translated and its truth checked over $\Phi(\Sigma)$ in an FOL-model J', or whether the truth of F is checked over Σ in the reduction of J'.

Similarly to the inter-theory level, we obtain: For all ML-signatures Σ, all Σ-sentences F, and all $\Phi(\Sigma)$-models J', we have

$$\beta_\Sigma(J') \models^{\mathcal{ML}}_\Sigma F \quad \text{iff} \quad J' \models^{\mathcal{FOL}}_{\Phi(\Sigma)} \alpha_\Sigma(F)$$

as in



	Intra-Theory	Inter-Theory	**Inter-Logic**
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.3.3 Institutions

Institution Comorphisms Abstracting from the translations of syntax and model theory, we obtain the definition of translations between institutions. For historical reasons, these are called **institution comorphisms**. An institution comorphism from $\mathbb{I} = (\mathit{Sig}, \mathit{Sen}, \mathit{Mod}, \models)$ to $\mathbb{I}' = (\mathit{Sig}', \mathit{Sen}', \mathit{Mod}', \models')$ is a triple (Φ, α, β) such that

- $\Phi : \mathit{Sig} \to \mathit{Sig}'$ is a functor,
- $\alpha : \mathit{Sen} \to \mathit{Sen}' \circ \Phi$ is a natural transformation,
- $\beta : \mathit{Mod} \to \mathit{Mod}' \circ \Phi$ is a natural transformation,
- for all $\mathbb{I}$-signatures Σ, all Σ-sentences F, and all $\Phi(\Sigma)$-models I':

$$\beta_\Sigma(I') \models^{\mathbb{I}}_\Sigma F \quad \text{iff} \quad I' \models^{\mathbb{I}'}_{\Phi(\Sigma)} \alpha_\Sigma(F).$$

An institution comorphism is said to have the **model expansion** property if every β_Σ is surjective on objects. For comorphisms with model expansion, we can prove the following theorem that captures the intuition of a truth-preserving inter-logic translation. For all $\mathbb{I}$ theories (Σ, Δ) and all Σ-sentences F:

$$\Delta \models^{\mathbb{I}}_\Sigma F \quad \text{iff} \quad \alpha_\Sigma(\Delta) \models^{\mathbb{I}'}_{\Phi(\Sigma)} \alpha_\Sigma(F)$$

where $\alpha_\Sigma(\Delta) = \{\alpha_\Sigma(G) \mid G \in \Delta\}$. This is called the **borrowing** theorem ([CM97]) because it means that the institution $\mathbb{I}$ can borrow the truth definition (and thus a proof system) of the institution $\mathbb{I}'$.

	Intra-Theory	Inter-Theory	**Inter-Logic**
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.3.4 Proof Theory

The proof translation from ML to FOL consists of a family τ_Σ of maps for every ML-signature Σ. τ_Σ maps every judgment and every proof in $\mathcal{C}^{\mathcal{ML}}(\Sigma)$ to a judgment or proof in $C^{\mathcal{FOL}}(\Phi(\Sigma)$. Proof translations are tricky because they can be syntactically complex. Writing them down without the support of a proof theoretical logical framework is so difficult that it is hardly worth doing at all. This is where LF shows its full strength as a logical framework. Therefore, we will give the maps τ_Σ as LF translations right away in Sect. 2.2.3.5.

	Intra-Theory	Inter-Theory	**Inter-Logic**
Syntax Example	2.2.1.1	2.2.2.1	2.2.3.1
Model Theory Example	2.2.1.2	2.2.2.2	2.2.3.2
Institutions	2.2.1.3	2.2.2.3	2.2.3.3
Proof Theory Example	2.2.1.4	2.2.2.4	2.2.3.4
Dependent Type Theory	2.2.1.5	2.2.2.5	2.2.3.5

2.2.3.5 Dependent Type Theory

Because in LF, both logics and theories are LF languages, no strict conceptual difference between logic translations and theory translations is needed. Assume that Φ translates every LF language representing an $\mathbb{I}$-signature to an LF language representing an $\mathbb{I}'$-signature. Then a proof translation between the calculi $C^{\mathbb{I}}$ and $C^{\mathbb{I}'}$ is a family τ_Σ of LF translations from $C^{\mathbb{I}}(\Sigma)$ to $C^{\mathbb{I}'}(\Phi(\Sigma))$.

Sentence Translation Since proofs contain formulas, we can only use LF to define a proof translation if we also represent the sentence translation in LF. For example, for an ML-signature Σ, the translation of the formulas can be represented in LF as follows:

$$\begin{aligned}
&\tau_\Sigma(o) := \iota \to o\\
&\tau_\Sigma(\neg) := \lambda_{f:\iota\to o}\, \lambda_{w:\iota}\, \neg f(w)\\
&\tau_\Sigma(\Rightarrow) := \lambda_{f:\iota\to o}\, \lambda_{g:\iota\to o}\, \lambda_{w:\iota}\, (f(w) \Rightarrow g(w))\\
&\tau_\Sigma(\Box) := \lambda_{f:\iota\to o}\, \lambda_{w:\iota}\, \forall \lambda_{w':\iota}\, (Acc(w,w') \Rightarrow f(w'))\\
&\tau_\Sigma(p) := \lambda_{w:\iota}\, @_p(w) \qquad \text{for } p \in \Sigma
\end{aligned}$$

$$\tau_\Sigma(true) := \lambda_{f:\iota\to o}\, true(\forall \lambda_{w:\iota}\, f(w))$$

Because $\tau_\Sigma(o) = \iota \to o$, the symbols $\neg$, $\Rightarrow$, and $\Box$, which take arguments of type o must be translated to terms taking arguments of type $\iota \to o$. It is easy to see the translation is just such that every Σ-proposition F is translated to $\lambda_{w:\iota}\, \overline{F}^w$ where $\overline{F}^w$ is the syntax translation defined in Sect. 2.2.3.1.

The most interesting part is the translation of the judgment *true*. Since its type in $\mathcal{C}^{\mathcal{ML}}(\Sigma)$ is $o \to \texttt{type}$, its translation $\tau_\Sigma(true)$ must have type $(\iota \to o) \to \texttt{type}$. Inspecting the value of $\tau_\Sigma(true)$ shows that it represents exactly the toplevel step used in Sect. 2.2.3.1.

Proof Translation Finally, the translation of the Σ-proofs to $\Phi(\Sigma)$-proofs is handled by extending τ_Σ to an LF translation from $\mathcal{C}^{\mathcal{ML}}(\Sigma)$ to $C^{\mathcal{FOL}}(\Phi(\Sigma))$ by adding translations for the symbols A through K from Sect. 2.2.2.6. As on the inter-theory level, the proof rules must be mapped to derived proof rules. This is tedious but straightforward, and we just give the only trivial case as an example:

$$\tau_\Sigma(o) \ := \ \Rightarrow Elim.$$

2.3 Institutions, top-down

In this section, we introduce the main definitions regarding institutions in a formally precise way for future reference and to make this text self-contained. This cannot replace an introductory course on the subjects and is only intended to ground the following on exact foundations. In particular, the section on category theory will probably only be readable to readers who already know the content. We have conveyed the main intuitions behind these concepts as far as they are relevant for this text in Sect. 2.2.

For more elaborate introductions, the standard reference for category theory is [Lan98], and we also recommend [Awo06] as an introductory textbook. The original paper introducing institutions is [GB92], and it is well-readable as an introductory text. The current state of the art in institutions is collected in the upcoming book [Dia08].

We do not give a similar development for dependent type theory because we will introduce our own variant in Sect. 4 anyway. The best references for the LF variant of dependent type theory and its use as a logical framework are [HHP93], which introduces LF, and the handbook article on proof-theoretical logical frameworks [Pfe01].

2.3.1 Category Theory

Definition 2.4 (Foundation)**.** As the **foundation**, we will use Mac Lane-Feferman set theory ([Lan69, Fef69]). In particular, there are three syntactic levels: sets, classes, and conglomerates. Every set is a class, and every class is a conglomerate. Classes are collections of sets, and conglomerates are collections of classes. For every property about sets, there is the class of all sets with that property; and for every property about classes, there is the conglomerate of all classes with that property. The axiom of choice is present for all three levels.

Definition 2.5 (Categories)**.** A **category** $\mathcal{C}$ is a tuple of

- a class $|\mathcal{C}|$ of objects,
- a class $Hom_{\mathcal{C}}(A, B)$ of morphisms from A to B for every two objects A, B,
- an identity morphism $id_A \in Hom_{\mathcal{C}}(A, A)$ for every object A,
- a composition operation $\circ : Hom_{\mathcal{C}}(B, C) \times Hom_{\mathcal{C}}(A, B) \to Hom_{\mathcal{C}}(A, C)$ for every three objects A, B, C, which is written in infix notation,

such that

- $id_B \circ f = f$ and $f \circ id_A = f$ for every morphism $f \in Hom_{\mathcal{C}}(A, B)$ (i.e., id_A is a neutral element for composition),
- $h \circ (g \circ f) = (h \circ g) \circ f$ for every three morphisms f, g, h for which the compositions are defined (i.e., when defined, composition is associative).

Notation 2.6 (Morphisms). If f is a morphism from A to B in a category that is clear from the context, we write $f : A \to B$. We also write $A \to B$ if f is clear from the context or to refer to morphisms from A to B in general.

Example 2.7 (Sets). The category $\mathcal{SET}$ has as objects the sets, as morphisms from A to B the mappings from A to B, and as identity and composition the corresponding operations for mappings.

Example 2.8 (Classes). The category $\mathcal{CLASS}$ has as objects the classes, as morphisms from A to B the mappings from A to B, and as identity and composition the corresponding operations for mappings. Technically, $\mathcal{CLASS}$ is only a quasi-category because $|\mathcal{CLASS}|$ is a proper conglomerate, i.e., not a class. We will drop this detail from our notation.

Definition 2.9 (Opposite Category)**.** For a category $\mathcal{C}$, the category $\mathcal{C}^{op}$ has the same objects and identities as $\mathcal{C}$, but all morphisms and morphism composition are reversed. Precisely,

$$Hom_{\mathcal{C}^{op}}(A, B) := Hom_{\mathcal{C}}(B, A)$$

and

$$g \circ^{\mathcal{C}^{op}} f := f \circ^{\mathcal{C}} g.$$

Definition 2.10 (Functors)**.** For two categories $\mathcal{C}$ and $\mathcal{C}'$, a **functor** F from $\mathcal{C}$ to $\mathcal{C}'$ is a family of mappings, all of which are denoted by F, such that

- $F : |\mathcal{C}| \to |\mathcal{C}'|$,
- $F : Hom_{\mathcal{C}}(A, B) \to Hom_{\mathcal{C}}(F(A), F(B))$ for every two objects $A, B \in |\mathcal{C}|$ (i.e., functors map all structure of $\mathcal{C}$ to structure of $\mathcal{C}'$),
- $F(id_A) = id_{F(A)}$ for every object $A \in |\mathcal{C}|$ (i.e., functors preserve identities),
- $F(g \circ f) = F(g) \circ F(f)$ for every two $\mathcal{C}$-morphisms f, g for which composition is defined (i.e., functors preserve composition).

Example 2.11 (Sets are Classes). The inclusion, which maps all sets and all mappings to themselves is a functor from $\mathcal{SET}$ to $\mathcal{CLASS}$.

Example 2.12 (Identity and Composition of Functors). For every category $\mathcal{C}$, the identity, which maps all objects and morphisms to themselves, is a functor from $\mathcal{C}$ to $\mathcal{C}$.

For every two functors F from $\mathcal{C}$ to $\mathcal{C}'$ and F' from $\mathcal{C}'$ to $\mathcal{C}''$, the composition $F' \circ F$, which maps all objects and morphisms of $\mathcal{C}$ by composing the corresponding mappings of F and F', is a functor from $\mathcal{C}$ to $\mathcal{C}''$.

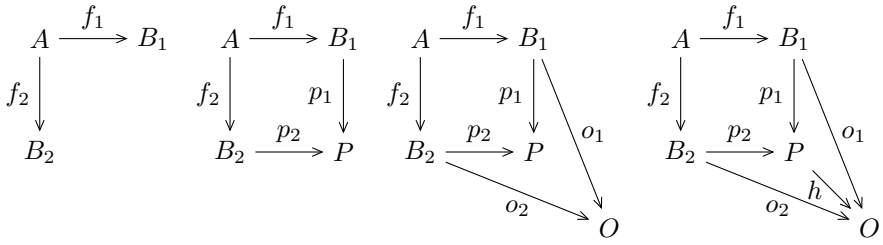
Example 2.13 (Category of Categories). The category $\mathcal{CAT}$ of categories has as objects categories, as morphisms from $\mathcal{C}$ to $\mathcal{C}'$ the functors from $\mathcal{C}$ to $\mathcal{C}'$, and as identity and composition the operations from Ex. 2.12. Technically, $\mathcal{CAT}$ is only a quasi-category just like $\mathcal{CLASS}$.

Example 2.14 (Forgetting Morphisms). The following is a functor: $|-| : \mathcal{CAT} \to \mathcal{CLASS}$, which maps every category $\mathcal{C}$ to its class of objects $|\mathcal{C}|$ and every functor to its restriction to objects. It is called a forgetful functor because it "forgets" the morphisms of categories.

Definition 2.15 (Diagrams)**.** For a category $\mathcal{C}$, a directed multi-graph in which the nodes are labelled with objects of $\mathcal{C}$, and the edges from A to B are labelled with $\mathcal{C}$-morphisms from A to B is called a (commuting) **diagram** over $\mathcal{C}$ iff the following holds: For every two paths from A to B that consist of the edges $(f_1, \ldots, f_n)$ and $(g_1, \ldots, g_n)$, it holds that $f_n \circ \ldots \circ f_1 = g_n \circ \ldots \circ g_1$.

Notation 2.16 (Diagrams). Instead of spelling out a set of properties about morphisms in a category, it is common and much easier to simply give a diagram and assert its commutativity. All diagrams in this text will be commutative unless stated otherwise.

Definition 2.17 (Pushout)**.** Consider the following diagrams in a category $\mathcal{C}$:



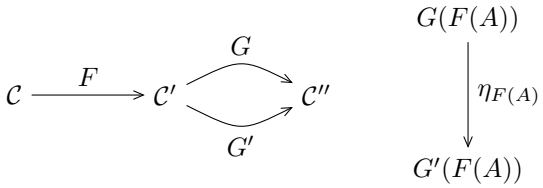
We say that (P, p_1, p_2) as in the second diagram is a **pushout** of the first diagram iff for every (O, o_1, o_2) as in the third diagram, there is a unique h as in the fourth diagram.

Definition 2.18 (Natural Transformations)**.** For two functors F and G from $\mathcal{C}$ to $\mathcal{C}'$, a **natural transformation** η from F to G is a family $(\eta_A)_{A\in|\mathcal{C}|}$ of $\mathcal{C}'$-morphisms $\eta_A : F(A) \to G(A)$ such that for every $\mathcal{C}$-morphism $f : A \to B$ the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array}$$

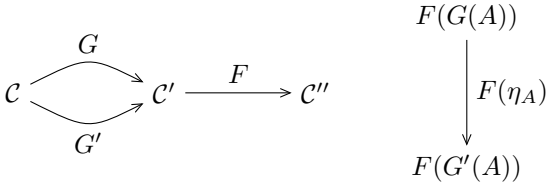
For every functor $F : \mathcal{C} \to \mathcal{C}'$, the family $(id_{F(A)})_{A\in|\mathcal{C}|}$ is a natural transformation from F to itself. And for two natural transformations $\eta : F \to G$ and $\eta' : G \to H$ between functors from $\mathcal{C}$ to $\mathcal{C}'$, the family $(\eta'_A \circ \eta_A)_{A\in|\mathcal{C}|}$ is a natural transformation from F to H. Thus, the functors from $\mathcal{C}$ to $\mathcal{C}'$, the natural transformations between them, and these identities and composition form a category.

Notation 2.19 (Composition of Functors and Natural Transformations). For a functor $F : \mathcal{C} \to \mathcal{C}'$, two functors $G, G' : \mathcal{C}' \to \mathcal{C}''$, and a natural transformation $\eta : G \to G'$, the family $(\eta_{F(A)})_{A\in|\mathcal{C}|}$ is a natural transformation $G \circ F \to G' \circ F$ as in the following (not necessarily commutative) diagrams.



We denote this natural transformation by η_F.

Similarly, for a functor $F : \mathcal{C}' \to \mathcal{C}''$, two functors $G, G' : \mathcal{C} \to \mathcal{C}'$, and a natural transformation $\eta : G \to G'$, the family $(F(\eta_A))_{A\in|\mathcal{C}|}$ is a natural transformation $F \circ G \to F \circ G'$ as in the following (not necessarily commutative) diagrams.



We denote this natural transformation by $F(\eta)$.

Furthermore, in Sect. 4, we need the advanced concepts of indexed sets, fibrations, locally cartesian closed categories, pullbacks, and adjoint functors, and some proofs use the Yoneda embedding and the Yoneda lemma, as well as sheaves and toposes. We will not define these notions and refer to the literature when they come up.

2.3.2 Institutions

Definition 2.20 (Institutions)**.** An **institution** consists of

- a signature category Sig,
- a sentence functor $\mathit{Sen} : \mathit{Sig} \to \mathcal{SET}$,
- a model category functor $\mathit{Mod} : \mathit{Sig} \to \mathcal{CAT}^{op}$,
- and for every $\Sigma \in |\mathit{Sig}|$, a satisfaction relation $\models_\Sigma \subseteq |\mathit{Mod}(\Sigma)| \times \mathit{Sen}(\Sigma)$.

These have to satisfy the satisfaction condition: For all signature morphisms $\sigma : \Sigma \to \Sigma'$, all Σ-sentences F, and all Σ'-models I', it holds that

$$\mathit{Mod}(\sigma)(I') \models_\Sigma F \text{ iff } I' \models_{\Sigma'} \mathit{Sen}(\sigma)(F).$$

Technically, Mod is only a (class-codable) quasi-functor because its codomain is a quasi-category, but we will drop this from the notation.

Notation 2.21. We will always assume that institutions $\mathbb{I}$ or $\mathbb{I}'$ are given as $(\mathit{Sig}, \mathit{Sen}, \mathit{Mod}, \models)$ or $(\mathit{Sig}', \mathit{Sen}', \mathit{Mod}', \models')$, respectively, unless stated otherwise.

Definition 2.22 (Comorphisms)**.** An institution **comorphism** $\mathbb{I} \to \mathbb{I}'$ consists of

- a functor $\Phi : \mathit{Sig} \to \mathit{Sig}'$ translating signatures,
- a natural transformation $\alpha : \mathit{Sen} \to \mathit{Sen}' \circ \Phi$ translating sentences,
- and a natural transformation $\beta : \mathit{Mod} \to \mathit{Mod}' \circ \Phi$ translating models,

such that for all $\Sigma \in |\mathit{Sig}|$, $F \in \mathit{Sen}(\Sigma)$, $I' \in |\mathit{Mod}'(\Phi(\Sigma))|$

$$\beta_\Sigma(I') \models_\Sigma F \text{ iff } I' \models_{\Phi(\Sigma)} \alpha_\Sigma(F).$$

$\mathcal{INS}$ denotes the category of institutions and institution comorphisms.

Notation 2.23. Our model category functor goes from Sig to $\mathcal{CAT}^{op}$ whereas the literature usually uses $\mathit{Mod} : \mathit{Sig}^{op} \to \mathcal{CAT}$. Similarly, our β_Σ is different from the one used in the literature. This is of course equivalent. Our notation has the advantage that Sen and Mod have the same domain, which is more elegant when we add a proof category functor in Sect. 3. The disadvantage is that for $\Sigma \to \Sigma'$, the functor $\mathit{Mod}(\sigma) : \mathit{Mod}(\Sigma) \to \mathit{Mod}(\Sigma')$ lives in $\mathcal{CAT}^{op}$ and has, intuitively, the wrong orientation. Therefore, we often identify $\mathit{Mod}(\sigma)$ with its dual functor $\mathit{Mod}(\Sigma') \to \mathit{Mod}(\Sigma)$ in the category $\mathcal{CAT}$. Diagrams involving the model functor are always drawn in $\mathcal{CAT}$.

Definition 2.24 (Comma Category)**.** For two functors $U_i : \mathcal{C}_i \to \mathcal{D}$, the **comma category** $(U_1 \downarrow U_2)$ has as objects tuples (A_1, A_2, a) for objects $A_i \in |\mathcal{C}_i|$, and a $\mathcal{D}$-morphism $a : U_1(A_1) \to U_2(A_2)$. Its morphisms from (A_1, A_2, a) to (A_1', A_2', a') are the pairs (f_1, f_2) for $f_i : A_i \to A_i'$ such that the following identity holds in A: $a' \circ U_1(f_1) = U_2(f_2) \circ a$.

Definition 2.25 (General Institutions)**.** Let U_1 and U_2 be as in Def. 2.24. Then a U_1-U_2-institution over *Sig* is a functor from *Sig* into $(U_1 \downarrow U_2)$. And for two such institutions $\mathbb{I}$, $\mathbb{I}'$ over *Sig* and *Sig*$'$, an U_1-U_2-institution comorphism consists of a functor $\Phi : Sig \to Sig'$ and a natural transformation $\rho : \mathbb{I} \to \mathbb{I}' \circ \Phi$. With the obvious identity and composition, these form the category $\mathcal{INS}\,[U_1, U_2]$.

Example 2.26. Now $\mathcal{INS}$ can be recovered as a special case. Let $\mathcal{REL}$ be the category of classes and relations, and let $|-|^r : \mathcal{CAT}^{op} \to \mathcal{REL}$ be the functor mapping categories to classes and functors to relations. Let $Incl_R : \mathcal{SET} \to \mathcal{REL}$ be the inclusion functor. Then we have $\mathcal{INS} \cong \mathcal{INS}\,[Incl_R, |-|^r]$. In particular, for every institution $\mathbb{I}$, the satisfaction relation $\models$ is a natural transformation from $Incl_R \circ Sen$ to $|-|^r \circ Mod$.

Definition 2.27 (Modifications)**.** If (Φ, ρ) and (Φ', ρ') are institution comorphisms from $\mathbb{I}$ to $\mathbb{I}'$, a **modification** from (Φ, ρ) to (Φ', ρ') is a natural transformation $m : \Phi \to \Phi'$ such that $\mathbb{I}'(m) \circ \rho = \rho'$ (which is an identity between natural transformations from $\mathbb{I}$ to $\mathbb{I}' \circ \Phi'$).

Terminology 2.28. The concept of modifications introduced in [Dia02] where a weaker condition was used. In [Tar96] a similar concept is called a representation map. Our definition is equivalent to the one given in [Mos05].

Part II

Combining Model and Proof Theory

Chapter 3

Logics and Logic Translations

3.1 Introduction

Institutions ([GB92, GR02]) provide an abstract model theoretical logical framework. In this section, we will extend institutions with an abstract notion of proof theory. For that purpose, we introduce proof categories in Sect. 3.2 and use them to define logics. In Sect. 3.3, we generalize the concept of institution comorphisms, i.e., translations between institutions, to logics. In Sect. 3.4, we discuss two related frameworks that extend institutions with proofs: the general logics of [Mes89] and the proof-theoretic institutions of [MGDT05, Dia06].

Our ultimate motivation is to give a logic for LF and use it as meta-logic as suggested in [Tar96]. We will do that in Sect. 4 and Sect. 5.

3.2 Logics

3.2.1 Proof Categories

Notation 3.1 (Products). If a category has products over the index set N, we write $(F_n)_{n\in N}$ or simply F_N for that product. Similarly, we write p_N for the universal morphism $(p_n)_{n\in N}$ into F_N.

Then we define proof categories as follows.

Definition 3.2 (Proof Categories)**.** A **proof category** is a category P that has products over all index sets whose cardinality is at most $|P|$. A morphism between proof categories is a functor preserving these products. The proof categories form a category $\mathcal{PFCAT}$ where identity and composition are defined as for functors.

We think of the irreducible objects of P as **judgments** that can be assumed and proved. And we think of a product F_N in P as a **multi-set of judgments**. The intuition of a morphism p_M from E_M to F_N is that it provides a **proof** of every judgment F_n for $n \in N$ assuming any judgments among E_M.

Example 3.3. We obtain a proof category functor Pf that maps every FOL-signature to a proof category as follows. Assume a FOL-signature Σ. The proof category $Pf(\Sigma)$ is the category of families $(F_n)_{n\in N}$ where N is some finite or countable set and $F_n \in Sen(\Sigma)$ for all $n \in N$. In other words, the objects of $Pf(\Sigma)$ are the mappings $N \to Sen(\Sigma)$ for a set N. The morphisms in $Pf(\Sigma)$ from E_M to F_N are the families $(p_n)_{n\in N}$ such that every p_n is a natural deduction proof of the sequent $E_{M^n} \vdash F_n$ for some finite $M^n \subseteq M$ and any ordering of the formulas in the family E_{M^n}. Then $Pf(\Sigma)$ has countable products by taking disjoint unions of the index sets.

In Def. 3.2, we only require products of limited cardinality. This is because formal systems typically have countable sets of judgments. There we find it enough to have countable products. Via the Curry-Howard correspondence ([CF58, How80]), countable products of judgments correspond to a context declaring a countable set of variables, where the variables act as names for hypotheses in the same way as the elements of the index sets do. For example, a morphism from E_M to F_N can be a tuple p_N of terms in context such that the context declares one variable for each formula in E_M and every p_n has type F_n. Then proof categories become Lawvere categories [Law63].

In general, we use the words judgment and proof in the widest possible sense. For example, proofs could be proof terms or semantical arguments, and judgments can be typing judgments, sequents, tableaux branches, or simply formulas. However, we commit a priori to a product structure in the proof category: Therefore, there are always proofs from (F, G) to F (weakening), from F to (F, F) (contraction), and from (F, G) to (G, F) (exchange). However, we do not see that as a bias towards certain logical systems and, e.g., against substructural logics like linear logic ([Gir87]): Such a restriction would only be the case if we intended to limit judgments to be formulas.

In particular, our definition applies to formal systems presented as a basic grammar generating expressions, a set of judgments about these expressions, and an inference system axiomatizing the judgments. And in this situation, the set of multi-sets of judgments naturally admits a product structure.

It is possible to relate the structure of the sentences or judgments of the logic to the structure of the proof categories. For example, the proof categories of Ex. 3.3 have the property that for finite M a proof from $(\ \vdash\ E_m)_{m\in M}$ to $\ \vdash\ F$ exists iff there is a proof from $()$ to $E_M\ \vdash\ F$. Similar results have been obtained for other logics (e.g., [LS86], [BdP00]). We pursued a related approach in [GMdP+07] and will not focus on this direction of research here.

3.2.2 Logics

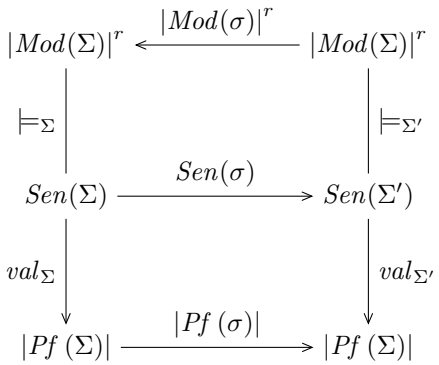
We will now define logics as institutions extended with proof categories. We will have a functor $Pf\ : Sig \to \mathcal{PFCAT}$ giving the proof category for every signature $\Sigma \in |Sig|$. Since the irreducible objects of $Pf\ (\Sigma)$ are not necessarily the sentences, we need to relate the sentences to the proof theory in some way. For that purpose, we use a map $val_\Sigma : Sen(\Sigma) \to |Pf\ (\Sigma)|$ assigning to every sentence an object in the proof category.

The intuition is that proofs, i.e., $Pf\ (\Sigma)$-morphisms, from $()$ to $val_\Sigma(F)$ represent the proofs of F. Thus, we obtain the following analogy. Sen defines the syntax, i.e., the propositions without regard for truth. Mod defines the model theory, and $\models$ provides the model theoretical definition of truth. And Pf defines the proof theory, and val provides the proof theoretical definition of truth.

Definition 3.4 (Logics)**.** Assume the definitions from Ex. 2.25 and identify $\mathcal{SET}$ with its two inclusions into $\mathcal{CLASS}$ and $\mathcal{REL}$. Then a **logic** is a tuple $\mathbb{I} = (Sig, Sen, Mod, \models, Pf\ , val)$ for a category Sig, functors $Sen : Sig \to \mathcal{SET}$, $Mod : Sig \to \mathcal{CAT}^{op}$, $Pf\ : Sig \to \mathcal{PFCAT}$, and natural transformations $\models: Sen \to |-|^r \circ Mod$ and $val : Sen \to |-| \circ Pf\ .$

Thus, a logic consists of a category of signatures Sig, a sentence functor Sen, a model category functor Mod, a proof category functor $Pf\ $, and model and proof theoretical definitions of truth $\models$ and val. We call $(Mod, \models)$ the **model theory**, and $(Pf\ , val)$ the **proof theory**. And we call $\models$ the **satisfaction relation** and val the **truth judgment**.

The structure of a logic can be visualized as follows for a signature morphism $\sigma : \Sigma \to \Sigma'$:



This is not a diagram in the sense of category theory. But if we treat the sets $Sen(\Sigma)$ and $Sen(\Sigma')$ as classes, then all arrows represent maps between classes. And the lines without arrow tips denote relations between classes. The naturality of *val* means that the lower half commutes. And the naturality of $\models$ means that the upper half commutes if regarded as a diagram in the category of classes and relations. The latter is equivalent to the satisfaction condition of institutions.

The symmetry between model and proof theory can be formalized by extending Def. 2.25. Logics are the U-V-institutions where U and V are as follows. $U : \mathcal{SET} \to \mathcal{REL} \times \mathcal{CLASS}$ maps a set S (namely, the set of sentences) to (S, S). And $V : \mathcal{CAT}^{op} \times \mathcal{PFCAT} \to \mathcal{REL} \times \mathcal{CLASS}$ maps a pair of a model and a proof category to itself but forgetting morphisms. Then the properties of comma categories yield exactly the two naturality conditions on $\models$ and *val*. More precisely, we can state this as follows.

Lemma 3.5. *The logics are in bijection with the U-V-institutions where U and V are as above.*

Proof. For a logic $(Sig, Sen, Mod, \models, Pf\ , val)$, a U-V-institution is obtained by taking the functor with domain Sig which maps a signature Σ to $\big(Sen(\Sigma), (Mod(\Sigma), Pf\ (\Sigma)), (\models_\Sigma, val_\Sigma)\big)$ and a signature morphism σ to $\big(Sen(\sigma), (Mod(\sigma), Pf\ (\sigma))\big)$. Conversely, a functor $\mathbb{I} : Sig \to \mathcal{INS}\,[U, V]$ induces a logic by decomposing the tuples into their components. □

3.2.3 Provability and Entailment

Definition 3.6 (Consequence)**.** For a given logic $\mathbb{I}$, and sets A and B of Σ-sentences, A **proves** B, written $A \vdash^{\mathbb{I}}_\Sigma B$, iff there is a $Pf\ (\Sigma)$-morphism from $\Pi val^{\mathbb{I}}_\Sigma(A)$ to $\Pi val^{\mathbb{I}}_\Sigma(B)$. And A **entails** B, written $A \models^{\mathbb{I}}_\Sigma B$, iff every model satisfying all sentences in A also satisfies all sentences in B.

Here ΠS denotes the product in $Pf\ (\Sigma)$ of a set S of objects indexed by itself, and $val^{\mathbb{I}}_\Sigma(A)$ is the set $\{val^{\mathbb{I}}_\Sigma(E) \mid E \in A\}$. We have $A \vdash^{\mathbb{I}}_\Sigma B$ iff $A \vdash^{\mathbb{I}}_\Sigma \{F\}$ for all $F \in B$, and we write $A \vdash^{\mathbb{I}}_\Sigma F$ if $B = \{F\}$. The same applies to entailment. We will drop the superscript $\mathbb{I}$ if it is clear from the context.

Terminology 3.7 (Entailment). The notion of entailment has been used with several different meanings in the literature. Sometimes it means the proof theoretical concept, which we call provability. And sometimes the right side of $A \models B$ is interpreted as a disjunction so that it corresponds to the use of $\vdash$ in sequent calculi. We use *entailment* here for lack of a generally accepted short name for *model theoretic consequence*.

Definition 3.8 (Theories)**.** Assume a logic $\mathbb{I}$. The category of $\mathbb{I}$-theories has:

- objects: pairs (Σ, Δ) of an $\mathbb{I}$-signature Σ and a set Δ of Σ-sentences,
- morphisms from (Σ, Δ) to (Σ', Δ'): signature morphisms $\sigma : \Sigma \to \Sigma'$ such that

$$\Delta' \models^{\mathbb{I}}_{\Sigma'} Sen(\Sigma)(\Delta),$$

- identity and composition: as for signature morphisms.

Proof-theoretically, we could alternatively define the notion of theory morphism in terms of the relation $\vdash$ instead of $\models$.

Terminology 3.9 (Theories). The term *theory* is not used consistently in the literature. Sometimes theories (e.g., in [GB92]) are required to be closed under semantic consequence, i.e., (Σ, Δ) is only a theory if Δ contains all sentences F such that $\Delta \models_\Sigma F$. Then theories in our sense are called *presentations*. The difference between the two definitions is not significant because every presentation uniquely determines a theory by adding all theorems to the set of axioms.

The most important properties of logics are the following:

Definition 3.10 (Soundness and Completeness)**.** A logic is **strongly sound** iff $A \vdash_\Sigma B$ implies $A \models_\Sigma B$ for all signatures Σ, and all sets of Σ-sentences A and B. If the property holds for fixed $A = \varnothing$, the logic is called (weakly) **sound**. A logic is **strongly complete** or (weakly) **complete** if the respective converse implication holds.

3.3 Logic Translations

3.3.1 Translations and Encodings

We distinguish — intuitively — between different kinds of translations from a logic $\mathbb{I}$ to a logic $\mathbb{I}'$. Simple **translations** from $\mathbb{I}$ to $\mathbb{I}'$ represent the formulas, proofs, and models of $\mathbb{I}$ as formulas, proofs, and models, respectively, of $\mathbb{I}'$. Such translations may have an inverse and the identity is always a translation of a logic to itself. **Encodings** are a special case where parts of the model or proof theory of $\mathbb{I}$ are represented using the syntax of $\mathbb{I}'$. Thus, $\mathbb{I}'$ acts as a meta-logic that is used to talk about $\mathbb{I}$. Encodings can not be inverted easily, and not all logics can encode themselves.

The importance of encodings that given a universal logic, other logics are represented in it using encodings. Thus, the same primitive notion — namely logic comorphisms — are used both for translating between logics and for encoding logics in a fixed meta-logic. Instead of meta-logic, we could also say universal logic or logical framework.

Definition 3.11. A **logic comorphism** from $\mathbb{I}$ to $\mathbb{I}'$ is a tuple $(\Phi, \alpha, \beta, \gamma)$ for a functor $\Phi : \mathit{Sig} \to \mathit{Sig}'$ and natural transformations $\alpha : \mathit{Sen} \to \mathit{Sen}' \circ \Phi$, $\beta : \mathit{Mod} \to \mathit{Mod}' \circ \Phi$, and $\gamma : \mathit{Pf} \to \mathit{Pf}' \circ \Phi$ such that

1. for all $\Sigma \in |\mathit{Sig}|$, $F \in \mathit{Sen}(\Sigma)$, $M' \in |\mathit{Mod}'(\Phi(\Sigma))|$:
$$\beta_\Sigma(M') \models_\Sigma F \quad \text{iff} \quad M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(F),$$
2. for all $\Sigma \in |\mathit{Sig}|$, $F \in \mathit{Sen}(\Sigma)$:
$$\gamma_\Sigma(\mathit{val}_\Sigma(F)) = \mathit{val}'_{\Phi(\Sigma)}(\alpha_\Sigma(F)).$$

With the obvious choices for identity and composition, we obtain the category $\mathcal{LOG}$ of logics and comorphisms. A forgetful functor $\mathit{LtoI}(-) : \mathcal{LOG} \to \mathcal{INS}$ is given by $\mathit{LtoI}(\mathit{Sig}, \mathit{Sen}, \mathit{Mod}, \models, \mathit{Pf}, \mathit{val}) := (\mathit{Sig}, \mathit{Sen}, \mathit{Mod}, \models)$ and $\mathit{LtoI}(\Phi, \alpha, \beta, \gamma) := (\Phi, \alpha, \beta)$.

A comorphism between logics consists of a signature translation $\Phi : \mathit{Sig} \to \mathit{Sig}'$, a sentence translation $\alpha_\Sigma : \mathit{Sen}(\Sigma) \to \mathit{Sen}'(\Phi(\Sigma))$, a model translation $\beta_\Sigma : \mathit{Mod}(\Sigma) \to \mathit{Mod}'(\Phi(\Sigma))$ (which lives in $\mathcal{CAT}^{op}$, i.e., is equivalent to $\beta_\Sigma : \mathit{Mod}'(\Phi(\Sigma)) \to \mathit{Mod}(\Sigma)$ in $\mathcal{CAT}$), and a proof translation $\gamma_\Sigma : \mathit{Pf}(\Sigma) \to \mathit{Pf}'(\Phi(\Sigma))$. Thus, the syntax and the proof theory are translated from $\mathbb{I}$ to $\mathbb{I}'$, whereas the model theory is translated in the opposite direction. The model translation must preserve the satisfaction relation, and the proof translation must preserve the truth judgment.

Then we can extend Lem. 3.5 as follows.

Lemma 3.12. *$\mathcal{LOG}$ and the category of U-V-institutions are isomorphic.*

Proof. For a logic comorphism $(\Phi, \alpha, \beta, \gamma)$, a U-V-institution morphism is obtained by taking (Φ, m) where m maps every Σ to $(\alpha_\Sigma, (\beta_\Sigma, \gamma_\Sigma))$. The inverse translation is accordingly. □

3.3.2 Borrowing

An important application of institution comorphisms is to derive sentences of an institution $\mathbb{I}$ using the proof system of a logic $\mathbb{I}'$.

Definition 3.13. An institution comorphism $(\Phi, \alpha, \beta) : \mathbb{I} \to \mathbb{I}'$ admits **model expansion** if for every $\mathbb{I}$-signature Σ, β_Σ is surjective on objects. We define model expansion for logic comorphisms accordingly.

The borrowing theorem ([CM97]) states that for institution comorphisms admitting model expansion, α preserves and reflects semantic consequence. Formally, this means

$$A \models_\Sigma B \quad \text{iff} \quad \alpha_\Sigma(A) \models'_{\Phi(\Sigma)} \alpha_\Sigma(B).$$

While the main application of borrowing is automated proof search, the borrowing result is usually stated without reference to proof theory in the sense of automated theorem proving. The following theorem recasts the result in terms of logics.

Theorem 3.14. *Let $\mathbb{I}$ be an institution, let $\mathbb{I}'$ be a logic, and let $\mu = (\Phi, \alpha, \beta) : \mathbb{I} \to LtoI(\mathbb{I}')$ be an institution comorphism. Then we obtain a logic $\mathbb{I}^*$ by extending $\mathbb{I}$ with $Pf^* := Pf' \circ \Phi$ and $val^* := val'_\Phi \circ \alpha$. Then*

1. *If $\mathbb{I}'$ is (strongly) complete, $\mathbb{I}^*$ is (strongly) complete.*
2. *If μ admits model expansion and $\mathbb{I}'$ is (strongly) sound, then $\mathbb{I}^*$ is (strongly) sound.*

Proof. Firstly, that $\mathbb{I}^*$ is indeed a logic, follows immediately. To prove (2), assume that μ admits model expansion, and $\mathbb{I}'$ is (strongly) sound. We need to prove the (strong) soundness of $\mathbb{I}^*$. So assume $A \vdash^*_\Sigma B$ and a Σ-model I satisfying all formulas in A. The definition of Pf^* and val^* yields $A \vdash^*_\Sigma B$ iff $\alpha_\Sigma(A) \vdash'_{\Phi(\Sigma)} \alpha_\Sigma(B)$. Then the soundness of $\mathbb{I}'$ and the surjectivity of β_Σ on objects yield $A \models^*_\Sigma B$. (1) is proved similarly. □

3.3.3 Meta-Logics

In the context of institutions, the idea of meta-logics has been studied in, e.g., [Tar96]. The idea is to use a universal institution in which other institutions are encoded via comorphisms. Then comorphisms can be encoded in the universal institution as institution comorphism modifications.

We assume an institution comorphism $\mu = (\Phi, \alpha, \beta) : \mathbb{I}^1 \to \mathbb{I}^2$ and two institution comorphisms $\mu^1 = (\Phi^1, \alpha^1, \beta^1) : \mathbb{I}^1 \to \mathbb{I}$ and $\mu^2 = (\Phi^2, \alpha^2, \beta^2) : \mathbb{I}^2 \to \mathbb{I}$, which encode $\mathbb{I}^1$ of $\mathbb{I}^2$ in $\mathbb{I}$. Then μ is encoded as an institution comorphism modification m from μ^1 to $\mu^2 \circ \mu$.

m is a family of $\mathbb{I}$-signature morphisms $m_\Sigma : \Phi^1(\Sigma) \to \Phi^2(\Phi(\Sigma))$ for every $\Sigma \in |Sig^1|$ such that the following diagrams commute for all $\Sigma \in |Sig^1|$:

$$\begin{array}{ccc} Sen^1(\Sigma) & \xrightarrow{\alpha_\Sigma} & Sen^2(\Phi(\Sigma)) \\ \downarrow{\scriptstyle \alpha^1_\Sigma} & & \downarrow{\scriptstyle \alpha^2_{\Phi(\Sigma)}} \\ Sen(\Phi^1(\Sigma)) & \xrightarrow{Sen(m_\Sigma)} & Sen(\Phi^2(\Phi(\Sigma))) \end{array} \qquad \begin{array}{ccc} Mod^1(\Sigma) & \xleftarrow{\beta_\Sigma} & Mod^2(\Phi(\Sigma)) \\ \uparrow{\scriptstyle \beta^1_\Sigma} & & \uparrow{\scriptstyle \beta^2_{\Phi(\Sigma)}} \\ Mod(\Phi^1(\Sigma)) & \xleftarrow{Mod(m_\Sigma)} & Mod(\Phi^2(\Phi(\Sigma))) \end{array}$$

Here the right diagram is drawn in $\mathcal{CAT}$ so that all arrows point in the direction of model reduction (see Not. 2.23).

Intuitively, m_Σ is a signature morphism in the meta-institution such that the $Sen(m_\Sigma)$ has the same effect as α_Σ. This is particularly intuitive if the encodings α^1_Σ and $\alpha^2_{\Phi(\Sigma)}$ are identities or bijections. Similarly, the model reduction $Mod(m_\Sigma)$ must have the same effect as β_Σ.

We obtain the definition of logic comorphism modifications by adding the corresponding requirement for proof categories. More precisely, we have the following.

Definition 3.15 (Modifications)**.** Assume μ, μ^1, and μ^2 as above. A **logic comorphism modification** m from μ^1 to $\mu^2 \circ \mu$ is an institution comorphism modification between the corresponding institution comorphisms such that in addition the following diagram commutes.

$$\begin{array}{ccc} Pf^{\,1}(\Sigma) & \xrightarrow{\gamma_\Sigma} & Pf^{\,2}(\Phi(\Sigma)) \\ \Big\downarrow{\scriptstyle \gamma^1_\Sigma} & & \Big\downarrow{\scriptstyle \gamma^2_{\Phi(\Sigma)}} \\ Pf\,(\Phi^1(\Sigma)) & \xrightarrow{Pf\,(m_\Sigma)} & Pf\,(\Phi^2(\Phi(\Sigma))) \end{array}$$

3.4 Conclusion

We added a proof category functor to institutions that is very similar to the model category functor. This similarity is affirmed by a natural transformation singling out a proof judgment for every sentence, which corresponds to the satisfaction relation. Thus, we obtain a striking correspondence of models and proofs as well as of satisfaction and truth.

Our proof categories use judgments as objects and products of judgments to denote a set of hypotheses. This has the effect that in the simplest cases, we obtain multi-sets of sentences as the objects in the proof categories. It would be much simpler to use sets instead. Indeed, our definition of proof categories is very similar to the one given in [MGDT05], which uses sets. While multi-sets are somewhat more complicated, we use them to overcome a subtle difficulty with the definition in [MGDT05].

This difficulty is due to a mistake we discovered in [Dia06]. There, a specification language is given which permits to construct proof categories in the sense of [MGDT05] inductively. This yields an institutional notion of what it means to build proofs from proof rules that is very similar to ours below. However, this does not work for signature translations that induce non-injective sentence translations. Therefore, the final version of [Dia06] restricts the main theorem to injective sentence translations.

To see what goes wrong, take a Σ-proof $\{p_1, p_2\}$ from E to $\{F_1, F_2\}$. Here, intuitively, p_j is a proof of F_j for $j = 1, 2$. Further assume a translation $\sigma : \Sigma \to \Sigma'$ that maps both sentences F_1 and F_2 to F, but does not identify the proofs p_1 and p_2. Then σ cannot induce a functor $Pf\,(\sigma) : Pf\,(\Sigma) \to Pf\,(\Sigma')$ between the proof categories: Naturally, we have $Pf\,(\sigma)(\{F_1, F_2\}) = \{F\}$, but when defining the image of $\{p_1, p_2\}$ under $Pf\,(\sigma)$, there is no canonical choice between $\{Pf\,(\sigma)(p_1)\}$ and $\{Pf\,(\sigma)(p_2)\}$. Thus, the construction in [Dia06] is only functorial if it is restricted to injective translations, which is enough in many situations but unsatisfactory in general. Our definition avoids this problem: Using families instead of sets, we have $Pf\,(\sigma)((F_1, F_2)) = (F, F)$ and $Pf\,(\sigma)((p_1, p_2)) = (Pf\,(\sigma)(p_1), Pf\,(\sigma)(p_2))$.

Our definition of logic is also similar to the one given in [Mes89]. There the proof structure is represented abstractly by an entailment system $\vdash$ between sets of sentences and sentences. This corresponds to our definition of provability, which is always an entailment system in the sense of [Mes89].

A more concrete representation of proof structure is also given in [Mes89]: It adds to an institution a category Str and a functor $P : Th \to Str$, a functor $proofs : Th \to \mathcal{SET}$, and a natural transformation $\pi : proofs \to Sen$ (where Sen is the functor $Th \to \mathcal{SET}$). A logic in our sense always induces such a proof structure: For every theory (Σ, Δ), take $Str := \mathcal{PFCAT}$, and $P(\Sigma, \Delta) := Pf\,(\Sigma)$; then let $proofs(\Sigma, \Delta)$ be the set of $Pf\,(\Sigma)$-morphisms whose domain is the product of the set $val_\Sigma(\Delta)$ and whose codomain is $val_\Sigma(F)$ for some F; finally, $\pi_{(\Sigma,\Delta)}$ maps every proof to that F. (Technically, that requires val_Σ to be injective, but that is a reasonable assumption in practice.) Our definition is less general but simpler and appears to us to be more elegant: It only adds two concepts (Pf and val) instead of four and yields an appealing symmetry of model and proof theory.

Acknowledgements While the definition of logic is ours, its conception owes to discussions with Răzvan Diaconescu, Joseph Goguen, Till Mossakowski, and Andrzej Tarlecki. An early idea to use set-indexed families of sentences is due to Till Mossakowski.

Chapter 4

Dependent Type Theory

4.1 Introduction and Related Work

Martin-Löf type theory, MLTT, is a dependent type theory ([ML74]). The main characteristic is that there are kinded function symbols that take terms as input and return types as output. This is enriched with further type constructors such as dependent sum and product. The syntax of dependent type theory is significantly more complex than that of simple type theory because well-formed types and terms and both their equalities must be defined in a single joint induction.

The semantics of MLTT is similarly complicated. In [See84], the connection between MLTT and locally cartesian closed, LCC, categories was first established. LCC categories interpret contexts Γ as objects $[\![\Gamma]\!]$, types in context Γ as objects in the slice category over $[\![\Gamma]\!]$, substitution as pullback, and dependent sum and product as left and right adjoint to pullback. But this does not address the difficulty that these three operations are not independent: Substitution of terms into types is associative and commutes with sum and product formation, which is not necessarily respected by the choices for the pullbacks and their adjoints. This is known as the coherence or strictness problem. In incoherent models, equal types are interpreted as isomorphic, but not necessarily equal objects such as in [Cur89]. In [Car86], coherent models for MLTT were given using categories with attributes. And in [Hof94], a category with attributes is constructed for every LCC category. Several other model classes and their coherence properties have been studied in, e.g., [Str91] and [Jac90]. In [Pit00], an overview is given.

These model classes have in common that they are rather abstract and have a more complicated structure than LCC categories. It is desirable to have simpler, more concrete models. But it is a hard problem to equip a given LCC category with choices for pullbacks and adjoints that are both natural and coherent. Our motivation is to find a simple concrete class of LCC categories for which such a choice can be made, and which is still complex enough to be complete for MLTT.

Mathematically, our main results can be summarized very simply: Using a theorem from topos theory, it can be shown that MLTT is complete with respect to — not necessarily coherent — models in the LCC categories of the form $\mathcal{SET}^P$ for posets P. And for these rather simple models, we can give a solution to the coherence problem. $\mathcal{SET}$ can be equipped with a coherent choice of pullback functors, and hence the categories $\mathcal{SET}^P$ can be as well. Deviating subtly from the well-known constructions, we can make coherent choices for the required adjoints to pullback. Finally, rather than working in the various slices $\mathcal{SET}^P/A$, we use the isomorphism $\mathcal{SET}^P/A \cong \mathcal{SET}^{\int_P A}$, where $\int_P A$ is the Grothendieck construction: Thus we can formulate the semantics of dependent types uniformly in terms of the simple categories of indexed sets $\mathcal{SET}^Q$ for various posets Q.

In addition to being easy to work with, this has the virtue of capturing the idea that a dependent type S in context Γ is in some sense a type-valued function on Γ: Our models interpret

Signatures	Σ	::=	$\cdot \mid \Sigma, c{:}S \mid \Sigma, a{:}(\Gamma)\mathtt{type}$
Contexts	Γ	::=	$\cdot \mid \Gamma, x{:}S$
Signature Morphisms	σ	::=	$\cdot \mid \sigma, c/s \mid \sigma, a/A$
Substitutions	γ	::=	$\cdot \mid \gamma, x/s$
Type families	A, S	::=	$a \mid A\ \gamma \mid \lambda_{x:S}\, A \mid 1 \mid Id(s, s') \mid \Sigma_{x:S}\, S' \mid \Pi_{x:S}\, S'$
Terms	s	::=	$c \mid x \mid * \mid \mathit{refl}(s) \mid \langle s, s' \rangle \mid \pi_1(s) \mid \pi_2(s) \mid \lambda_{x:S}\, s \mid s\ s'$

Figure 4.1: Basic Grammar

Γ as a poset $[\![\Gamma]\!]$ and S as an indexed set $[\![\Gamma|S]\!] : [\![\Gamma]\!] \to \mathcal{SET}$. We speak of Kripke models because these models are a natural extension of the well-known Kripke models for intuitionistic first-order logic ([Kri65]). Such models are based on a poset P of worlds, and the universe is given as a P-indexed set. This can be seen as the special case of our semantics when there is only one base type.

In fact, our results are also interesting in the special case of simple type theory ([Chu40]). Contrary to Henkin models [Hen50, MS89], and the models given in [MM91], which like ours use indexed sets on posets, our models are standard: $[\![\Gamma|S \to S']\!]$ is the exponential of $[\![\Gamma|S]\!]$ and $[\![\Gamma|S']\!]$. And contrary to the models in [Fri75, Sim95], our completeness result holds for theories with more than only base types.

A different notion of Kripke-models for dependent type theory is given in [Lip92], which is related to [All87]. There, the MLTT types are translated into predicates in an untyped first-order language. The first-order language is then interpreted in a Kripke-model, i.e., there is one indexed universe of which all types are subsets. Such models correspond roughly to non-standard set-theoretical models.

We give the syntax of MLTT in Sect. 4.2 and 4.3 and some categorical preliminaries in Sect. 4.4. Then we derive the coherent functor choices in Sect. 4.5 and use them to define the interpretation in Sect. 4.6. We give our main results regarding the interpretation of substitution, soundness, and completeness in Sect. 4.7, 4.8, and 4.9. In Sect. 4.10, we collect the pieces and obtain a logic for dependent type theory.

4.2 Syntax Overview

The basic syntax for MLTT expressions is given by the grammar in Fig. 4.1. The vocabulary of the syntax is declared in signatures and contexts: Signatures Σ declare globally accessible names c for constants of type S and names a for kinded constants with a list Γ of argument types. Contexts Γ locally declare typed variables x. Signature morphisms σ from Σ to Σ' and substitutions γ from Γ to Γ' translate between the global and local vocabularies: σ provides a Σ'-term or Σ'-type family for every Σ-symbol; and γ provides a Γ'-term for every variable in Γ. We treat signatures, contexts, signature morphisms, and substitutions as lists, and occasionally we will do an induction from the left. $\cdot$ represents the empty list in all four cases.

Relative to a signature Σ and a context Γ, there are three syntactical levels: kinds, kinded type families, and typed terms. Types are type families of the kind $\mathtt{type}$.

Type families are the type family constants a and lambda abstractions $\lambda_{x:S}\, A$. Base types are formed by application $A\ \gamma$ of a type family to a list of argument terms. The composed types are the unit type 1, the identity types $Id(s, s')$, the dependent product types $\Sigma_{x:S}\, T$, and the dependent function types $\Pi_{x:S}\, T$. Terms are constants c, variables x, the element $*$ of the unit type, the element $\mathit{refl}(s)$ of the type $Id(s, s)$, pairs $\langle s, s' \rangle$, projections $\pi_1(s)$ and $\pi_2(s)$, lambda abstractions $\lambda_{x:S}\, s$, and function applications $s\ s'$. We do not need equality axioms $s \equiv s'$ because they can be given as constants of type $Id(s, s')$. For simplicity, we omit equality axioms for types.

Judgment	Intuition
$\vdash \Sigma\ \mathtt{Sig}$	Σ is a well-formed signature
$\vdash_\Sigma \Gamma\ \mathtt{Ctx}$	Γ is a well-formed context over Σ
$\vdash \sigma : \Sigma \to \Sigma'$	σ is a well-formed signature morphism from Σ to Σ'
$\vdash_\Sigma \gamma : \Gamma \to \Gamma'$	γ is a well-formed substitution over Σ from Γ to Γ'
$\Gamma \vdash_\Sigma A : (\Gamma_0)\mathtt{type}$	type family A with kind Γ_0 is well-formed over Σ and Γ
$\Gamma \vdash_\Sigma A \equiv A'$	type families A and A' are equal over Σ and Γ
$\Gamma \vdash_\Sigma s : S$	term s is well-formed with type S over Σ and Γ
$\Gamma \vdash_\Sigma s \equiv s'$	terms s and s' are equal over Σ and Γ

Figure 4.2: Judgments

The judgments defining well-formed syntax are listed in Fig. 4.2.

Notation 4.1. It is common to use Σ as a meta-variable for signatures. And it is also common to use Σ as a syntax symbol for dependent product types. Since these two uses of Σ can always be disambiguated from the context, we will Σ in both cases.

Notation 4.2. We will adhere to the following conventions for meta-variable names for expressions in context:

Context	Term	Type	Type family
Γ	s	S	A
Γ	u	$\Sigma_{x:S}\, T$	
Γ	f	$\Pi_{x:S}\, T$	
$\Gamma, x{:}S$	t	T	

If we need more than one such object, the second one will be primed. Substitutions γ will always be from Γ to Γ'. Whenever one of these meta-variables is used, we assume implicitly that it represents an expression of the corresponding syntactic class.

4.3 Well-Formed Expressions

Our formulation and notation follow that for LF in [Pfe01] closely: We use signatures — but we add signature morphisms — and we use kinds, application, and lambda abstraction for type families. The remaining type constructors are as given for MLTT in [See84]. Thus, we obtain a system that subsumes both LF and MLTT.

Contrary to [Pfe01], we use a spine form for kinds: We write the kinding judgment $A : \Pi_{x_1:S_1} \ldots \Pi_{x_n:S_n}\ \mathtt{type}$ of LF as $A{:}(x_1{:}S_1, \ldots, x_n{:}S_n)\mathtt{type}$ so that we can use contexts instead of kinds, which simplifies the semantics. Nonetheless, we retain α-renaming in kinds.

We begin by giving the formal definitions regarding substitutions.

Definition 4.3 (Substitution Application)**.** The **application** of a substitution γ to a term, type family, context, or substitution is defined as follows.

Substitution in terms:

$$\begin{array}{lll}
\gamma(c) & := c & \\
\gamma(x) & := s & \text{for } x/s \text{ in } \gamma \\
\gamma(*) & := * & \\
\gamma(\mathit{refl}(s)) & := \mathit{refl}(\gamma(s)) & \\
\gamma(\langle s, s'\rangle) & := \langle \gamma(s), \gamma(s')\rangle & \\
\gamma(\pi_1(s)) & := \pi_1(\gamma(s)) & \\
\gamma(\pi_2(s)) & := \pi_2(\gamma(s)) & \\
\gamma(\lambda_{x:S}\, t) & := \lambda_{x:\gamma(S)}\, \gamma^x(t) & \\
\gamma(f\ s) & := \gamma(f)\ \gamma(s) &
\end{array}$$

Substitution in types:

$$\begin{array}{ll}\gamma(1) & := 1\\ \gamma(Id(s,s')) & := Id(\gamma(s),\gamma(s'))\\ \gamma(\Sigma_{x:S}\,T) & := \Sigma_{x:\gamma(S)}\,\gamma^x(T)\\ \gamma(\Pi_{x:S}\,T) & := \Pi_{x:\gamma(S)}\,\gamma^x(T)\\ \gamma(a) & := a\\ \gamma(A\;\gamma_0) & := \gamma(A)\;\gamma(\gamma_0)\\ \gamma(\lambda_{x:S}\,A) & := \lambda_{x:\gamma(S)}\,\gamma^x(A)\end{array}$$

Substitution in contexts:

$$\begin{array}{ll}\gamma(\cdot) & := \cdot\\ \gamma(x:S,\Gamma_0) & := x:\gamma(S),\gamma^x(\Gamma_0)\end{array}$$

Substitution in substitutions:

$$\begin{array}{ll}\gamma(\cdot) & := \cdot\\ \gamma(x_1/s_1,\ldots,x_n/s_n) & := x_1/\gamma(s_1),\ldots,x_n/\gamma(s_n)\end{array}$$

Here $\gamma^x(-)$ abbreviates $\gamma, x/x(-)$. The application to a variable x is undefined if γ does not contain x/s. Such cases are excluded by the type system. For the case, where only one variable is to be substituted in an expression e in context $\Gamma, x:S$, we define

$$e[x/s] := (id_\Gamma, x/s)(e).$$

Definition 4.4 (Category of Contexts and Substitutions)**.** We define the **identity** substitution of the Σ-context $\Gamma = x_1:S_1,\ldots,x_n:S_n$ by

$$id_\Gamma := x_1/x_1,\ldots,x_n/x_n.$$

We define the **composition** of two substitutions by

$$\gamma' \circ \gamma := \gamma'(\gamma).$$

And we say that two substitutions $x_1/s_1,\ldots,x_n/s_n$ and $x_1/s_1,\ldots,x_n/s'_n$ from Γ to Γ' are **equal** if

$$\Gamma' \vdash_\Sigma s_i \equiv s'_i$$

for all i. Then the contexts and substitutions form a category.

Notation 4.5. We will use id_Γ as a grammar-level abbreviation. Thus, we can use it as a component of other substitutions, e.g., $id_\Gamma, \gamma_0 : \Gamma, \Gamma_0 \to \Gamma$.

Definition 4.6 (Signature Morphisms)**.** We define identity and composition of signature morphisms, and the application $\sigma(-)$ of signature morphisms as for substitutions except that they replace signature symbols and keep variables unchanged.

The rules for signatures, signature morphisms, contexts, and substitutions are given in Fig. 4.3. A **signature** is a list of declarations of type families a or term constants c. For example $a:(\Gamma)\texttt{type}$ means that a can be applied to arguments with types given by Γ and returns a type. The domain of a signature is defined by $dom(\cdot) = \varnothing$, $dom(\Sigma, a:(\Gamma)\texttt{type}) = dom(\Sigma) \cup \{a\}$, and $dom(\Sigma, c:S) = dom(\Sigma) \cup \{c\}$. To simplify the rules for signature morphisms, we assume that the judgment $\vdash \sigma : \Sigma \to \Sigma'$ is only meaningful when $\vdash \Sigma\ \texttt{Sig}$ and $\vdash \Sigma'\ \texttt{Sig}$ have already been established. Then a **signature morphism** σ from Σ to Σ' provides a type family A over Σ for every type family symbol a of Σ, and a term s over Σ' for every term symbol c of Σ.

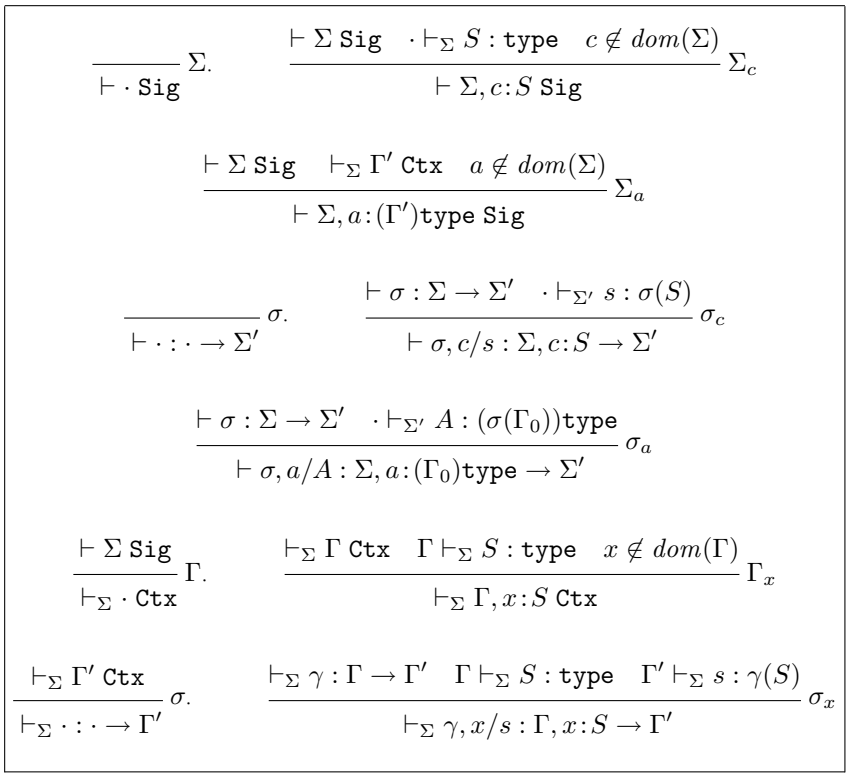


Figure 4.3: Signatures, Signature Morphisms, Contexts, Substitutions

Contexts are similar to signatures except that they only declare variables ranging over terms. The domain of a context is defined as for signatures. A **substitution** from Γ to Γ' is a list of terms in context Γ' such that each term is typed by the corresponding type in Γ. Note that in a context $x_1{:}S_1, \ldots, x_n{:}S_n$, the variable x_i may occur in $S_{i+1}, \ldots, S_n$.

Fig. 4.4 gives the formation rules for **types** and **type families**. Our rule for the application of type families is different from [Pfe01]. In context Γ, a type family $A{:}(\Gamma_0)\texttt{type}$ can be applied to a list γ_0 of argument terms and returns a type. γ_0 must be such that id_Γ, γ_0 is a substitution from Γ, Γ_0 into Γ; that means that γ_0 provides a list of terms in context Γ whose types match those in Γ_0. This corresponds to fully applying a type family to arguments in the typing system of [Pfe01]. The partial application of a type family $A{:}(x_1{:}S_1, \ldots, x_n{:}S_n)\texttt{type}$ to a term s of type S_1 can be defined as an abbreviation; for example if A and s are closed, as

$$A\ s := \lambda_{y_2:\gamma(S_2)}\ \ldots \lambda_{y_n:\gamma(S_n)}\ (A\ \gamma)$$

where $\gamma := x_1/s, x_2/y_2, \ldots, x_n/y_n$.

We write $\Sigma_{x:S}\, S'$ and $\Pi_{x:S}\, S'$ as $S \times S'$ and $S \to S'$, respectively, if x does not occur free in S'. We also write the kinding judgment $A{:}(x_1{:}S_1, \ldots, x_n{:}S_n)\texttt{type}$ as $A{:}\texttt{type}$ if $n = 0$, and as $A{:}S_1 \to \ldots \to S_n \to \texttt{type}$ if the variables do not occur in the types.

Fig. 4.5 gives the **term** formation rules. The congruence and conversion rules for **equality of terms** are given in Fig. 4.6. η-conversion, reflexivity, symmetry, transitivity, and congruence rules for the other term constructors are omitted because they are derivable or admissible. In particular, η-conversion is implied by functional extensionality $e_{funcext}$. To make the rules easier to read, we do not have a subexpression property for the equality judgment, i.e., we assume that all terms occuring in Fig. 4.6 are well-formed without making that explicit in the rules.

Finally, Fig. 4.7 gives a simple axiomatization of **equality of types and type families**.

$$\frac{a{:}(\Gamma_0)\texttt{type} \text{ in } \Sigma \quad \vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma a : (\Gamma_0)\texttt{type}}\, T_a \qquad \frac{\Gamma,\ x{:}S \vdash_\Sigma A : (\Gamma_0)\texttt{type}}{\Gamma \vdash_\Sigma \lambda_{x:S}\, A : (x{:}S,\ \Gamma_0)\texttt{type}}\, T_\lambda$$

$$\frac{\Gamma \vdash_\Sigma A : (\Gamma_0)\texttt{type} \quad \vdash_\Sigma id_\Gamma, \gamma_0 : \Gamma, \Gamma_0 \to \Gamma}{\Gamma \vdash_\Sigma A\ \gamma_0 : \texttt{type}}\, T_{app}$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma 1 : \texttt{type}}\, T_1 \qquad \frac{\Gamma \vdash_\Sigma s : S \quad \Gamma \vdash_\Sigma s' : S}{\Gamma \vdash_\Sigma Id(s, s') : \texttt{type}}\, T_{Id(-,-)}$$

$$\frac{\Gamma, x{:}S \vdash_\Sigma T : \texttt{type}}{\Gamma \vdash_\Sigma \Sigma_{x:S}\, T : \texttt{type}}\, T_\Sigma \qquad \frac{\Gamma, x{:}S \vdash_\Sigma T : \texttt{type}}{\Gamma \vdash_\Sigma \Pi_{x:S}\, T : \texttt{type}}\, T_\Pi$$

Figure 4.4: Type Families

Since we only have β and η-conversions, no type variables, and no type equality axioms, this is very simple. In particular, equality of types is decidable iff the equality of terms is.

It is easy to see that the following rule is derivable from the rules for equality of types:

$$\frac{\Gamma \vdash_\Sigma S : \texttt{type} \quad \begin{array}{c}\gamma = x_1/s_1, \ldots, x_n/s_n \\ \gamma' = x_1/s'_1, \ldots, x_n/s'_n\end{array} \quad \Gamma' \vdash_\Sigma s_i \equiv s'_i \text{ for } i = 1, \ldots, n}{\Gamma' \vdash_\Sigma \gamma(S) \equiv \gamma'(S)}$$

We mention this rule explicitly because its soundness expresses our coherence result.

Finally, we list some structural properties of the formal system.

Lemma 4.7. *We have the following subexpression properties:*

- $\vdash_\Sigma \Gamma\ \texttt{Ctx}$ *implies*
  - $\vdash \Sigma\ \texttt{Sig}$, *and*
  - $\Gamma \vdash_\Sigma S : \texttt{type}$ *for all* $x{:}S$ *in* Γ.
- $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ *implies*
  - $\vdash_\Sigma \Gamma\ \texttt{Ctx}$,
  - $\vdash_\Sigma \Gamma'\ \texttt{Ctx}$, *and*
  - $\Gamma' \vdash_\Sigma s : \gamma(S)$ *for every* x/s *in* γ *and* $x{:}S$ *in* Γ.
- $\Gamma \vdash_\Sigma A : (\Gamma_0)\texttt{type}$ *and* $dom(\Gamma) \cap dom(\Gamma_0) = \varnothing$ *implies* $\vdash_\Sigma \Gamma, \Gamma_0\ \texttt{Ctx}$.
- $\Gamma \vdash_\Sigma s : S$ *implies* $\Gamma \vdash_\Sigma S : \texttt{type}$.

Proof. This is proved by a straightforward induction on the typing derivations. □

Further properties such as weakening, exchange, and substitution could be established.

$$\frac{c{:}S \text{ in } \Sigma \quad \vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma c : S}\, t_c \qquad \frac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad x{:}S \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : S}\, t_x$$

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma * : 1}\, t_* \qquad \frac{\Gamma \vdash_\Sigma s : S}{\Gamma \vdash_\Sigma \mathit{refl}(s) : \mathit{Id}(s,s)}\, t_{\mathit{refl}(-)}$$

$$\frac{\Gamma \vdash_\Sigma s : S \quad \Gamma, x{:}S \vdash_\Sigma T : \texttt{type} \quad \Gamma \vdash_\Sigma t : T[x/s]}{\Gamma \vdash_\Sigma \langle s,t\rangle : \Sigma_{x:S}\, T}\, t_{\langle -,-\rangle}$$

$$\frac{\Gamma \vdash_\Sigma u : \Sigma_{x:S}\, T}{\Gamma \vdash_\Sigma \pi_1(u) : S}\, t_{\pi_1} \qquad \frac{\Gamma \vdash_\Sigma u : \Sigma_{x:S}\, T}{\Gamma \vdash_\Sigma \pi_2(u) : T[x/\pi_1(s)]}\, t_{\pi_2}$$

$$\frac{\Gamma, x{:}S \vdash_\Sigma t : T}{\Gamma \vdash_\Sigma \lambda_{x:S}\, t : \Pi_{x:S}\, T}\, t_\lambda \qquad \frac{\Gamma \vdash_\Sigma f : \Pi_{x:S}\, T \quad \Gamma \vdash_\Sigma s : S}{\Gamma \vdash_\Sigma f\ s : T[x/s]}\, t_{app}$$

Figure 4.5: Terms

4.4 Categorical Preliminaries

In this section, we repeat some well-known definitions and results about indexed sets and fibrations over posets (see e.g., [Joh02]). We assume the basic notions of category theory (see, e.g., [Lan98]). We use a set-theoretical pairing function (a,b) and define tuples as left-associatively nested pairs, i.e., $(a_1,\ldots,a_n)$ abbreviates $(\ldots(a_1,\ldots),a_n)$.

Definition 4.8 (Indexed Sets)**.** $\mathcal{POSET}$ denotes the category of partially ordered sets. We treat posets as categories and write $p \le p'$ for the uniquely determined morphism $p \to p'$. If P is a poset, $\mathcal{SET}^P$ denotes the category of functors $P \to \mathcal{SET}$ and natural transformations. These functors are also called **P-indexed sets**.

It is often convenient to replace an indexed set A over P with the disjoint union of all sets $A(p)$ for $p \in P$. This is a special case of a construction by Mac Lane ([LM92]) usually called the Grothendieck construction.

Definition 4.9 (Grothendieck Construction)**.** For an indexed set A over P, we define a poset $\int_P A := \{(p,a) \mid p \in P, a \in A(p)\}$ with

$$(p,a) \le (p',a') \qquad \text{iff} \qquad p \le p' \text{ and } A(p \le p')(a) = a'.$$

We also write $\int A$ instead of $\int_P A$ if P is clear from the context.

Using the Grothendieck construction, we can work with sets indexed by indexed sets: We write $P|A$ if A is an indexed set over P, and $P|A|B$ if additionally B is an indexed set over $\int_P A$.

Definition 4.10. Assume $P|A|B$. We define an indexed set $P|(A \ltimes B)$ by

$$(A \ltimes B)(p) = \{(a,b) \mid a \in A(p), b \in B(p,a)\}$$

and

$$(A \ltimes B)(p \le p') : (a,b) \mapsto \big(a', B\big((p,a) \le (p',a')\big)(b)\big) \qquad \text{for } a' = A(p \le p')(a).$$

$$\frac{\Gamma \vdash_\Sigma v : Id(s, s')}{\Gamma \vdash_\Sigma s \equiv s'}\, e_{Id(-,-)} \qquad \frac{\Gamma \vdash_\Sigma v : Id(s, s') \quad \Gamma \vdash_\Sigma v' : Id(s, s')}{\Gamma \vdash_\Sigma v \equiv v'}\, e_{id-uniq}$$

$$\frac{\Gamma \vdash_\Sigma s : 1}{\Gamma \vdash_\Sigma s \equiv *}\, e_* \qquad \frac{}{\Gamma \vdash_\Sigma \langle \pi_1(u), \pi_2(u) \rangle \equiv u}\, e_{\langle -,- \rangle}$$

$$\frac{}{\Gamma \vdash_\Sigma \pi_1(\langle s, s' \rangle) \equiv s}\, e_{\pi_1} \qquad \frac{}{\Gamma \vdash_\Sigma \pi_1(\langle s, s' \rangle) \equiv s'}\, e_{\pi_2}$$

$$\frac{}{\Gamma \vdash_\Sigma (\lambda_{x:S}\, t)\ s \equiv t[x/s]}\, e_\beta \qquad \frac{\Gamma \vdash_\Sigma f \equiv f' \quad \Gamma \vdash_\Sigma s \equiv s'}{\Gamma \vdash_\Sigma f\ s \equiv f'\ s'}\, e_{app}$$

$$\frac{\Gamma \vdash_\Sigma f : \Pi_{x:S} T \quad \Gamma \vdash_\Sigma f' : \Pi_{x:S} T \quad \Gamma, y{:}S \vdash_\Sigma f\ y \equiv f'\ y}{\Gamma \vdash_\Sigma f \equiv f'}\, e_{funcext}$$

$$\frac{\Gamma \vdash_\Sigma s : S \quad \Gamma \vdash_\Sigma s \equiv s' \quad \Gamma \vdash_\Sigma S \equiv S'}{\Gamma \vdash_\Sigma s' : S'}\, e_{typing}$$

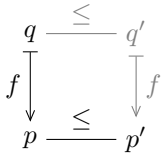
Figure 4.6: Equality of Terms

And we define a natural transformation $\pi_B : A \ltimes B \to A$ by

$$(\pi_B)_p : (a, b) \mapsto a.$$

Definition 4.11 (Fibrations)**.** A **fibration** over a poset P is a functor $f : Q \to P$ with the following property: For all $p' \in P$ and $q \in Q$ such that $f(q) \leq p'$, there is a unique $q' \in Q$ such that $q \leq q'$ and $f(q') = p'$. We call f **normal** iff f is the first projection of $Q = \int_P A$ for some $P|A$.

Technically, what we call fibrations here are discrete opfibrations, these are necessarily split. The situation in a fibration is illustrated by the following diagram where the gray parts exist uniquely in the black situation. In particular, if we keep q (and thus p) fixed, then q' is a function of p'.



For every indexed set A over P, the first projection $\int_P A \to P$ is a fibration. Conversely, every fibration $f : Q \to P$ defines an indexed set over P by mapping $p \in P$ to its preimage under f. This leads to a well-known equivalence of indexed sets and fibrations over P. If we only permit normal fibrations, we obtain an isomorphism as follows.

$$\frac{a:(\Gamma_0)\mathtt{type}\ \text{in}\ \Sigma}{\Gamma \vdash_\Sigma a \equiv a}\,E_a \qquad \frac{\Gamma \vdash_\Sigma S \equiv S' \quad \Gamma, x{:}S \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma \lambda_{x:S}\, A \equiv \lambda_{x:S'}\, A'}\,E_\lambda$$

$$\frac{}{\Gamma \vdash_\Sigma (\lambda_{x:S}\, A)\ (x/s, \gamma_0) \equiv A[x/s]\ \gamma_0}\,E_\beta$$

$$\frac{\Gamma \vdash_\Sigma A : (x_1{:}S_1, \ldots, x_n{:}S_n)\mathtt{type} \quad \gamma = id_\Gamma, x_1/y_1, \ldots, x_n/y_n}{\Gamma \vdash_\Sigma \lambda_{y_1:\gamma(S_1)}\ \ldots \lambda_{y_n:\gamma(S_n)}\,(A\ x_1/y_1, \ldots, x_n/y_n) \equiv A}\,E_\eta$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A' \quad \begin{array}{l}\gamma_0 = x_1/s_1, \ldots, x_n/s_n \\ \gamma_0' = x_1/s_1', \ldots, x_n/s_n'\end{array} \quad \Gamma \vdash_\Sigma s_i \equiv s_i' \ \text{ for } i = 1, \ldots, n}{\Gamma \vdash_\Sigma A\ \gamma_0 \equiv A'\ \gamma_0'}\,E_{app}$$

$$\frac{}{\Gamma \vdash_\Sigma 1 \equiv 1}\,E_1 \qquad \frac{\Gamma \vdash_\Sigma s_1 \equiv s_1' \quad \Gamma \vdash_\Sigma s_2 \equiv s_2'}{\Gamma \vdash_\Sigma Id(s_1, s_2) \equiv Id(s_1', s_2')}\,E_{Id(-,-)}$$

$$\frac{\Gamma \vdash_\Sigma S \equiv S' \quad \Gamma, x{:}S \vdash_\Sigma T \equiv T'}{\Gamma \vdash_\Sigma \Sigma_{x:S}\, T \equiv \Sigma_{x:S'}\, T'}\,E_\Sigma \qquad \frac{\Gamma \vdash_\Sigma S \equiv S' \quad \Gamma, x{:}S \vdash_\Sigma T \equiv T'}{\Gamma \vdash_\Sigma \Pi_{x:S}\, T \equiv \Pi_{x:S'}\, T'}\,E_\Pi$$

Figure 4.7: Equality of Type Families

Lemma 4.12. *If we restrict the objects of $\mathcal{POSET}/P$ to be normal fibrations and the morphisms to be (arbitrary) fibrations, we obtain a full subcategory $Fib(P)$ of $\mathcal{POSET}/P$. There are isomorphisms*

$$F(-) : \mathcal{SET}^P \to Fib(P) \qquad \text{and} \qquad I(-) : Fib(P) \to \mathcal{SET}^P.$$

Proof. It is straightforward to show that $Fib(P)$ is a full subcategory. The identity in $\mathcal{POSET}$ and the composition of two fibrations are fibrations. Thus, it only remains to show that if $f \circ \varphi = f'$ in $\mathcal{POSET}$ where f and f' are fibrations and φ is a morphism in $\mathcal{POSET}$, then φ is a fibration as well. This is easy.

For $A : P \to \mathcal{SET}$, we define the fibration $F(A) : \int_P A \to P$ by $(p, a) \mapsto p$. And for a natural transformation $\eta : A \to A'$, we define the fibration $F(\eta) : \int_P A \to \int_P A'$ satisfying $F(A) \circ F(\eta) = F(A')$ by $(p, a) \mapsto (p, \eta_p(a))$.

For $f : Q \to P$, we define an indexed set $I(f)$ by $I(f)(p) := \{a \mid f(p, a) = p\}$ and $I(f)(p \leq p') : a \mapsto a'$ where a' is the uniquely determined element such that $(p, a) \leq (p', a') \in Q$. And for a morphism φ between fibrations $f : Q \to P$ and $f' : Q' \to P$, we define a natural transformation $I(\varphi) : I(f) \to I(f')$ by $I(\varphi)_p : a \mapsto a'$ where a' is such that $\varphi(p, a) = (p, a')$.

Then it is easy to compute that I and F are inverse functors. □

Definition 4.13 (Indexed Elements)**.** Assume $P|A$. The **P-indexed elements** of A are given by

$$Elem(A) := \left\{ \left(a_p \in A(p)\right)_{p \in P} \mid a_{p'} = A(p \leq p')(a_p) \text{ whenever } p \leq p' \right\}.$$

We denote the P-indexed set that maps all $p \in P$ to $\{\varnothing\}$ by 1_P. Then the indexed elements of A are in bijection with the natural transformations $1_P \to A$. In particular, if P has a least

element p_0, there is exactly one indexed element of A for every $a_0 \in A(p_0)$. For $a \in \mathit{Elem}(A)$, we will write $F(a)$ for the fibration $P \to \int A$ mapping p to (p, a_p).

Example 4.14. We exemplify the introduced notions by Fig. 4.8. P is a totally ordered set visualized as a horizontal line with two elements $p_1 \leq p_2 \in P$. For $P|A$, $\int A$ becomes a blob over P. The sets $A(p_i)$ correspond to the vertical lines in $\int A$, and $a_i \in A(p_i)$. The action of $A(p \leq p')$ and the poset structure of $\int A$ are horizontal: If we assume $A(p_1 \leq p_2) : a_1 \mapsto a_2$, then $(p_1, a_1) \leq (p_2, a_2)$ in $\int A$. In general, $A(p_1 \leq p_2)$ need not be injective or surjective. The action of $F(A)$ is vertical: $F(A)$ maps (p_i, a_i) to p_i.

For $P|A|B$, $\int B$ becomes a three-dimensional blob over $\int A$. The sets $B(p_i, a_i)$ correspond to the dotted lines, and $b_i \in B(p_i, a_i)$. The action of $B((p_1, a_1) \leq (p_2, a_2))$ and the poset structure of $\int B$ are horizontal, and $F(B)$ projects $\int B$ to $\int A$.

$\int_P (A \ltimes B)$ is isomorphic to $\int_{\int_P A} B$: Their elements differ only in the bracketing, i.e., $(p_i, (a_i, b_i))$ and $((p_i, a_i), b_i)$, respectively. We have $(a_i, b_i) \in (A \ltimes B)(p_i)$, and $(A \ltimes B)(p \leq p') : (a_1, b_1) \mapsto (a_2, b_2)$. Thus, the sets $(A \ltimes B)(p_i)$ correspond to the two-dimensional gray areas. Up to this isomorphism, the projection $F(A \ltimes B)$ is the composite $F(A) \circ F(B)$.

Indexed elements $a \in \mathit{Elem}(A)$ are families $(a_p)_{p \in P}$ and correspond to horizontal curves through $\int A$ such that $F(a)$ is a section of $F(A)$. The naturality of a means that one point of the curve determines all points to the right of it. For example, $a_{p_1} = a_1$ requires $a_{p_2} = a_2$. Indexed elements of B correspond to two-dimensional vertical areas in $\int B$ (intersecting the dotted lines exactly once), and indexed elements of $A \ltimes B$ correspond to horizontal curves in $\int B$ (intersecting the gray areas exactly once).

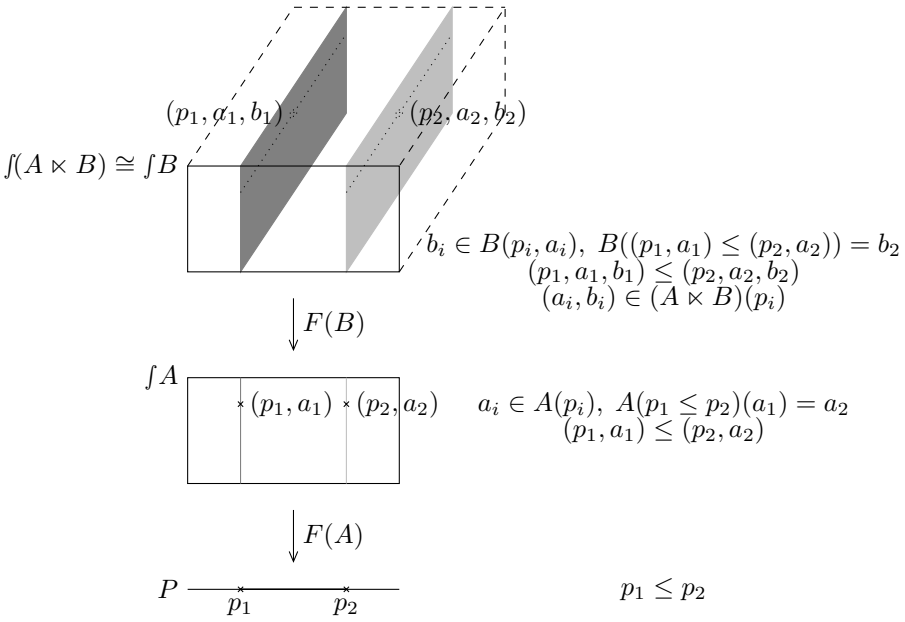


Figure 4.8: Indexed Sets and Fibrations

We will use Lem. 4.12 frequently to switch between indexed sets and fibrations. In particular, we will use the following two corollaries.

Lemma 4.15. *Assume $P|A$. Then*

$$\mathit{Elem}(A) \cong \mathit{Hom}_{\mathit{Fib}(P)}(id_P, F(A)) = \{f : P \to \textstyle\int_P A \mid F(A) \circ f = id_P\}.$$

and

$$\mathcal{SET}^P/A \cong \mathcal{SET}^{\int A}$$

Proof. Both claims follow from Lem. 4.12 by using $Elem(A) = Hom_{\mathcal{SET}^P}(1_P, A)$ as well as $Fib(P)/F(A) \cong Fib(\int_P A)$, respectively. □

Finally, we say that a category is **locally cartesian closed** (LCC) if it and all of its slice categories are cartesian closed (in particular, it has a terminal object). Then we have the following well-known result.

Lemma 4.16. *$\mathcal{SET}^P$ is LCC.*

Proof. The terminal object is given by 1_P. The product is taken pointwise: $A \times B : p \mapsto A(p) \times B(p)$ and similarly for morphisms. The exponential object is given by: $B^A : p \mapsto Hom_{\mathcal{SET}^P}(A^p, B^p)$ where A^p and B^p are as A and B but restricted to $P^p := \{p' \in P \mid p \leq p'\}$. $B^A(p \leq p')$ maps a natural transformation, which is a family of mappings over P^p, to its restriction to $P^{p'}$. This proves that $\mathcal{SET}^P$ and so also $Fib(P)$ is cartesian closed for any P. By Lem. 4.15, we obtain the same for all slice categories. □

4.5 Operations on Indexed Sets

Because $\mathcal{SET}^P$ is LCC, we know that it has pullbacks and that the pullback along a fixed natural transformation has left and right adjoints ([Hof94]). However, these functors are only unique up to isomorphism, and it is non-trivial to pick coherent choices for them.

Pullbacks Assume $P|A_1$ and $P|A_2$ and a natural transformation $h : A_2 \to A_1$. The pullback along h is a functor $\mathcal{SET}^P/A_1 \to \mathcal{SET}^P/A_2$. Using Lem. 4.15, we can avoid dealing with slice categories of $\mathcal{SET}^P$ and give a functor

$$h^* : \mathcal{SET}^{\int A_1} \to \mathcal{SET}^{\int A_2}$$

and call it the pullback along h. h^* is given by precomposition.

Definition 4.17. Assume A_1 and A_2 indexed over P, and a natural transformation $h : A_2 \to A_1$. Then we put for $B \in \mathcal{SET}^{\int A_1}$

$$h^*B := B \circ F(h) \in \mathcal{SET}^{\int A_2}.$$

As usual we define the action of h^* on morphisms via the universal property of the pullback (which is proved below). And we define a natural transformation between P-indexed sets by

$$h \ltimes B : A_2 \ltimes h^*B \to A_1 \ltimes B, \qquad (h \ltimes B)_p : (a_2, b) \mapsto (h_p(a_2), b).$$

The application of $h \ltimes B$ is independent of its second argument, which is only needed in the notation to determine the domain and codomain of $h \ltimes B$.

Lemma 4.18 (Pullbacks)**.** *In the situation of Def. 4.17, the following is a pullback in $\mathcal{SET}^P$.*

$$\begin{array}{ccc} A_2 \ltimes h^*B & \xrightarrow{h \ltimes B} & A_1 \ltimes B \\ \downarrow{\scriptstyle \pi_{h^*B}} & & \downarrow{\scriptstyle \pi_B} \\ A_2 & \xrightarrow{\quad h \quad} & A_1 \end{array}$$

Furthermore, we have the following coherence properties for every natural transformation $g : A_3 \to A_2$*:*

$$id_{A_1}{}^* B = B, \qquad id_{A_1} \ltimes B = id_{A_1 \ltimes B},$$
$$(h \circ g)^* B = g^*(h^* B), \qquad (h \circ g) \ltimes B = (h \ltimes B) \circ (g \ltimes h^* B).$$

Proof. The following is a pullback in $\mathcal{POSET}$:

$$\begin{array}{ccc} \int A_2 \ltimes h^* B & \xrightarrow{F(h \ltimes B)} & \int A_1 \ltimes B \\ \downarrow{\scriptstyle F(\pi_{h^* B})} & & \downarrow{\scriptstyle F(\pi_B)} \\ \int A_2 & \xrightarrow{F(h)} & \int A_1 \end{array} \qquad \begin{array}{ccc} (p, (a_2, b)) & \xmapsto{F(h \ltimes B)} & (p, (h_p(a_2), b)) \\ \downarrow{\scriptstyle F(\pi_{h^* B})} & & \downarrow{\scriptstyle F(\pi_B)} \\ (p, a_2) & \xmapsto{F(h)} & (p, h_p(a_2)) \end{array}$$

If we turn this square into a cocone on P by adding the canonical projections $F(A_2)$ and $F(A_1)$, it becomes a pullback in $Fib(P)$. Then the result follows by Lem. 4.12. The coherence properties can be verified by simple computations. □

Equivalently, using the terminology of [Pit00], we can say that for every P the tuple

$$(\mathcal{SET}^P, \mathcal{SET}^{\int A}, A \ltimes B, \pi_B, h^* B, h \ltimes B)$$

forms a type category (where A, B, h indicate arbitrary arguments). Then giving coherent adjoints to the pullback means to show that this type category admits dependent sums and products.

Adjoints To interpret MLTT, the adjoints to h^*, where $h : A_2 \to A_1$, are only needed if h is a projection, i.e., $A_1 := A$, $A_2 := A \ltimes B$, and $h := \pi_B$ for some $P|A|B$. We only give adjoint functors for this special case because we use this restriction when defining the right adjoint. Thus, we give functors

$$\mathcal{L}_B, \mathcal{R}_B : \mathcal{SET}^{\int A \ltimes B} \to \mathcal{SET}^{\int A} \text{ such that } \mathcal{L}_B \dashv \pi_B{}^* \dashv \mathcal{R}_B.$$

Definition 4.19. We define the functor $\mathcal{L}_B$ as follows. For an object C, we put $\mathcal{L}_B C := B \ltimes (C \circ assoc)$ where *assoc* maps elements $((p, a), b) \in \int B$ to $(p, (a, b)) \in \int A \ltimes B$; and for a morphism, i.e., a natural transformation $\eta : C \to C'$, we put

$$(\mathcal{L}_B \eta)_{(p,a)} : (b, c) \mapsto (b, \eta_{(p,(a,b))}(c)) \qquad \text{for } (p, a) \in \int A.$$

Lemma 4.20 (Left Adjoint)**.** *$\mathcal{L}_B$ is left adjoint to $\pi_B{}^*$. Furthermore, for any natural transformation $g : A' \to A$, we have the following coherence property (the Beck-Chevalley condition)*

$$g^*(\mathcal{L}_B C) = \mathcal{L}_{g^* B}(g \ltimes B)^* C.$$

Proof. It is easy to show that $\mathcal{L}_B$ is isomorphic to precomposition, for which the adjointness is well-known. In particular, we have the following diagram in $\mathcal{SET}^P$:

$$\begin{array}{ccc} (A \ltimes B) \ltimes C & \overset{\cong}{\leftrightarrow} & A \ltimes \mathcal{L}_B C \\ \downarrow{\scriptstyle \pi_C} & & \\ A \ltimes B & & \swarrow{\scriptstyle \pi_{\mathcal{L}_B C}} \\ \downarrow{\scriptstyle \pi_B} & & \\ A & & \end{array}$$

The coherence can be verified by direct computation. □

The right adjoint is more complicated. Intuitively, $\mathcal{R}_B C$ must represent the dependent functions from B to C. The natural candidate for this is $Elem(C) \cong Hom(1_{\int B}, C)$ (i.e., $Hom(B, C)$ in the simply-typed case), but this is not a $\int A$-indexed set. There is a well-known construction how to remedy this, but we use a subtle modification to achieve coherence, i.e., the analogue of the Beck-Chevalley condition. To do that, we need an auxiliary definition.

Definition 4.21. Assume $P|A|B$, $P|A \ltimes B|C$, and an element $x := (p, a) \in \int A$. Let $y^x \in \mathcal{SET}^P$ and a natural transformation $i : y^x \to A$ be given by

$$y^x(p') = \begin{cases} \{\varnothing\} & \text{if } p \leq p' \\ \varnothing & \text{otherwise} \end{cases} \qquad i_{p'} : \varnothing \mapsto A(p \leq p')(a).$$

Then we define indexed sets $P|y^x|B^x$ and $P|y^x \ltimes B^x|C^x$ by:

$$B^x := i^* B, \qquad C^x := (i \ltimes B)^* C$$

and put $d^x := \int y^x \ltimes B^x$ for the domain of C^x.

The left diagram in Fig. 4.9 shows the involved P-indexed sets, the right one gives the actions of the natural transformations for an element $p' \in P$ with $p \leq p'$. Coherence will hold because B^x and C^x contain tuples in which a' is replaced with $\varnothing$.

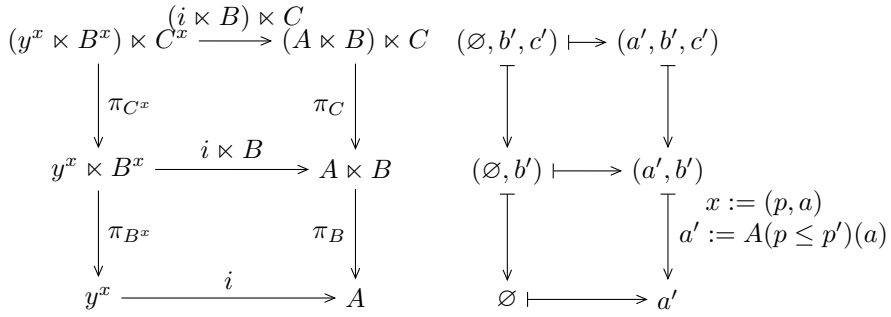


Figure 4.9: The Situation of Def. 4.21

Definition 4.22. Assume $P|A|B$. Then we define the functor $\mathcal{R}_B : \mathcal{SET}^{\int A \ltimes B} \to \mathcal{SET}^{\int A}$ as follows. Firstly, for an object C, we put for $x \in \int A$

$$(\mathcal{R}_B C)(x) := Elem(C^x).$$

In particular, $f \in (\mathcal{R}_B C)(x)$ is a family $(f_y)_{y \in d^x}$ for $f_y \in C^x(y)$. For $x \leq x' \in \int A$, we have $d^x \supseteq d^{x'}$ and put

$$(\mathcal{R}_B C)(x \leq x') : (f_y)_{y \in d^x} \mapsto (f_y)_{y \in d^{x'}}.$$

Secondly, for a morphism, i.e., a natural transformation $\eta : C \to C'$, we define $\mathcal{R}_B \eta : \mathcal{R}_B C \to \mathcal{R}_B C'$ as follows: For $x := (p, a) \in \int A$ and $f \in (\mathcal{R}_B C)(x)$, we define $f' := (\mathcal{R}_B \eta)_x(f) \in (\mathcal{R}_B C')(x)$ by

$$f'_{(p', (\varnothing, b'))} := \eta_{(p', (a', b'))}(f_{(p', (\varnothing, b'))}) \qquad \text{for } (p', (\varnothing, b')) \in d^x \text{ and } a' := A(p \leq p')(a).$$

Lemma 4.23 (Right Adjoint)**.** *$\mathcal{R}_B$ is right adjoint to $\pi_B{}^*$. Furthermore, for every natural transformation $g : A' \to A$, we have the following coherence property*

$$g^*(\mathcal{R}_B C) = \mathcal{R}_{g^*B}(g \ltimes B)^* C.$$

Proof. Assume $P|A|B$, $P|A \ltimes B|C$, and $x = (p, a) \in \int A$. Let $y(x) \in \mathcal{SET}^{\int A}$ be the covariant representable functor of x mapping $x' \in \int A$ to a singleton iff $x \leq x'$ and to the empty set otherwise. Since we know the right adjoint exists, we can use the Yoneda lemma for covariant functors to derive sufficient and necessary constraints for $\mathcal{R}_B$ to be a right adjoint:

$$\begin{aligned}(\mathcal{R}_B C)(x) &\cong Hom_{\mathcal{SET}^{\int A}}(y(x), \mathcal{R}_B C) \cong Hom_{\mathcal{SET}^{\int A \ltimes B}}(\pi_B{}^* y(x), C) \\ &\cong Hom_{Fib(\int A \ltimes B)}(F(\pi_B{}^* y(x)), F(C)).\end{aligned}$$

Let i be as in Def. 4.21. Let $Fib'(Q)$ be the category of (not necessarily normal) fibrations on Q. Then it is easy to check that $F(i \ltimes B)$ seen as a fibration with domain d^x and $F(\pi_B{}^* y(x))$ are isomorphic in $Fib'(\int A \ltimes B)$. (They are not isomorphic in $Fib(\int B)$ because the former is not normal.) Using the fullness of $Fib(Q)$, we obtain

$$\begin{aligned}(\mathcal{R}_B C)(x) &\cong Hom_{Fib'(\int A \ltimes B)}(F(i \ltimes B), F(C)) \\ &= \{f : d^x \to \int C \mid F(C) \circ f = F(i \ltimes B)\}.\end{aligned}$$

And using the definition of C^x as a pullback, we obtain

$$(\mathcal{R}_B C)(x) \cong \{f : d^x \to \int C^x \mid F(C^x) \circ f = id_{d^x}\} \cong Elem(C^x).$$

And this is indeed how $\mathcal{R}_B C$ is defined. The value of $\mathcal{R}_B C$ on morphisms is verified similarly.

To show the coherence property, assume $P|A'$, $g : A' \to A$, and $x' := (p, a') \in \int A'$. We abbreviate as follows: $a := g_p(a')$, $x := (p, a)$, $B' := g^* B$, and $C' := (g \ltimes B)^* C$. Furthermore, we denote by i' the natural transformation from Def. 4.21 used to construct the indexed sets $B'^{x'}$ and $C'^{x'}$.

Now coherence requires $g^* \mathcal{R}_B C = \mathcal{R}_{B'} C'$. And that follows if we show that

$$B'^{x'} = B^x \qquad \text{and} \qquad C'^{x'} = C^x.$$

Using Lem. 4.18, this follows from $g \circ i' = i$, which is an equality between natural transformations from $y^x = y^{x'}$ to A in $\mathcal{SET}^P$. And to verify the latter, assume $o \in P$. The maps $g_o \circ i'_o$ and i_o have domain $\varnothing$ or $\{\varnothing\}$. In the former case, there is nothing to prove. In the latter case, put

$$a'_o := i'_o(\varnothing) = A'(p \leq o)(a') \qquad \text{and} \qquad a_o := i_o(\varnothing) = A(p \leq o)(a).$$

Then we need to show $g_o(a'_o) = a_o$. And that is indeed the case because of the naturality of g as indicated in

$$\begin{array}{ccc} a' & \xmapsto{A'(p \leq o)} & a'_o \\ \Big\downarrow{\scriptstyle g_p} & & \Big\downarrow{\scriptstyle g_o} \\ a & \xmapsto{A(p \leq o)} & a_o \end{array}$$

□

The adjointness implies $Elem(\mathcal{R}_B C) \cong Elem(C)$. We spell out this isomorphism explicitly because we will use it lateron.

Lemma 4.24. *Assume $P|A|B$ and $P|A \ltimes B|C$. For $t \in Elem(C)$ and $x := (p,a) \in \int A$, let $t^x \in Elem(C^x)$ be given by*

$$(t^x)_{(p',(\varnothing,b'))} = t_{(p',(a',b'))} \qquad \text{where } a' := A(p \leq p')(a).$$

And for $f \in Elem(\mathcal{R}_B C)$ and $x := (p,(a,b)) \in \int A \ltimes B$, we have $f_{(p,a)} \in Elem(C^x)$; thus, we can put

$$f^x := (f_{(p,a)})_{(p,(\varnothing,b))} \ \in \ C(p,(a,b)).$$

Then the sets $Elem(C)$ and $Elem(\mathcal{R}_B C)$ are in bijection via

$$Elem(C) \ \ni \ t \ \overset{sp(-)}{\longmapsto} \ (t^x)_{x \in \int A} \ \in \ Elem(\mathcal{R}_B C)$$

and

$$Elem(\mathcal{R}_B C) \ \ni \ f \ \overset{am(-)}{\longmapsto} \ (f^x)_{x \in \int A \ltimes B} \ \in \ Elem(C)$$

Proof. This follows from the right adjointness by easy computations. □

Intuitively, $sp(t)$ turns $t \in Elem(C)$ into a $\int A$-indexed set by splitting it into components. And $am(f)$ glues such a tuple of components back together. Syntactically, these operations correspond to currying and uncurrying, respectively.

Then we need one last notation. For $P|A$, indexed elements $a \in Elem(A)$ behave like mappings with domain P. We can precompose such indexed elements with fibrations $f : Q \to P$ to obtain Q-indexed elements of $Elem(A \circ f)$.

Definition 4.25. Assume $P|A$, $f : Q \to P$, and $a \in Elem(A)$. $a * f \in Elem(A \circ f)$ is defined by: $(a * f)_q := a_{f(q)}$ for $q \in Q$.

4.6 Model Theory

Using the operations from Sect. 4.5, the definition of the semantics is straightforward. To demonstrate its simplicity, we will spell it out in an elementary way. The models are Kripke-models, i.e., a Σ-model I is based on a poset P^I of worlds, and provides interpretations $[\![c]\!]^I$ and $[\![a]\!]^I$ for all symbols declared in Σ. I extends to a function $[\![-]\!]^I$, which interprets all Σ-expressions. We will omit the index I if no confusion is possible. The interpretation is such that

- for a context Γ, $[\![\Gamma]\!]$ is a poset,
- for a substitution γ from Γ to Γ', $[\![\gamma]\!]$ is a poset morphism from $[\![\Gamma']\!]$ to $[\![\Gamma]\!]$,
- for a type family A of kind Γ_0, $[\![\Gamma|A]\!]$ is an indexed set on $[\![\Gamma, \Gamma_0]\!]$,
- in particular, for a type S, $[\![\Gamma|S]\!]$ is an indexed set on $[\![\Gamma]\!]$,
- for a term s of type S, $[\![\Gamma|s]\!]$ is an indexed element of $[\![\Gamma|S]\!]$.

If $\Gamma = x_1 : S_1, \ldots, x_n : S_n$, an element of $[\![\Gamma]\!]$ has the form $(p,(a_1,\ldots,a_n))$ where $p \in P$, $a_1 \in [\![\cdot|S_1]\!](p)$, $\ldots$, $a_n \in [\![x_1 : S_1, \ldots, x_{n-1} : S_{n-1}|S_n]\!](p,(a_1,\ldots,a_{n-1}))$. Intuitively, a_i is an assignment to the variable x_i in world p. For a typed term $\Gamma \vdash_\Sigma s : S$, both $[\![\Gamma|s]\!]$ and $[\![\Gamma|S]\!]$ are indexed over $[\![\Gamma]\!]$. And if an assignment (p,α) is given, the interpretations of s and S satisfy $[\![\Gamma|s]\!]_{(p,\alpha)} \in [\![\Gamma|S]\!](p,\alpha)$. This is illustrated in the left diagram in Fig. 4.10.

If γ is a substitution $\Gamma \to \Gamma'$, then $[\![\gamma]\!]$ maps assignments $(p,\alpha') \in [\![\Gamma']\!]$ to assignments $(p,\alpha) \in [\![\Gamma]\!]$. And a substitution in types and terms is interpreted by pullback, i.e., composition. This is illustrated in the right diagram in Fig. 4.10; its commutativity expresses the coherence.

The poset P of worlds plays the same role as the various posets $[\![\Gamma]\!]$ — it interprets the empty context. In this way, P can be regarded as interpreting an implicit or relative context. This is in keeping with the practice of type theory (and category theory), according to which closed expressions may be considered relative to some fixed but unspecified context (respectively, base category).

For a type family and a kind $\Gamma \vdash_\Sigma A : (\Gamma_0)\mathtt{type}$ with n variables in Γ_0, $[\![\Gamma|A]\!]$ assigns a set to every tuple $(p, (\alpha, a_1, \ldots, a_n))$. Here p is a world, (p, α) is an assignment for Γ, and $a_1, \ldots, a_n$ are the values of the arguments that can be provided to A. This is illustrated in Fig. 4.11.

Sum types are interpreted naturally as the dependent sum of indexed sets given by the left adjoint. And pairing and projections have their natural semantics. Product types are interpreted as exponentials using the right adjoint. A lambda abstraction $\lambda_{x:S}\, t$ is interpreted by first interpreting t and then splitting it as in Lem. 4.24. And an application $f\ s$ is interpreted by amalgamating the interpretation of f as in Lem. 4.24 and using the composition from Def. 4.25.

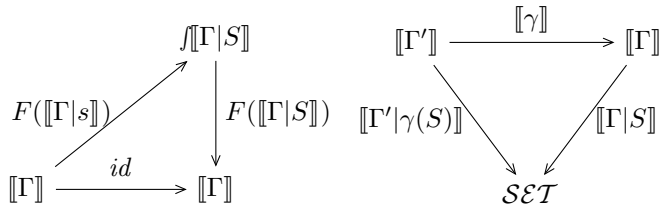


Figure 4.10: Semantics of Terms, Types, and Substitution

Definition 4.26 (Models)**.** For a signature Σ, Σ**-models** are defined as follows:

- A model I for the empty signature $\cdot$ is a poset P^I.
- A model I for the signature $\Sigma, c : S$ consists of a Σ-model I_Σ and an indexed element $[\![c]\!]^I \in \mathit{Elem}([\![\cdot|S]\!]^{I_\Sigma})$.
- A model I for the signature $\Sigma, a : (\Gamma_0)\mathtt{type}$ consists of a Σ-model I_Σ and an indexed set $[\![a]\!]^I$ over $[\![\Gamma_0]\!]^{I_\Sigma}$.

We call a model **simple** if P^I is a singleton.

Simple models yield the naive, i.e., not indexed, set theoretical semantics of MLTT: Types are interpreted as sets and terms as elements. This semantics is not complete in general, but often simple models are all that one needs. Our semantics is sound for any poset P. And completeness would still hold if we restricted P to complete Heyting algebras.

Definition 4.27 (Model Extension)**.** The extension of a model is defined by induction on the typing derivations. We assume in each case that all occurring expressions are well-formed. For example in the case for $[\![\Gamma|f\ s]\!]$, f has type $\Pi_{x:S}\, T$ and s has type S.

- Contexts:
$$[\![\cdot]\!] := 1_P \qquad [\![\Gamma, x:S]\!] := [\![\Gamma]\!] \ltimes [\![\Gamma|S]\!]$$
This defines $[\![\Gamma]\!]$ as a P-indexed set, but we will usually use $[\![\Gamma]\!]$ to refer to the induced poset. Also, we usually omit the occurrence of $\varnothing$ in elements of these posets.
- Substitutions $\gamma = x_1/s_1, \ldots, x_n/s_n$ from Γ to Γ':
$$[\![\gamma]\!] : (p, \alpha') \mapsto \big(p, ([\![\Gamma'|s_1]\!]_{(p,\alpha')}, \ldots, [\![\Gamma'|s_n]\!]_{(p,\alpha')})\big) \qquad \text{for } (p, \alpha') \in [\![\Gamma']\!]$$

- Type families:

$$\begin{aligned}
[\![\Gamma|a]\!] &:= [\![a]\!] \circ [\![id_{\Gamma_0}]\!] && \text{where } \vdash_\Sigma id_{\Gamma_0} : \Gamma_0 \to \Gamma, \Gamma_0 \\
[\![\Gamma|A\ \gamma_0]\!] &:= [\![\Gamma|A]\!] \circ [\![id_\Gamma, \gamma_0]\!] \\
[\![\Gamma|\lambda_{x:S}\ A]\!] &:= [\![\Gamma, x:S|A]\!]
\end{aligned}$$

- Composed types:

$$\begin{aligned}
[\![\Gamma|1]\!](p,\alpha) &:= \{\varnothing\} \\
[\![\Gamma|Id(s,s')]\!](p,\alpha) &:= \begin{cases} \{\varnothing\} & \text{if } [\![\Gamma|s]\!]_{(p,\alpha)} = [\![\Gamma|s']\!]_{(p,\alpha)} \\ \varnothing & \text{otherwise} \end{cases} \\
[\![\Gamma|\Sigma_{x:S}\ T]\!] &:= \mathcal{L}_{[\![\Gamma|S]\!]}[\![\Gamma, x:S|T]\!] \\
[\![\Gamma|\Pi_{x:S}\ T]\!] &:= \mathcal{R}_{[\![\Gamma|S]\!]}[\![\Gamma, x:S|T]\!]
\end{aligned}$$

  $[\![\Gamma|1]\!]$ and $[\![\Gamma|Id(s,s')]\!]$ are only defined on objects; their extension to morphisms is uniquely determined.

- Elementary terms:

$$[\![\Gamma|c]\!]_{(p,\alpha)} := [\![c]\!]_p, \qquad [\![x_1:S_1, \ldots, x_n:S_n|x_i]\!]_{(p,(a_1,\ldots,a_n))} := a_i$$

- Composed terms:

$$\begin{aligned}
[\![\Gamma|*]\!]_{(p,\alpha)} &:= \varnothing \\
[\![\Gamma|\mathit{refl}(s)]\!]_{(p,\alpha)} &:= \varnothing \\
[\![\Gamma|\langle s, s'\rangle]\!]_{(p,\alpha)} &:= ([\![\Gamma|s]\!]_{(p,\alpha)}, [\![\Gamma|s']\!]_{(p,\alpha)}) \\
[\![\Gamma|\pi_i(u)]\!]_{(p,\alpha)} &:= a_i \quad \text{where } [\![\Gamma|u]\!]_{(p,\alpha)} = (a_1, a_2) \\
[\![\Gamma|\lambda_{x:S}\ t]\!] &:= sp([\![\Gamma, x:S|t]\!]) \\
[\![\Gamma|f\ s]\!] &:= am([\![\Gamma|f]\!]) * (\mathit{assoc} \circ F([\![\Gamma|s]\!]))
\end{aligned}$$

  Here *assoc* maps $((p,\alpha),a)$ to $(p,(\alpha,a))$.

Since the same expression may have more than one well-formedness derivation, the well-definedness of Def. 4.27 must be proved in a joint induction with the proof of Thm. 4.31 below. And because of the use of substitution, e.g., for application of function terms, the induction must be intertwined with the proof of Thm. 4.28 as well.

4.7 Substitution Lemma

Theorem 4.28 (Substitution)**.** *Assume* $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$*. Then:*

1. *for a substitution* $\vdash_\Sigma \gamma' : \Gamma' \to \Gamma''$*:* $[\![\gamma' \circ \gamma]\!] = [\![\gamma]\!] \circ [\![\gamma']\!]$,
2. *for a type family* $\Gamma \vdash_\Sigma A : (\Gamma_0)\mathtt{type}$*:* $[\![\Gamma'|\gamma(A)]\!] = [\![\Gamma|A]\!] \circ [\![\gamma, id_{\gamma(\Gamma_0)}]\!]$,
3. *in particular, for a type* $\Gamma \vdash_\Sigma S : \mathtt{type}$*:* $[\![\Gamma'|\gamma(S)]\!] = [\![\Gamma|S]\!] \circ [\![\gamma]\!]$
4. *for a term* $\Gamma \vdash_\Sigma s : S$*:* $[\![\Gamma'|\gamma(s)]\!] = [\![\Gamma|s]\!] * [\![\gamma]\!]$.

For every substitution γ from Γ to Γ', the $\mathcal{POSET}$-morphism $[\![\gamma]\!] : [\![\Gamma']\!] \to [\![\Gamma]\!]$ induces a natural transformation between the P-indexed sets induced by Γ and Γ', which we denote by $I([\![\gamma]\!])$ in the sequel.

Before we prove Thm. 4.28, we establish two auxiliary lemmas.

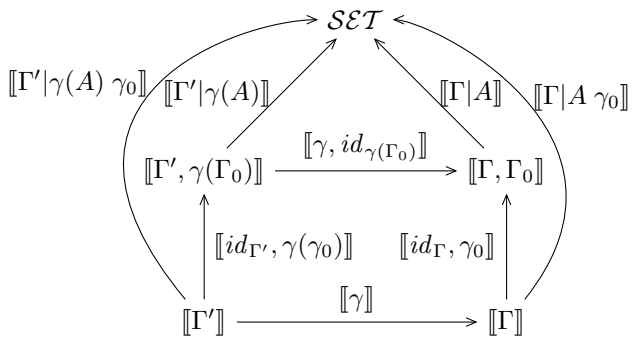


Figure 4.11: Semantics of Type Families and Substitution

Lemma 4.29. *Assume* $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ *and* $\Gamma \vdash_\Sigma S : \mathtt{type}$ *and thus also*

$$\vdash_\Sigma\ \gamma, x/x\ :\ \Gamma, x{:}S\ \to\ \Gamma', x{:}\gamma(S)\ .$$

Furthermore, assume the induction hypothesis of Thm. 4.28 for the involved expressions. Then we have:

$$[\![\gamma, x/x]\!] = F(I([\![\gamma]\!]) \ltimes [\![\Gamma|S]\!]).$$

Proof. This follows from direct computation. □

Lemma 4.30. *Assume* $P|A|B$, $P|A \ltimes B|C$, $P|A'$, *a natural transformation* $g : A' \to A$, *and* $t \in Elem(C)$. *Then for* $x' \in \int A'$*:*

$$sp(t * F(g \ltimes B))_{x'} = sp(t)_{F(g)(x')}.$$

Proof. This follows by direct computation. □

Proof of Thm. 4.28.

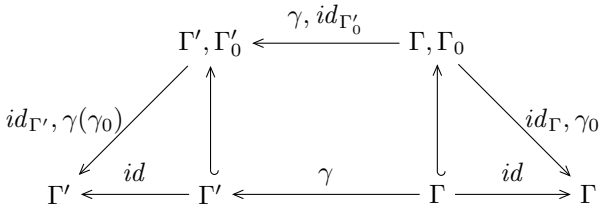
Proof. The proofs of all subtheorems are intertwined in an induction on the typing derivations; in addition, the induction is intertwined with the proof of Thm. 4.31.

For the case of a substitution γ', assume $\gamma = x_1/s_1, \ldots, x_n/s_n$ and $(p, \alpha'') \in [\![\Gamma'']\!]$. Then applying the composition of substitutions, the semantics of substitutions, the induction hypothesis for terms, and the semantics of substitutions, respectively, yields:

$$\begin{aligned}&[\![\gamma' \circ \gamma]\!](p, \alpha'') = [\![x_1/\gamma'(s_1), \ldots, x_n/\gamma'(s_n)]\!](p, \alpha'') = \left([\![\Gamma''|\gamma'(s_i)]\!]_{(p,\alpha'')}\right)_{i=1,\ldots,n}\\ &= \left([\![\Gamma'|s_i]\!]_{[\![\gamma]\!](p,\alpha'')}\right)_{i=1,\ldots,n} = ([\![\gamma]\!] \circ [\![\gamma']\!])(p, \alpha'')\end{aligned}$$

The cases for type families are as follows:

- a: Clear because $\gamma(a) = a$ and the semantics of a projects the context away.
- $A\ \gamma_0$: Assume $\Gamma \vdash_\Sigma A : (\Gamma_0)\mathtt{type}$, and abbreviate $\Gamma'_0 := \gamma(\Gamma_0)$. The commutativity of the following diagram can be directly verified:



Using the induction hypothesis for the composition of substitutions, the interpretation turns it into a diagram in $\mathcal{POSET}$. From that, we obtain the needed equality because (using the semantics of application)

$$[\![\Gamma|A\ \gamma_0]\!] \circ [\![\gamma]\!] = [\![\Gamma|A]\!] \circ [\![id_\Gamma, \gamma_0]\!] \circ [\![\gamma]\!]$$

and (using the definition of substitution, the semantics of application, and the induction hypothesis, respectively)

$$\begin{aligned}&[\![\Gamma'|\gamma(A\ \gamma_0)]\!] = [\![\Gamma'|\gamma(A)\ \gamma(\gamma_0)]\!]\\ &= [\![\Gamma'|\gamma(A)]\!] \circ [\![id_{\Gamma'}, \gamma(\gamma_0)]\!] = [\![\Gamma|A]\!] \circ [\![\gamma, id_{\Gamma'_0}]\!] \circ [\![id_{\Gamma'}, \gamma(\gamma_0)]\!]\end{aligned}$$

- $\lambda_{x:S}\, A$: Applying the definition of substitution, the semantics of lambda abstraction, and the induction hypothesis, respectively, we obtain:

$$\begin{aligned}&[\![\Gamma'|\gamma(\lambda_{x:S}\ A)]\!] = [\![\Gamma'|\lambda_{x:\gamma(S)}\ \gamma^x(A)]\!]\\ &= [\![\Gamma', x{:}\gamma(S)|\gamma^x(A)]\!] = [\![\Gamma, x{:}S|A]\!] \circ [\![\gamma^x, id_{\gamma^x(\Gamma_0)}]\!]\end{aligned}$$

  Then the needed equality follows from $[\![\Gamma, x : S|A]\!] = [\![\Gamma|\lambda_{x:S}\, A]\!]$ and $\gamma^x, id_{\gamma^x(\Gamma_0)} = \gamma, id_{x\gamma(S),\gamma(\Gamma_0)}$.

- 1: Trivial.

- $Id(s, s')$: This follows directly from the induction hypothesis for s and s'.

- $\Sigma_{x:S}\, T$: This follows directly by combining the induction hypothesis, Lem. 4.20, and Lem. 4.29.

- $\Pi_{x:S}\, T$: This follows directly by combining the induction hypothesis, Lem. 4.23, and Lem. 4.29.

And for the cases of a term s, let us assume a fixed $(p, \alpha') \in [\![\Gamma']\!]$ and $(p, \alpha) := [\![\gamma]\!](p, \alpha')$. Then we need to show

$$[\![\Gamma'|\gamma(s)]\!]_{(p,a')} = [\![\Gamma|s]\!]_{(p,a)}.$$

- c: Clear because $\gamma(c) = c$.

- x: Assume x occurs in position i in Γ, and let x/s be in γ. Further, assume $\alpha' = (a'_1, \ldots, a'_n)$ and $\alpha = (a_1, \ldots, a_n)$. Then by the properties of substitutions: $[\![\Gamma'|\gamma(x)]\!]_{(p,\alpha')} = [\![\Gamma'|s]\!]_{(p,\alpha')} = a_i$. And that is equal to $[\![\Gamma|x]\!]_{(p,\alpha)}$.

- $refl(s)$: Trivial.

- $*$: Trivial.

- $\langle s, s'\rangle$: Because $\gamma(\langle s, s'\rangle) = \langle\gamma(s), \gamma(s')\rangle$, this case follows immediately from the induction hypothesis.

- $\pi_i(u)$ for $i = 1, 2$: Because $\gamma(\pi_i(s)) = \pi_i(\gamma(s))$, this case follows immediately from the induction hypothesis.

- $\lambda_{x:S}\, t$: By the definition of substitution, the semantics of lambda abstraction, the induction hypothesis, and Lem. 4.29, respectively, we obtain:

$$\begin{aligned}[\![\Gamma'|\gamma(\lambda_{x:S}\, t)]\!] = [\![\Gamma'|\lambda_{x:\gamma(S)}\, \gamma^x(t)]\!] &= sp([\![\Gamma', x{:}\gamma(S)|\gamma^x(t)]\!])\\ &= sp([\![\Gamma, x{:}S|t]\!] * [\![\gamma, x/x]\!])\\ &= sp([\![\Gamma, x{:}S|t]\!] * F(I([\![\gamma]\!]) \ltimes [\![\Gamma|S]\!])).\end{aligned}$$

Furthermore, we have $[\![\Gamma|\lambda_{x:S}\, t]\!] = sp([\![\Gamma, x:S|t]\!])$. Then the result follows by using Lem. 4.30 and $F(I([\![\gamma]\!])) = [\![\gamma]\!]$.

- $f\ s$: We evaluate both sides of the needed equation. On the left-hand side, we obtain by the definition of substitution, the semantics of application, and the induction hypothesis, respectively:

$$[\![\Gamma'|\gamma(f\ s)]\!] = [\![\Gamma'|\gamma(f)\ \gamma(s)]\!] = am([\![\Gamma'|\gamma(f)]\!]) * (assoc \circ F([\![\Gamma'|\gamma(s)]\!]))$$
$$= am([\![\Gamma|f]\!] * [\![\gamma]\!]) * (assoc \circ F([\![\Gamma|s]\!] * [\![\gamma]\!])).$$

To compute the value at (p, α') of this indexed element, we first compute $([\![\Gamma|s]\!] * [\![\gamma]\!])_{(p,\alpha')}$, say we obtain b. Then we can compute $am([\![\Gamma|f]\!] * [\![\gamma]\!])_{(p,(\alpha',b))}$. Using the notation from Lem. 4.24, this yields $\left([\![\Gamma|f]\!] * [\![\gamma]\!]\right)^{(p,(\alpha',b))}$, which is equal to $([\![\Gamma|f]\!]_{(p,\alpha)})_{(p,(\varnothing,b))}$.

On the right-hand side, we have by the semantics of application:

$$[\![\Gamma|f\ s]\!] = am([\![\Gamma|f]\!]) * (assoc \circ F([\![\Gamma|s]\!])).$$

When computing the value at (p, α) of this indexed element, we obtain in a first step $am([\![\Gamma|f]\!])_{(p,(\alpha,b))}$. And evaluating further, this yields $([\![\Gamma|f]\!]_{(p,\alpha)})_{(p,(\varnothing,b))}$.

Thus, the equality holds as needed.

□

4.8 Soundness

Theorem 4.31 (Soundness)**.** *Assume a signature Σ, and a context Γ. If $\Gamma \vdash_\Sigma S \equiv S'$ for two well-formed types S, S', then in every Σ-model:*

$$[\![\Gamma|S]\!] = [\![\Gamma|S']\!] \in \mathcal{SET}^{[\![\Gamma]\!]}.$$

And if $\Gamma \vdash_\Sigma s \equiv s'$ for two well-formed terms s, s' of type S, then in every Σ-model:

$$[\![\Gamma|s]\!] = [\![\Gamma|s']\!] \in Elem([\![\Gamma|S]\!]).$$

Proof. The soundness is proved by induction over all derivations; the induction is intertwined with the proof of Thm. 4.28. An instructive example is the rule e_{typing}. Its soundness states the following: If $[\![\Gamma|s]\!] \in Elem([\![\Gamma|S]\!])$ and $[\![\Gamma|s]\!] = [\![\Gamma|s']\!]$ and $[\![\Gamma|S]\!] = [\![\Gamma|S']\!]$, then also $[\![\Gamma|s']\!] \in Elem([\![\Gamma|S']\!])$. And this clearly holds.

Among the remaining rules for terms, the soundness of some rules is an immediate consequence of the semantics. These are: all rules from Fig. 4.5 except for maybe t_λ and t_{app}, and from Fig. 4.6 the rules $e_{Id(-,-)}$, $e_{id-uniq}$, e_*, $e_{\langle -,-\rangle}$, e_{π_1}, e_{π_2}, and e_{app}.

The soundness of the rules t_λ and t_{app} follows by applying the semantics and Lem. 4.24. That leaves the rules e_β and $e_{funcext}$, the soundness of which we will prove in detail.

For e_β, we interpret $(\lambda_{x:S}\, t)\ s$ by applying the definition:

$$[\![\Gamma|(\lambda_{x:S}\, t)\ s]\!] = am([\![\Gamma|\lambda_{x:S}\, t]\!]) * (assoc \circ F([\![\Gamma|s]\!]))$$
$$= am(sp([\![\Gamma, x:S|t]\!])) * (assoc \circ F([\![\Gamma|s]\!]))$$

$am(sp([\![\Gamma, x:S|t]\!]))$ is equal to $[\![\Gamma, x:S|t]\!]$ by Lem. 4.24. Furthermore, we have $t[x/s] = \gamma(t)$ where $\gamma = id_\Gamma, x/s$ is a substitution from $\Gamma, x:S$ to Γ. And interpreting γ yields $[\![\gamma]\!](p, \alpha) = (p, (\alpha, [\![\Gamma|s]\!]_{(p,\alpha)}))$, i.e., $[\![\gamma]\!] = assoc \circ F([\![\Gamma|s]\!])$. Therefore, using Thm. 4.28 for terms yields

$$[\![\Gamma|t[x/s]]\!] = [\![\Gamma, x:S|t]\!] * (assoc \circ F([\![\Gamma|s]\!])),$$

which concludes the soundness proof for e_β.

To understand the soundness of $e_{funcext}$, let us look at the interpretations of f in the contexts Γ and $\Gamma, y{:}S$:

$$am([\![\Gamma|f]\!]) \in Elem([\![\Gamma, x{:}S|T]\!]), \qquad am([\![\Gamma, y{:}S|f]\!]) \in Elem([\![\Gamma, y{:}S, x{:}S|T]\!]).$$

Let γ be the inclusion substitution from Γ to $\Gamma, y{:}S$. Then $[\![\gamma]\!]$ is the projection $[\![\Gamma, y{:}S]\!] \to [\![\Gamma]\!]$ mapping elements $(p, (\alpha, a))$ to (p, α). Applying Thm. 4.28 yields for arbitrary $(p, \alpha) \in [\![\Gamma]\!]$ and $a', a \in [\![\Gamma|S]\!](p, \alpha)$:

$$am([\![\Gamma, y{:}S|f]\!])_{(p,(\alpha,a',a))} = am([\![\Gamma|f]\!])_{(p,(\alpha,a))}.$$

And we have

$$[\![\Gamma, y{:}S|y]\!]_{(p,(\alpha,a'))} = a', \qquad \text{and} \qquad F([\![\Gamma, y{:}S|y]\!])(p, (\alpha, a')) = (p, (\alpha, a'), a').$$

Putting these together yields

$$\begin{aligned}[\![\Gamma, y{:}S|f\ y]\!]_{(p,(\alpha,a'))} &= \big(am([\![\Gamma, y{:}S|f]\!]) * (assoc \circ F([\![\Gamma, y{:}S|y]\!]))\big)_{(p,(\alpha,a'))} \\ &= am([\![\Gamma, y{:}S|f]\!])_{(p,(\alpha,a',a'))} = am([\![\Gamma|f]\!])_{(p,(\alpha,a'))}\end{aligned}$$

Therefore, the induction hypothesis applied to $\Gamma, y{:}S \vdash_\Sigma f\ y \equiv f'\ y$ yields

$$am([\![\Gamma|f]\!]) = am([\![\Gamma|f']\!]).$$

And then Lem. 4.24 yields

$$[\![\Gamma|f]\!] = [\![\Gamma|f']\!]$$

concluding the soundness proof for $e_{funcext}$.

Regarding the rules for types in Fig. 4.4 and Fig. 4.7, the soundness proofs are straightforward except for E_β and E_η. For E_β, we apply the definition of the semantics on the left side:

$$[\![\Gamma|(\lambda_{x:S}\, A)\ (x/s, \gamma_0)]\!] = [\![\Gamma|\lambda_{x:S}\, A]\!] \circ [\![id_\Gamma, x/s, \gamma_0]\!] = [\![\Gamma, x{:}S|A]\!] \circ [\![id_\Gamma, x/s, \gamma_0]\!]$$

And on the right side, we apply Thm. 4.28 and the definition of the semantics:

$$[\![\Gamma|A[x/s]\ \gamma_0]\!] = [\![\Gamma|A[x/s]]\!] \circ [\![id_\Gamma, \gamma_0]\!] = [\![\Gamma, x{:}S|A]\!] \circ [\![id_\Gamma, x/s, id_{\Gamma_0[x/s]}]\!] \circ [\![id_\Gamma, \gamma_0]\!]$$

And applying the definition of the composition of substitutions and its semantics as given in Thm. 4.28, we see that the two sides are the same, which shows the soundness of E_β.

For E_η, let $\Gamma_0 := x_1 : S_1, \ldots, x_n : S_n$, $S_i^y := \gamma(S_i)$, $\Gamma_0^y := y_1 : S_1^y, \ldots, y_n : S_n^y$, and $\gamma_0 := x_1/y_1, \ldots, x_n/y_n$; i.e., γ_0 and Γ_0^y express renaming the x_i in Γ_0 to y_i. We apply on the left side of the equation the definition of the semantics (twice) and Thm. 4.28 for the inclusion substitution from Γ to Γ, Γ_0^y, respectively:

$$\begin{aligned}&[\![\Gamma|\lambda_{y_1:S_1^y}\ \ldots \lambda_{y_n:S_n^y}\ (A\ \gamma_0)]\!] = [\![\Gamma, \Gamma_0^y|A\ \gamma_0]\!] = [\![\Gamma, \Gamma_0^y|A]\!] \circ [\![id_{\Gamma,\Gamma_0^y}, \gamma_0]\!] \\ &= [\![\Gamma|A]\!] \circ [\![id_\Gamma, id_{\Gamma_0}]\!] \circ [\![id_{\Gamma,\Gamma_0^y}, \gamma_0]\!]\end{aligned}$$

Here id_Γ, id_{Γ_0} is a substitution from Γ, Γ_0 to $\Gamma, \Gamma_0^y, \Gamma_0$, and $id_{\Gamma,\Gamma_0^y}, \gamma_0$ is a substitution from $\Gamma, \Gamma_0^y, \Gamma_0$ to Γ, Γ_0^y. Then using $[\![\Gamma, \Gamma_0]\!] = [\![\Gamma, \Gamma_0^y]\!]$ reduces the above to $[\![\Gamma|A]\!]$. □

4.9 Completeness

Definition 4.32. We call a functor $F : \mathcal{C} \to \mathcal{D}$ LCC if $\mathcal{C}$ is LCC and if F preserves that structure, i.e., F maps terminal object, products and exponentials in all slices $\mathcal{C}/A$ to corresponding constructions in $\mathcal{D}/F(A)$. An LCC functor is called an LCC embedding if it is injective on objects, full, and faithful.

Lemma 4.33. *For every LCC category $\mathcal{C}$, there are a poset P and an LCC embedding $E : \mathcal{C} \to \mathcal{SET}^P$.*

Proof. Clearly, the composition of LCC embeddings is an LCC embedding. We obtain $E : \mathcal{C} \to \mathcal{SET}^P$ as a composite $E_3 \circ E_2 \circ E_1$. $E_1 : \mathcal{C} \to \mathcal{SET}^{\mathcal{C}^{op}}$ is the Yoneda embedding, which maps $A \in |\mathcal{C}|$ to $Hom(-, A)$. This is well-known to be an LCC embedding. E_2 maps a presheaf on $\mathcal{C}$ to a sheaf on a topological space S. E_2 is the inverse image part of the spatial cover of the topos $\mathcal{SET}^{\mathcal{C}^{op}}$ of presheaves on $\mathcal{C}$. This construction rests on a general topos-theoretical result established in [BM99], and we refer to [Awo00] for the details of the construction of S, the definition of E_2, and the proof that E_2 is an LCC embedding. Finally $E_3 : sh(S) \to \mathcal{SET}^{O(S)^{op}}$ includes a sheaf on S into the category of presheaves on the poset $O(S)$ of open sets of S. That E_3 is an LCC embedding, can be verified directly. Finally, we put $P := O(S)^{op}$ so that E becomes an LCC embedding into $\mathcal{SET}^P$. □

Definition 4.34 (Term-Generated)**.** A Σ-model I is called term-generated if for all closed Σ-types S and every indexed element $e \in Elem([\![\cdot|S]\!]^I)$, there is a Σ-term s of type S such that $[\![\cdot|s]\!]^I = e$.

Theorem 4.35 (Model Existence)**.** *For every signature Σ, there is a term-generated model I such that for all types $\Gamma \vdash_\Sigma S : \mathtt{type}$*

$$Elem([\![\Gamma|S]\!]^I) \neq \varnothing \qquad \text{iff} \qquad \Gamma \vdash_\Sigma s : S \text{ for some } s,$$

and for all such terms $\Gamma \vdash_\Sigma s : S$ and $\Gamma \vdash_\Sigma s' : S$

$$[\![\Gamma|s]\!]^I = [\![\Gamma|s']\!]^I \qquad \text{iff} \qquad \Gamma \vdash_\Sigma s \equiv s'.$$

Proof. It is well-known how to obtain a syntactical category $\mathcal{C}$ from Σ and Γ. The objects of $\mathcal{C}$ are given by the set of all types S such that $\Gamma \vdash_\Sigma S : \mathtt{type}$ modulo the equivalence relation $\Gamma \vdash_\Sigma S \equiv S'$. We will write $[S]$ for the equivalence class of S.

The $\mathcal{C}$-morphisms from $[S]$ to $[S']$ are given by the terms f such that $\Gamma \vdash_\Sigma f : S \to S'$ modulo the equivalence relation $\Gamma \vdash_\Sigma f \equiv f'$. We will write $[f]$ for the equivalence class of f.

It is straightforward to check that $\mathcal{C}$ is LCC (see, e.g., [See84]). For example, the exponential $f_2^{f_1}$ of two slices $\Gamma \vdash_\Sigma f_1 : S_1 \to S$ and $\Gamma \vdash_\Sigma f_2 : S_2 \to S$ is given by

$$\lambda_{u:U}\, \pi_1(u) \qquad \text{where} \qquad U := \Sigma_{x:S} \left(\Sigma_{y_1:S_1} Id(x, f_1\ y_1) \to \Sigma_{y_2:S_2} Id(x, f_2\ y_2)\right).$$

By Lem. 4.33 there are a poset P and an LCC embedding $E : \mathcal{C} \to \mathcal{SET}^P$. From those, we construct the needed model I over P. Essentially, I arises by interpreting every term or type as its image under E.

Firstly, assume a declaration $c : S$ in Σ. Since $\mathcal{C}$ only uses types and function terms, E cannot in general be applied to c. But using the type 1, every term c of type S can be seen as the function term $\lambda_{x:1}\, c$ of type $1 \to S$. Therefore, we define $E'(c) := E(\lambda_{x:1}\, c)$, which is an indexed element of $E(1 \to S)$. Since $Elem(E(1 \to S))$ and $Elem(E(S))$ are in bijection, $E'(c)$ induces an indexed element of $E(S)$, which we use to define $[\![c]\!]^I$.

Secondly, assume a declaration $a : (\Gamma_0)\mathtt{type}$ in Σ. $[\![a]\!]^I$ must be an indexed set over $[\![\Gamma_0]\!]^I$. For the same reason as above, E cannot always be applied to the symbol a. But E can be applied to

$a\ \gamma_0$ for every substitution γ_0 from Γ into $\cdot$. And because E is full, the interpretations $[\![\cdot|a\ \gamma_0]\!]^I$ for all γ_0 determine $[\![a]\!]^I$ uniquely.

That I is term-generated, follows in a straightforward way from the fullness of E.

Finally, we must show that I has the two required properties. These follows easily from the fullness and the faithfulness of E. □

Theorem 4.36 (Completeness)**.** *For all terms* $\Gamma \vdash_\Sigma s : S$ *and* $\Gamma \vdash_\Sigma s' : S$*: If* $[\![\Gamma|s]\!]^I = [\![\Gamma|s']\!]^I$ *holds for all* Σ*-models* I*, then* $\Gamma \vdash_\Sigma s \equiv s'$*.*

Proof. This follows immediately from Thm. 4.35. □

Functional Completeness The fact that the model I in Thm. 4.35 is term-generated can be interpreted as functional completeness of the semantics: A natural transformation that exists in every model is syntactically definable. To see what this means, let I be the model constructed in Thm. 4.35, and assume a natural transformation $\eta : [\![\cdot|S]\!]^I \to [\![\cdot|S']\!]^I$ for some Σ-types S and S'. Then there exists a Σ-term f of type $S \to S'$ such that η arises from $[\![\cdot|f]\!]^I$ as follows. Put $\eta' := am([\![\cdot|f]\!]^I) \in Elem([\![x : S|S']\!]^I)$. Then η' maps pairs (p, a) to elements of $[\![x{:}S|S']\!]^I(p, a) = [\![\cdot|S']\!]^I(p)$ for $a \in [\![\cdot|S]\!]^I(p)$. Then we obtain η as $\eta_p : a \mapsto \eta'(p, a)$.

4.10 A Logic for DTT

In this section, we define a logic $\mathbb{D}$ for DTT based on the syntax and semantics introduced before. Using the propositions-as-types paradigm, the closed types are the sentences over a given signature. The model theory is given by the models described in Sect. 4.6, and the proof theory is given by the inference system of Sect. 4.3.

However, we add an empty type 0 to our type theory. Thus, we have the Curry-Howard analogue to falsity and thus to first-order logic, in particular to negation. Since this introduces the danger of inconsistencies, we have deliberately omitted this type in the preceding sections so that we can encapsulate it here.

4.10.1 Syntax

Infinite Vocabularies It is desirable to be able to reason about infinite signatures and contexts. Therefore, we use infinite (albeit countable) lists of declarations. We use the following notation for **infinite lists**.

Notation 4.37 (Infinite Lists). We use $D_1, \ldots$ to denote finite or countably infinite lists (including the empty sequence). A **prefix** of a list $D_1, \ldots$ is a list $D_1, \ldots, D_n$ for some $n \in \mathbb{N}$ (including $n = 0$, in which case the prefix is empty). The meta-variable X^* always ranges over prefixes of X.

From now on, we use Σ, Γ, σ, and γ for infinite signatures, contexts, signature morphisms, and substitutions. Since our judgments can only talk about finite lists, we generalize them to the infinite case.

Definition 4.38. We define for possibly infinite Σ, Σ', σ, Γ, Γ', or γ:

- $\vdash \Sigma$ `Sig` iff $\vdash \Sigma^*$ `Sig` for every prefix Σ^*,
- $\vdash \sigma : \Sigma \to \Sigma'$ iff $\vdash \Sigma$ `Sig` and $\vdash \Sigma'$ `Sig` and for every prefix σ^* there are prefixes Σ^*, Σ'^* such that $\vdash \sigma^* : \Sigma^* \to \Sigma'^*$,
- $\vdash_\Sigma \Gamma$ `Ctx` iff $\vdash \Sigma$ `Sig` and for every prefix Γ^* there is a prefix Σ^* such that $\vdash_{\Sigma^*} \Gamma^*$ `Ctx`,
- $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$ iff $\vdash_\Sigma \Gamma$ `Ctx` and $\vdash_\Sigma \Gamma'$ `Ctx` and for every prefix γ^* there are prefixes Σ^*, Γ^*, Γ'^* such that $\vdash_{\Sigma^*} \gamma^* : \Gamma^* \to \Gamma'^*$,

Type	Formula
1	true
$Id(s, s')$	$s \equiv s'$
$a\ x_1/s_1, \ldots, x_n/s_n$	$a(s_1, \ldots, s_n)$
$S \times S'$	$S \wedge S'$
$\Sigma_{x:S}\, T$	$\exists x{:}S.T$
$S \to S'$	$S \Rightarrow S'$
$(S \to S') \times (S' \to S)$	$S \Leftrightarrow S'$
$\Pi_{x:S}\, T$	$\forall x{:}S.T$
0	false
$S \to 0$	$\neg S$
$\neg S \to \neg S' \to 0$	$S \vee S'$

Figure 4.12: Judgments as Types

- the judgments for typing, kinding, and equality are generalized accordingly.

Propositions as Types The type-constructors of MLTT internalize logical formulas via the judgments-as-types principle as in Fig. 4.12. Since negation is not covered in MLTT, we add a type 0, which is intended to correspond to falsity. Then negation can be introduced as an abbreviation.

Therefore, we add the two rules from Fig. 4.13 to the syntax. These add the type 0 and functions from it into every other type. Therefore, if 0 is inhabited, every type is inhabited and in particular all terms of the same type are equal. We consider 0 and the terms $!!_S$ as an (infinite) set of extra signature symbols that are preserved by signature morphisms. Then it is easy to see that Thm. 4.28, 4.31, and 4.36 also hold for this extended version of the type theory.

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\Gamma \vdash_\Sigma 0 : \texttt{type}}\, T_0 \qquad \frac{\Gamma \vdash_\Sigma S : \texttt{type}}{\Gamma \vdash_\Sigma !!_S : 0 \to S}\, t_0$$

$$\frac{}{\Gamma \vdash_\Sigma 0 \equiv 0}\, E_0$$

Figure 4.13: The 0 type

Definition 4.39 (Signatures)**.** We define the category $\mathit{Sig}^{\mathbb{D}}$ as follows.

- Objects: lists Σ of declarations such that $\vdash \Sigma\ \texttt{Sig}$.
- Morphisms from Σ to Σ': lists σ of assignments such that $\vdash \sigma : \Sigma \to \Sigma'$. Two signature morphisms are equal iff they are equal component-wise.
- Identity and composition are defined in the obvious way.

Definition 4.40 (Sentences)**.** We define a functor $\mathit{Sen}^{\mathbb{D}} : \mathit{Sig}^{\mathbb{D}} \to \mathcal{SET}$ as follows:

- For $\Sigma \in \mathit{Sig}^{\mathbb{D}}$, the set $\mathit{Sen}^{\mathbb{D}}(\Sigma)$ is the set of types S such that

$$\cdot \vdash_\Sigma S : \texttt{type}$$

modulo the equivalence relation

$$\cdot \vdash_\Sigma S \equiv S'.$$

- For a signature morphism σ, the mapping $Sen^{\mathbb{D}}(\sigma)$ is given by $\sigma(-)$.

4.10.2 Model Theory

Now we define the model theoretical semantics of $\mathbb{D}$ by giving $Mod^{\mathbb{D}}$ and $\models^{\mathbb{D}}$.

Definition 4.41 (Models)**.** The Σ-models are defined by induction on Σ.

- For finite Σ, models are defined as in Def. 4.26.
- For infinite Σ, a Σ-model is an infinite list such that every prefix is a model of the corresponding prefix of Σ.
- We do not prescribe interpretations for the type 0 and the terms $!!_S$; as for Σ-symbols, every model must provide interpretations for them.

Finally, we define $Mod^{\mathbb{D}}(\Sigma)$ as the class of all Σ-models I satisfying that $Elem([\![0]\!]^I) = \varnothing$. $Mod^{\mathbb{D}}$ extends to a functor $Sig \to \mathcal{CAT}^{op}$ if we treat each model class $Mod^{\mathbb{D}}(\Sigma)$ as a discrete category.

Thus, Σ-models may interpret 0 in any way. Only when defining $Mod^{\mathbb{D}}(\Sigma)$, we throw away all models for which 0 is inhabited. This way it is easier to reuse the results obtained before. Note that we do not require models to satisfy $[\![0]\!](p) = \varnothing$ in every world p — only $[\![0]\!]$ as an indexed set may not have indexed elements. Intuitively, this means that some worlds of the Kripke model may be inconsistent as long as the model as a whole is consistent.

Definition 4.42 (Model Reduction)**.** For a signature morphism $\vdash \sigma : \Sigma \to \Sigma'$, the mapping $Mod^{\mathbb{D}}(\sigma) : Mod^{\mathbb{D}}(\Sigma') \to Mod^{\mathbb{D}}(\Sigma)$ is defined as follows. I' is mapped to I given by

- $P^I := P^{I'}$,
- $[\![c]\!]^I := [\![\cdot|\sigma(c)]\!]^{I'}$ for term constants c,
- $[\![a]\!]^I := [\![\cdot|\sigma(a)]\!]^{I'}$ for type family constants a.

This is well-defined because every signature morphism maps 0 to 0, and thus reduces models in $Mod^{\mathbb{D}}(\Sigma')$ to models in $Mod^{\mathbb{D}}(\Sigma)$.

Definition 4.43 (Satisfaction)**.** For a signature $\Sigma \in Th^{\mathbb{D}}$, the satisfaction relation between Σ-models I and Σ-sentences, i.e., closed Σ-types S, is defined by

$$I \models^{\mathbb{D}}_\Sigma S \quad \text{iff} \quad Elem([\![\cdot|S]\!]^I) \neq \varnothing.$$

In particular, 0 holds in no model $I \in Mod^{\mathbb{D}}(\Sigma)$.

We define the category $Th^{\mathbb{D}}$ of $\mathbb{D}$-theories in the usual way. The objects are pairs $(\Sigma; \Delta)$ of a signature Σ and a set Δ of sentences.

4.10.3 Proof Theory

Now we define the proof theoretical semantics of $\mathbb{D}$ by defining $Pf^{\mathbb{D}}$ and $val^{\mathbb{D}}$. $Pf^{\mathbb{D}}(\Sigma)$ is the category of contexts and substitutions.

Definition 4.44 (Proof Category)**.** For every signature $\Sigma \in |Sig^{\mathbb{D}}|$, $Pf^{\mathbb{D}}(\Sigma)$ is given as follows.

- Objects: contexts Γ such that $\vdash_\Sigma \Gamma$ `Ctx` and such that all types in Γ are closed.

- Morphisms from Γ to Γ': substitutions γ such that

$$\vdash_\Sigma \gamma : \Gamma' \to \Gamma$$

  where two morphisms are equal if they are equal component-wise.

- Identity and composition are defined as for substitutions.

For a signature morphism σ form Σ to Σ', the functor $Pf^{\mathbb{D}}(\sigma) : Pf^{\mathbb{D}}(\Sigma) \to Pf^{\mathbb{D}}(\Sigma')$ is given by

- on objects: $Pf^{\mathbb{D}}(\sigma)(\Gamma) := \sigma(\Gamma)$,
- on morphisms: $Pf^{\mathbb{D}}(\sigma)(\gamma) := \sigma(\gamma)$.

We write $\Gamma \vdash_\Sigma S$ iff there is a term s such that $\Gamma \vdash_\Sigma s : S$. The morphisms in $Pf^{\mathbb{D}}(\Sigma)$ from Γ to Γ' are substitutions from Γ' to Γ. The point of the reversed direction is to follow the style of Lawvere categories ([Law63]) by using as morphisms from Γ to Γ' tuples of Γ-terms typed according to Γ'. It is easy to show that the empty context is a terminal object in $Pf^{\mathbb{D}}(\Sigma)$. Furthermore, $x{:}0$ is initial. Products in $Pf^{\mathbb{D}}(\Sigma)$ are obtained by merging contexts, in particular, if Γ is finite, Γ, Γ' is a product of Γ and Γ'. We use Γ, Γ' to denote products even if Γ is infinite.

Definition 4.45. For $\Sigma \in |Sig^{\mathbb{D}}|$, we define

$$val^{\mathbb{D}}_\Sigma(S) \quad := \quad x{:}S.$$

4.10.4 Collecting the Pieces

Finally, we can collect the definitions to obtain the logic $\mathbb{D}$.

Theorem 4.46. *$(Sig^{\mathbb{D}}, Sen^{\mathbb{D}}, Mod^{\mathbb{D}}, \models^{\mathbb{D}}, Pf^{\mathbb{D}}, val^{\mathbb{D}})$ is a logic.*

Proof. To show the satisfaction condition, assume $\vdash \sigma : \Sigma \to \Sigma'$, as well as $I' \in Mod^{\mathbb{D}}(\Sigma)$ and $S \in Sen(\Sigma)$. Put $I := Mod(\sigma)(I')$ and $S' := Sen(\sigma)(S)$. Then $[\![\cdot|S]\!]^I = [\![\cdot|S']\!]^{I'}$, and therefore $I \models^{\mathbb{D}}_{\Sigma'} S$ iff $I' \models^{\mathbb{D}}_{\Sigma'} S'$. The naturality of $val^{\mathbb{D}}$ is obvious. □

So far we have used the symbol $\vdash$ in two different ways: for provability in the logic $\mathbb{D}$ and in the typing judgments defining $\mathbb{D}$. We can reconcile these two uses in the following lemma.

Lemma 4.47. *Let (Σ, Δ) be a $\mathbb{D}$-theory and $\vdash_\Sigma \Gamma$ `Ctx` such that $\Gamma = x_1{:}S_1 \ldots$ and $\Delta = \{S_1, \ldots\}$. Then for all $S \in Sen^{\mathbb{D}}(\Sigma)$:*

$$\Delta \vdash^{\mathbb{D}}_\Sigma S \quad \text{iff} \quad \Gamma \vdash_\Sigma s : S \text{ for some } s.$$

Proof. Clear. □

Theorem 4.48 (Strong Soundness)**.** *$\mathbb{D}$ is strongly sound.*

Proof. Assume $\Sigma \in Sig^{\mathbb{D}}$, $\Delta \subseteq Sen^{\mathbb{D}}(\Sigma)$, and $S \in Sen^{\mathbb{D}}(\Sigma)$. Let Γ be a context declaring a variable of type F for every sentence $F \in \Delta$. Assume $\Delta \vdash^{\mathbb{D}}_\Sigma S$. Then there must be a prefix $\Gamma^* = x_1 : S_1, \ldots, x_n : S_n$ and by Lem. 4.47 a term $\Gamma^* \vdash_\Sigma s : S$. Then by Thm. 4.31, there is $e \in Elem([\![\Gamma^*|S]\!]^I)$ for all Σ-models I. And if $I \in Mod^{\mathbb{D}}(\Sigma; \Delta)$, there are $e_i \in Elem([\![\cdot|S_i]\!]^I)$ for all i. Then

$$p \mapsto e_{(p, a_1, \ldots, a_n)} \qquad \text{where } a_i := (e_i)_p$$

is an indexed element of $[\![\cdot|S]\!]^I$. Therefore, $I \models^{\mathbb{D}}_{(\Sigma;\Delta)} S$. □

Theorem 4.49 (Strong Completeness)**.** *$\mathbb{D}$ is strongly complete.*

Proof. Assume $\Sigma \in \mathit{Sig}^{\mathbb{D}}$, $\Delta \subseteq \mathit{Sen}^{\mathbb{D}}(\Sigma)$, and $S \in \mathit{Sen}^{\mathbb{D}}(\Sigma)$. Assume $\Delta \models^{\mathbb{D}}_{\Sigma} S$. Then $I \models^{\mathbb{D}}_{(\Sigma;\Delta)} S$ for all $I \in \mathit{Mod}^{\mathbb{D}}(\Sigma;\Delta)$. Let Γ be the context declaring a variable of type F for every sentence $F \in \Delta$, and let I be the Σ-model constructed for Γ in Thm. 4.35. There are two cases:

- $\mathit{Elem}([\![\Gamma|0]\!]^I) = \varnothing$. Then $I \in \mathit{Mod}^{\mathbb{D}}(\Sigma)$. And clearly also $I \in \mathit{Mod}^{\mathbb{D}}(\Sigma;\Delta)$. Using the assumption and Thm. 4.35, there is a term $\Gamma \vdash_\Sigma s : S$.
- $\mathit{Elem}([\![\Gamma|0]\!]^I) \neq \varnothing$. Using Thm. 4.35, there is a term $\Gamma \vdash_\Sigma s : 0$. Then $\Gamma \vdash_\Sigma !!_S\, s : S$.

In both cases, the result follows using Lem. 4.47. □

The soundness and completeness of $\mathbb{D}$ can also be formulated in a way that is more balanced between proof and model theory. From the proof theoretical point of view, the Σ-contexts are the Σ-theories. Then the Grothendieck integral construction applied to the functor $\mathit{Pf}^{\mathbb{D}}$ yields the proof theoretical notion of the category of theories: An object of $\int_{\mathit{Sig}^{\mathbb{D}}} \mathit{Pf}^{\mathbb{D}}$ is a pair $(\Sigma;\Gamma)$ such that $\vdash_\Sigma \Gamma\ \mathtt{Ctx}$; and a morphism from $(\Sigma;\Gamma)$ to $(\Sigma';\Gamma')$ is a pair $(\sigma;\gamma)$ such that

$$\vdash_{\Sigma'} \gamma : \Gamma' \to \sigma(\Gamma).$$

Then soundness can be stated as a functor from $\int_{\mathit{Sig}^{\mathbb{D}}} \mathit{Pf}^{\mathbb{D}}$ to $\mathit{Th}^{\mathbb{D}}$. This functor maps $(\Sigma, x_1 : S_1, \ldots)$ to $(\Sigma, \{S_1, \ldots\})$ and (σ,γ) to σ. Stating completeness as a functor in the opposite direction is not as elegant: For a model theoretical theory morphism σ, a proof theoretical theory morphism (σ,γ) exists, but not uniquely. But if we identify all terms of the same type, then γ is uniquely determined, and we obtain a pair of adjoint functors between the model and the proof theoretical category of theories.

4.10.5 Subsystems

We can recover several systems as subsystems of $\mathbb{D}$. We will do that by removing certain signatures and signature morphisms from $\mathit{Sig}^{\mathbb{D}}$, and certain contexts and substitutions from $\mathit{Pf}^{\mathbb{D}}(\Sigma)$. We call these four concepts the vocabulary concepts. We may also remove some types from $\mathit{Sen}^{\mathbb{D}}(\Sigma)$. The definitions for $\models^{\mathbb{D}}$, $\models^{\mathbb{D}}$, and $\mathit{val}^{\mathbb{D}}$ can be restricted accordingly. Note that we only change the logic, not the underlying type theory. In particular, even if identity types are removed from the signatures and contexts, they can still be used to reason about the equality of terms.

Example 4.50. A logic for Martin-Löf Type Theory arises if all instances of vocabulary concepts that rely on T_λ or T_0 are removed, and all types that rely on T_0 are removed from $\mathit{Sen}(\Sigma)$. In particular, all type families are constants or types. This yields the same well-formed instances of the vocabulary concepts as the system used in [See84].

Example 4.51. A logic for LF arises if all instances of vocabulary concepts that rely on T_Σ, $T_{\mathit{Id}(-,-)}$, T_1, or T_0 are removed. Furthermore, the corresponding types are removed from $\mathit{Sen}^{\mathbb{D}}(\Sigma)$. This yields the same well-formed instances of the vocabulary concepts as the system used in [See84].

4.11 Conclusion

We have presented a concrete and intuitive semantics for MLTT in terms of indexed sets on posets. And we have shown soundness and completeness. Our semantics is essentially that proposed by Lawvere in [Law69] in the hyperdoctrine of posets, fibrations, and indexed sets on posets, but we have made particular choices for which the models are coherent. Our models use standard function spaces, and substitution has a very simple interpretation as composition. The same holds in the simply-typed case, which makes our models an interesting alternative to

Henkin models. In both cases, we strengthen the existing completeness results by restricting the class of models. Finally, we showed how this model theory leads to a logic for MLTT.

We assume that the completeness result can still be strengthened somewhat, e.g., to permit equality axioms between types. In addition, it is an open problem to find an elementary completeness proof, i.e., one that does not rely on topos-theoretical results.

Acknowledgements Steve Awodey suggested the research presented here. The main ideas, in particular for the completeness result, are due to him. And while the specific definitions and the remaining results are ours, the whole section benefited strongly from discussions with and concrete advice by him. With the exception of Sect. 4.10, this section is the core of [AR08].

Chapter 5

Dependent Type Theory as a Meta-Logic

In this section, we define a logic $\mathbb{L}$ such that logic comorphisms into $\mathbb{L}$ capture the intuition of logic encodings in LF. Building on $\mathbb{D}$, we give the definition of the logic in Sect. 5.2. Then we show how logics and logic translations are defined or encoded in $\mathbb{L}$ in Sect. 5.3 and 5.5, respectively. Both are followed by example encodings in Sect. 5.4 and 5.6, respectively. In Sect. 5.7 and 5.8, we summarize and discuss some directions for future research.

5.1 Introduction

No encodings of institutions, i.e, model theories, in LF have been studied before. But various encodings that do not consider the model theory already exist. Examples for syntax encodings of various fragments of first-order logic are given in [HHP93], [HST94], [AHMP92], [Pfe01], and [Pfe00]. In [AHMP98], several possibilities are given to encode modal logics, which is difficult because of rules with more complex side conditions and the lack of straightforward natural deduction. For higher-order logics, encodings are given in [HHP93] and [SS04]. The encodings of propositional categorical logics in [GMdP+07] use the logic DFOL ([Rab06]) as a meta-logic, which is itself encoded in LF.

We use a logic $\mathbb{L}$ for LF that is an extension of the subsystem of $\mathbb{D}$ given in Ex. 4.51. But since one of the strengths of LF is its simplicity, we will take special care to add only a few simple concepts and to make all additions as LF-compatible as feasible. To motivate the definition of $\mathbb{L}$, we look at an example LF-signature Σ encoding a fragment of FOL given in Fig. 5.1. There are two important structural properties of the encoding that are not explicit in Σ.

$$
\begin{array}{l}
i : \texttt{type} \\
o : \texttt{type} \\
\Rightarrow : o \to o \to o \\
\forall : (i \to o) \to o \\
true : o \to \texttt{type} \\
\Rightarrow I : \Pi_{F:o}\, \Pi_{G:o}\, \big((true\ F \to true\ G) \to true\ (\Rightarrow\ F\ G)\big) \\
\Rightarrow E : \Pi_{F:o}\, \Pi_{G:o}\, \big(true\ (\Rightarrow\ F\ G) \to true\ F \to true\ G\big) \\
\forall I : \Pi_{F:o}\, \big((\Pi_{x:i}\, true\ (Fx)) \to true\ (\forall\ F)\big) \\
\forall E : \Pi_{F:o}\, \big(true\ (\forall\ F) \to \Pi_{x:i}\, true\ (Fx)\big)
\end{array}
$$

Figure 5.1: Encoding of First-order Logic

Firstly, Σ has two parts: The first part declares symbols for the sorts (i), constants (not shown in Fig. 5.1; e.g., $comp : i \to i \to i$ for monoids), the type of formulas (o), connectives ($\Rightarrow$) and binders ($\forall$) of the encoded logic. These encode the domain of discourse, the syntax of FOL. When giving a model theory for FOL, these symbols correspond to specific semantic objects. In the simplest case, the sorts and constants correspond to sets and elements, respectively. The type of formulas corresponds to the set of truth values, and the connectives and binders to operation on the truth values. An assignment of semantic objects to the symbols is called a model, a structure, or an interpretation. We call this part the **object level** of Σ. The second part declares symbols for judgments ($true$) and proof rules ($\Rightarrow I$, $\Rightarrow E$, $\forall I$, $\forall E$). These encode an inference system that reasons about the object level. We call this part the **judgment level**. Here and in most other cases, the judgment level refers to the object level, but not vice versa.

Secondly, Σ has two distinguished symbols: o and $true$. o is the object level type of propositions. $true$ is the truth judgment: For a formula f, the type $true\ \ f$ encodes the judgment "f is true.". And giving evidence or proof for this judgment means to give a term of type $true\ f$. Thus, terms encode proofs. Semantically, the interpretation of o can be viewed as the set of truth values. And the interpretation of $true$ can be viewed as singling out some truth values, the designated ones. These two symbols are special in that they encode the essence of logic: propositions ans truth.

These two structural properties set apart logic encodings from other uses of LF. Currently this structure is marked up implicitly using natural language in comments or even by the choice of symbol names. But this is insufficient for automated processing. Therefore, we will introduce additional concepts to attach to a signature its structural semantics.

Furthermore, while Σ adequately encodes the syntax and proof theory of FOL, it falls short of describing the model theory. This is a principal characteristic of LF: Some aspects of the semantics cannot be described in it because it does not provide negation. For example, it is not possible to exclude inconsistent models in which all formulas are true. Therefore, it is desirable to use $\mathbb{D}$-theories (instead of just $\mathbb{D}$-signatures or just LF-signatures) to axiomatize the model theory.

Intuitively, we arrive at the following:

- The objects L of $Sig^{\mathbb{L}}$ are tuples of $\mathbb{D}$-theories $(\Sigma; \Delta)$, an assignment of roles to the symbols in Σ, and distinguished Σ-expressions $\underline{o}$ and $\underline{true}$ where $\underline{true} : \underline{o} \to \texttt{type}$.
- $Sen^{\mathbb{L}}(L)$ is as $Sen^{\mathbb{D}}(\Sigma)$ but restricted to the image of the distinguished type family $\underline{true}$. Similarly, we could say that $Sen^{\mathbb{L}}(L)$ contains the terms of the distinguished type $\underline{o}$.
- The proof category $Pf^{\mathbb{L}}(L)$ is $Pf^{\mathbb{D}}(\Sigma)$ but restricted to the judgment level types and terms.
- The models of L are the models in $Mod^{\mathbb{D}}(\Sigma; \Delta)$, but restricted to the simple models and using proof irrelevance when interpreting the judgment level.
- The truth judgment and the satisfaction relation are as for $\mathbb{D}$.

5.2 A Meta-Logic for LF

Now we define the logic $\mathbb{L} = (Sig^{\mathbb{L}}, Sen^{\mathbb{L}}, Mod^{\mathbb{L}}, \models^{\mathbb{L}}, Pf^{\mathbb{L}}, val^{\mathbb{L}})$. In the following we will define the components of $\mathbb{L}$ separately and finally prove that it is a logic. We will use an encoding of FOL as our running example.

5.2.1 Signatures

Definition 5.1 (Role Assignments)**.** A **role assignment** for a $\mathbb{D}$-signature Σ is a map R that assigns to every type family symbol of Σ one of *sort* and *judgment*. A role assignment is extended

to all type families as follows:

$$\begin{array}{ll} R(a\ \gamma_0) & := R(a) \\ R(\lambda_{x:S}\, A) & := R(A) \\ R(Id(s,s')) & := \mathit{judgment} \\ R(1) & := \mathit{judgment} \\ R(0) & := \mathit{judgment} \\ R(\Pi_{x:S}\, T) & := R(T) \\ R(\Sigma_{x:S}\, T) & := \begin{cases} \mathit{sort} & \text{if } R(S) = \mathit{sort} \text{ or } R(T) = \mathit{sort} \\ \mathit{judgment} & \text{otherwise} \end{cases} \end{array}$$

And for a Σ-term s of type S, we extend R by

$$R(s) := \begin{cases} \mathit{object} & \text{if } R(S) = \mathit{sort} \\ \mathit{proof} & \text{if } R(S) = \mathit{judgment} \end{cases}$$

The intuition is that objects and sorts form the object level, and proofs and judgments the judgment level. Here, the case of sum types is interesting: Only when both S and T are judgments, do we regard $\Sigma_{x:S}\, T$ as a judgment. In particular, if S is a sort and T a judgment, $\Sigma_{x:S}\, T$, which is Curry-Howard equivalent to an existential quantification over S, is a sort — intuitively, it is the subsort of S containing those elements for which T holds. Existential quantification is still possible: For all types S, a judgment expressing the inhabitation of S is given by double-negating S.

Thus, we distinguish four roles for expressions. Furthermore, we distinguish atomic and composed expressions. Then we have eight syntactic concepts.

Notation 5.2 (Meta-Variables). We use the following meta-variables for atomic (upper entry in each field) and composed (lower entry in each field) expressions:

	Terms	Types and type families
Object level	constant c	sort symbol a
	object s	sort S, sort family A
Judgment level	rule r	judgment symbol j
	proof p	judgment J, judgment family $-$

Then we are ready to define $\mathbb{L}$-signatures.

Definition 5.3 (Signatures)**.** $(\Sigma; \Delta; R; \underline{o}; \underline{\mathit{true}})$ is an element of $|\mathit{Sig}^{\mathbb{L}}|$ iff it satisfies the following conditions:

1. $(\Sigma; \Delta) \in \mathit{Th}^{\mathbb{D}}$ where Σ is an LF-signature as defined in Ex. 4.51.
2. R is a role assignment for Σ.
3. For all constants $c{:}S$, all rules $r{:}J$, and all sort or judgment symbols of kind Γ_0 in Σ, we have that S, J, and Γ_0 are produced by the following grammar

$$\begin{array}{ll} \Gamma_0 & ::= \cdot \mid \Gamma_0, x{:}S \\ S & ::= a\ \gamma_0 \mid J \to S \mid \Pi_{x:S}\, S \\ J & ::= j\ \gamma_0 \mid J \to J \mid \Pi_{x:S}\, J \\ a & ::= \text{sort symbol} \\ j & ::= \text{judgment symbol} \\ \gamma_0 & ::= \text{substitution} \end{array}$$

4. $\underline{o}$ and $\underline{\mathit{true}}$ are such that

$$\cdot \vdash_\Sigma \underline{o} : \mathtt{type}$$

is a sort and

$$\cdot \vdash_\Sigma \underline{\mathit{true}} : \underline{o} \to \mathtt{type}$$

is a judgment family.

The restriction of Σ to LF-signatures reflects our interest in logic encodings in LF; for the model theory encoded by Δ, no restriction is used. Furthermore, $\mathbb{D}$ expressions are allowed in $\underline{o}$ and $\underline{\mathit{true}}$.

Example 5.4. The signature from Fig. 5.1 is extended to an $\mathbb{L}$-signature as follows. Δ contains the axiom i, which states the non-emptiness of the universe. A role assignment is given by making i and o sorts and true a judgment. Furthermore, $\underline{o} := o$ and $\underline{\mathit{true}} := \mathit{true}$.

The intuition behind the grammar from Def. 5.3 is fortified by the following terminology.

Terminology 5.5 (Sorts and Judgments). We call

- sorts $J \to S$: **guarded sorts** with guard J and sort S,
- sorts $\Pi_{x:S} S'$: dependent **function sorts** with argument sort S and return sort S',
- judgments $J \to J'$: **hypothetical judgments** with hypothesis J and conclusion J',
- judgments $\Pi_{x:S} J$: **parametric judgments** taking a parameter of type S.

Furthermore, we call types of the form $\Sigma_{x:S} J$ **subsorts** of S with defining predicate J.

Example 5.6. In Fig. 5.1, $o \to o \to o$ is a function sort with two non-dependent arguments. An example for a guarded sort is $(\mathit{true}\ F) \to i$; an object of this sort would be a function that returns a new individual if provided with a proof of F. An example for a subsort is $\Sigma_{x:i} (\mathit{true}\ (F\ x))$, which encodes the sort of individuals x for which F is true. The type of $\Rightarrow I$ is a hypothetical judgment with a hypothesis which is itself a hypothetical judgment. The type of $\forall I$ has a hypothesis which is a judgment parametric in i.

The point of the grammar in Item 3 of Def. 5.3 is to restrict the interdependence of object and judgment level. Let us look at a special case first.

Definition 5.7. $(\Sigma; \Delta; R; \underline{o}; \underline{\mathit{true}}) \in |\mathit{Sig}^{\mathbb{L}}|$ is called **separated** if Σ does not use the production $S ::= J \to S$.

If $(\Sigma; \Delta; R; \underline{o}; \underline{\mathit{true}})$ is separated, the object level is independent of the judgment level in the following sense: Dropping all judgment and rule declarations from Σ yields a well-formed signature Σ^o; and if $\Gamma \vdash_{\Sigma^o} S : \mathtt{type}$ the S-terms over Σ and Σ^o are the same, i.e., the judgment level does not add new object level terms. In the terminology of [Vir96], a separated signature conforms to a dependency relation where proofs may not occur in judgment families, objects, or sort families.

While separation is a natural property, it is too strong for logic encodings in general. The most important example where this is insufficient are logics where the well-formedness of propositions is itself axiomatized in a way that is mutually recursive with the axiomatization of truth. For example, $\mathbb{D}$ is in this class: The well-formedness of the type $\mathit{Id}(s, s')$ depends on the axiomatization of equality between types and thus equality between terms. For such logics, there are two possible encodings. One way is to use $\underline{o} := \Sigma_{x:o} (\mathit{wff}\ x)$ where wff is the judgment for well-formed formulas. Intuitively, the propositions are a subsort of some type o. This is used in Ex. 5.31 below. Another way is to sacrifice separation and use constants that take proofs as arguments. This is used in Ex. 5.32 to encode a description operator, which takes a proof of unique existence as an argument.

Notation 5.8 (Meta-Variables). We will use L and L' as meta-variables for objects of $\mathit{Sig}^{\mathbb{L}}$, and we will always assume that they are given as $L = (\Sigma; \Delta; R; \underline{o}; \underline{\mathit{true}})$ and $L' = (\Sigma'; \Delta'; R'; \underline{o}'; \underline{\mathit{true}}')$, respectively.

5.2.2 Signature Morphisms

Definition 5.9 (Signature Morphisms)**.** A $Sig^{\mathbb{L}}$-morphism from L to L' is a $\mathbb{D}$-theory morphism from $(\Sigma;\Delta)$ to $(\Sigma';\Delta')$ such that

1. σ respects the role assignment: Constants, sort symbols, rules, and judgment symbols are mapped to objects, sort families, proofs, and judgment families, respectively.
2. σ respects $\underline{o}$ and $\underline{true}$ in the following sense: There is a term $\cdot \vdash_{\Sigma'} k : \sigma(\underline{o}) \to \underline{o}'$, such that

$$\cdot \vdash_{\Sigma'} \sigma(\underline{true}) \equiv \lambda_{x:\sigma(\underline{o})}\, (\underline{true}'\ (k\ x)).$$

Identity and composition in $Sig^{\mathbb{L}}$ are inherited from $Th^{\mathbb{D}}$. That makes $Sig^{\mathbb{L}}$ a category.

It would be natural to require a $Sig^{\mathbb{L}}$-morphism to satisfy the following stronger conditions regarding $\underline{o}$ and $\underline{true}$:

$$\cdot \vdash_{\Sigma'} \sigma(\underline{o}) \equiv \underline{o}' \qquad \text{and} \qquad \cdot \vdash_{\Sigma'} \sigma(\underline{true}) \equiv \underline{true}'.$$

Inter-theory translations typically satisfy these conditions. But they are too strong for inter-logic translations. In the latter case, it is often useful to permit $\sigma(\underline{o}) \not\equiv \underline{o}'$ and use a fixed term k to bridge the gap between $\sigma(\underline{o})$ and $\underline{o}'$. Then $\sigma(\underline{true})$ is determined by $\sigma(\underline{o})$ and k. Of course, k might simply be the identity. An example is given in Ex. 5.37 in Sect. 5.6.

Example 5.10. Based on the signature in Fig. 5.1, the FOL-signatures for monoids and groups and a signature morphism between them are obtained naturally. This was already described in Sect. 2.2.2.5. Indeed, this signature morphism maps $\underline{o}$ to $\underline{o}'$ and $\underline{true}$ to $\underline{true}'$.

5.2.3 Sentences

The object part of the sentence functor is easy: The sentences are the terms f of type $\underline{o}$ modulo equality. However, we will the types $\underline{true}\ f$ instead because that permits slightly more general translations.

Definition 5.11 (Sentences)**.** The functor $Sen^{\mathbb{L}} : Sig^{\mathbb{L}} \to \mathcal{SET}$ is defined by

$$Sen^{\mathbb{L}}(L) := \{\underline{true}\ f \mid \cdot \vdash_{\Sigma} f : \underline{o}\} \qquad \text{modulo} \qquad \cdot \vdash_{\Sigma} \underline{true}\ f \equiv \underline{true}\ f'$$

$$Sen^{\mathbb{L}}(\sigma)(\underline{true}\ f) := \sigma(\underline{true}\ f)$$

5.2.4 Proof Theory

The proof theory of $\mathbb{L}$ is like the one of $\mathbb{D}$ except that only judgments are used.

Definition 5.12 (Proof Categories)**.** $Pf^{\mathbb{L}}(L)$ is the full subcategory of $Pf^{\mathbb{D}}(\Sigma)$ induced by those contexts that declare only variables of judgment types. For a signature morphism $\sigma : L \to L'$, the proof translation functor $Pf^{\mathbb{L}}(\sigma)$ is the appropriate restriction of $Pf^{\mathbb{D}}(\sigma)$. The existence and preservation of products are clear.

Definition 5.13 (Truth)**.** $val^{\mathbb{L}}_L(F) := val^{\mathbb{D}}_{\Sigma}(F)$.

Example 5.14. Under the encoding of FOL in $\mathbb{L}$, a proof from a set of formulas $f_1, \ldots$ to a set of formulas $f'_1, \ldots$ is a substitution from $x_1 : true\ f'_1, \ldots$ to $x_1 : true\ f_1, \ldots$, i.e., a family $p_1, \ldots$ of terms such that

$$x_1 : true\ f'_1, \ldots \vdash_{\Sigma} p_i : true\ f_i \qquad \text{for } i = 1, \ldots.$$

And a sentence $true\ f$ is valid if there is a closed term of type $true\ f$.

5.2.5 Model Theory

The set Δ of an $\mathbb{L}$-signature is not used in the definition of sentences or proof theory. It is only used to limit the available models. Furthermore, we add axioms to Δ that enforce proof irrelevance, i.e., models must interpret all proofs of the same judgments in the same way.

Definition 5.15 (Models)**.** $Mod^{\mathbb{L}}$ is given as follows. $Mod^{\mathbb{L}}(L)$ is the full subcategory of $Mod^{\mathbb{D}}(\Sigma;\Delta,\Delta^+)$ induced by the simple models; here Δ^+ contains an axiom

$$\Pi_{x_1:S_1}\ \ldots\Pi_{x_n:S_n}\,\Pi_{x:J}\,\Pi_{x':J}\,Id(x,x') \qquad \text{where } J := j\ id_{\Gamma_0}$$

for every judgment symbol $j:(\Gamma_0)\texttt{type}$ in Σ. $Mod^{\mathbb{L}}(\sigma)$ is the appropriate restriction of $Mod^{\mathbb{D}}(\sigma)$.

Definition 5.16 (Satisfaction)**.** For an $\mathbb{L}$-signature L, a judgment J in context Γ, and an L-model I, we define:

$$I \models^{\mathbb{L}}_L \Gamma|J \qquad \text{iff} \qquad Elem([\![\Gamma|J]\!]^I) \neq \varnothing.$$

In particular, this defines the satisfaction of $\mathbb{L}$-sentences.

Notation 5.17. Since $\mathbb{L}$-models are simple, we will always omit the unique p when writing $(p,\alpha)\in[\![\Gamma]\!]$.

Example 5.18. For FOL, models are interpretations of the signature from Fig. 5.1. The interpretation of i is the universe, $[\![o]\!]^I$ is the set of truth values. It can be reasonable to use further axioms that make $[\![o]\!]^I$ a set with two elements. But that is not necessary because $[\![true]\!]^I$ already singles out a subset of $[\![o]\!]^I$. Model-theoretically, this set can be regarded as the set of designated truth values. It might seem necessary to add axioms that axiomatize the meaning of $\Rightarrow$ and $\forall$. But these axioms are already present in form of the rules. Thus, we obtain a model theoretical interpretation of the proof rules. For the satisfaction of a FOL-formula f, we obtain $I \models^{\mathbb{L}}_L \cdot|\underline{true}\ f$ iff $[\![\cdot|true\ \ f]\!]^I \neq \varnothing$, i.e., f is true iff $[\![\cdot|f]\!]^I$ is in the set of designated truth values.

5.2.6 Collecting the Pieces

Then we conclude the definition of $\mathbb{L}$ with the following result.

Theorem 5.19. *$\mathbb{L}$ is a strongly sound logic.*

Proof. This follows from the corresponding properties of $\mathbb{D}$. □

$\mathbb{L}$ cannot be complete because we restricted attention to simple models. For our current motivation, the gained simplicity of the model theory outweighs the lack of completeness. We will come back to that in Sect. 5.7.2.

Finally, we give a simple result that formalizes the intuition behind composed sorts and judgments.

Theorem 5.20. *For an $\mathbb{L}$-signature L, a context Γ, an L-model I, and $\alpha\in[\![\Gamma]\!]^I$ we have*

1. *Judgments are interpreted as booleans, and proofs are irrelevant, i.e., for all judgments J: $[\![\Gamma|J]\!]^I(\alpha)$ is a singleton or empty.*
2. *Hypothetical judgments correspond to implication:*
$$I_\alpha \models^{\mathbb{L}}_L \Gamma|J\to J' \qquad \text{iff} \qquad I_\alpha \not\models^{\mathbb{L}}_L \Gamma|J \ \text{ or } \ I_\alpha \models^{\mathbb{L}}_L \Gamma|J'.$$
3. *Parametric judgments correspond to universal quantification:*
$$I_\alpha \models^{\mathbb{L}}_L \Gamma|\Pi_{x:S}\,J \qquad \text{iff} \qquad I_{(\alpha,a)} \models^{\mathbb{L}}_L \Gamma,x:S|J \ \text{ for all } a\in[\![\Gamma|S]\!]^I(\alpha).$$

4. Subsorts correspond to subsets:

$$[\![\Gamma|\Sigma_{x:S}\, J]\!]^I(\alpha) \cong \{a \in [\![\Gamma|S]\!]^I(\alpha) \mid I_{(\alpha,a)} \models^{\mathbb{L}}_L \Gamma, x:S|J\}.$$

Proof. The first claims follow by a straightforward induction over the judgments. As an example, we give the induction step for hypothetical judgments. From the induction hypothesis of Claim 1, we know that $[\![\Gamma|J]\!]^I(\alpha)$ and $[\![\Gamma|J']\!]^I(\alpha)$ are a singleton or empty. Therefore, so is $[\![\Gamma|J \to J']\!]^I(\alpha)$, and it is empty iff the former is non-empty and the latter is. For parametric judgments, it is shown similarly that there can be at most one element in $[\![\Gamma|\Pi_{x:S}\, J]\!]^I$, and such an element exists iff all $[\![\Gamma, x:S|J]\!]^I$ are non-empty.

Claim 4: Because of Claim 1, if $a \in [\![\Gamma|S]\!]^I(\alpha)$, then $[\![\Gamma|\Sigma_{x:S}\, J]\!]^I(\alpha)$ contains at most one pair (a, b). And it does so iff $[\![\Gamma, x:S|J]\!]^I(\alpha, a)$ is non-empty. □

5.3 Defining and Encoding Logics

$\mathbb{L}$ can be used both to encode logics and to define them. A logic definition consists of a signature category *Sig* and a functor $\Phi : Sig \to Sig^{\mathbb{L}}$.

Theorem 5.21 (Defining Logics)**.** *For every functor* $\Phi : Sig \to Sig^{\mathbb{L}}$*,*

$$\mathbb{L} \circ \Phi := (Sig,\ Sen^{\mathbb{L}} \circ \Phi,\ Mod^{\mathbb{L}} \circ \Phi,\ \models^{\mathbb{L}}_{\Phi},\ Pf^{\mathbb{L}} \circ \Phi,\ val^{\mathbb{L}}_{\Phi})$$

is a strongly sound logic.

Recall Not. 2.19 regarding the composition of Φ and the natural transformations $\models$ and *val*.

Proof. This follows immediately from Thm. 5.19 and basic properties of functors and natural transformations. □

Thus, a logic definition starts with a signature category and borrows all other notions from $\mathbb{L}$. A logic encoding starts with a whole logic $\mathbb{I} = (Sig, Sen, Mod, \models, Pf, val)$ and a functor $\Phi : Sig \to Sig^{\mathbb{L}}$. That yields two logics $\mathbb{I}$ and $\mathbb{L} \circ \Phi$. For the encoding to be useful, these have to be related in some way. This is formalized by logic comorphisms.

Definition 5.22 (Encoding Logics)**.** An **encoding** of a logic $\mathbb{I}$ is a logic comorphism $(\Phi, \alpha, \beta, \gamma) : \mathbb{I} \to \mathbb{L}$.

While a logic comorphism expresses a certain structural similarity between $\mathbb{I}$ and $\mathbb{L} \circ \Phi$, it is not enough to make the consequence relations equivalent. For that we need the following.

Definition 5.23 (Adequacy)**.** A logic encoding $(\Phi, \alpha, \beta, \gamma)$ is called **model-theoretically adequate** if it admits models expansion, i.e., each β_Σ is surjective on objects. It is called **proof-theoretically adequate** iff each γ_Σ is full, i.e., surjective on morphisms.

Proof theoretical adequacy is simply called adequacy in a proof theoretical setting. Actually, to obtain Thm. 5.24, the fullness of γ_Σ can be relaxed: It is sufficient if $\pi_\Sigma \circ \gamma_\Sigma$ is full where π_Σ is the projection that quotients $Pf(\Sigma)$ into a poset.

Theorem 5.24. *For a model- and proof-theoretically adequate logic encoding* $(\Phi, \alpha, \beta, \gamma)$ *of* $\mathbb{I}$*, an* $\mathbb{I}$*-signature* Σ*, and sets of sentences* $A, B \subseteq Sen^{\mathbb{I}}(\Sigma)$*:*

$$A \models^{\mathbb{I}}_{\Sigma} B \quad \text{iff} \quad \alpha_\Sigma(A) \models^{\mathbb{L}}_{\Phi(\Sigma)} \alpha_\Sigma(B)$$
$$A \vdash^{\mathbb{I}}_{\Sigma} B \quad \text{iff} \quad \alpha_\Sigma(A) \vdash^{\mathbb{L}}_{\Phi(\Sigma)} \alpha_\Sigma(B)$$

Proof. The two results are independent and only require the respective adequacy assumption. The model theoretical result is well-known and proved similarly to Thm. 3.14. The proof theoretical result is easy to prove considering that γ_Σ is a functor $Pf^{\mathbb{I}}(\Sigma) \to Pf^{\mathbb{L}}(\Phi(\Sigma))$ and that morphisms in the proof categories are used to define provability. Note that in both cases, the left-to-right implication holds even without adequacy. □

Besides defining and encoding logics, there are intermediate cases: A logic can be partially encoded and partially defined. For example, an institution can be encoded by an institution comorphism, which can then be used to borrow the proof categories of $\mathbb{L}$.

Theorem 5.25. *If an institution comorphism (Φ, α, β) into $LtoI(\mathbb{L})$ admits model expansion, then $(Sig^{\mathbb{I}}, Sen^{\mathbb{I}}, Mod^{\mathbb{I}}, \models^{\mathbb{I}}, Pf^{\mathbb{L}} \circ \Phi, val^{\mathbb{L}}_{\Phi} \circ \alpha)$ is a strongly sound logic.*

Proof. This is a special case of Thm. 3.14. □

This case is particularly interesting because it permits to use LF as a specification language for proof categories of a given institution.

5.4 Examples

Notation 5.26 (Twelf Syntax). When giving the examples, we will use the Twelf ASCII syntax ([PS99]) instead of the paper syntax for LF. It is defined as follows:

$s{:}S$	$\rightsquigarrow$	`s: S`
$A{:}(\ldots, x_i{:}S_i, \ldots)$`type`	$\rightsquigarrow$	`A: ... {xi: Si} ... type`
`type`	$\rightsquigarrow$	`type`
$A\ \ldots, x_i/s_i, \ldots$	$\rightsquigarrow$	`(A ... s1 ... )`
$\lambda_{x:S}\, A$	$\rightsquigarrow$	`[x: S] A`
$\Pi_{x:S}\, T$	$\rightsquigarrow$	`{x: S} T`
$S \to S'$	$\rightsquigarrow$	`S -> S'`
$\lambda_{x:S}\, t$	$\rightsquigarrow$	`[x: S] t`
$f\ s$	$\rightsquigarrow$	`f s`

Furthermore, `%infix a n c` declares infix notation for the symbol c with binding strength n and associativity a. Similarly, prefix notation is declared. Finally, the examples use implicit arguments: If an unbound upper case variable X of type S' occurs in S, then $c{:}S$ abbreviates $c{:}\Pi_{X:S'}\, S$. When c is applied, the values of the implicit arguments are not provided to c; rather, they are inferred from the context. Similarly, types of bound variables may be omitted because they can be reconstructed.

Example 5.27 (FOL). We give an encoding $(\Phi, \alpha, \beta, \gamma)$ of FOL in $\mathbb{L}$.

Φ maps a FOL-signature Z to the $\mathbb{L}$-signature $(\Sigma; \Delta; R; \underline{o}; \underline{true})$. Σ extends the signature from Fig. 5.2 with constants $f : i \to \ldots \to i \to i$ and $p : i \to \ldots \to i \to o$ for the function and predicate symbols in Z. This part is well-known (see, e.g., [Pfe00]). Δ contains the axioms i and $\neg(true\ ff)$. The former states the non-emptiness of the universe, and the latter states that there is no proof of ff, i.e., consistency. This also implies that $[\![o]\!]$ is not a singleton. R, $\underline{o}$, and $\underline{true}$ are as in Ex. 5.6. α_Z is the obvious bijection. Furthermore, α_Z extends to an encoding of formulas with free variables by mapping all variables to themselves.

To define β_Z, assume an $\mathbb{L}$-model I'; we have to define a FOL-model $I := \beta_Z(I')$. I consists of a universe U^I and interpretations s^I for the function and predicate symbols in Z. For the universe, we put $U^I := [\![i]\!]^{I'}$. For a function symbol f, we put $f^I := [\![f]\!]^{I'}$. For a predicate symbol p taking n arguments, we put $(u_1, \ldots, u_n) \in p^I$ iff $[\![true]\!]^{I'}([\![p]\!]^{I'}(u_1, \ldots, u_n)) \neq \varnothing$.

To show the satisfaction condition, assume a Z-formula F with free variables $x_1, \ldots, x_n$, and a $\Phi(Z)$-model I'. Let $I := \beta_Z(I')$, and $u_1, \ldots, u_n \in U^I$. The satisfaction condition is a special case of

$$I_{(u_1,\ldots,u_n)} \models_Z^{\mathcal{FOL}} F \qquad \text{iff} \qquad {I'}_{(u_1,\ldots,u_n)} \models^{\mathbb{L}}_{\Phi(Z)} x_1 : i, \ldots, x_n : i \mid \mathit{true}\ \alpha_\Sigma(F).$$

This is shown by induction on F. The rules in Σ are used to establish the various cases. This is easy to see using the intuition given by Thm. 5.20. The case for $F = \mathit{ff}$ requires the axiom $\neg(\mathit{true}\ \mathit{ff})$, which implies $[\![\mathit{true}\ \mathit{ff}]\!]^{I'} = \varnothing$.

To show the model expansion property, assume a Z-model I. I induces to a $\Phi(Z)$-model I' by putting $[\![i]\!]^{I'} := U^I$ for the universe, $[\![o]\!]^{I'} = \{\varnothing, \{\varnothing\}\}$ for the truth values, and $[\![s]\!]^{I'} := s^I$ for a function or predicate symbol s. Furthermore, we interpret all connectives and quantifiers appropriately and put $[\![\mathit{true}]\!]^{I'} := \mathit{id}$ Then it is straightforward to check that all rules can be interpreted and that $\beta_Z(I') = I$.

γ_Σ maps from $Pf^{\mathcal{FOL}}(\Sigma)$ as defined in Ex. 3.3 to $Pf^{\mathbb{L}}(\Phi(\Sigma))$. γ_Σ maps a family $(F_i)_{i\in I}$ to the context declaring $x_i : \mathit{true}\ \alpha_\Sigma(F_i)$ for all $i \in I$. And γ_Σ maps a family $(p_j)_{j\in J}$ to the substitution containing $x_j/\overline{p_j}$ for every $j \in J$. Here $\overline{p_j}$ is the Curry-Howard representation of the proof p_j as an LF-term. Proof theoretical adequacy is easy to show.

```
% object level
i : type.
o : type.
tt : o.
ff : o.
and : o -> o -> o.          %infix left 10 and.
or : o -> o -> o.           %infix left 10 or.
imp : o -> o -> o.          %infix left 10 imp.
not : o -> o.               %prefix 15 not.
forall : (i -> o) -> o.
exists : (i -> o) -> o.
eq : i -> i -> o.           %infix none 20 eq.

% judgment level
true : o -> type.           %prefix 0 true.

ttI : true top.
ffE : true bot -> true A.

andI : true A -> true B -> true A and B.
andEl : true A and B -> true A.
andEr : true A and B -> true B.

orIl : true A -> true A or B.
orIr : true B -> true A or B.
orE : (true A -> true C) -> (true B -> true C) -> (true A or B -> true C).

impI : (true A -> true B) -> true A imp B.
impE : true A imp B -> true A -> true B.

notI : (true A -> true bot) -> true not A.

forallI : ({x} true A x) -> true forall A.
forallE : true forall A -> {x} true A x.

existsI : {x} true A x -> true exists A.
existsE : {x} (true A x -> true C) -> (true exists A -> true C).

refl : true X eq X.
sym : true X eq Y -> true Y eq X.
trans : true X eq Y -> true Y eq Z -> true X eq Z.
cong : true X eq Y -> true (F X) eq (F Y).

tnd : true A or not A.
```

Figure 5.2: Encoding of FOL

Example 5.28 (ML). Now we sketch an encoding $(\Phi, \alpha, \beta, \gamma)$ of ML (as defined in Sect. 2.2.2.6) in $\mathbb{L}$. Φ maps a FOL-signature Z to the $\mathbb{L}$-signature $(\Sigma; \Delta; R; \underline{o}; \underline{true})$. Σ extends the base signature given in Fig. 5.3 with declarations of the form $p:o$ for every boolean variable $p \in Z$. Contrary to the encoding of FOL, the model theoretical semantics is unrelated to the interpretation of the proof rules: While Σ declares formulas and proofs, there is nothing in Σ that would make $[\![o]\!]^I$ the set of functions from some set of worlds to the set $\{0,1\}$. This was to be expected because Kripke models are only one possible way to interpret modal logic. Therefore, we use Δ to axiomatize the semantics, which is given in Fig. 5.4. The intended interpretation is for w the set of worlds, for b the set $\{0,1\}$, for the judgment acc the accessibility relation, and for v the evaluation of formulas in worlds. Since Δ is a model theoretical concept, we give it in the syntax of Fig. 4.12. α, β, and γ are straightforward.

```
% object level
w : type.
b : type.
0 : b.
1 : b.
o : type.
v : o -> w -> b.
not : o -> o.
imp : o -> o -> o.          %infix left 10 imp.
box : o -> o.

% for all p ∈ Z
p   : o.

% judgment level
true : o -> type.
acc : w -> w -> type.
A : true (F imp (not F imp G)).
B : true ((F imp G) imp ((G imp H) imp (F imp H))).
C : true ((not F imp F) imp F).
K : true (imp (imp (box (imp F G)) (box F)) (box G)).
MP : true (imp F G) -> true F -> true G.
Nec : true F -> true (box F).
```

Figure 5.3: Encoding of ML

$$
\begin{array}{l}
w \\
\neg 0 \equiv 1 \\
\forall x{:}b.(x \equiv 0 \vee x \equiv 1) \\
(v\ (not\ F)\ W) \equiv 1 \ \Leftrightarrow\ (v\ F\ W) \equiv 0 \\
(v\ (F\ imp\ G)\ W) \equiv 1 \ \Leftrightarrow\ ((v\ F\ W) \equiv 0 \vee (v\ G\ W) \equiv 1) \\
(v\ (F\ box\ G)\ W) \equiv 1 \ \Leftrightarrow\ \forall W'{:}w.((acc\ W\ W') \Rightarrow (v\ F\ W')) \\
true\ F \ \Leftrightarrow\ \forall W{:}w.(v\ F\ W) \equiv 1
\end{array}
$$

Figure 5.4: Axiomatization of ML Models

Example 5.29 (HOL). We sketch an encoding of a higher-order logic with standard models. Fig. 5.5 gives an LF signature that encodes formulas by $\underline{o} := tm\ o$ and the truth judgment by $\underline{true} := true$. This is an example where $\underline{o}$ is not a constant. It is easy to see that $[\![lam]\!]^I$ and $[\![@]\!]^I$ are bijections between $[\![\cdot | tm\ (A \Rightarrow B)]\!]^I$ and the set of functions from $[\![\cdot | tm\ A]\!]^I$ to $[\![\cdot | tm\ B]\!]^I$.

It remains open how to define variants of HOL with non-standard function spaces, i.e., Henkin models. It seems promising to use a variant of $\mathbb{L}$ where all those $\mathbb{D}$-models I are used for which P^I has a least element. Then taking the interpretation of types in the least element

yields models that are very closely related to Henkin models, presumably the model classes are in a natural bijection.

```
% object level
tp : type.
o : tp.
=> : tp -> tp -> tp.                    %infix left 10 =>.
tm : tp -> type.
lam : (tm A -> tm B) -> tm (A => B).
@ : tm (A => B) -> (tm A -> tm B).
== : tm A -> tm A -> tm o.

% judgment level
true : tm o -> type.                    %prefix 0 true.
refl : true S == S.
beta : true @ (lam F) S == F S.
eta : true lam (@ F) == F.
```

Figure 5.5: Encoding of HOL

As an example of an institution defined in LF, we give DFOL ([Rab06]).

Example 5.30 (DFOL). DFOL is defined by a functor $\Phi : Sig \to Sig^{\mathbb{L}}$.

Sig is given as follows: Objects are the LF-signatures, which extend the base signature given in Fig. 5.6 with declarations of the following form:

- Sort declarations: $s{:}\Pi_{x_1:tm\ s_1} \dots \Pi_{x_n:tm\ s_n} S$,
- Constant declarations: $c{:}\Pi_{x_1:tm\ s_1} \dots \Pi_{x_n:tm\ s_n}\ tm\ s$,
- Predicate declarations: $p{:}\Pi_{x_1:tm\ s_1} \dots \Pi_{x_n:tm\ s_n}\ o$.

Morphisms are $\mathbb{D}$-signature morphisms that are the identity for the symbols in Σ_B. Actually, Fig. 5.6 omits the proof rules. Those arise by extending the rules for first-order logic in a straightforward way. Details are given in [Rab06].

Φ maps a $Z \in |Sig|$ to $(Z; \Delta; R; \underline{o}; \underline{true})$ where Δ, R, $\underline{o}$, $\underline{true}$ are given as follows. Δ contains only the axiom $\neg(true\ ff)$. R makes S, tm, and o sorts, and $true$ a judgment. $\underline{o} := o$, and $\underline{true} := true$. Φ maps Sig-morphisms to themselves.

While [Rab06] uses LF to define DFOL, it does not make use of the framework we have presented here. Comparing the definitions in [Rab06] and Ex. 5.30 shows how we can extend the elegance and simplicity of proof theoretical logic definitions in LF to model theoretical logics.

```
% object level
S : type.
tm : S -> type.
o : type.
tt : o.
ff : o.
and : o -> o -> o.
or : o -> o -> o.
impl : o -> o -> o.
not : o -> o.
forall : (tm A -> o) -> o.
exists : (tm B -> o) -> o.
eq : tm A -> tm A -> o.

% judgment level
true : o -> type.
```

Figure 5.6: Definition of DFOL

Ex. 5.29 already gave an example for an encoding where $\underline{o}$ is a composed sort and not just a sort symbol. An even more complex case arises when we consider dependent type theory, e.g., by encoding $\mathbb{D}$ itself. We will not go into the details here and only give a fragment.

Example 5.31 (DTT). Fig. 5.7 gives a fragment of an encoding of $\mathbb{D}$ in $\mathbb{L}$. On the object level, it declares three syntactical sort symbols for terms, type families, and kinds, respectively, as well as a constant *tp* for the special kind `type`. On the judgment level, it declares three judgments for the well-formedness of kinds, the kinding of type families, and the typing of terms, respectively. Here we have $\underline{o} := \Sigma_{S:tf}\,(\mathit{ofkind}\ S\ \mathit{tp})$, i.e., the sentences are those type families for which the judgment that they are well-formed with kind *tp* holds. The subsort of well-formed terms of type S is given by $\mathit{wft}(S) := \Sigma_{s:tm}\,(\mathit{oftype}\ s\ S)$. And we have $\underline{\mathit{true}} := \lambda_{p:\underline{o}}\,\neg\neg\mathit{wft}(\pi_1(p))$, i.e., $\underline{\mathit{true}}$ assigns to every sentence $p = \langle S, - \rangle$ the judgment that there is a well-formed term of type S. The double negation expresses the judgment that the sort $\mathit{wft}(S)$ is inhabited.

```
% object level
tm : type.
tf : type.
kd : type.
tp : kd.
% judgment level
wfk : kd -> type.
ofkind : tf -> kd -> type.
oftype : tm -> tf -> type.
```

Figure 5.7: Encoding of DTT

Finally, we give an example of an extension of FOL with implicit definitions that we can use to define set theories. It is also an example of an $\mathbb{L}$-signature that is not separated.

Example 5.32 (FOL with Implicit Definitions). We define a logic FOL with implicit definitions by extending the signature in Fig. 5.2 with the declarations from Fig. 5.8. *existsU* is an abbreviation for the quantifier of unique existence. *implDef* takes two arguments, a formula f in a free variable, and a proof of *existsU* f; it returns a new individual. And *implAx1* *implAx2* give an implicit definition its intended meaning. *implDef* can also be regarded as a description operator.

```
existsU : (i -> o) -> o = [f : i -> o] exists [x : i] (f x and forall [y : i] f y imp y eq x).
implDef : {f : i -> o} true existsU f -> i.
implAx1 : true (implDef F P) eq (implDef F P').
implAx2 : true F (implDef F P).
```

Figure 5.8: Implicit Definitions

5.5 Defining and Encoding Logic Translations

Given two logics that are defined or encoded in $\mathbb{L}$, we can define or encode logic comorphisms between them.

Definition 5.33 (Defining Comorphisms)**.** For two functors $\Phi^i : Sig^i \to Sig^{\mathbb{L}}$ where $i = 1, 2$, a comorphism definition is a pair (Φ, m) where $\Phi : \Phi^1 \to \Phi^2$ is a functor and $m : \Phi^1 \to \Phi^2 \circ \Phi$ is a natural transformation.

Theorem 5.34. *If (Φ, m) is a comorphism definition from Φ^1 to Φ^2, then*

$$(\Phi, Sen^{\mathbb{L}}(m), Mod^{\mathbb{L}}(m), Pf^{\mathbb{L}}(m))$$

is a logic comorphism from $\mathbb{L} \circ \Phi^1$ to $\mathbb{L} \circ \Phi^2$.

Proof. This result does not depend on the details of $\mathbb{L}$ at all. It follows by basic operation on categories using Lem. 3.5. □

Thus, a comorphism between logics can be defined by giving a functor and a family of $\mathbb{L}$-morphisms. This is parallel to the definition of logics in $\mathbb{L}$.

Definition 5.35 (Encoding Comorphisms)**.** Assume a logic comorphism $\mu : \mathbb{I}^1 \to \mathbb{I}^2$, and two encodings μ^1 and μ^2 of $\mathbb{I}^1$ and $\mathbb{I}^2$. An encoding of μ is a logic comorphism modification from μ^1 to $\mu^2 \circ \mu$.

Thus, logics are encoded as logic comorphisms, and logic comorphisms are encoded as logic comorphism modifications. Another way to think of such encodings is that instead of working in the category $\mathcal{LOG}$, we work in the slice category $\mathcal{LOG}/\mathbb{L}$.

5.6 Examples

We finally come full circle by encoding the translation from modal logic to first-order logic in $\mathbb{L}$.

Notation 5.36. A $Sig^{\mathbb{L}}$-morphism from L to L' is a mapping from Σ-symbols c to Σ'-terms s. We write such a morphism σ as a list of pairs $(c, \sigma(c))$, and we use `c => s` as ASCII syntax for such pairs.

Example 5.37 (Translating ML to FOL). To encode the comorphism μ (with signature translation Φ) from ML to FOL as given in Sect. 2.2.3, assume the encodings μ^1 of ML (with signature translation Φ^1) and μ^2 of FOL (with signature translation Φ^2) as given in Ex. 5.28 and Ex. 5.27. We need to give a logic comorphism modification from μ^1 to $\mu^2 \circ \mu$. More concretely, we need for every ML-signature Z an $Sig^{\mathbb{L}}$-signature morphism m_Z from $\Phi^1(Z)$ to $\Phi^2(\Phi(Z))$. The pairs comprising m_Z are given in Fig. 5.9 where we omit the translations of the rules.

To check that m_Z is a theory morphism, we need to show that the axioms of $\mu^1(Z)$ hold after their translation via m_Z. We omit that step. The role preservation of m_Z is easy to check. Finally, Def. 5.9 requires a term k of type $m_Z(\underline{o}) \to \underline{o}'$, which in this case means $(i \to o) \to o$: We have $k = \lambda_{f:i\to o}\, (forall\ f)$.

An important property of this encoding is the translation of o and *true*: We do not have $m_Z(o) = o$ and $m_Z(true) = true$. Instead, o is mapped to $i \to o$. We can now explain the remark after Def. 5.9 and argue why a stricter definition would not work. The term *forall* used in $m_Z(true)$ above corresponds exactly to the last step of the definition of α_Σ in Sect. 2.2.3.1: There we defined $\alpha_\Sigma(F) = \forall w \overline{F}^w$.

More abstractly, the translation of formulas from ML to FOL is not compositional. It consists of a compositional translation (namely $F \mapsto \overline{F}^w$) followed by one toplevel step (namely $\overline{F}^w \mapsto \forall w \overline{F}^w$). Such translations are typical when the semantics of one logic is encoded using

```
% object level
w    => i
b    => o
0    => ff
1    => tt
o    => i -> o
v    => [f: i -> o] x
not  => [f: i -> o] [x: i] (not (f x))
imp  => [f: i -> o] [g: i -> o] [x: i] ((f x) imp (g x))
box  => [f: i -> o] [x: i]  forall [y: i] ((acc x y) imp (f y))

% for all p ∈ Z
p    => @p

% judgment level
true => [f: i -> o] true (forall f)
acc  => [x: i] [y: i] true (acc x y)
...
```

Figure 5.9: Encoding of the Translation from ML to FOL

the syntax of another logic. In those cases sentences are first translated to some syntactical expressions (which is often a compositional translation); and a last step turns the intermediate result into a sentence. The last step makes the whole translation non-compositional. And since $Sig^{\mathbb{L}}$-morphisms can only represent compositional translations, we need to accommodate for this last step in some other way. Therefore, $Sen^{\mathbb{L}}$ uses the types $\underline{true}\ f$ as the sentences and not the terms f as one might expect.

Another simple example for such a translation is a translation from many-valued propositional logic MVPL to first-order logic. The compositional part of the translation maps the MVPL sentences to FOL terms. And a unary FOL-predicate *Des* is used to encode the designated truth values of MVPL. If o is the type of propositions in the encoding of MVPL, we would have $m_Z(o) = i$ and $m_Z(true) = \lambda_{x:i}\,(true\ Des\ x)$.

As an example for the use of subsorts, we sketch a translation from HOL to ZFC. Here we mean by ZFC the logic that arises from Ex. 5.32 by taking (i) only those signatures that contain a binary infix predicate symbol $\in$ and the axioms of Zermelo-Fraenkel set theory with choice, and (ii) only those signature morphisms that preserve $\in$ and the axioms.

Example 5.38 (Translating HOL to ZFC). The translation (Φ, m) encodes the interpretation of HOL in standard ZFC-models. First, we define some abbreviations over ZFC:

- For a ZFC-term x of type i, let $\overline{x} := \Sigma_{y:i}\,(true\ y \in x)$, i.e., $\overline{x}$ abbreviates the subsort of the ZFC-universe containing the elements of x.
- The function set constructor $Func : i \to i \to i$ is implicitly definable in ZFC.
- For ZFC-terms x of type i (the domain) and f of type $\overline{x} \to i$ (the function), a λ-abstraction is implicitly definable in ZFC (by using the axiom of replacement), i.e., there is a term $func : \Pi_{x:i}\,(\overline{x} \to i) \to i$.

Then we have for a HOL-signature Z that $\Phi(Z)$ contains a symbol of type i for every base type of HOL. And the $Sig^{\mathbb{L}}$-morphism m_Z maps in particular:

- $m_Z(tp) = i$, i.e., a type is mapped to an element of the ZFC-universe.
- $m_Z(tm) = \lambda_{x:i}\,\overline{x}$, i.e., a sort $tm\ A$ is mapped to the subsort of i containing the elements of $m_Z(A)$.
- $m_Z(\Rightarrow) = Func$.

- $m_Z(lam) = \lambda_{f:\overline{A}\to\overline{B}}\,\langle func\ A\ (\lambda_{x:\overline{A}}\,\pi_1(f\ x)), p\rangle$, i.e., HOL-$\lambda$-abstraction is mapped to the λ-abstraction of ZFC. Here $m_Z(lam)$ has return type $\overline{Func\ A\ B}$, and p is the ZFC-proof that if f has domain A and B includes the image of f, then $func\ A\ f$ is a function from A to B.

Subsorts are needed in a lot of translations, e.g., when translating sorted first-order logic to unsorted first-order logic. Another example for such translations are translations from a higher-order logic in which every type is encoded as a separate LF-type to a higher-order logic in which all types are encoded by the same type. The former is more convenient but only possible if type-checking is decidable. An example for such a translation is the translation from HOL to Nuprl ([Gor88, CAB+86, NSM01, SS04]). Our solution to this problem tries to be minimally invasive to retain compatibility with existing LF encodings. Therefore, we did not consider a framework that extends LF with explicit subtypes such as the one given in [AC01].

5.7 Future Work

5.7.1 Logical Libraries

So far our example representations of logics and logic translations do not go very deep. An obvious next step is to use the framework to create a **logical library**. However, we find that our framework does not scale to that level. Therefore, Part III will focus on designing a scalable logical framework, in particular one that scales to the web.

Then future work can concentrate on using our scalable architecture as the basis of a logical library. Such libraries have been worked on before, and it will be most beneficial to merge our architecture with existing ones. We intend to focus on two libraries in particular, the LF-based proof theoretical library Logosphere ([PSK+03]), and the institution-based model theoretical Hets system ([MML07]).

Hets As mentioned in Sect. 1.3.1, Hets ([MML07]) is a tool for the heterogeneous theory development. It can work with translations on the inter-theory and inter-logic level. And since it is parametric in the institution for the intra-theory level, it can act as a logical framework. However, currently Hets does not support the dynamic specification of logics and logic translations — these are hard-coded.

A natural extension would make Hets understand logic definitions in $\mathbb{L}$. For that it will be necessary to make the internal representations of logics instantiable so that new logics can be created at run time. Then it is possible to parse logic definitions and generate the data structures for new logics dynamically. Logic translations can be treated accordingly.

One important advantage of this is that $\mathbb{L}$-definitions are purely declarative whereas the current definitions in Hets are programmed. On the other hand, this makes Hets more expressive, which is especially important for logic translations. We expect that extending our framework to Delphin ([PS08]), a functional programming language based on LF, will provide a good compromise.

This also offers another application, namely to use the modular development features of Hets to build logics. For example, colimits such as pushouts can be used to combine different logics as in [Tar96] and [MTP97]. In Part III, we will pick up on the idea of using the same modular development both for the inter-theory and the inter-logic level.

Finally, Hets can prove the correctness of inter-theory translations by combining internal reasoning and calling external provers. But it does not offer strong support for proof objects and proof-checking. Here for $\mathbb{L}$-defined logics, LF can act as a small trusted proof checker.

Logosphere The Logosphere database is designed to contain logics, logic translations, and theorems with proofs. Currently, the database contains some logics, and users can upload

theorem files via the web site. For the latter, the logic must be referenced, which is implemented by interpreting a comment on the first line of the theorem file as the file name of the logic. Then the two files are concatenated and checked by the Twelf ([PS99]) implementation of LF. Retrieval and Google-like search are planned but not implemented yet.

We see two ways how to extend Logosphere. Firstly, Logosphere works mainly on the intra-theory level, and the support for the higher levels is not provided in a systematic way. Here the techniques developed for institutions can provide the right intuitions. It is also a good example of an application that can benefit from a more scalable infrastructure, in particular techniques for databases and search.

5.7.2 Completeness Analysis

When using $\mathbb{L}$ to define a logic, it is a crucial question whether $\mathbb{L} \circ \Phi$ is complete. The same question arises when encoding institutions as in Thm. 5.25. Completeness would always hold if $\mathbb{L}$ were complete, but that is not the case due to the restriction to simple models.

We could extend $\mathbb{L}$ to a complete logic by admitting all Kripke models, but this would pose further problems: Logic encodings in $\mathbb{L}$ must reduce all $\mathbb{L}$-models, and this can be hard or even impossible if there are too many models in the meta-logic. While some models can be thrown out by using axioms, this is not always sufficient to obtain a comorphism into $\mathbb{L}$. Furthermore, the non-simple models can be regarded as an irrelevant artifact, for example when encoding first-order logic.

However, in many situations it is irrelevant whether $\mathbb{L}$ is complete or not. The completeness of the meta-logic is not a prerequisite for reasoning about the completeness of an encoded logic. This is because only the signatures in the image of Φ are relevant for the completeness of $\mathbb{L} \circ \Phi$, and completeness for that special case may be much easier to obtain than in general. In fact, the completeness of $\mathbb{L}$ could even be a problem because it would always make $\mathbb{L} \circ \Phi$ complete, which would preclude the encoding of incomplete logics.

The most promising direction of further research seems to be to use $\mathbb{L}$ as a framework in which to carry out a general completeness proof for a large class of logics. Some completeness proofs have been generalized in this way (e.g., in [Pet07] for Gödel-completeness in the framework of institutions). $\mathbb{L}$ may help with such generic completeness proofs because it unifies model and proof theory in a logical framework and provides the concrete syntax needed to populate a canonical model.

It might even be possible to reuse the canonical model we gave for $\mathbb{D}$. For example, the canonical model is always built upon a poset with a least element (in fact, it is a complete Heyting algebra). It seems possible to reduce a $\mathbb{D}$-Kripke model to a FOL-model or a Henkin model of higher-order logic by evaluating all terms and types in this least element.

Another interesting idea is to restrict the interpretation of $\mathbb{D}$-types S such that all $[\![\cdot|S]\!]^I(p \leq p')$ must be surjective. Such models are better behaved and can be reduced to the models of an encoded logic more easily. (The technical reason is that for every $p \in P^I$ every element of $[\![\cdot|S]\!]^I(p)$ occurs as a component in some indexed element of $[\![\cdot|S]\!]^I$.) This property is preserved by unit, product, and sum, but not by identity type formation. If $\mathbb{D}$ without identity types is complete for such models, this might make completeness proofs much easier.

5.8 Conclusion

Our motivation was to combine model and proof theory. For that purpose, we introduced an abstract definition of logic and logic translations. By extending institutions with abstract notions of proofs and judgments, our development is conservative over the framework of institutions. But institutions are too abstract to benefit from the existing proof theoretical encodings, which are most elegantly given in dependent type theory (DTT). Therefore, we obtained a model

theory for DTT. And we used that model theory to give a more concrete notion of proof theory that is conservative over the Edinburgh logical framework (LF).

From the point of view of proof theory, we have given a model theoretical semantics for logic encodings in LF by using institutions. And from the point of view of model theory, we have given a specification language for institutions by using LF. From a neutral point of view, these results yield a logical framework that combines model theory and proof theory.

Thus, we can **evaluate** our requirements **(R1)** – **(R3)** from Sect. 1.3.1 in the positive. However, as already indicated in Sect. 1.3.1 and 5.7.1, the scalability requirement **(R4)** requires a separate investigation, for which we have reserved Part III.

As already argued in Sect. 1.3.1, our framework has some **limitations**. (i) Substructural logics cannot be encoded naturally. This can be remedied in the future by using CLF ([WCPW02]) instead of LF as the base language. However, giving a model theory for CLF is complicated and not understood yet. (ii) The support for non-compositional translations is limited. This can be addressed by extending the notion of signature morphisms in $\mathbb{L}$ with translations that are defined by case distinction and recursion. Since the signatures stay the same, the model theory would not have to be changed. Such a language along with a Twelf-compatible implementation has recently been developed ([PS08]), and we plan to integrate it into our framework. (iii) Our definition of object and judgment level excludes interesting logic encodings, namely those where the judgment level makes judgments about itself. Of course, some hierarchy of judgment levels would be necessary to prevent this from being cyclic. The most important example are some encodings of modal logic, for example one given in [AHMP98]: The "boxed assumptions judgment" uses a judgment about proofs that asserts that a proof only used certain assumptions. We exclude such encodings because we favor the gained simplicity. In particular, our definition permits to use proof irrelevance. (iv) It remains to be seen whether stronger support for subsorts will become necessary when attempting more complex translations. (v) The necessity to restrict $\mathbb{L}$ to simple models is unappealing but has proved necessary so far. In Sect. 5.7.2, we have outlined some ideas how to remedy this in the future.

Acknowledgements While the work presented in this section is original, it owes much to discussions with Frank Pfenning and Till Mossakowski as well as Valeria de Paiva, Carsten Schürmann, and Andrzej Tarlecki.

Part III

Logical Knowledge Management

Chapter 6

A Module System for Logical Knowledge

6.1 Introduction

In this section, we introduce the module system MMT for formal knowledge that addresses the shortcomings of current systems. The main technical innovations are that we use a web-scalable naming scheme, make theory morphisms first-class citizens, and introduce the notion of meta-theories to account for logics and meta-logical frameworks.

MMT is expressive enough to represent all modules systems presented in Sect. 1.1.3.2 except for the higher-order systems and stronger forms of hiding. Our modules are called **theories**, and we use named, interspersed imports, with free, explicit, partial instantiations. These are called **structures**. We also provide for **views**, and **theory morphisms** are formed by composing structures and views. Axioms are treated as special symbols following the Curry-Howard correspondence, and we permit hiding. MMT is defined by a formal abstract syntax, and for the concrete syntax, we build on the OMDoc (Open Mathematical Documents, [Koh06]) format, which already integrates much of the state of the art reviewed so far into a coherent content-oriented markup format for mathematical knowledge.

In the rest of this section we will present the problems that motivated the work in this section and preview our solutions.

Semantic Referencing In mathematical languages, we need to be able to refer to (i.e., identify) content objects in order to state the semantic relations. In principle, there are two ways to access mathematical knowledge: **by location** (relative to a particular document or file) and **by context** (relative to a mathematical theory). The first one essentially makes use of the organization structure of file systems, and the second makes use of mathematical structuring principles supplied by the representation format. Both approaches to resource identification have their justification: Conceptually, resource identification by location can be inherited from the XML substratum and is desirable, since it can be readily mapped to current practice and transport protocols of the internet. But resource identification by location is brittle for content-oriented formats, where the distribution of knowledge over file system is a secondary consideration. It is standard practice in mathematics to develop mathematical theories decentrally, and change their location, e.g., by publishing them. Once there is a definition of a theory in place (e.g., in an academic journal), other researchers add theorems to the theory in other documents. Furthermore, it is a necessary requirement for a representation format to be closed under concatenation so that we can aggregate mathematical knowledge in databases, e.g., for computer-supported learning. This breaks the other assumption in URI-based systems for identification by location:

that we can rely on document-unique identifiers for document fragments.

The difficulty for resource identification by context is that it should still be compatible with the URI-based approach which is sometimes called the "plumbing of the Web", since it mediates all resource transport over the internet. Note that the OpenMath 2 standard ([BCC+04]) already pioneered resource identification by context for symbols: A symbol is identified by the symbol name and content dictionary, which in turn is identified by the CD name and the CD base, i.e., the URI where the CD is located. A mapping from this symbol identification triple to a URI is only straightforward via the one-CD-one-file restriction imposed by OpenMath, which is too restrictive in general. The current OMDoc version 1.2 restricts theory names to be document-unique enabling OpenMath-style resource identification by context for symbols; statement- and theory-level objects are relegated to identification by location.

A major problem for identification by content in OMDoc is that it uses unnamed imports with renaming, which significantly hinders multiple inheritance. Therefore, we introduce a new way for reference by context that allows for multiple inheritance and hierarchic namespaces and is easy to implement. Thus, every declaration including those that are imported from other theories is uniquely identifiable by a URI. Reference by location is supported as a special case.

Libraries and Queries Our main concept is that of a **library**, which formalizes the idea of a database for mathematical knowledge. All operations center around the current library. In particular, all references to named entities are resolved via special lookup functions that query the library. This is an important aspect of the scalability of Mmt: It permits to distribute knowledge management services over different machines that interconnect via the lookup functions — possibly indirectly with a component maintaining the library in between.

Furthermore, libraries serve to encapsulate state by storing the symbols defined in the context so that implementations of specific object languages can focus on the mathematically challenging parts, e.g., type inference and proof search.

Named Imports as Objects Consider the running example from Fig. 1.1. The theory `monoid` is inherited twice by the theory `ring`. Thus, for example, the composition operation of `monoid` must give rise to two different operations in the theory `ring`, namely addition and multiplication. This was not supported by OMDoc 1.2.

The integration of named imports into OMDoc has two consequences. Firstly, the theory graph structure becomes more complicated and makes the need of a fully formalized notion of well-formedness even bigger than it had been. And secondly, imports become much closer conceptually to symbols. Thus, imports play a twofold role. On the level of theories, they are morphisms that translate between theories. On the level of declarations, they behave like symbols in that they have declarative character. The latter nature is more adequately captured by calling them structures, and we are so far lacking a name that better reflects their double nature. Officially, we will call them *structures*, and we will appeal to the import metaphor where helpful for the intuition.

It is a crucial and novel feature of our system that this dual role of imports is fully captured and utilized: Whenever possible, structures are treated like symbols, and we even changed our ontology to reflect this: We use the concept **constant** for symbols in the usual sense as they are used to represent sorts, operations, etc., and we use the concept **symbol** to unify the concepts *constant* and *structure*. In particular, structures are themselves subject to instantiations: Instantiations map constants to terms and structures to theory morphisms.

To provide a clearer intuition for the instantiation of imports with theory morphisms, let us look at the pseudo code fragment of the object-oriented program in the left part of Fig. 6.1. It corresponds to the situation of Fig. 1.2. The class `Order` provides the interface of an ordering relation, which is implemented by the class `OrdNat` for the ordered natural numbers. The class `OrdList` provides the functionality of ordered lists after being instantiated with an object

implementing the class `Order`. The constructor of `Main` declares first an instance n of `OrdNat` and then uses it to obtain an instance l of `OrdList` that implements the ordered lists of natural numbers. Thus, `Main` instantiates `Order` twice, namely via n and $l.e$. Furthermore, the second instantiation passes n, which is itself an instantiation, as an argument to the constructor of `OrdList`.

```
class Order {
  ...
}

class OrdNat implements Order {
  ...
}

class OrdList {
  object e : Order
  constructor OrdList(e : Order) {this.e := e}
  ...
}

class Main {
  constructor Main {
    object n = new OrdNat
    object l = new OrdList(n)
  }
}
```

```
<theory name="Order">
  ...
</theory>

<theory name="OrdNat">
  ...
</theory>

<view name="m" from="Order" to="OrdNat"/>
  ...
</view>

<theory name="OrdList">
  <import name="e" from="Order"/>
  ...
</theory>

<theory name="Main">
  <import name="n" from="OrdNat"/>
  <import name="l" from="OrdList">
    <maps name="e">Main/n ∘ m</maps>
  </import>
</theory>
```

Figure 6.1: Instantiations

We apply this principle of instantiation to mathematical theories. In our mathematical module system, using a simplified version of its XML encoding (see Sect. 8.1), the above example would look as in the right part of Fig. 6.1. `Main` imports from `OrdNat` and `OrdList`, and `OrdList` imports from `Order`. The fact that `OrdNat` implements `Order` is expressed by an explicit theory morphism m (which we call a view; in the terminology of development graphs, it is a theorem link). Thus, there are two theory morphisms from `Order` to `Main`: the compositions $m_1 := \texttt{Main}/l \circ \texttt{OrdList}/e$ and $m_2 := \texttt{Main}/n \circ m$. The semantics of the instantiation of e with $\texttt{Main}/n \circ m$ in the import l is that m_1 and m_2 are equal. Note that m_1 corresponds to the dereferentiation $l.e$.

Meta-Theories We are interested in a representation system for mathematical knowledge that is based on a structural view of formulas. It is one of the attractive features of such a system that we can concentrate on structural issues and leave lexical ones to an external definition mechanism like content dictionaries and theories. In particular, this allows us to operate without choosing a particular foundational logical system, as we can just supply content dictionaries for the symbols in the particular logic. Thus, logics and in the same way logical frameworks become theories, and we speak of the **logics-as-theories** approach. This is particularly attractive in a setting where the content dictionaries are allowed to be informally given in natural language as it allows us to mimic mathematical practice where the meta-mathematics are usually given that way.

But conceptually, it is helpful to distinguish levels here. To state a property in the theory `ring` like commutativity of the operation $\circ$ over the base set R in $\forall a, b \in R. a \circ b = b \circ a$, we use symbols $\forall$ and $=$ from first-order logic together with $\circ$ and R from ring theory. Even though it is structurally possible to just build a theory of rings by simply importing first-order logic, this would fail to describe the meta-relationship between the theories. But this relation is crucial when interpreting `ring`: The symbols of the meta-language are not interpreted because a fixed interpretation is given in the context. Therefore we think of first-order logic as the **meta-theory**

of the theory `ring`. And we permit every theory to refer to another theory as its meta-theory.

For example, in Fig. 6.2, the running example is extended by adding meta-theories. The theory `fol` for first-order logic is the meta-theory for `monoid` and `ring`. And the theory `LF` for the logical framework LF is the meta-theory of `fol` and the theory `hol` for higher-order logic. Now the crucial advantage of the logics-as-theories approach is that on all three levels the same module system can be used: For example, the morphisms m and m' indicate possible translations on the levels of logical frameworks and logics, respectively.

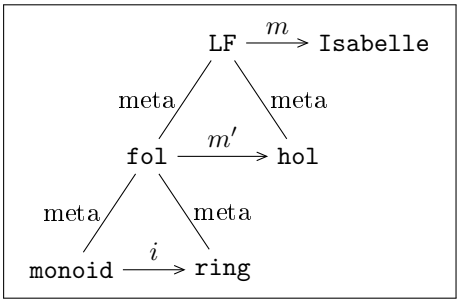


Figure 6.2: Meta-Theories

Hiding Hiding in formal specifications has been studied in [ST88], [GR04], [CoF04], and [MAH06]. A natural way to model hiding is by a symbol mapping $\mu : S \to T$ where T contains the internally used symbols and S contains the publicly visible ones. Often μ is an inclusion, e.g., in OMDoc 1.2. Hiding has the technical difficulty that usually T is given first and S is obtained from T by hiding some symbols. Thus, the direction of the mapping μ is against the direction of the development. This is responsible, for example, for the loss of flattenability in development graphs with hiding ([MAH06]).

Our motivation is to integrate the main functionality of hiding while retaining conceptual simplicity. Therefore, instead of using hiding maps $\mu : S \to T$, we model hiding by partial maps from T to S, which are undefined for the symbols that are to be hidden. Thus, we are able to treat hiding as a special case of instantiation. In particular, all our theory graphs remain flattenable.

Hiding is usually applied to symbols whose values can be inferred from the other parts of the specification. For example, in the example in Fig. 1.3, the values of the hidden operations are uniquely determined by the axioms. In general, it is reasonable to hide symbols added in conservative extensions, i.e., symbols that have direct or indirect (via axioms) definitions. It is also possible to hide symbols that are loosely axiomatized. To reason about such hidings, the proof system given in [MAH06] uses an oracle for conservative extensions.

An important property of theory morphisms μ from S to T is that they induce a mapping from S-expressions to T-expressions. This property is usually lost if μ uses hiding. However, if μ hides a defined symbol c, it is still possible to map all S-objects to T-objects: c can be replaced by its definition before applying μ. If c is only determined indirectly by axioms, this is not always possible because the axiomatic characterization of c may or not be constructive. However, an axiomatic definition can be turned into a direct one if the meta-theory of S and T has a description operator. Similarly, if c is loosely axiomatized and hidden, it is still possible to map c from S to T if there is a choice operator in T. Therefore, we contend that by using meta-theories with description or choice operators, the above cases can be reduced to the hiding of defined symbols.

Since it is easy to check syntactically whether a constant has a definition, it would be possible to restrict hiding to defined constants. We go one step further and permit the hiding of any constants. But we still retain the property that all morphisms μ extend to maps between the objects: Whenever μ is applied to an object that contains an undefined hidden symbols, the whole object is hidden. Of course, then we have to keep track of hidden objects.

Propositions as Types and Proofs as Terms We do not provide syntax for axioms and proofs. Instead, we use the **Curry-Howard correspondence** ([CF58, How80]) in order to express formulas and proofs as terms. Then the relation "p proves F" between proofs and

formulas is reduced to the typing relation "p has type *true* F" where *true* is a type constructor defined by the respective meta-theory, i.e., the logic. Declaring an axiom F is reduced to declaring a constant of type *true* F. And declaring a theorem F with proof p is reduced to declaring a constant of type *true* F with definiens p. Furthermore, proof rules can be specified as constants as well if an appropriate meta-theory for a logical framework is used. That significantly reduces the conceptual complexity without losing expressivity.

6.2 Syntax

In this section we cover the syntactical aspects of our module system: We specify a formal syntax for the MMT module system that we use in the mathematical analysis of the MMT language.

6.2.1 A Four-Level Model of Mathematical Knowledge

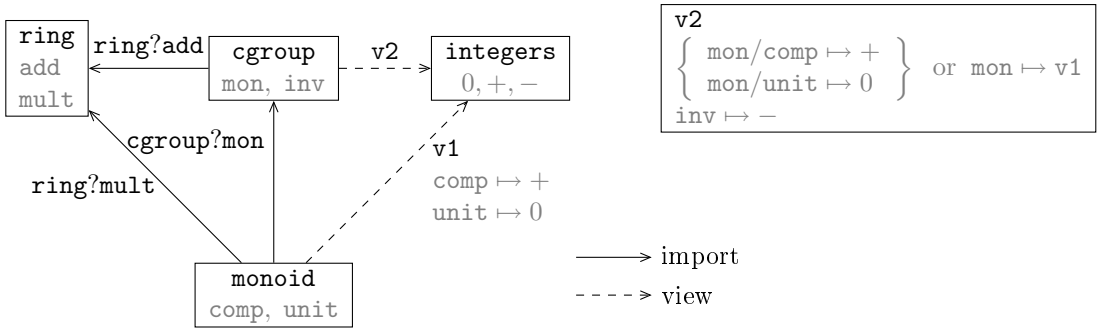


Figure 6.3: Example

Example 6.1. We begin the exposition of our language with a simple motivating example that already shows some of the crucial features of MMT and that we will use as a running example: Fig. 6.3 gives a theory graph for a portion of the algebraic hierarchy. The bottom node represents the theory of monoids, which declares operations for composition and unit. (We omit the axioms and the types here.) The theory `cgroup` for commutative groups arises by importing from `monoid` and adding an operation `inv` for the inverse element. This theory does not have to declare the operations for composition and unit again because they are imported from the theory of monoids. The import is named `mon`, and it can be referenced by the qualified name `cgroup?mon`; it induces a theory morphism from monoids to commutative groups.

Then the theory of rings can be formed by importing from `monoid` (via an import named `mult` that provides the multiplicative structure) and from `cgroup` (via an import named `add` that provides the additive structure). Because all imports are named, different import paths can be distinguished: By concatenating import and symbol names, the theory `ring` can access the symbols `add/mon/comp` (i.e., addition), `add/mon/unit` (i.e, zero), `add/inv` (i.e., additive inverse), `mult/comp` (i.e., multiplication), and `mult/unit` (i.e., one).

The node on the right side of the graph represents a theory for the integers declaring the operations 0, +, and −. The fact that the integers are a monoid is represented by the view `v1`. It is a theory morphism that is explicitly given by its interpretations of `comp` as + and of `unit` as 0. (If we did not omit axioms, this view would also have to interpret all the axioms of `monoid` as — using Curry-Howard representation — proof terms.)

The view `v2` is particularly interesting because there are two ways to represent the fact that the integers are a commutative group. Firstly, all operations of `cgroup` can be interpreted

as terms over `integers`: This means to interpret `inv` as $-$ and the two imported operations `mon/comp` and `mon/unit` as $+$ and 0, respectively. Secondly, `v2` can be constructed along the modular structure of `cgroup` and use the existing view `v1` to interpret all operations imported by `mon`. In MMT, this can be expressed elegantly by the interpretation $\texttt{mon} \mapsto \texttt{v1}$, which interprets a named import with a theory morphism. The intuition behind such an interpretation is that it makes the right triangle commute: `v2` is defined such that $\texttt{v2} \circ \mathit{cgroup?mon} = \texttt{v1}$. Clearly, both ways lead to the same theory morphism; the second one is conceptually more complex but eliminates redundancy. (This redundancy is especially harmful when axioms are considered, which must be interpreted as proofs.)

Ontology and Grammar We characterize mathematical theories on four levels: the **document**, **module**, **symbol**, and **object** level. On each level, there are several kinds of expressions. In Fig. 6.4, the relations between the MMT-concepts of the first three levels are defined in an ontology. The MMT knowledge items are grouped into six primitive concepts. **Documents** (*Doc*, e.g., the whole graph in Fig. 6.3) comprise the document level. **Theories** (*Thy*, e.g., `monoid` and `integers`) and **views** (*Viw*, e.g., `v1` and `v2`) comprise the module level. And finally **constants** (*Con*, e.g., `comp` and `inv`) and **structures** (*Str*, e.g., `mon` and `add`) as well as **assignments** to them (*ConAss*, e.g., $\texttt{inv} \mapsto -$, and *StrAss*, e.g., $\texttt{mon} \mapsto \texttt{v2}$) comprise the symbol level.

In addition, we define four unions of concepts: First **modules** (*Mod*) unite theories and views, **symbols** (*Sym*) unite constants and structures, and **assignments** (*Ass*) unite the assignments to constants and structures. The most interesting union is that of **links** (*Lnk*): They unite the module level concept of views and the symbol level concept of structures. Structures being both symbols and links makes our approach balanced between the two different ways to understand them.

These higher three levels are called the structural levels because they represent the structure of mathematical knowledge: Documents are sets of modules, theories are sets of symbols, and links are sets of assignments. The actual mathematical objects are represented at the fourth level: They occur as arguments of symbols and assignments. MMT provides a formalization of the structural levels while being parametric in the specific choice of objects used on the fourth level.

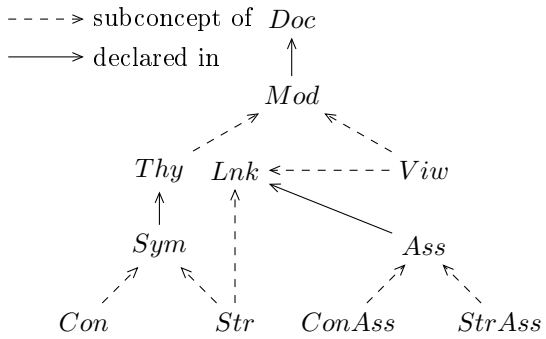


Figure 6.4: The MMT Ontology

The declarations of the four levels along with the meta-variables we will use to reference them are given in Fig. 6.5 and 6.6. The MMT knowledge items of the structural levels are declared with unqualified names (underlined Latin letter) in a certain scope and referenced by qualified names (Latin letter). Greek letters are used as meta-variables for composed expressions.

The grammar for MMT is given in Fig. 6.7 where *, $^+$, $|$, and $[-]$ denote repetition, non-empty repetition, alternative, and optional parts, respectively. The rules for the non-terminals URI and pchar are defined in RFC 3986. Thus, g produces a URI without a query or a fragment. (The query and fragment components of a URI are those starting with the special characters ? and #, respectively.) pchar, essentially, produces any Unicode character, possibly using percent-encoding for reserved characters. (Thus, percent-encoding is necessary for the characters ?/#[]% and all characters generally illegal in URIs.) In this section, we will describe the syntax of MMT and the intuition behind it in a bottom-up manner, i.e., from the object to the document level. Alternatively, the following subsections can be read in top-down order.

Level	Declared concept
Document	document g
Module	theory T, S, R, M
	view v
	structure or view m
Symbol	symbol s
	constant c
	structure h, i, j
	assignment to a constant
	assignment to a structure
Object	variable x

Figure 6.5: Meta-Variables for References to MMT Declarations

Variable	Type
Λ	library (list of document declarations)
γ	document body (list of module declarations)
ϑ	theory body (list of symbol declarations)
σ	view/structure body (list of assignments to symbols)
ω	term
μ	morphism

Figure 6.6: Meta-Variables for MMT Expressions

6.2.1.1 The Object Level

We distinguish two kinds of objects: **terms** and **morphisms**. Atomic terms are references to declared constants, and general terms are built up from the constants via operations such as application and binding as detailed below. Atomic morphisms are references to structures and views, and general morphisms are built up using structures, views, identity, and composition as detailed below.

The MMT objects are typed. For terms, we do not distinguish terms and types syntactically: Rather, typing is a binary relation on terms that is not further specified by MMT. Instead, it is left to be determined by the **foundation**, and MMT is parametric in the specific choice of typing relation. In particular, terms may be untyped or may have multiple types. For morphisms, the domain theory doubles as the type. Thus, every morphism has a unique type.

Well-formedness of objects is checked relative to a home theory (see Sect. 6.3). For a term ω, the home theory must declare or import all symbols that occur in ω. For a morphism, the home theory is the codomain. Objects with home theory T are also called **objects over** T. These relations are listed in Fig. 6.8.

Terms MMT-terms are a generalization of a fragment of OPENMATH objects ([BCC+04]). They can be

- **constants** c declared or imported by the home theory,
- **variables** x declared by a binder,
- **applications** $@(\omega, \omega_1, \ldots, \omega_n)$ of ω to arguments ω_i,
- **bindings** $\beta(\omega_1, \Upsilon, \omega_2)$ by a binder ω_1 of a list of variables Υ with scope ω_2,

Library	Λ	::=	Doc^*
Document	Doc	::=	$g := \{\gamma\}$
Document body	γ	::=	Mod^*
Module	Mod	::=	$Thy \mid Viw$
Theory	Thy	::=	$\underline{T} \stackrel{[T]}{:=} \{\vartheta\}$
View	Viw	::=	$\underline{v} : T \to T \stackrel{[\mu]}{:=} \{\sigma\} \mid \underline{v} : T \to T := \mu$
Theory body	ϑ	::=	Sym^*
Symbol	Sym	::=	$Con \mid Str$
Link body	σ	::=	Ass^*
Assignment	Ass	::=	$ConAss \mid StrAss$
Constant	Con	::=	$\underline{c} : \omega := \omega \mid \underline{c} : \omega \mid \underline{c} := \omega \mid \underline{c}$
Structure	Str	::=	$\underline{i} : T \stackrel{[\mu]}{:=} \{\sigma\} \mid \underline{i} : T := \mu$
Ass. to constant	$ConAss$	::=	$\underline{c} \mapsto \omega$
Ass. to structure	$StrAss$	::=	$\underline{i} \mapsto \mu$
Term	ω	::=	$\top \mid c \mid x \mid \omega^{\mu} \mid @(\omega, \omega^{\mp})$
			$\mid \beta(\omega, \Upsilon, \omega) \mid \alpha(\omega, \omega \mapsto \omega)$
Variable context	Υ	::=	$\cdot \mid \Upsilon, \omega$ if $str(\omega)$ of the form x
Morphism	μ	::=	$id_T \mid i \mid v \mid \mu \bullet \mu$
Document reference	g	::=	URI, no query, no fragment
Module reference	T, v	::=	$g?\underline{T} \mid g?\underline{v}$
Symbol reference	c, i	::=	$T?\underline{c} \mid T?\underline{i}$
Assignment reference	a	::=	$v?\underline{c} \mid v?\underline{i}$
Local name	$\underline{T}, \underline{v}, \underline{c}, \underline{i}$	::=	$C^+[/C^+]^*$
Variable name	x	::=	C^+
Character	C	::=	pchar
	URI, pchar		see RFC 3986 [BLFM05]

Figure 6.7: The Grammar for Raw MMT Expressions

- **attributions** $\alpha(\omega_1, \omega_2 \mapsto \omega_3)$ to a term ω_1 with key ω_2 and value ω_3,
- **morphism applications** ω^{μ} of μ to ω,
- **special term** $\top$.

A term over T, may use the constant $T?\underline{c}$ to refer to a previously declared symbol $\underline{c}$. And if $\underline{i}$ is a previously declared structure instantiating S, and $\underline{c}$ is a constant declared in S, then T may use $T?\underline{i}/\underline{c}$ to refer to the imported constant. By concatenating structure names, any indirectly imported constant has a unique qualified name.

The attributions of OPENMATH mean that every term can carry a list of key-value pairs that are themselves OPENMATH objects. In particular, attributions are used to attach types to bound variables. In OPENMATH, the keys must be symbols, which we relax to obtain a more uniform syntax. Because OPENMATH specifies that nested attributions are equivalent to a single attribution, we can introduce attributions of multiple key-value pairs as abbreviations:

$$\alpha(\omega, \omega_1 \mapsto \omega_1', \ldots, \omega_n \mapsto \omega_n') := \alpha(\ldots(\alpha(\omega, \omega_1 \mapsto \omega_1'), \ldots), \omega_n \mapsto \omega_n')$$

We use the auxiliary function $str(\cdot)$ to strip toplevel attributions from terms, i.e.,

$$str(\alpha(\omega, \_)) = str(\omega) \qquad str(\omega) = \omega \text{ otherwise.}$$

	Atomic object	Composed object	Type	Checked relative to
Terms	constant	term	term	home theory
Morphisms	structure/view	morphism	domain	codomain

Figure 6.8: The Object Level

This is used in the grammar to make sure that only attributed variables and not arbitrary terms may occurs in the context bound in a binding.

The special term $\top$ is used for terms that are inaccessible because they refer to constants that have been hidden by a structure or view. $\top$ is also used to subsume hidings under assignments (see below).

Example 6.2 (Continued). The running example only contains the atomic terms given by symbols. Composed terms arise when types and axioms are covered. For example, the type of the inverse in a commutative group is $@(\to, \iota, \iota)$. Here $\to$ represents the function type constructor and ι the carrier set. These two constants are not declared in the example. Instead, we will add them later by giving `cgroup` a meta-theory, in which these symbols are declared. A more complicated term is the axiom for left-neutrality of the unit:

$$\omega_e \ := \ \beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(e?\texttt{monoid}?\texttt{comp}, e?\texttt{monoid}?\texttt{unit}, x), x)).$$

Here $\forall$ and $=$ are further constants that must be declared in the so far omitted meta-theory. The same applies to `oftype`, which is used to attribute the type ι to the bound variable x. We assume that the example is located in a document with URI e. Thus, for example, $e?\texttt{monoid}?\texttt{comp}$ is used to refer to the constant `comp` in the theory `monoid`.

Morphisms Morphisms are built up by compositions $\mu_1 \bullet \mu_2$ of structures i, views m, and the identity id_T of T. Here μ_1 is applied before μ_2, i.e., $\bullet$ is composition in diagram order. Morphisms are not used in OpenMath, which only has an informal notion of theories, namely the content dictionaries, and thus no need to talk about theory morphisms. A morphism application ω^μ takes a term ω over S and a morphism μ from S to T, and returns a term over T. Similarly, a morphism over S, e.g., a morphism μ' from R to S becomes a morphism over T by taking the composition $\mu' \bullet \mu$. The normalization given below will permit to eliminate all morphism applications.

Example 6.3 (Continued). In the running example, an example morphism is

$$\mu_e \ := \ e?\texttt{cgroup}?\texttt{mon} \bullet e?\texttt{v2}.$$

It has domain $e?\texttt{monoid}$ and codomain $e?\texttt{integers}$. The intended semantics of the term ${\omega_e}^{\mu_e}$ is that it yields the result of applying μ_e to ω_e, i.e.,

$$\beta(\forall, \alpha(x, \texttt{oftype} \mapsto \iota), @(=, @(+, 0, x), x)).$$

Here, we assume μ_e has no effect on those constants that are inherited from the meta-theory. We will make that more precise below.

6.2.1.2 The Symbol Level

We distinguish symbol declarations and assignments to symbols. **Declarations** are the constituents of theories: They introduce named objects, i.e., the **constants** and **structures**. Similarly, **assignments** are the constituents of links: A link from S to T can be defined by a sequence of assignments that instantiate constants or structures declared in S with terms or morphisms, respectively, over T. This yields four kinds of knowledge items which are listed in Fig. 6.9. Both constant and structure declarations are further subdivided as explained below.

	Declaration	Assignment
Terms	of a constant Con	to a constant $\underline{c} \mapsto \omega$
Morphisms	of a structure Str	to a structure $\underline{i} \mapsto \mu$

Figure 6.9: The Statement Level

Declarations There are two kinds of symbols:

- **Constant declarations** $\underline{c} : \tau := \delta$ declare a constant $\underline{c}$ of type τ with definition δ. Both the type and the definition are optional yielding four kinds of constant declarations. If both are given, then δ must have type τ. In order to unify these four kinds, we will sometimes write $\bot$ for an omitted type or definition.

- **Structure declarations** $\underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\}$ declare a structure $\underline{i}$ from the theory S defined by assignments σ. Such structures can have an optional meta-morphism μ (see below). Alternatively, structures may be introduced using an existing morphism: $\underline{i} : S := \mu$, which simply means that $\underline{i}$ serves as an abbreviation for μ; we call these structures **defined structures**. While the domain of a structure is always given explicitly (in the style of a type), the codomain is always the theory in which the structure is declared. Consequently, if $\underline{i} : S := \mu$ is declared in T, μ must be a morphism from S to T.

In well-formed theory bodies (see Sect. 6.3), the declared or imported names must be unique.

Assignments Parallel to the declarations, there are two kinds of assignments that can be used to define a link m:

- **Assignments to constants** of the form $\underline{c} \mapsto \omega$ express that m maps the constant $\underline{c}$ of S to the term ω over T. Assignments of the form $\underline{c} \mapsto \top$ express that the constant $\underline{c}$ is **hidden**, i.e., m is undefined for $\underline{c}$.

- **Assignments to structures** of the form $\underline{i} \mapsto \mu$ for a structure $\underline{i}$ declared in S and a morphism μ over (i.e., into) T express that m maps the structure $\underline{i}$ of S to μ. This results in the commuting triangle $S?\underline{i} \bullet m = \mu$.

Both kinds of assignments must type-check. For a link m with domain S and codomain T defined by among others an assignment $\underline{c} \mapsto \omega$, the term ω must type-check against τ^m where τ is the type of $\underline{c}$ declared in S. This ensures that typing is preserved along links. For an assignment $\underline{i} \mapsto \mu$ where $\underline{i}$ is a structure over S of type R, type-checking means that μ must be a morphism from R to T.

Virtual Symbols Intuitively, the semantics of a structure $\underline{i}$ with domain S declared in T is that all symbols of S are imported into T. For example, if S contains a symbol $\underline{s}$, then $\underline{i}/\underline{s}$ is available as a symbol in T. In other words, the slash is used as the operator that dereferences structures. Another way to say it is that structures create virtual or induced symbols. This is significant because these virtual symbols and their properties must be inferred, and this is non-trivial because it is subject to the translations given by the structure. While these induced symbols are easily known to systems built for a particular formalism, they present great difficulties for generic knowledge management services.

Similarly, every assignment to a structure induces virtual assignments to constants. Continuing the above example, if a link with domain T contains an assignment to $\underline{i}$, this induces assignments to the imported symbols $\underline{i}/\underline{s}$. Furthermore, assignments may be **deep** in the following sense: If $\underline{c}$ is a constant of S, a link with domain T may also contain assignments to the

virtual constant $\underline{i}/\underline{c}$. Of course, this could lead to clashes if a link contains assignments for both $\underline{i}$ and $\underline{i}/\underline{c}$; links with such clashes are not well-formed.

Example 6.4 (Continued). The symbol declarations in the theory `cgroup` are written formally like this:

$$\texttt{inv} : @(\rightarrow, \iota, \iota) \quad \text{and} \quad \texttt{mon} : e?\texttt{monoid} := \{\}.$$

The latter induces the virtual symbols $e?\texttt{cgroup}?\texttt{mon/comp}$ and $e?\texttt{cgroup}?\texttt{mon/unit}$.

Using an assignment to a structure, the assignments of the view `v2` look like this:

$$\texttt{inv} \mapsto e?\texttt{integers}? - \quad \text{and} \quad \texttt{mon} \mapsto e?\texttt{v1}.$$

The latter induces virtual assignments for the virtual symbols $e?\texttt{cgroup}?\texttt{mon/comp}$ as well as $e?\texttt{cgroup}?\texttt{mon/unit}$. For example, $e?\texttt{cgroup}?\texttt{mon/comp}$ is mapped to $e?\texttt{monoid}?\texttt{comp}^{e?\texttt{v1}}$.

The alternative formulation of the view `v2` arises if two deep assignments to the virtual constants are used instead of the assignment to the structure `mon`:

$$\texttt{mon/comp} \mapsto e?\texttt{integers}? + \quad \text{and} \quad \texttt{mon/unit} \mapsto e?\texttt{integers}?0$$

6.2.1.3 The Module Level

The module level consists of two kinds of declarations: theory and view declarations.

- **Theory declarations** $\underline{T} \stackrel{[M]}{:=} \{\vartheta\}$ declare a theory $\underline{T}$ defined by a list of symbol declarations ϑ, which we call the body of $\underline{T}$. Theories have an optional meta-theory M.
- **View declarations** $\underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma\}$ declare a link $\underline{v}$ from S to T defined by a list of assignments σ. If S has a meta-theory M, a meta-morphism μ from M to T must be provided. Just like structures, views may also be defined by an existing morphism: $\underline{v} : S \to T := \mu$.

Meta-Theories Above, we have already mentioned that theories may have meta-theories and that links may have meta-morphisms. Meta-theories provide a second dimension in the graph induced by theories and links. If M is the meta-theory of T, then there is a special structure instantiating M in T, which we denote by $T?...$ M provides the syntactic material that T can use to define the semantics of its symbols: T can refer to a symbol $\underline{s}$ of M by $T?../\underline{s}$. While meta-theories could in principle be replaced with structures altogether, it is advantageous to make them explicit because the conceptual distinction pervades mathematical discourse. For example, systems can use the meta-theory to determine whether they understand a specific theory they are provided as input.

Because theories S with meta-theory M implicitly import all symbols of M, a link from S to T must provide assignments for these symbols as well. This is the role of the meta-morphism: Every link from S to T must provide a meta-morphism, which must be a morphism from M to T. Defined structures or views with definition μ do not need a meta-morphism because a meta-morphism is already implied by the meta-morphisms of the links occurring in μ.

Example 6.5 (Continued). An MMT theory for the logical framework LF could be declared like this

$$\texttt{lf} := \{\texttt{type}, \texttt{funtype}, \ldots\}$$

where we only list the constants that are relevant for our running example. If this theory is located in a document with URI m, we can declare a theory for first-order logic in a document with URI f like this:

$$\texttt{fol} \stackrel{m?\texttt{lf}}{:=} \{\texttt{i} : ??../\texttt{type}, \texttt{o} : ??../\texttt{type}, \texttt{equal} : @(??../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}, ??\texttt{o}), \ldots\}$$

Here we already use relative names (see Sect. 8.1.2) in order to keep the notation readable: Every name of the form ??$\underline{s}$ is relative to the enclosing theory: For example, ??`i` resolves to f?`fol`?`i`. Furthermore, we use the special structure name .. to refer to those constants inherited from the meta-theory. Again we restrict ourselves to a few constant declarations: types `i` and `o` for terms and formulas and the equality operation that takes two terms and returns a formula.

Then the theories `monoid`, `cgroup`, and `ring` can be declared using f?`fol` as their meta-theory. For example, the declaration of the theory `cgroup` finally looks like this:

$$\texttt{cgroup} \overset{f?\texttt{fol}}{:=} \left\{ \texttt{inv} : @(??../../\texttt{funtype}, ??\texttt{i}, ??\texttt{i}),\ \texttt{inv} : e?\texttt{monoid} \overset{e?\texttt{cgroup}?..}{:=} \{\} \right\}$$

Here ../../`funtype` refers to the function type constant declared in the meta-theory of the meta-theory. And the structure `mon` must have a meta-morphism with domain f?`fol` and codomain e?`cgroup`. This is trivial because the meta-theory of e?`cgroup` is also f?`fol`: The meta-morphism is simply the implicit structure e?`cgroup`?.. via which e?`cgroup` inherits from f?`fol`.

A more complicated meta-morphism must be given in the view `v1` if we assume that the meta-theory of `integers` is some other theory, i.e., a representation of set theory.

Structures and Views Both structures and views from S to T are defined by a list of assignments σ that assigns T-objects to the symbols declared in S. And both induce theory morphisms from S to T that permit to map all objects over S to objects over T. The major difference between structures and views is that a view only relates two fixed theories without changing either one. On the other hand, structures from S to T occur within T because they change the theory T. Structures have **definitional flavor**, i.e., the symbols of S are imported into T. In particular, if σ contains no assignment for a constant $\underline{c}$, this is equivalent to copying (and translating) the declaration of $\underline{c}$ from S to T. If σ does provide an assignment $\underline{c} \mapsto \omega$, the declaration is also copied, but in addition the imported constant receives ω as its definiens.

Views, on the other hand, have **theorem flavor**: σ *must* provide assignments for the symbols of S. If a constant $\underline{c} : \tau$ represents an axiom stating τ, giving an assignment $\underline{c} \mapsto \pi$ means that π is a proof of the translation of τ.

Therefore, the assignments defining a structure may be (and typically are) partial whereas a view should be total. This leads to a crucial technical difficulty in the treatment of structures: Contrary to views from S to T, the assignments by themselves in a structure from S to T do not induce a theory morphism from S to T — only by declaring the structure do the virtual symbols become available in T that serve as the images of (some of) the symbols of S. This is unsatisfactory because it makes it harder to unify the concepts of structures and views.

Therefore, we admit **partial views** as well. As it turns out, this is not only possible, but indeed desirable. A typical scenario when working with views is that some of the specific assignments making up the view constitute proof obligations and must be found by costly procedures. Therefore, it is reasonable to represent partial views, namely views where some proof obligations have already been discharged whereas others remain open. Thus, we use hiding to obtain a semantics for partial views: All constants for which a view does not provide an assignment are implicitly hidden, i.e., $\top$ is assigned to them.

If a link m from S to T is applied to an S-constant that is hidden, there are two cases: If the hidden symbol has a definition in S, it is replaced by this definition before applying the link. If it does not have a definition, it is mapped to $\top$. Hiding is strict: If a subterm is mapped to $\top$, then so is the whole term. In that case, we speak of hidden terms.

6.2.1.4 The Document Level

Document declarations are of the form $g := \{\gamma\}$ where γ is a document body and g is a URI identifying the document. The meaning of a document declaration is that γ is accessible via

the name g. Since g is a URI, it is not necessarily only the name, but can also be the primary location of γ. By forming lists of documents, we obtain **libraries**, which represent mathematical knowledge bases. Special cases of libraries are single self-contained documents and the internet seen as a library of MMT documents.

Documents provide the border between formal and informal treatment: How documents are stored, copied, cached, mirrored, transferred, and combined is subject to knowledge management services that may or may not take the mathematical semantics of the document bodies in the processed documents into account. For example, libraries may be implemented as web servers, file systems, databases, or any combination of these. The only important thing is that they provide the query interface described below.

Theory graphs are the central notion of MMT. The theory graph is a directed acyclic multigraph. The nodes are all theories of all documents in the library. And similarly, the edges are the structures and views of all documents. Then theory morphisms can be recovered as the paths in the theory graph.

6.2.2 Querying a Library

MMT is designed to scale to a mathematical web. This means that we define access functions to MMT libraries that have the form of HTTP requests to a RESTful web server [Fie00]. Specifically, there is a lookup function that takes a library Λ and a URI U as arguments and returns an MMT fragment $\Lambda(U)$. This specifies the behavior of a web server hosting an MMT library in response to GET requests. Furthermore, all possible changes to an MMT library can be formulated as POST, PUT, and DELETE requests that add, change, and delete knowledge items, respectively.

It is non-trivial to add such a RESTful interface to formal systems a posteriori. It requires the rigorous use of URIs as identifiers for all knowledge items that can be the object of a change. And it requires to degrade gracefully if the documents in a library are not in the same main memory or on the same machine. In large applications, it is even desirable to load only the relevant parts of a document into the main memory and request further fragments on demand from a low-level database service. In MMT, web-scalability is built into the core: All operations on a library Λ including the definition of the semantics only depend on the lookup function $\Lambda(-)$ and not on Λ itself. In particular, two libraries that respond in the same way to lookup requests are indistinguishable by design. Therefore, MMT scales well to web-based scenarios.

As a motivating example, we trace the lookup of the name $g?\underline{T}?\underline{i_1}/\ldots/\underline{i_n}/\underline{s}$. First, g is looked up returning a document. Within this document, a theory with name $\underline{T}$ is looked up returning a theory $g?\underline{T}$, let us call it S_0. Then a structure named $\underline{i_1}$ is looked up in this theory's body. If this structure has domain S_1, a structure named $\underline{i_2}$ is looked up in the body of S_1, and so on. This means to successively look up the structure $S_{r-1}?\underline{i_r}$, i.e., $g?\underline{T}?\underline{i_1}/\ldots/\underline{i_r}$ with domain S_r for $r = 1,\ldots,n$. Finally, the lookup algorithm finds the symbol $S_n?\underline{s}$.

This symbol is not yet the result of the lookup — rather, it must be translated back along the traversed structures. In the example, it must be translated along the morphism $\mu := S_{n-1}?\underline{i_n} \bullet \ldots \bullet S_0?\underline{i_1}$. For a constant declaration $\underline{s} : \tau := \delta$, this means to apply μ to τ and δ ultimately returning the constant $\underline{i_1}/\ldots/\underline{i_n}/\underline{s} : \tau^\mu := \delta^\mu$. And for a structure declaration $\underline{s} : R := \{\sigma\}$, this means to take the composition $S_n?\underline{s} \bullet \mu$ and thus to return $\underline{i_1}/\ldots/\underline{i_n}/\underline{s} : R := S_n?\underline{s} \bullet \mu$.

In the following we will give the formal definition of the lookup by specifying the functions listed in Fig. 6.10. These functions define the domain and the result of the lookup, grouped according to the level of the looked up knowledge item.

Document and Module Level The lookup of a document in a library Λ is straightforward:

$$\Lambda(g) := \gamma \qquad \text{if } g := \{\gamma\} \text{ in } \Lambda$$

And we define $Doc(\Lambda)$ as the set of document names g for which $\Lambda(g)$ is defined.

$Doc(\Lambda)$	the set of valid document names over Λ
$\Lambda(g)$	the lookup of $g \in Doc(\Lambda)$
$Thy(\Lambda)$	the set of valid theory names over Λ
$Mor(\Lambda)$	the set of valid link names over Λ
$\Lambda(x)$	the lookup of $x \in Thy(\Lambda) \cup Mor(\Lambda)$
$Con^\Lambda(T)$	the set of valid unqualified constant names over $T \in Thy(\Lambda)$
$Str^\Lambda(T)$	the set of valid unqualified structure names over $T \in Thy(\Lambda)$
$Sym^\Lambda(T)$	union of $Con^\Lambda(T)$ and $Str^\Lambda(T)$
$\Lambda^T(\underline{s})$	the lookup of $\underline{s} \in Sym^\Lambda(T)$
$Sym^\Lambda(m)$	the set of unqualified names instantiated by $m \in Mor(\Lambda)$
$\Lambda^m(\underline{s})$	the lookup of $\underline{s} \in Sym^\Lambda(m)$

Figure 6.10: Lookup Functions for a Library Λ

Similarly, $\Lambda(-)$ is a partial map from module names to module declarations. To conserve space, we will adopt a more compact notation for this. For theory names T, $\Lambda(T)$ is a pair $([M], \vartheta)$ of the meta-theory and the theory body occurring in the declaration of T. We also put $meta^\Lambda(T) := M$ and $body^\Lambda(T) := \vartheta$. For view names v, $\Lambda(v)$ is a 4-tuple $(S, T, [\mu], B)$ of domain, codomain, meta-morphism, and body, and we put $dom^\Lambda(m) := S$, $cod^\Lambda(m) := T$, $meta^\Lambda(m) := \mu$, $body^\Lambda(m) := B$. Here, B is a list of assignments or a morphism depending on how the view is defined; in the latter case μ is the meta-morphism implicitly given by B.

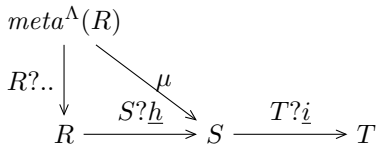
We unify the cases with and without meta-theories by using square brackets. If the optional M or μ is missing, we leave $meta^\Lambda(T)$ or $meta^\Lambda(m)$ undefined and write $-$.

The double nature of structures as being both links and symbols is reflected in the fact that structures can also be addressed by module names and returned as the result of module level lookups. If there is a structure declaration $\underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\}$ in a theory $g?\underline{T}$, then $\Lambda(g?\underline{T}/\underline{i})$ returns $(S, g?\underline{T}, [\mu], \sigma)$ and similarly for defined structures.

Formally, we can define this as follows:

$$
\begin{array}{lll}
\Lambda(g?\underline{T}) & := ([M], \vartheta) & \text{if } \underline{T} \stackrel{[M]}{:=} \{\vartheta\} \text{ in } \Lambda(g) \\
\Lambda(g?\underline{v}) & := (S, T, [\mu], \sigma) & \text{if } \underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma\} \text{ in } \Lambda(g) \\
\Lambda(g?\underline{v}) & := (S, T, meta^\Lambda(\mu), \mu) & \text{if } \underline{v} : S \to T := \mu \text{ in } \Lambda(g) \\
\Lambda(g?\underline{T}/..) & := (meta^\Lambda(g?\underline{T}), g?\underline{T}, -, \cdot) & \\
\Lambda(g?\underline{T}/\underline{i}) & := (S, g?\underline{T}, [\mu], \sigma) & \text{if } \underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\} \text{ in } body^\Lambda(g?\underline{T}) \\
\Lambda(g?\underline{T}/\underline{i}) & := (S, g?\underline{T}, meta^\Lambda(\mu), \mu) & \text{if } \underline{i} : S := \mu \text{ in } body^\Lambda(g?\underline{T}) \\
\Lambda(g?\underline{T}/\underline{i}/\underline{h}) & := (R, T, [\mu \bullet g?\underline{T}?\underline{i}], S?\underline{h} \bullet g?\underline{T}?\underline{i}) & \text{if } dom^\Lambda(g?\underline{T}/\underline{i}) = S,\ \Lambda(S?\underline{h}) = (R, S, \mu, \_)
\end{array}
$$

In the order listed, the cases look up theories, views defined by assignments and morphisms, meta-imports, and declared, defined, and imported structures. Meta-imports are implicit structures with the special name ..: This is a structure instantiating the meta-theory of T, it is used so that T can access the symbols of its meta-theory. The case of imported structures is the most interesting one: The theory T imports from the theory S via a structure $\underline{i}$, and S imports from the theory R via a structure $\underline{h}$. This produces an imported structure $\underline{i}/\underline{h}$ in the theory T. Its semantics is that of the composition $S?\underline{h} \bullet T?\underline{i}$ as shown in the following diagram.



Now we could use the lists of assignments of $body^\Lambda(T?\underline{i})$ and $body^\Lambda(S?\underline{h})$ and compute their composition to obtain $body^\Lambda(g?\underline{T}/\underline{i}/\underline{h})$. However, by doing so we would destroy the modular structure. Therefore, we use the value $S?\underline{h} \bullet T?\underline{i}$ in order to defer the computation of the composition as long as reasonable and to give implementations control over when to compute it. A further advantage is that the lookup functions become significantly easier to implement.

For the example from the beginning of this section, the lookup of $g?\underline{T}/\underline{i_1}/\ldots/\underline{i_n}$ would yield

$$(S_n, S_0, -, S_{n-1}?\underline{i_n} \bullet \ldots \bullet S_0?\underline{i_1}).$$

We denote by $Thy(\Lambda)$ the set of module names for which the lookup returns a theory declaration, and by $Mor(\Lambda)$ the set of module names for which the lookup returns a link declaration.

Finally, we extend the functions $meta^\Lambda(-)$, $dom^\Lambda(-)$, and $cod^\Lambda(-)$ to morphisms. For the latter two, this is done in the obvious way. $meta^\Lambda(-)$ is defined by

$$\begin{array}{ll} meta^\Lambda(id_{g?\underline{T}}) := g?\underline{T}/.. & \text{if } meta^\Lambda(g?\underline{T}) \text{ defined} \\ meta^\Lambda(m \bullet \mu) := meta^\Lambda(m) \bullet \mu & \end{array}$$

Thus, we have that if μ is a morphism from S to T and $meta^\Lambda(S) = M$, then $meta^\Lambda(\mu)$ is a morphism from M to T.

Symbol Level Intuitively, the lookup of a symbol name $T?\underline{s}$ returns the declaration of that symbol. To emphasize that this lookup is done in a theory, we will usually write $\Lambda^T(\underline{s})$ instead of $\Lambda(T?\underline{s})$. Similarly, the lookup of an assignment name $v?\underline{s}$ returns the assignment to $\underline{s}$ by the view v, and we write $\Lambda^v(\underline{s})$ instead of $\Lambda(v?\underline{s})$.

The above-mentioned double nature of structures is crucial here. A structure $\underline{i}$ declared in $g?\underline{T}$ can be accessed both as $\Lambda(g?\underline{T}?\underline{i})$ and as $\Lambda(g?\underline{T}/\underline{i})$ — the former emphasizes the symbol, the latter the link nature of a structure. However, both names differ in the way in which they can be further dereferenced: The lookup $\Lambda(g?\underline{T}?\underline{i}/\underline{s})$ returns the imported symbol $\underline{i}/\underline{s}$ of the theory $g?\underline{T}$, whereas $\Lambda(g?\underline{T}/\underline{i}?\underline{s})$ returns the assignment to the symbol $\underline{s}$ by the structure $g?\underline{T}/\underline{i}$. In particular, the former is always defined if such a structure is present, whereas the latter is only defined if $g?\underline{T}/\underline{i}$ provides an assignment to $\underline{s}$.

Again we use a more compact notation for the declarations and assignments returned by the lookup functions. If $\Lambda^T(\underline{s})$ returns a constant declaration $\underline{c} : \tau := \delta$, we simply write $\Lambda^T(\underline{s}) = (\tau, \delta)$. Similarly, if it returns a structure, we use the same abbreviation as above. And if the lookup in a link m returns an assignment $\underline{s} \mapsto \omega$ or $\underline{s} \mapsto \mu$, we write $\Lambda^m(\underline{s}) = \omega$ and $\Lambda^m(\underline{s}) = \mu$, respectively. We use $\bot$ if the type or definition of a constant is omitted or if no assignment is present in a link.

Lookup in Theories For the lookup of constant names in a theory, we first give an auxiliary definition that does not take hiding into account. Assume a theory $T = g?\underline{T} \in Thy(\Lambda)$ with $body^\Lambda(T) = \vartheta$. Then $\Lambda^T_*(\underline{c})$ is defined by:

$$\begin{array}{lll} \Lambda^T_*(\underline{c}) & := (\tau, \delta) & \text{if } \underline{c} : \tau := \delta \text{ in } \vartheta \\ \Lambda^T_*(\underline{i}/\underline{c}) & := \left\{ \begin{array}{ll} (\tau^{T?\underline{i}}, \delta') & \text{if } \delta' \neq \bot \\ (\tau^{T?\underline{i}}, \delta^{T?\underline{i}}) & \text{otherwise} \end{array} \right\} & S = dom^\Lambda(g?\underline{T}/\underline{i}),\ \Lambda^S_*(\underline{c}) = (\tau, \delta),\ \delta' = \Lambda^{g?\underline{T}/\underline{i}}(\underline{c}) \\ \Lambda^T_*(../\underline{c}) & := (\tau^{T?..}, \delta^{T?..}) & \text{if } M = meta^\Lambda(T),\ \Lambda^M_*(\underline{c}) = (\tau, \delta) \end{array}$$

The first case is trivial: It looks up declared constants. For imported constants (second case), two subcases must be distinguished: If there is an assignment for $\underline{c}$ in $\underline{i}$ — i.e., $\delta' \neq \bot$ —, that assignment is used as the definition (first subcase). Otherwise, the definition of $\underline{c}$ is translated along $\underline{i}$ (second subcase). In both cases, the type is translated. Here we can avoid a special treatment of the cases $\delta = \bot$ and $\tau = \bot$ by agreeing that $\bot^\mu$ is just another way to say $\bot$.

To handle hiding, we have to hide all constants that depend on hidden constants. This means the actual lookup function $\Lambda^T(-)$ agrees with $\Lambda^T_*(-)$ except that it is defined only for a smaller set of names. The recursive check whether a term is hidden is handled by the normalization function (see Sect. 6.2.3) which maps ω to $\overline{\omega}$: If $\overline{\omega} = \top$, then ω is a hidden term. Therefore, we finally define

$$\Lambda^T(\underline{c}) := \Lambda^T_*(\underline{c}) \quad \text{if} \quad \Lambda^T_*(\underline{c}) = (\tau, \delta),\ \overline{\tau} \neq \top,\ \overline{\delta} \neq \top.$$

Finally, the lookup of structure names in theories is defined by appealing to the module level lookup above: $\Lambda(g?\underline{T}?\underline{i}) := \Lambda(g?\underline{T}/\underline{i})$.

Lookup in Links Now we define $\Lambda^m(\underline{c})$ for $m \in Mor(\Lambda)$, $dom^\Lambda(m) = S$, and constant names $\underline{c} \in Con^\Lambda(S)$. We put $body^\Lambda(m) =: B$ which can be either a list of assignments σ or a morphism μ. Then we define:

$$\begin{array}{lll}
\Lambda^m(\underline{c}) & := (S?\underline{c})^\mu & \text{if } B = \mu \\
\Lambda^m(\underline{c}) & := \omega & \text{if } \underline{c} \mapsto \omega \text{ in } B = \sigma \\
\Lambda^m(../\underline{c}) & := (M?\underline{c})^\mu & \text{if } meta^\Lambda(S) = M,\ meta^\Lambda(m) = \mu \\
\Lambda^m(\underline{i}/\underline{c}) & := (R?\underline{c})^\mu & \text{if } \underline{i} \mapsto \mu \text{ in } B = \sigma,\ dom^\Lambda(S?\underline{i}) = R
\end{array}$$

The intuitions are as follows. If $B = \mu$, then μ is applied to $\underline{c}$ (first case). If $B = \sigma$ and there is an assignment for $\underline{c}$ in σ, it is returned (second case). If S has a meta-theory, then the meta-morphism is used to translate constants of the meta-theory (third case). The most interesting case arises when the argument to $\Lambda^m(-)$ is of the form $\underline{i}/\underline{c}$ and there is an assignment $\underline{i} \mapsto \mu$ for $\underline{i}$: Then μ is applied to $R?\underline{c}$ where R is the domain of $\underline{i}$. To be well-formed (see Sect. 6.3), the second and the fourth case must be mutually exclusive.

The lookup of structure names in links is defined accordingly.

Example 6.6 (Continued). If Λ is a library containing the three documents with URIs m, f, and e from the running example, we obtain the following lookup results:

- $\Lambda(e?\texttt{monoid}) = (f?\texttt{fol},\ \vartheta)$ where ϑ contains the declarations for `comp` and `unit`,
- $\Lambda(e?\texttt{cgroup/mon}) = (e?\texttt{monoid},\ e?\texttt{cgroup},\ e?\texttt{cgroup}?..,\ \cdot)$,
  i.e., $e?\texttt{cgroup/mon}$ is a morphism from $e?\texttt{monoid}$ to $e?\texttt{cgroup}$ with the meta-morphism $e?\texttt{cgroup}?..$ and without any assignments,
- $\Lambda^{e?\texttt{monoid}}(\texttt{unit}) = (e?\texttt{monoid}?../\texttt{i},\ \bot)$,
  i.e., the theory `monoid` has a constant `unit` with type $e?\texttt{monoid}?../\texttt{i}$ and no definition,
- $\Lambda^{e?\texttt{cgroup}}(\texttt{mon/unit}) = (e?\texttt{monoid}?../\texttt{i}^{e?\texttt{cgroup}?\texttt{mon}},\ \bot)$,
  i.e., the type of the virtual constant `mon/unit` arises by translating the type from the source theory along the importing structure,
- $\Lambda^{e?\texttt{cgroup/mon}}(\texttt{unit}) = \bot$,
  i.e., the lookup is undefined because the structure $e?\texttt{cgroup/mon}$ does not have an assignment for `unit`,
- the lookup $\Lambda^{e?\texttt{v2}}(\texttt{mon/unit})$ yields $e?\texttt{integers}?0$ if the variant with the deep assignment $\texttt{mon/unit} \mapsto e?\texttt{integers}?0$ is used to define `v2`, and $e?\texttt{monoid}?\texttt{unit}^{e?\texttt{v1}}$ if the variant with the structure assignment $\texttt{mon} \mapsto e?\texttt{v1}$ is used.

Uniqueness of Names To avoid confusions, we already mention Lem. 6.12 from Sect. 6.3.6, which states that for all well-formed libraries (see Sect. 6.3) and all MMT names the lookup is either undefined or uniquely determined. If the lookup is applied to an ill-formed library, we rule for the sake of definiteness that the left-most possible name resolution be preferred if multiple resolutions are possible.

In order to ensure this name uniqueness, the type system will use some auxiliary functions. Firstly, $Nam^-(-)$ is the set of unqualified names that are already declared in a scope. $Nam^\Lambda(g)$ is the set of theory or view names declared in $\Lambda(g)$. $Nam^\Lambda(T)$ is the set of symbol names declared in $body^\Lambda(T)$ (excluding virtual symbols). And for a link m, $Nam^\Lambda(m)$ is the set of names for which there are assignments in $body^\Lambda(m)$ (excluding virtual assignments).

And secondly, the prefixes of an unqualified name are its initial /-terminated segments:

$$pref(n_1/\ldots/n_r) := \{n_1,\ n_1/n_2,\ \ldots, n_1/\ldots/n_r\}.$$

We will sometimes apply $pref(-)$ to a set $Nam^-(-)$ to obtain the set of all prefixes of names declared in a scope.

6.2.3 Normalization

Assume a library Λ and a term ω over Λ. We write $\overline{\omega}^\Lambda$ for the normal form of ω, and we will omit Λ if it is clear from the context. Normalization eliminates all morphism applications, expands all definitions, and enforces the strictness of hiding (A term with a hidden subterm is also hidden.). Thus, the normalization provides the MMT-specific part of the axiomatization of equality. It is also an important theoretical result that the MMT-concepts can be eliminated.

For a fixed library Λ, we give the algorithm by structural induction on ω, grouping the cases in three groups. The normalization results in an error if ω contains references to symbols for which the lookup is not defined.

1. The algorithm works from the inside of the term to the outside: First all subterms are normalized, and if one of them yields $\top$, the whole term normalizes to $\top$. If a constant, has a definition, it is expanded.

$$\begin{array}{lll}
\overline{\top} & := \top & \\
\overline{x} & := x & \\
\overline{T?\underline{c}} & := \begin{cases} \text{undefined} & \text{if } \Lambda^T(\underline{c}) \text{ undefined} \\ \overline{\delta} & \text{if } \Lambda^T(\underline{c}) = (_, \delta),\ \delta \neq \bot \\ T?\underline{c} & \text{otherwise} \end{cases} \\
\overline{@(\omega_1, \ldots, \omega_n)} & := \begin{cases} @(\overline{\omega_1}, \ldots, \overline{\omega_n}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i \\ \top & \text{otherwise} \end{cases} \\
\overline{\beta(\omega_1, x_1, \ldots, x_n, \omega_2)} & := \begin{cases} \beta(\overline{\omega_1}, \overline{x_1}, \ldots, \overline{x_n}, \overline{\omega_2}) & \text{if } \overline{\omega_i} \neq \top \text{ and } \overline{x_i} \neq \top \text{ for all } i \\ \top & \text{otherwise} \end{cases} \\
\overline{\alpha(\omega_1, \omega_2 \mapsto \omega_3)} & := \begin{cases} \alpha(\overline{\omega_1}, \overline{\omega_2} \mapsto \overline{\omega_3}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i \\ \top & \text{otherwise} \end{cases}
\end{array}$$

2. The case of morphism applications ω^μ is defined by two subinductions: first on the structure of μ, where all links in μ are applied separately, starting from the inside; and then for a single link m on the structure of ω.

$$\begin{array}{ll}
\overline{\omega^{id_T}} & := \overline{\omega} \\
\overline{\omega^{\mu \bullet \mu'}} & := \overline{\overline{\omega^\mu}^{\mu'}} \\
\overline{\top^m} & := \top \\
\overline{x^m} & := x \\
\overline{@(\omega_1, \ldots, \omega_3)^m} & := \overline{@({\omega_1}^m, \ldots, {\omega_3}^m)} \\
\overline{\beta(\omega_1, \Upsilon, \omega_2)^m} & := \overline{\beta({\omega_1}^m, \Upsilon^m, {\omega_2}^m)} \\
\overline{\alpha(\omega_1, \omega_2 \mapsto \omega_3)^m} & := \overline{\alpha({\omega_1}^m, {\omega_2}^m \mapsto {\omega_3}^m)}
\end{array}$$

Here Υ^m abbreviates component-wise morphism application to a list of attributed variables.

3. Finally, the most interesting subcase is the application of a link m to a constant $c = S?\underline{c}$. Let $\Lambda^S(\underline{c}) := (\_, \delta)$ and $\Lambda^m(\underline{c}) = \delta'$. (If the former lookups is undefined or if the lookup $\Lambda(m)$ is undefined, the normalization remains undefined.) Three cases are distinguished:

   - If $\delta \neq \bot$, then c has a definiens that is expanded before m is applied. This has two reasons: The image of c under m is not provided by an assignment in m, instead it is induced by the application of m to δ. Furthermore, if c is hidden by m, then c must be eliminated before applying m.
   - If $\delta = \bot$, then $\underline{c}$ is undefined. If m provides an assignment for c, i.e., $\delta' \neq \bot$, then c normalizes to $\overline{\delta'}$.
   - If $\delta = \delta' = \bot$, we distinguish two subcases:
     - If m refers to a structure, e.g., $m = T?\underline{i}$, the intended semantics of partiality is that $\underline{c}$ is imported by m. Therefore, c^m normalizes to the name of the imported constant: $T?\underline{i}/\underline{c}$. (Actually, the normalization is called one more time on $T?\underline{i}/\underline{c}$ to make sure that that name has a defined lookup.)
     - If m is a view, the intended semantics of partiality is that $\underline{c}$ is hidden. Therefore, c^m normalizes to $\top$.

   Formally, we write this as:

$$\overline{c^m} := \begin{cases} \overline{\delta^m} & \text{if } \delta \neq \bot \\ \overline{\delta'} & \text{if } \delta = \bot,\ \delta' \neq \bot \\ \left\{ \begin{array}{ll} \overline{T?\underline{i}/\underline{c}} & \text{if } m \text{ structure} \\ \top & \text{if } m \text{ view} \end{array} \right\} & \text{if } \delta = \delta' = \bot \end{cases}$$

In implementations, it is reasonable and easy to avoid the expansion of definitions in the first group of cases. We do it here in order to formalize the MMT-semantics of definitions.

6.3 Well-formed MMT Expressions

In this section we present an inference system to define the **well-formed** or **valid** MMT expressions. The organization of the inference system is top-down. That means there is one primary judgment $\rhd \Lambda$ for well-formed libraries. All other judgments are secondary and axiomatize how well-formed libraries can be extended with documents, modules, symbols, and assignments can be added to a library (see Fig. 6.11). All such extensions occur in the right-most positions. Thus, the inference system can be seen as the specification of an MMT parser. In particular, all information in Λ is processed in one left-to-right pass over a library.

The remaining judgments in Fig. 6.11 define well-formed terms and morphisms relative to a library and a home theory or to domain and codomain, respectively. Terms are also relative to a variable context, which we omit if it is empty. The two judgments for equality and typing of terms are special because MMT is parametric in them: Their definition is given by a foundation Φ, which is provided externally so that MMT can be instantiated with any specific non-modular object language. Therefore, all judgments from Fig. 6.11 are actually parametric in Φ. But since Φ is always fixed, we omit it from the notation. The details of Φ are given in Sect. 6.3.4.2. We will explain the auxiliary judgments, given in Fig. 6.12, when they become relevant.

We will introduce the rules for the judgments in a mutual induction in the next sections according to Fig. 6.11.

Judgment	Intuition	Section
$\triangleright \Lambda$	Λ is a well-formed library.	6.3.2
$\Lambda \triangleright Doc$	The document Doc can be added at the end of Λ.	6.3.2
$\Lambda \triangleright Mod$	The module Mod can be added at the end of the last document of Λ.	6.3.2
$\Lambda \triangleright Sym$	The symbol Sym can be added at the end of the last theory of Λ.	6.3.3
$\Lambda \triangleright Ass$	The assignment Ass can be added to the link at the end of Λ.	6.3.3
$\Lambda; \Upsilon \triangleright_T \omega$	ω is a structurally well-formed term over Λ with variables from Υ and with home theory T.	6.3.4.1
$\Lambda \triangleright \mu : S \to T$	μ is a well-formed morphism from S to T.	6.3.4.3
$\Lambda \triangleright_T \omega \equiv \omega'$	The well-formed terms ω and ω' are equal over Λ and T.	6.3.4.2
$\Lambda \triangleright_T \omega : \omega'$	ω is a well-formed term of well-formed type ω' over Λ and T.	6.3.4.2

Figure 6.11: Main Judgments of MMT

Judgment	Intuition	Section
$\Lambda \triangleright^g n$ new	The module name n can be added to the document g.	6.3.2
$\Lambda \triangleright^T \underline{s}$ new	The symbol name $\underline{s}$ can be added to the theory T.	6.3.3
$\Lambda \triangleright^m \underline{s}$ new	An assignment for the symbol name $\underline{s}$ can be added to the link m.	6.3.3
$\Lambda \triangleright_T \omega \mid^m \omega'$	A constant with definiens ω can be mapped to ω' by the link m.	6.3.3
$\Lambda \triangleright_T i \mid^m \mu : R$	Structure i with domain R can be mapped to μ by the morphism m.	6.3.3

Figure 6.12: Auxiliary Judgments of MMT

6.3.1 Adding Knowledge Items to Libraries

Our inference system is such that the initial segment of every document, theory, view, or structure is already part of the library as soon as it is checked: First, an empty document, theory, view, or structure is added to the library, and then it is extended step by step. This is possible because later document fragments can never invalidate an already checked initial segment.

In order to write these extensions with a more compact notation, we introduce the abbreviation $\Lambda + X$ for the operation of adding the knowledge item X in the right-most position of the library Λ. Formally, we write for a document declaration Doc

$$\Lambda + \ Doc \quad := \quad \Lambda, \ Doc,$$

for a module Mod

$$\Lambda, \ g := \{\gamma\} \ + \ Mod \quad := \quad \Lambda, \ g := \{\gamma, \ Mod\},$$

for a symbol Sym

$$\Lambda, \ g := \left\{\gamma, \ \underline{T} \overset{[M]}{:=} \{\vartheta\}\right\} \ + \ Sym \quad := \quad \Lambda, \ g := \left\{\gamma, \ \underline{T} \overset{[M]}{:=} \{\vartheta, \ Sym\}\right\},$$

and for an assignment Ass

$$\Lambda,\ g := \left\{\gamma,\ \underline{v} : S \to T \overset{[\mu]}{:=} \{\sigma\}\right\} + Ass \quad := \quad \Lambda,\ g := \left\{\gamma,\ \underline{v} : S \to T \overset{[\mu]}{:=} \{\sigma,\ Ass\}\right\},$$
$$\Lambda,\ g := \left\{\gamma,\ \underline{T} \overset{[M]}{:=} \left\{\vartheta, \underline{i} : S \overset{[\mu]}{:=} \{\sigma\}\right\}\right\} + Ass \quad :=$$
$$\Lambda,\ g := \left\{\gamma,\ \underline{T} \overset{[M]}{:=} \left\{\vartheta,\ \underline{i} : S \overset{[\mu]}{:=} \{\sigma,\ Ass\}\right\}\right\}.$$

Thus, every library can be written as $\cdot + X_1 + X_2 + \ldots$ (where $+$ associates to the left).

Furthermore, we put

$$\begin{aligned}
&\mathit{lastDoc}\,(_,\ g := \{_\}) && := && g\\
&\mathit{lastThy}\left(_,\ g := \left\{_,\ \underline{T} \overset{[_]}{:=} \{_\}\right\}\right) && := && g?\underline{T}\\
&\mathit{lastLink}\left(_,\ g := \left\{_,\ \underline{v} : _ \to _ \overset{[_]}{:=} \{_\}\right\}\right) && := && g?\underline{v}\\
&\mathit{lastLink}\left(_,\ g := \left\{_,\ \underline{T} \overset{[_]}{:=} \left\{_,\ \underline{i} : _ \overset{[_]}{:=} \{_\}\right\}\right\}\right) && := && g?\underline{T}?\underline{i}
\end{aligned}$$

Intuitively, these functions give the name of the right-most document, theory, or link in Λ.

Then a typical rule of the inference system is

$$\frac{\Lambda \rhd g := \{\gamma\} \quad \Lambda + g := \{\gamma\} \rhd \mathit{Mod}}{\Lambda \rhd g := \{\gamma, \mathit{Mod}\}}\,\mathit{Mod}$$

Its verbalization is: If $g := \{\gamma\}$ can be added to Λ, and if after adding it, Mod can be added to $g := \{\gamma\}$, then $g := \{\gamma, \mathit{Mod}\}$ can be added to Λ. Then the intelligence of the inference system lies mainly with the judgment $\Lambda \rhd \mathit{Mod}$, which defines which modules may be added, and with its analogues for symbols and assignments.

Instead of this top-down structure, a bottom-up system could be used. This would first type-check expressions of a lower level and then encapsulate and name them to form expressions of a higher level. When checking the library

$$\Lambda,\ g := \{\gamma,\ \underline{T} := \{\vartheta,\ \underline{c} : \tau := \beta(\omega, \Upsilon, \omega_1)\}\},$$

typical judgments would have the form

$$\Lambda \rhd \gamma \qquad \Lambda; \gamma \rhd \vartheta \qquad \Lambda; \gamma; \vartheta \rhd \omega \qquad \Lambda; \gamma; \vartheta; \Upsilon \rhd \omega_1.$$

It turned out that this yields an inconvenient formulation. Besides the four-partite contexts, the problem is that the names of documents, theories, and views as well as the meta-theories and meta-morphisms occur at the beginning of a declaration. It is more intuitive and closer to implementations if all information is processed in left-to-right order and stored in the context, and if the context is identified with the current library.

6.3.2 Document and Module Level

The rules for the construction of libraries and documents are given in Fig. 6.13. The rules $\mathit{Lib}_\varnothing$ and Doc construct a library by successively adding well-formed documents. And similarly, the rules $\mathit{Doc}_\varnothing$ and Mod construct a document by successively adding well-formed modules.

The rules for the construction of theories and views are given in Fig. 6.14. The rules $\mathit{Thy}_\varnothing$ and Sym construct theories by successively adding well-formed symbols. The rules $\mathit{Viw}_\varnothing$ and $\mathit{Viw}_{\mathit{Ass}}$ construct views by successively adding well-formed assignments. Alternatively, Viw_μ

$$\frac{}{\rhd \cdot}\,Lib_\varnothing \qquad \frac{\rhd \Lambda \quad \Lambda \rhd Doc}{\rhd \Lambda + Doc}\,Doc$$

$$\frac{\rhd \Lambda \quad g \notin Doc(\Lambda)}{\Lambda \rhd g := \{\cdot\}}\,Doc_\varnothing \qquad \frac{\Lambda \rhd g := \{\gamma\} \quad \Lambda + g := \{\gamma\} \rhd Mod}{\Lambda \rhd g := \{\gamma, Mod\}}\,Mod$$

Figure 6.13: Structure of Libraries and Documents

adds a view that is defined by an existing morphism. In all rules we unify the cases with and without meta-theories by using square brackets: For example, $Thy_\varnothing$ handles empty theories with or without meta-theory, and in the former case there is one additional premise. $Viw_\varnothing$ makes sure that views from a theory S are only possible if a well-formed morphism μ from a possible meta-theory is given.

The judgment $\Lambda \rhd^g n$ new defined in rule new_g clarifies which modules names are still available in a document. This is the case if a module name n satisfies two conditions: Firstly, no prefix of n may already have been declared. This is expressed by $pref(n) \cap Nam^\Lambda(g) = \varnothing$. And secondly, n may not occur as a prefix of any name already declared. This is expressed by $n \notin pref(Nam^\Lambda(g))$. We will give examples for these conditions below when describing the corresponding rule for declarations in a theory. This rule also ensures that only the last document may be extended.

In addition to rule new_g, rule new'_g permits names $\underline{T}/n$ even if the prefix $\underline{T}$ is already declared. Such names are only permitted if n is not resolvable in $\underline{T}$. The usefulness of this rule is that structures $\underline{i}$ from S over $\underline{T}$ can be turned into views $\underline{T}/\underline{i}$. This is relevant for the flattening (see Sect. 6.4).

6.3.3 Symbol Level

The symbol level rules define which symbols may be added to theories and which assignments may be added to links, see Fig. 6.15 and Fig. 6.16.

Symbol Declarations In Fig. 6.15, the rule Con says that constant declarations $\underline{c} : \tau := \delta$ are well-formed if δ has type τ (which implies that the two are well-formed), and if $\underline{c}$ does not clash with existing names. To avoid case distinctions, we use $\bot$ to indicate that τ or δ are omitted and assume that the typing judgment about τ and δ is defined in such cases as well. This gives the foundation the possibility to reject undefined or untyped constants.

The rules for structures correspond to those for views: $Str_\varnothing$ and Str_{Ass} construct structures by successively adding well-formed assignments. Alternatively, Str_μ adds a structure that is defined by an existing morphism. Again we use square brackets to unify the cases when importing from theories with or without meta-theories.

The judgment $\Lambda \rhd^T \underline{s}$ new defined in rule new_T governs which symbol names can be declared. A name $\underline{s}$ must satisfy two conditions to be available for a new symbol: Firstly, no prefix of $\underline{s}$ may already have been declared. This is expressed by $pref(\underline{s}) \cap Nam^\Lambda(T) = \varnothing$. For example, if $\underline{s} = \underline{i}/\underline{c}$, and a structure $\underline{i}$ has already been declared, a declaration of $\underline{s}$ could clash with a constant $\underline{c}$ imported via $\underline{i}$. Even if no such $\underline{c}$ exists, such names are forbidden because they would make name resolution (and management of change) inefficient. And secondly, $\underline{s}$ may not occur as a prefix of any name already declared. This is expressed by $\underline{s} \notin pref(Nam^\Lambda(T))$. For example, if $\underline{s}/\underline{c}$ is already declared, no declaration for $\underline{s}$ is allowed.

At this point the reader may wonder why names of the form $\underline{s}/\underline{c}$ should be legal at all. Indeed, it would be much easier to forbid any names containing slashes. However, the flattening (see

$$\frac{\rhd \Lambda \quad \Lambda \rhd^g \underline{T} \text{ new} \quad [M \in Thy(\Lambda)]}{\Lambda \rhd \underline{T} \stackrel{[M]}{:=} \{\cdot\}} Thy_\varnothing$$

$$\frac{\Lambda \rhd \underline{T} \stackrel{[M]}{:=} \{\vartheta\} \quad \Lambda + \underline{T} \stackrel{[M]}{:=} \{\vartheta\} \rhd Sym}{\Lambda \rhd \underline{T} \stackrel{[M]}{:=} \{\vartheta, Sym\}} Sym$$

$$\frac{\rhd \Lambda \quad \Lambda \rhd^g \underline{v} \text{ new} \quad S \in Thy(\Lambda) \quad T \in Thy(\Lambda) \quad [\Lambda \rhd \mu : meta^\Lambda(S) \to T]}{\Lambda \rhd \underline{v} : S \to T \stackrel{[\mu]}{:=} \{\cdot\}} Viw_\varnothing$$

$$\frac{\Lambda \rhd \underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma\} \quad \Lambda + \underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma\} \rhd Ass}{\Lambda \rhd \underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma, Ass\}} Viw_{Ass}$$

$$\frac{\rhd \Lambda \quad \Lambda \rhd^g \underline{v} \text{ new} \quad \Lambda \rhd \mu : S \to T}{\Lambda \rhd \underline{v} : S \to T := \mu} Viw_\mu$$

$$\frac{g = lastDoc\,(\Lambda) \quad pref(n) \cap Nam^\Lambda(g) = \varnothing \quad n \notin pref(Nam^\Lambda(g))}{\Lambda \rhd^g n \text{ new}} new_g$$

$$\frac{g = lastDoc\,(\Lambda) \quad n \notin Sym^\Lambda(g?\underline{T}) \quad n \notin pref(Nam^\Lambda(g))}{\Lambda \rhd^g \underline{T}/n \text{ new}} new'_g$$

Figure 6.14: Structure of Theories and Views

Sect. 6.4) will replace a structure $\underline{i}$ with lists of symbols for names of the form $\underline{i}/\underline{c}$. Therefore, we permit such names.

Assignments to Symbols Fig. 6.16 gives the most important rules, the ones governing assignments. $ConAss$ is used to add an assignment $\underline{c} \mapsto \delta'$ to the last link m in γ, where m has domain S and codomain T. In the simplest case, $\underline{c}$ is declared in S, but it is also permitted that $\underline{c}$ is of the form $\underline{i}/\underline{c}'$, i.e., arises from a structure into S. The rule has five premises. The first one assumes a well-formed library Λ. The second one executes three lookups in Λ; in particular, it looks up the type τ and definition δ of $\underline{c}$. The third premise guarantees that there is no assignment for $\underline{c}$ in m already. The fourth premise type-checks δ' against the translation of τ along m. Since τ usually contains other symbols declared in S, the translation of τ to T must use the assignments already provided in m. The fifth hypothesis and rule $\mathcal{M}|_{Con}$ govern the relationship between the S-definiens δ and the T-definiens δ': If $\delta = \bot$, i.e., $\underline{c}$ is not defined, any assignment is permitted; otherwise, the translated old definiens δ^m must be equal to the new one δ'; finally, in both cases $\underline{c}$ may be hidden, i.e., $\delta = \top$.

As before, to avoid case distinctions, we handle the cases of omitted τ or δ in a constant declaration by putting $\tau = \bot$ or $\delta = \bot$ even though $\bot$ is not a well-formed term. We agree that $\bot^m$ is just another way to say $\bot$, and we permit the judgment $\Lambda \rhd_T \bot \mid^m \delta'$ (which always holds). Then this rule also motivates the notation for $\bot$ and $\top$: Intuitively, they are the least

$$\frac{\rhd \Lambda \quad \Lambda \rhd^T \underline{c}\ \text{new} \quad \Lambda \rhd_T \delta : \tau}{\Lambda \rhd \underline{c} : \tau := \delta}\ Con$$

$$\frac{\rhd \Lambda \quad \Lambda \rhd^T \underline{i}\ \text{new} \quad S \in Thy(\Lambda) \quad [\Lambda \rhd \mu : meta^{\Lambda}(S) \to T]}{\Lambda \rhd \underline{i} : S \overset{[\mu]}{:=} \{\cdot\}}\ Str_{\varnothing}$$

$$\frac{\Lambda \rhd \underline{i} : S \overset{[\mu]}{:=} \{\sigma\} \quad \Lambda + \underline{i} : S \overset{[\mu]}{:=} \{\sigma\} \rhd Ass}{\Lambda \rhd \underline{i} : S \overset{[\mu]}{:=} \{\sigma, Ass\}}\ Str_{Ass}$$

$$\frac{\rhd \Lambda \quad \Lambda \rhd^T \underline{i}\ \text{new} \quad \Lambda \rhd \mu : S \to T}{\Lambda \rhd \underline{i} : S := \mu}\ Str_{\mu}$$

$$\frac{T = lastThy\,(\Lambda) \quad pref(\underline{s}) \cap Nam^{\Lambda}(T) = \varnothing \quad \underline{s} \notin pref(Nam^{\Lambda}(T))}{\Lambda \rhd^T \underline{s}\ \text{new}}\ new_T$$

Figure 6.15: Declarations in a Theory

and greatest element of an ordering (in which all other non-equal terms are incomparable). Then assignments to defined constants are well-formed if the new definition is at least as great as the old one.

The rule $StrAss$ is very similar to $ConAss$: It permits to add an assignment $\underline{i} \mapsto \mu$ of a morphism to a structure. The intuition of such an assignment is that it makes the diagram on the right commute. All premises correspond exactly to those of the rule $ConAss$. In particular, R corresponds to τ as the type of $\underline{i}$. Then the fourth premise checks the type of μ against R.

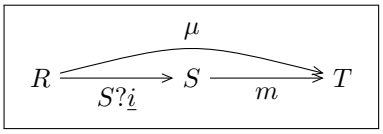


Figure 6.17: Semantics of $\underline{i} \mapsto \mu$

To understand the last premise and its definition by rule $\mathcal{M}|_{Str}$, first note that m is defined by a partial list of assignments, i.e., not totally determined yet. Further note that if $S?\underline{i}$ and μ are given, it is not necessarily possible to find any morphism m such that the diagram commutes. Then the intuitive meaning of the auxiliary judgment $\Lambda \rhd_T S?\underline{i}\ |^m\ \mu : R$ is: In the situation of Fig. 6.17, the morphism μ can be factored as $S?\underline{i} \bullet m'$ in a way such that m' agrees with m for all arguments for which m is already determined.

$\Lambda \rhd_T S?\underline{i}\ |^m\ \mu : R$ is defined in rule $\mathcal{M}|_{Str}$. Here we use double square brackets to indicate hypothetical judgments in order to distinguish them from optional parts. The hypothesis checks for all constants $\underline{c}$ in R that are imported via $\underline{i}$ whether they may be redefined to the result of applying μ to $\underline{c}$. This rule is very inefficient because all constants of R must be checked. Below we will supplement it with a more efficient variant that exploits the theory graph structure.

Finally, the rule new_m is essentially the same as the analogous rule for theories. The rule defines which names are not already affected by assignments in m. An assignment for a name $\underline{s}$ is permitted if the following two conditions hold. Firstly, there may not already be an assignment for a prefix of $\underline{s}$. For example, an assignment $\underline{i} \mapsto \mu$ determines the assignments to all constants $\underline{i}/\underline{c}$, which may therefore not be instantiated separately anymore. Secondly, $\underline{s}$ may not occur as a prefix of any name for which there already is an assignment. For example, if $\underline{s} = \underline{i}/\underline{c}$, and

$$\frac{\begin{array}{c} m = \mathit{lastLink}\,(\Lambda) \\ \rhd\,\Lambda \quad \Lambda(m) = (S, T, \_, \_) \\ \Lambda^S(\underline{c}) = (\tau, \delta) \end{array} \qquad \Lambda \rhd^m \underline{c}\ \text{new} \qquad \Lambda \rhd_T \delta' : \tau^m \qquad \Lambda \rhd_T \delta \mid^m \delta'}{\Lambda \rhd \underline{c} \mapsto \delta'}\ \mathit{ConAss}$$

$$\frac{\delta = \bot \quad \text{or} \quad \Lambda \rhd_T \delta^m \equiv \delta' \quad \text{or} \quad \delta' = \top}{\Lambda \rhd_T \delta \mid^m \delta'}\ \mathcal{M}|_{Con}$$

$$\frac{\begin{array}{c} m = \mathit{lastLink}\,(\Lambda) \\ \rhd\,\Lambda \quad \Lambda(m) = (S, T, \_, \_) \\ \mathit{dom}^\Lambda(S?\underline{i}) = R \end{array} \qquad \Lambda \rhd^m \underline{i}\ \text{new} \qquad \Lambda \rhd \mu : R \to T \qquad \Lambda \rhd_T S?\underline{i} \mid^m \mu : R}{\Lambda \rhd \underline{i} \mapsto \mu}\ \mathit{StrAss}$$

$$\frac{\begin{array}{c} [\![\Lambda^S(\underline{i}/\underline{c}) = (\_, \delta)]\!] \\ \vdots \\ \Lambda \rhd_T \delta \mid^m R?\underline{c}^\mu \end{array}}{\Lambda \rhd_T S?\underline{i} \mid^m \mu : R}\ \mathcal{M}|_{Str}$$

$$\frac{m = \mathit{lastLink}\,(\Lambda) \quad \mathit{pref}(\underline{s}) \cap \mathit{Nam}^\Lambda(m) = \varnothing \quad \underline{s} \notin \mathit{pref}(\mathit{Nam}^\Lambda(m))}{\Lambda \rhd \underline{s}\ \text{new}}\ \mathit{new}_m$$

Figure 6.16: Assignments in a Link

there is an assignment for $\underline{i}$, then the assignments to $\underline{s}$ is already determined by the assignments to $\underline{i}$.

Theory-Level Reasoning As mentioned above, we give an alternative rule for the judgment $\Lambda \rhd_T S?\underline{i} \mid^m \mu : R$:

$$\frac{\begin{array}{cc} [\![\Lambda^S(\underline{i}/\underline{c}) = (\_, \delta), \quad \underline{c}\ \text{declared in}\ R]\!] & [\![\underline{h}\ \text{declared in}\ R]\!] \\ \vdots & \vdots \\ \Lambda \rhd_T \delta \mid^m R?\underline{c}^\mu & \Lambda \rhd_T S?\underline{i}/\underline{h} \mid^m R?\underline{h} \bullet \mu : Q \end{array}}{\Lambda \rhd_T S?\underline{i} \mid^m \mu : R}\ \mathcal{M}|_{Str}$$

Here the first hypothesis is as in rule $\mathcal{M}|_{Str}$ but restricted to constants declared in R. All constants imported into R are handled by the second hypothesis, which recurses for every structure declared in R. It is easy to see that this rule is derivable from $\mathcal{M}|_{Str}$: Unwinding the recursion leads to exactly the same set of hypotheses.

Thus, there is no efficiency gain yet. But in this form, we can apply the knowledge about the theory graph to avoid some recursive calls. For example, it is simple to add reasoning about the equality of morphisms using rules for associativity, identity elimination, and definition expansion. Then if $S?\underline{i}/\underline{h} \bullet m \equiv R?\underline{h} \bullet \mu$, we can immediately infer $\Lambda \rhd_T S?\underline{i}/\underline{h} \mid^m R?\underline{h} \bullet \mu : Q$ without going into a recursive call.

Thus, we decompose a judgment about all constants in R into separate judgments about the constants declared locally in R and the structures into R, and then discharge the judgments about structures by arguing about the theory graph. This approach is very similar to the decomposition of global theorem links in development graphs ([AHMS99]).

6.3.4 Object Level

6.3.4.1 Structural Well-Formedness of Terms

The inference rules in Fig. 6.18 formalize the notion of a structurally well-formed term. This is a level of validation between the context-free grammar check (as used in OMDoc or OpenMath) and the type checking in formal systems. Structural well-formedness means that a term only uses symbols and variables that are in scope; but it does not say anything about a possible semantics of the term. Our rules mainly follow the OpenMath standard but restricted to a home theory T and a context Υ. Only symbols over T or variables declared in Υ may occur in well-formed terms.

The rule $\mathcal{T}_\beta$ is particularly interesting since it is not totally clear how to interpolate the treatment of bound variables from the OpenMath standard to an inference system using variable contexts. We chose the least restrictive interpretation and permit every bound variable to occur in the attributions of every other variable bound by the same binding operation. This is useful to represent mutually recursive let bindings. However, we forbid duplicate variable names within the same binding operation since they are hardly useful anyway and can easily be confusing in the presence of attributions. Note that $\mathcal{T}_\beta$ checks the well-formedness of the bound variables; and since Mmt only uses closed terms, we do not need a judgment for well-formed contexts in $\mathcal{T}_\top$, $\mathcal{T}_{Var}$, and $\mathcal{T}_{Con}$.

The rules $\mathcal{T}_\top$ and $\mathcal{T}_\mu$ formalize our extensions of OpenMath: The special term $\top$ is a well-formed term, and morphisms move terms between theories. We use Υ^μ to abbreviate the component-wise application of a morphism to a list of variables.

$$\frac{T \in Thy(\Lambda)}{\Lambda;\Upsilon \rhd_T \top}\,\mathcal{T}_\top \qquad \frac{x \in \Upsilon \quad T \in Thy(\Lambda)}{\Lambda;\Upsilon \rhd_T str(x)}\,\mathcal{T}_{Var}$$

$$\frac{\underline{c} \in Con^\Lambda(T)}{\Lambda;\Upsilon \rhd_T T?\underline{c}}\,\mathcal{T}_{Con} \qquad \frac{\Lambda;\Upsilon \rhd_T \omega_i}{\Lambda;\Upsilon \rhd_T @(\omega_1,\ldots,\omega_n)}\,\mathcal{T}_@$$

$$\frac{\Lambda;\Upsilon \rhd_T \omega \quad \Upsilon' = x_1,\ldots,x_n \quad \begin{array}{c} str(x_i) \in Var \\ str(x_i) \neq str(x_j) \text{ for } i \neq j \end{array} \quad \begin{array}{c} \Lambda;\Upsilon,\Upsilon' \rhd_T x_i \\ \Lambda;\Upsilon,\Upsilon' \rhd_T \omega' \end{array}}{\Lambda;\Upsilon \rhd_T \beta(\omega,\Upsilon',\omega')}\,\mathcal{T}_\beta$$

$$\frac{\Lambda;\Upsilon \rhd_T \omega_i}{\Lambda;\Upsilon \rhd_T \alpha(\omega_1,\omega_2 \mapsto \omega_3)}\,\mathcal{T}_\alpha \qquad \frac{\Lambda;\Upsilon \rhd_S \omega \quad \Lambda \rhd \mu : S \to T}{\Lambda;\Upsilon^\mu \rhd_T \omega^\mu}\,\mathcal{T}_\mu$$

Figure 6.18: Well-formed Terms

6.3.4.2 Typing and Equality of Terms

The rules given so far do not address the typing and equality of terms, i.e., we do not derive the judgments

$$\Lambda; \Upsilon \rhd_T \omega \equiv \omega' \qquad \text{and} \qquad \Lambda; \Upsilon \rhd_T \omega : \omega'.$$

This follows an essential design choice in OpenMath and OMDoc, which do not commit to any particular formal system for these judgments. We stay parametric in the type system by using a concept we call **foundations**. Foundations provide an abstraction from those (non-modular) formal systems that can be specified in terms of judgments for typing and equality.

An intuition for foundations is as follows: Any formal or software system Sys that uses or implements MMT must provide a set D of theory names and a foundation Φ for D. The names in D constitute the theories that Sys understands. These theories should be declared in a library Λ_0 that is published as the interface of Sys. Then the input of Sys consists of documents that are well-formed relative to Λ_0. In this way a wide variety of formal systems can be represented in MMT.

Definition 6.7. For a set D of URIs and a library Λ, we define the meta-import from D to T by

$$mi^{\Lambda}_D(T) := T?../\ldots/..$$

where the number $N(T)$ of repetitions of .. is given by

$$N(T) := \begin{cases} 0 & \text{if } T \in D \\ N(meta^{\Lambda}(T)) + 1 & \text{otherwise} \end{cases}$$

Thus, $mi^{\Lambda}_D(T)$ is obtained by traversing the meta-theories of T until a theory $M \in D$ is found. Then $mi^{\Lambda}_D(T)$ is the morphism from M to T via which M is a meta-theory of T. The intuition behind $mi^{\Lambda}_D(T)$ is that if $i = mi^{\Lambda}_D(T)$, a foundation for D must know the semantics of $i/\underline{c}$, because it refers to the symbol $\underline{c}$ in a theory within D.

Definition 6.8. For a set D of URIs, an MMT **foundation** Φ is a component that defines for the following input:

- a meta-import $i := mi^{\Lambda}_D(T)$ from some $M \in D$ to some theory T declared in some library Λ,
- the lookup function for Λ,
- two terms ω and ω' that are well-formed over Λ and T,

two yes/no values, which we denote by

$$\Phi^{\Lambda,i}(\omega \equiv \omega') \qquad \text{and} \qquad \Phi^{\Lambda,i}(\omega : \omega').$$

Definition 6.9. We say that a foundation for D **covers** a library Λ if $mi^{\Lambda}_D(T)$ is defined for all theories in Λ.

Relative to a foundation Φ for D, equality and typing are defined as follows. Whenever the judgment $\Lambda \rhd_T \omega \equiv \omega'$ or $\Lambda \rhd_T \omega : \omega'$ is encountered in a library Λ covered by Φ, the meta-import $i := mi^{\Lambda}_D(T)$ of T is determined. Then Φ is called with the lookup functions for Λ and i, ω and ω' as input. The lookup functions are passed to Φ so that Φ can call back to MMT in order to look up all symbols it encounters in ω and ω'. Formally, this is defined by the rules in Fig. 6.19.

Only the lookup function of Λ are passed to Φ, not Λ itself. In implementations, this is simply a pointer to a callback function. This is crucial to achieve scalability because sophisticated

$$\frac{\Lambda \rhd_T \omega \quad \Lambda \rhd_T \omega' \quad i = mi^{\Lambda}_D(T) \quad \Phi^{\Lambda,i}(\omega \equiv \omega')}{\Lambda \rhd_T \omega \equiv \omega'}\ \mathcal{T}_{\equiv}$$

$$\frac{\Lambda \rhd_T \omega \quad \Lambda \rhd_T \omega' \quad i = mi^{\Lambda}_D(T) \quad \Phi^{\Lambda,i}(\omega : \omega')}{\Lambda \rhd_T \omega : \omega'}\ \mathcal{T}_{:}$$

Figure 6.19: Equality and Typing

management services and optimizations can be implemented in a foundation-independent way. Then the implementation of the specific foundation only needs to implement the core functionality of type and equality checking. In particular, a foundation-independent implementation of MMT can handle the tasks of parsing, storing, updating, and querying the current library and thus encapsulate the global state. Furthermore, it remains transparent to the foundation how the current library is distributed both over local short and long term storage media and the internet.

While our main motivation is the use of MMT as an interface language for software systems, we do not assume in general that a foundation is decidable. A foundation may also axiomatize undecidable judgments of equality and typing. But of course, not every foundation is useful, and some choices have to be excluded. Therefore, we employ the following conditions on foundations.

Definition 6.10. A foundation Φ is called **regular** if it satisfies the following conditions for any Λ it covers and any $T \in Thy(\Lambda)$:

- if $\Phi^{\Lambda,i}(\omega_1 : \omega_2)$, and Λ' arises from Λ by adding documents, theories, or views in a well-formed way, then also $\Phi^{\Lambda',i}(\omega_1 : \omega_2)$, and similarly for equality,
- equality is reflexive, symmetric, and transitive,
- equality is congruent with respect to application, binding, and attribution, and if $\overline{\omega} \neq \top$ then $\Phi^{\Lambda,i}(\omega \equiv \overline{\omega})$,
- equality respects α-renaming: If $x'_1, \ldots, x'_n$, and ω'_1 arise from $x_1, \ldots, x_n$, and ω_1 by parallel capture-avoiding substitution of every reference to the variable x_i with a reference to x'_i, then
$$\Phi^{\Lambda,i}(\beta(\omega, x_1, \ldots, x_n, \omega_1) \equiv \beta(\omega, x'_1, \ldots, x'_n, \omega'_1))$$
- typing and equality are related via
$$\frac{\Phi^{\Lambda,i}(\omega_1 : \omega_2) \quad \Phi^{\Lambda,i}(\omega_1 \equiv \omega'_1) \quad \Phi^{\Lambda,i}(\omega_2 \equiv \omega'_2)}{\Phi^{\Lambda,i}(\omega'_1 : \omega'_2)}$$
- if $\Lambda^T(\underline{c}) = (\tau, \_)$, then $\Phi^{\Lambda,i}(T?\underline{c} : \tau)$ for the appropriate i,
- $\top$ is only equal to itself,
- $\top$ is typed by every term, and only $\top$ is typed by $\top$.

In Def. 6.10, all requirements are intuitively plausible except for the last one. $\top$ must have every type so that in rule *ConAss* constants of every type can be hidden. This also means that $\top$ is a proof of everything and thus can be used as a place holder for every undischarged proof obligation.

In rule *Con*, we called the type-checking judgment even though the arguments τ and δ might be omitted. Permitting that the inputs to the typing judgments can also be $\bot$, gives the foundations the freedom to decide what kind of constant declarations may be added. This is described in Fig. 6.20.

Instance	Effect
$\Phi^{\Lambda,i}(\bot:\bot)$	theories may declare undefined, untyped constants
$\Phi^{\Lambda,i}(\bot:\omega)$	theories may declare undefined constants of type ω
$\Phi^{\Lambda,i}(\omega:\bot)$	theories may define untyped constants as ω

Figure 6.20: Special Cases in Foundations

For regular foundations, the terms ω and ω' can always be normalized without changing $\Phi^{\Lambda,i}(\omega:\omega')$ or $\Phi^{\Lambda,i}(\omega\equiv\omega')$. Therefore, we can normalize ω and ω' before passing them to the foundation. And since regular foundations make equality a congruence relation, it is also possible to always normalize the type and definition returned by lookup calls of the form $\Lambda^T(\underline{c})$. Then it is possible to use a foundation in such a way that it will never encounter any morphisms, and the only necessary lookups are the ones of the form $\Lambda^T(\underline{c})$. And since normalization does not depend on the foundation, a regular foundation does not have to handle — and not even understand — morphisms at all. Then MMT yields a module system for every non-modular language for which we can give a regular foundation.

Example 6.11. An important example is the default foundation Φ^{Def} for the set of all possible theory names. Φ^{Def} can be used by systems if the semantics of a theory cannot be determined otherwise. It is defined as follows:

- $(\Phi^{Def})^{\Lambda,i}(\omega\equiv\omega')$ iff $\overline{\omega}$ and $\overline{\omega'}$ are identical modulo α-renaming of bound variables,
- $(\Phi^{Def})^{\Lambda,i}(\omega:\omega')$ according to:

$\overline{\omega}\backslash\overline{\omega'}$	$\bot$	ω	$\top$
$\bot$	+	+	-
ω	+	-	-
$\top$	+	+	+

Thus, equality is syntactical equality with α-conversion. And no non-degenerate typing relation holds. Φ^{Def} is regular and covers all libraries.

Foundations are an important tool to use MMT as an interface language between software systems. For a translation from a system A and B, an export from A to MMT would be implemented as a part of A. The output would refer to a meta-theory M_A for the logic underlying A and be interpreted with respect to a foundation for $\{M_A\}$. The foundation is documented separately or within M_A using informal annotations (see also Sect. 6.5.7). Then an import from MMT is implemented as part of B. The semantics of M_A is hard-coded in this import. Of course, some information may be lost when filtering through MMT compared to a direct translation from A to B. But a translation via MMT will typically be much easier than the direct translation — in particular, the feasibility threshold may lie in between the two ways to implement the translation.

6.3.4.3 Morphisms

Fig. 6.21 shows the rules for the construction of morphisms. The rule $\mathcal{M}_m$ handles links. $\mathcal{M}_{id}$ and $\mathcal{M}_\bullet$ give identity and composition of morphisms. Composition $\bullet$ is written in diagrammatic order, i.e., from the domain to the codomain.

$$\frac{\Lambda(m) = (S, T, \_, \_)}{\Lambda \rhd m : S \to T}\,\mathcal{M}_m$$

$$\frac{T \in Thy(\Lambda)}{\Lambda \rhd id_T : T \to T}\,\mathcal{M}_{id} \qquad \frac{\Lambda \rhd \mu : R \to S \quad \Lambda \rhd \mu' : S \to T}{\Lambda \rhd \mu \bullet \mu' : R \to T}\,\mathcal{M}_\bullet$$

Figure 6.21: Morphisms

6.3.5 Validity Levels

Thus, we obtain three levels of validity for MMT documents. This induces the corresponding validity notions for OMDOC documents if only the fragment of OMDOC is considered that lies within the image of the XML-encoding of MMT given in Sect. 8.1.

- A document that conforms to the MMT grammar (i.e., is produced from *Doc*) is called OMDOC-valid. OMDOC-validity can be checked by standardized XML validation.
- A document *Doc* is **structurally well-formed** or MMT-valid relative to the library Λ if $\Lambda \rhd Doc$ holds with respect to the inconsistent foundation, i.e., the foundation in which the typing and equality judgments always hold.
- Φ-validity is defined like MMT-validity but with respect to a foundation Φ.

Then Φ-validity implies MMT-validity, and that implies OMDOC-validity. A system is called OMDOC, MMT, or Φ-aware if it can identify documents that are invalid with respect to the corresponding validity level.

6.3.6 Structural Properties

Lemma 6.12 (Uniqueness of Names)**.** *For well-formed Λ and any g, T, m, $\underline{s}$: $\Lambda(g)$, $\Lambda(T)$, $\Lambda(m)$, $\Lambda^T(\underline{s})$, and $\Lambda^m(\underline{s})$ are either undefined or uniquely determined.*

Proof. This follows easily by inspecting the rules new_g, new_T, and new_m. □

In particular, names within a scope (i.e., a library, document, theory, structure, or view) are always unique, and the order of documents, declarations, or assignments within the same scope is transparent to the lookup functions.

Theorem 6.13 (Subexpression Property)**.** *We have the following subexpression properties:*

1. *Assume $\rhd \Lambda$. Then any subexpression of Λ that arises by dropping the right-most document, declaration, or assignment is also a well-formed library.*
2. *Assume $\rhd \Lambda$ and $\Lambda \rhd_T \omega$ and a subterm ω' of ω. Let Υ be the concatenation of the contexts in the binders governing ω' (starting with the outermost binder) or the empty context if there is none. Let S be the domain of the morphism in the innermost morphism application governing ω' or T if there is none. Then $\Lambda; \Upsilon \rhd_S \omega'$.*
3. *Assume $\rhd \Lambda$ and $\Lambda \rhd \mu : S \to T$ and a subexpression μ' of μ with domain S' and codomain T'. Then $\Lambda \rhd \mu' : S' \to T'$.*

Proof. These follow by straightforward inductions on the derivations. □

We do not establish a subterm property for terms with free variables. This is possible, too, but less interesting because libraries cannot contain terms with free variables.

Lemma 6.14 (Uniqueness of Derivations)**.** *For every judgment, there is at most one derivation.*

Proof. This follows easily by inspecting the rules. □

Theorem 6.15 (Decidability)**.** *If there is a decidable foundation that covers all theories in Λ, then the judgment $\triangleright \Lambda$ is decidable. In particular, it is decidable for the default foundation Φ^{Def}.*

Proof. This is easy to see because the inference rules are designed to specify the implementation of the decision procedure. □

Theorem 6.16 (Weakening)**.** *Assume a regular foundation. Let Λ and Λ' be two well-formed libraries such that Λ' arises from Λ by adding documents, theories, or views. Then the judgments for well-formed terms and morphisms, typing, and equality with respect to Λ imply their analogues with respect to Λ'.*

Proof. Clearly every derivation over Λ becomes a derivation over Λ' by replacing every occurrence of Λ with Λ'. The cases for typing and equality judgments follow from the regularity of the foundation. □

6.4 Library Transformations

6.4.1 Modular and Flat Libraries

The representation of theory graphs introduced in the last section is geared towards expressing mathematical knowledge in its most general form and with the least redundancy: constants can be shared by inheritance (i.e., via imports), and terms can be moved between theories via morphisms. This style of writing mathematics has been cultivated by the Bourbaki group ([Bou68, Bou74]) and lends itself well to a systematic development of theories.

However, it also has drawbacks: Items of mathematical knowledge are often not where or in the form in which we expect them, as they have been generalized to a different context. For example, a constant c need not be explicitly represented in a theory T, if it is induced as the image of a constant c' under some import into T.

In this section, we introduce operations on libraries to flatten a library. In particular, this involves adding all induced knowledge items to every theory thus making all theories self-contained (but hugely redundant between theories). For a given MMT-library Λ, we can view the flattening of Λ as its semantics, since flattening eliminates the specific MMT-representation infrastructure of imports, meta-theories, and morphisms, and reduces theories to collections of constants, possibly with types and definitions, which conforms to the non-modular logical view of theories. Formally, we define flat libraries as follows.

Definition 6.17. A library Λ is called **flat** if it does not contain meta-theories, structures, morphism applications, or hidings, i.e., it is produced by the restricted grammar given in Fig. 6.22 (where the productions for names are omitted).

6.4.2 Equivalence of Libraries

Definition 6.18. Two libraries Λ and Λ' are called **structurally equivalent** if the following holds:

- $Doc(\Lambda) = Doc(\Lambda')$,

Λ	::=	Doc^*
Doc	::=	$g := \{\gamma\}$
γ	::=	Mod^*
Mod	::=	$Thy \mid Viw$
Thy	::=	$\underline{T} := \{\vartheta\}$
Viw	::=	$\underline{v} : T \to T := \{\sigma\} \mid \underline{v} : T \to T := \mu$
ϑ	::=	Con^*
σ	::=	$ConAss^*$
Con	::=	$\underline{c} : \omega := \omega \mid \underline{c} : \omega \mid \underline{c} := \omega \mid \underline{c}$
$ConAss$	::=	$\underline{c} \mapsto \omega$
ω	::=	$c \mid x \mid @(\omega, \omega^{\mp}) \mid \beta(\omega, \Upsilon, \omega) \mid \alpha(\omega, \omega \mapsto \omega)$
Υ	::=	$\cdot \mid \Upsilon, \omega$ if $str(\omega)$ of the form x
μ	::=	$id_T \mid i \mid v \mid \mu \bullet \mu$

Figure 6.22: The Grammar for Raw MMT Expressions

- $Thy(\Lambda) = Thy(\Lambda')$,
- $Mor(\Lambda) = Mor(\Lambda')$,
- for all $T \in Thy(\Lambda)$: $Con^{\Lambda}(T) = Con^{\Lambda'}(T)$,
- for all $m \in Mor(\Lambda)$: $dom^{\Lambda}(m) = dom^{\Lambda'}(m)$ and $cod^{\Lambda}(m) = cod^{\Lambda'}(m)$.

Note that structural equivalence implies that a theory T has the same constant names $\underline{i}/\underline{c}$ in both Λ and Λ', but leaves open whether a constant of name $\underline{i}/\underline{c}$ is declared or whether a constant $\underline{c}$ is imported via a structure $\underline{i}$. The intuition behind structural equivalence of libraries is given by the following lemma: Structurally equivalent libraries have the same objects.

Lemma 6.19. *Assume two structurally equivalent libraries Λ and Λ' that are well-formed for a fixed foundation. Then for all $S, T \in Thy(\Lambda)$:*

- $\Lambda \rhd_T \omega$ *iff* $\Lambda' \rhd_T \omega$,
- $\Lambda \rhd \mu : S \to T$ *iff* $\Lambda' \rhd \mu : S \to T$.

Proof. This follows by a straightforward induction on the derivations. □

Having the same objects does not say that two libraries are semantically equivalent. For example, two theories might have the same constants but with different types. The next definition refines this.

Definition 6.20. Two structurally equivalent libraries Λ and Λ' are called **semantically equivalent** if the following holds:

- For all $T \in Thy(\Lambda)$ and all $\underline{c} \in Con^{\Lambda}(T)$ such that $\Lambda^T(\underline{c}) = (\tau, \delta)$ and $\Lambda'^T(\underline{c}) = (\tau', \delta')$:

  $\overline{\delta}^{\Lambda}$ and $\overline{\delta'}^{\Lambda'}$ as well as $\overline{\tau}^{\Lambda}$ and $\overline{\tau'}^{\Lambda'}$ are identical up to α-variants.

- For all $m \in Mor(\Lambda)$ and all $\underline{c} \in Con^{\Lambda}(dom^{\Lambda}(m))$:

  $\overline{\Lambda^m(\underline{c})}^{\Lambda}$ and $\overline{\Lambda'^m(\underline{c})}^{\Lambda'}$ are identical up to α-variants.

Intuitively, if a regular foundation is used, then semantically equivalent libraries are indistinguishable by the lookup functions. That is formalized in the following result:

Theorem 6.21. *Assume two semantically equivalent libraries Λ and Λ' and a regular foundation. Then for all g and γ:*

$$\rhd \Lambda, g := \{\gamma\} \qquad \text{iff} \qquad \rhd \Lambda', g := \{\gamma\}\,.$$

Proof. Assume a well-formedness derivation D for $\rhd \Lambda, g := \{\gamma\}$. Let D' arise from D by replacing every Λ with Λ'. We claim that every subtree of D' is a well-formedness derivation for its respective root. Then in particular, D' is a well-formedness derivation for $\rhd \Lambda', g := \{\gamma\}$. This is shown by induction on D. The induction steps for most rules are trivial because the library only occurs as a fixed parameter. For the rules regarding new names, the induction step follows from the structural equivalence. And for the rules $\mathcal{T}_{\dot{=}}$ and $\mathcal{T}_{\equiv}$, it follows from the semantical equivalence and the regularity of the foundation. □

This provides systems working with MMT libraries with an invariant for semantically indiscernible library transformations. Systems maintaining libraries can apply such transformations to increase the efficiency of storage or lookup without affecting their outward appearance. Further applications are management of change systems (e.g., [AHMS02, MK08]), which are given an easily implementable criterion to analyze the semantic relevance of a change.

Of course, Def. 6.20 is just a sufficient criterion for semantic indistinguishability. If a foundation adds equalities between terms, then libraries that are distinguished by Def. 6.20 become equivalent with respect to that foundation. But the strength of Def. 6.20 and Thm. 6.21 is that they are foundation-independent. Therefore, semantic equivalence can be implemented by low-level knowledge management services and tools that do not know about the semantics of the objects they are processing.

The most important example of semantically equivalent libraries are reorderings.

Theorem 6.22. *If Λ and Λ' are well-formed libraries that differ only in the order of documents, modules, symbols, or assignments, then they are semantically equivalent.*

Proof. Clear since the lookup functions are insensitive to reorderings. □

However, note that not all reorderings preserve the well-formedness of libraries.

6.4.3 Flattening

In the following we give several instances of semantic equivalence. Taken together, they permit to transform every MMT-library into a semantically equivalent flat one.

Flattening Meta-Theories Intuitively, declaring a meta-theory M of T is just a special way of importing M. Thus, we can drop the meta-relation and add a structure with domain M at the beginning of T. This is made precise in the following lemma.

Lemma 6.23. *Assume a well-formed library*

$$\Lambda \quad = \quad \Lambda_0,\ g := \left\{\gamma_0,\ \underline{S} \overset{M}{:=} \{\vartheta\}\,,\ \gamma_1\right\},\ \Lambda_1$$

where M has no meta-theory. Let Λ' be obtained as follows:

- *$\underline{S} \overset{M}{:=} \{\vartheta\}$ is replaced with $\underline{S} := \{.. : M := \{\cdot\},\ \vartheta\}$,*
- *every view $\underline{v} : g?\underline{S} \to T \overset{\mu}{:=} \{\sigma\}$ anywhere in Λ is replaced with $\underline{v} : g?\underline{S} \to T := \{.. \mapsto \mu, \sigma\}$,*
- *every structure $\underline{i} : g?\underline{S} \overset{\mu}{:=} \{\sigma\}$ anywhere in Λ is replaced with $\underline{i} : g?\underline{S} := \{.. \mapsto \mu, \sigma\}$.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. Since we give the new structure the special name .., all references to the meta-import $g?\underline{T}?..$ in Λ are also well-formed in Λ'. For the same reason, all references from within $\underline{T}$ to names of M stay well-formed. Similarly, the replacement of meta-morphisms with assignments retains semantic equivalence. Then the details of the proof are straightforward. □

Flattening Structures Intuitively, importing from a theory S is just an abbreviation of copying and translating the body of S. Thus, a structure $\underline{i} : S := \{\_\}$ can be replaced with a translated copy of $body^{\Lambda}(S)$. Since we need to preserve the uniqueness of names, we replace every symbol name $\underline{s}$ of S with $\underline{i}/\underline{s}$. For example $\underline{c} : \tau := \bot$ becomes $\underline{i}/\underline{c} : \tau' := \bot$, where τ' arises from τ by changing all names referring to symbols of S to the newly generated names. Then references to $\underline{c}$ of S via the structure $\underline{i}$ take the same form as references to the new symbol $\underline{i}/\underline{c}$. If $\underline{i}$ has an assignment $\underline{c} \mapsto \delta$, we generate a defined constant: $\underline{i}/\underline{c} : \tau' := \delta$.

Lemma 6.24. *Let $T := g?\underline{T}$, and assume a well-formed library*

$$\Lambda \quad = \quad \Lambda_0,\ g := \left\{\gamma_0,\ \underline{T} \stackrel{[M]}{:=} \{\vartheta_0,\ \underline{i} : S := \{\sigma\},\ \vartheta_1\},\ \gamma_1\right\},\ \Lambda_1$$

such that the morphism $T?\underline{i}$ does not occur in σ or ϑ_1 (Symbols of the form $T?\underline{i}/\underline{s}$ may occur.) and such that S has no meta-theory and $body^{\Lambda}(S)$ does not contain structures. Let Λ' be obtained as follows:

1. *$\underline{i} : S := \{\sigma\}$ is replaced with the list of declarations containing $\underline{i}/\underline{c} : \overline{\tau} := \overline{\delta}$ for every $\underline{c}$ such that $\Lambda^T(\underline{i}/\underline{c}) = (\tau, \delta)$ (in some order that makes it well-formed).*
2. *$\underline{T}/\underline{i} : S \to T := \{\sigma'\}$ is added before γ_1 where σ' contains $\underline{c} \mapsto T?\underline{i}/\underline{c}$ for every constant $\underline{c}$ declared in $body^{\Lambda}(S)$.*
3. *Every assignment $\underline{i} \mapsto \mu$ in a link m with domain T is replaced with the list of assignments containing $\underline{i}/\underline{c} \mapsto \overline{\omega}$ for every $\underline{c}$ such that $\Lambda^{T?\underline{i}}(\underline{c}) = \omega$.*
4. *Every assignment $\underline{j}/\underline{i} \mapsto \mu$ in a link from T' such that $T'?\underline{j}$ has domain T is treated as in Case 3.*
5. *If $m = g'?\underline{T}'?\underline{i}'$ is a structure with domain T, all occurrences of the imported structure $g'?\underline{T}'?\underline{i}'/\underline{i}$ are replaced with $g?\underline{T}/\underline{i} \bullet m$, and a view $g'?\underline{T}'/\underline{i}'/\underline{i}$ from S to $g'?\underline{T}'$ is added as in Case 2.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. Straightforward. (Note that hiding is covered by appealing to $\Lambda^T(-)$ in Step 1.) □

Lem. 6.24 is limited to the case where S does not declare any structures. Therefore, structures into S must be flattened recursively before a structure from S can be flattened. However, the flattening of structures leads to an exponential blow-up, which should be prevented whenever possible. The lemma is given here for its conceptual clarity but should not be implemented literally. Instead, a generalized version of Lem. 6.24 is possible, which permits structure declarations in the body of S. Then a structure $\underline{h}$ importing from R to S can be translated to a structure $\underline{i}/\underline{h} : R := \mu$ where $\mu = body^{\Lambda}(T?\underline{i}/\underline{h})$. In particular, if σ contains $\underline{h} \mapsto \mu'$, then $\mu = \mu'$.

Another important enhancement that is relevant in practice is that δ and τ in Case 1 of Lem. 6.24 do not have to be normalized. Rather, it is sufficient to remove all occurrences of $T?\underline{i}$. And the latter is more efficient and preserves more modular structure.

In Lem. 6.24, we exclude occurrences of the structure $T?\underline{i}$ after its declaration. All such occurrences can be eliminated using other flattening steps as seen below. Thus, we first eliminate all references to $T?\underline{i}$ and then its declaration. It is still easy for an implementation to flatten all structures using only one pass over the library.

Flattening Terms Flattening terms means to replace terms with their normalization.

Lemma 6.25. *Assume a well-formed library containing a constant declaration $c : \tau := \delta$. Let Λ' arise from Λ by replacing $c : \tau := \delta$ with $c : \overline{\tau} := \overline{\delta}$.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. Clear. □

Lemma 6.26. *Assume a well-formed library containing an assignment $\underline{c} \mapsto \omega$ to a constant. Let Λ' arise from Λ by replacing $\underline{c} \mapsto \omega$ with $\underline{c} \mapsto \overline{\omega}$.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. Clear. □

Flattening Hiding Hiding is flattened by omitting knowledge items that refer to $\top$.

Lemma 6.27. *Assume a well-formed library containing a constant $\underline{c} : \_ := \top$ or $\underline{c} : \top := \_$. Let Λ' arise from Λ by omitting that constant.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. The lookup $\Lambda^{g?\underline{T}}(\underline{c})$ is not defined so that $g?\underline{T}?\underline{c}$ cannot occur in any term. Neither can it be instantiated by a link since rule *ConAss* also depends on the lookup to exist. Therefore, the result follows immediately. □

Lemma 6.28. *Assume a well-formed library containing a view with an assignment $\underline{c} \mapsto \top$. Let Λ' arise from Λ by omitting that assignment.*

Then Λ' is well-formed and Λ and Λ' are semantically equivalent.

Proof. Clear because the semantics of partial views is defined by hiding. □

Flattening Libraries Finally, we obtain:

Theorem 6.29. *Every library is semantically equivalent to a flat one.*

Proof. This is easy to show by iteratively applying the above lemmas. □

Note that the flattened library of a library Λ contains views for all elements of $Mor(\Lambda)$, but these views are not used anywhere (and could thus be dropped). Of course, the size of the flattened library is exponential in the height of the theory graph. This reminds of the importance of modular theory design and of implementations that preserve the modular structure in their internal data structures.

Finally, note that the existence of the flattened library is in fact trivial. We can construct it immediately as follows: Given a library Λ, for every $T \in Thy(\Lambda)$, we take a theory T and fill it with declarations $\underline{c} : \overline{\tau} := \overline{\delta}$ for every $\Lambda^T(\underline{c}) = (\tau, \delta)$. Similarly, for every link $m \in Mor(\Lambda)$, we take a view m and fill it with assignments according to $\Lambda^m(-)$.

The most important practical aspect of the flattening is not its existence but that it can be applied flexibly: Single equivalence transformations can be carried out without flattening the whole library. Thus, the modular structure can be preserved as long as possible or necessary, and the exponential blowup is avoided.

6.5 Future Work

The version of MMT presented here is a core language. Nothing more can be expected (or should be attempted for that matter) when designing a language from scratch. There are several non-trivial extensions — some envisioned, some already designed — that will prove important in the future. An important observation in this respect is the extensibility of MMT: Now that a fully formal syntax for a core language is given, more complex features can be easily added step by step.

Some of these features, such as roles, structured proofs, and informal documents, can be seen as reconstructing OMDOC 1.2 in a fully formal setting. Others, such as functors and abstractions, go beyond OMDOC 1.2. In this section, we give an overview over these upcoming developments, ordered by descending maturity of the ideas.

6.5.1 Implementation

Before extending the language, we will focus on an implementation. Since one of our main concerns is scalability, it is important to feed back practical experiences into the language design process. The implementation will read in an MMT library (using the XML syntax from Sect. 8.1), validate it, and flatten it. A plugin architecture will be used to handle foundations.

In fact, the upcoming implementation is already the second implementation round. A first fully functional prototype was developed in an earlier stage and has recently become largely obsolete. For example, the switch from a bottom-up to a top-down inference system was made after experimenting with the prototype. Another lesson learned from the prototype was the use of the Curry-Howard correspondence in order to reduce the number of primitive concepts.

The most important short-term goal of the implementation is to achieve a design of the internal data structures that combines the modular and the flattened view. Neither extreme scales to large input sizes. Therefore, the flexible flattening results from Sect. 6.4 are crucial.

6.5.2 Small Conservative Changes

Renaming upon Imports After importing via $\underline{i} : S := \{\sigma\}$, it is often desirable to access a symbol $\underline{c}$ of S without using the qualifier $\underline{i}$. For example, when importing from monoid to group via an import `mon`, it is confusing to say `mon/comp` instead of simply `comp`. Therefore, we permit an additional kind of assignments, namely

$$Ass ::= \underline{c} \rightsquigarrow \underline{c'}$$

An import declaration $\underline{i} : S := \{\sigma\}$ where σ contains $\underline{c} \rightsquigarrow \underline{c'}$ is well-formed if a constant of name $\underline{c'}$ could be declared after $\underline{i}$. And then its semantics is that $\underline{c'}$ abbreviates $\underline{i}/\underline{c}$.

Conservative Extensions of Theories Extensions of theories with defined constants — which include theorems by the Curry Howard correspondence (see Sect. 6.5.3) — are often done outside the theory. For example, a theory T is defined first, then views in and out of T are obtained that eventually lead to an important theorem that should be added to the theory. Another example is that a user provides a small core theory, e.g., for ZFC, that is enriched step by step and by different authors with new definitions.

Currently, this is only possible by defining a new theory which imports T and then adds a new symbol. But it is desirable, to add the symbol directly to T so that existing views from T can be used to move it into other theories. Since the only requirement for such an operation is that a fresh name is chosen for the new symbol of T, this feature can easily be added to MMT.

6.5.3 Roles

Judgments as Types and Proofs as Terms MMT does not provide syntax for axioms, proof rules, or judgments. Instead, these concepts are subsumed by constants and terms via the Curry-Howard correspondence. However, it is often desirable to treat constants differently depending on whether they are intended to represent mathematical objects or axioms, in particular, when interfacing to a system that does not use the Curry-Howard correspondence (which includes most humans).

Therefore, we introduce the concept of **semantic roles**. Constants may be declared to have one of the roles listed in Fig. 6.23. These roles correspond to those of Def. 5.1. While the roles can be used for various management services, they are transparent to the formal definition of MMT.

	Value roles	Type roles
Objects	term	sort
Judgments	proof	judgment

Figure 6.23: Semantic Roles

Thus, the axioms, assertions, and proofs of non-Curry-Howard (nCH) languages (such as OMDoc 1.2) are subsumed by MMT constants. Fig. 6.24 gives the correspondence between MMT constants and nCH concepts. In particular, the definition of an MMT constant with role "proof" corresponds to the proof of an nCH theorem, and proof checking is reduced to type checking. In particular, the requirement that views must provide well-typed assignments for every constant means that a view must provide a proof in the codomain theory for every axiom of the domain theory. This is exactly the defining property of theory morphisms.

Note that the restriction to Curry-Howard languages is no loss of generality. Every language that talks about formulas and proofs can be made into a Curry-Howard language by adding a new constant `true` with role "judgment". Then the relation that p proves F can be regarded as the typing relation p:`true` F.

	Typed by atomic judgment	Typed by composed judgment
Undefined	axiom	proof rule
Defined	theorem	derived proof rule, proof

Figure 6.24: Axioms and Theorems

6.5.4 Unnamed Imports

The structures of MMT arose out of the need to provide a rigorous foundation for OMDoc. OMDoc used unnamed imports with instantiations, and this led to difficulties with disambiguating multiple imports. To alleviate the ensuing necessity of renaming, MMT replaced them with structures as named imports.

We now know that this was not the complete solution yet. A disadvantage of named imports is that multiple imports are always different, and references to imported symbols need to be qualified with the import name. In many cases this is not satisfactory, in particular when importing theories that should not be instantiated. For example, a theory for the natural numbers may be imported in theories S and S'. If a theory T imports both S and S', the two copies of the natural numbers must be explicitly equated. This is unnatural because it would not make sense to have two different copies of the natural numbers.

Therefore, it is not the right decision to replace OMDoc 1.2 imports with the new Mmt structures. Rather, the OMDoc 1.2 imports should be kept aside the new structures — except that the old imports should not carry morphisms anymore. Thus, the syntax of Mmt should be extended with productions

$$\vartheta ::= Imp^*, Sym^*$$

$$Imp ::= \texttt{import}\ S$$

where `import` S represents an unnamed import from S.

The relation "T imports from S" can be formalized as an ordering $S \leq T$ on the theory names. Then if $R \leq S$, (i) views from S to T are only well-formed if also $R \leq T$, and (ii) if T declares a structure of type S, this forces $R \leq T$ (but not necessarily $S \leq T$). It does not matter how often and along which path T imports S: All such unnamed imports are indistinguishable. This is reflected in the way how T can refer to the imported symbols: T refers to a symbol $\underline{s}$ of S simply by $S?\underline{s}$.

6.5.5 Subtheories

A previous version of Mmt already permitted subtheories, i.e., theory declarations that occur in other theories. The principal idea of the semantics of subtheories is that a subtheory $\underline{S} := \{\vartheta\}$ occurring within the body of a theory $g?\underline{T}$ is equivalent to a toplevel theory $g?\underline{T}/\underline{S} \stackrel{M}{:=} \{\vartheta\}$. Here M is the part of $g?\underline{T}$ that precedes the declaration of $\underline{S}$. Thus, $\underline{S}$ can refer to all symbols $\underline{s}$ of $\underline{T}$ declared before $\underline{S}$ via $../\underline{s}$. Views or imports from T ignore the subtheories of T.

Theories can import from their own subtheories so that they provide an alternative way to represent local functions: Subtheories and their instantiations can be used to represent function declarations and applications, respectively. It can also be very convenient to have global parameters, i.e., some symbols declared at the beginning of a theory that are shared by several subtheories.

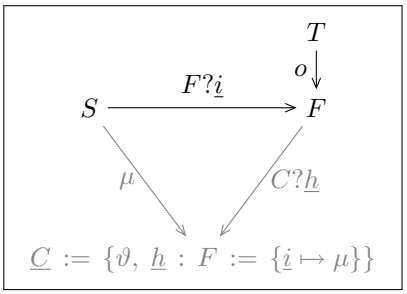
Another application of subtheories is that they can serve as a general scoping mechanism. A theory T can contain a subtheory for anything that locally declares new symbols that should not be copied when T is imported. Examples for such scoping constructs in mathematical documents are examples and exercises but also theorems and proofs.

However, the decision that the list of $\underline{T}$-declarations preceding $\underline{S}$ should comprise the meta-theory of $\underline{S}$ turned out to be unfortunate. In particular, this definition behaves very badly under reorderings. Therefore, we dropped subtheories for now. A promising redesign will consider only those symbols preceding $\underline{S}$ that are actually used in $\underline{S}$ as the meta-theory. However, this requires a stronger dependency analysis than employed so far: The meta-theory has to be computed because $\underline{S}$ may depend on symbols that do not occur literally in ϑ. Since such a formal dependency calculus will be needed for the management of change anyway (see Sect. 8.4), we defer the reintroduction of subtheories.

6.5.6 Functors

Mmt can represent functors via the pushout semantics employed by various module systems, e.g., OBJ ([GWM+93]). That means that a functor from a theory S to a theory T consists of a triple (i, F, o) where:

- F is a theory, the main expression representing the functor.
- i is a structure from S in F with no assignments. This structure imports the input interface S into the functor as uninstantiated symbols.
- o is a morphism from T to F. This morphism yields an interpretation of the output interface T in terms of F.



Consider the diagram on the right where C represents the local context, in which the functor (i, F, o) is applied to the morphism μ from S to C (i.e., to a morphism μ of type S over C). This is done by declaring a structure $\underline{h} : F := \{\underline{i} \mapsto \mu\}$, which instantiates the input interface S with the concrete interpretation of S via μ. And the result of this application is the morphism $o \bullet C?\underline{h}$, which is a morphism from T to C.

While this yields an elegant representation of functors, it is not all that is desirable. MMT is foundation-independent, and foundation-specific functors typically require a slightly different form. For example, the category of monoids can be represented as a theory M in MMT (say with first-order logic as the meta-theory). And every specific monoid can be represented as a theory morphism from M to ZFC where ZFC is a theory for the specific foundation — e.g., Zermelo-Fraenkel set theory — in which the monoid is expressed. Similarly, groups are represented as theory morphisms from a theory G to ZFC.

Then the well-known theory morphism from M to G induces a model reduction functor from the category of groups to the category of monoids. This functor — like all model reduction functors — can be easily expressed in MMT by composition. However, other functors cannot, e.g., the functor in the opposite direction assigning to every monoid its unit group. The latter functor takes a morphism $G \to ZFC$ as input and returns a morphism $M \to ZFC$. Very similar situations arise with other kinds of functors, e.g., SML functors have the same form with a theory SML instead of ZFC.

For that purpose, a stronger module system based on lambda abstractions over morphisms was developed in ([Aga08]). A typical expression in this language looks like this: $\lambda m : M \to ZFC.\sigma : G \to ZFC$, which has type $(M \to ZFC) \Rightarrow (G \to ZFC)$. This module system uses a simplified version of MMT that has still to be merged into the main version.

A generalized version could permit to combine statement level λ-abstraction over morphisms and object level λ-abstraction over terms. This could also remedy the problem that currently no quantification over morphisms is possible. For example, it is desirable to have an axiom of the form "For all x of type τ over T there is a view from S to T that maps the constant $\underline{c}$ of S to x.".

6.5.7 Informal Documents

One of the strongest aspects of OMDOC is that it scales to the case where informal methods must be used because a fully formal treatment is not feasible. Thus, adding informal elements to MMT is an important extension. However, this must be done in a controlled way: Too generous a relaxation of the syntax makes the format unintelligible for machines and thus useless. Three kinds of informal languages are particularly useful: natural language, programming languages, and program output.

In a first step, informal elements should only occur at the object level, i.e., terms and morphisms could be given informally. For example, constant definitions could be in semi-formal or informal natural language, which is especially useful for proofs (see also Sect. 6.5.8). This can be realized by adding a production $\omega ::= (Txt|\omega)^*$ where Txt is any string representing sentence fragments in natural language. For example, a semi-formal quantification typical of mathematical documents could be expressed as

$$\beta(\text{ for all },\alpha(x,\text{ with the property that } \mapsto P(x)),\text{ we have that } Q(x)).$$

Similarly, a semi-formal or informal specification of a software system could be expressed as a theory T that declares a constant of type R for every requirement R. Then views out of T must provide proofs for these requirements. Since the requirements are informal, the correctness

of such a proof — be it formal or informal — cannot be checked, but checking whether it is present or whether it changed compared to the last version is already extremely useful.

Software programs can be represented in MMT by representing, e.g., classes as theories and fields of a class as symbols. Invariants and other formal or informal comments can be attached to implementations as theorems. Then views from a specification to a program can relate every function symbol to the function implementing it, and every axiom to a proof. Here the proofs consist of natural language possibly referring to the theorems given about the implementation.

6.5.8 Structured Proofs

Currently MMT only supports proof terms. This is in contrast to the elaborate markup language for structured proofs provided by OMDOC 1.2. However, the formal semantics of the latter is unclear (but see [Coe05] for a partial semantics), and an important future project will be to extend the formal semantics of MMT to structured proofs.

Such proofs will be trees with judgments as nodes and proofs connecting every node with its children. Via the Curry-Howard treatment of judgments and proofs, it becomes possible to use terms for both judgments and proofs. This goes beyond OMDOC 1.2, which does not formalize the notion of inference rules.

This will be particularly useful in conjunction with informal documents as described in Sect. 6.5.7. Then arbitrary proof steps can be written informally. Simple examples of informal proof steps are references to the literature, omitted simple steps, or conjectures. If there are formal judgments that act as interfaces for the informal proofs by giving the assumptions and conclusion of the proof fragment formally, a validation of the whole proof relative to the correctness of the informal steps is possible. It is even possible to use flexible trust levels that permit researchers to select which informal steps a proof checker should accept.

We will look at two kinds of informal proof steps in more detail: borrowing steps and prover traces. These are interesting because they are relatively close to formal proof steps and are routine accepted as quasi-formal by many practitioners.

Borrowings translate proof goals to a different logic. Typically, the translation of the proof goals is significantly easier than the proof that the borrowing is sound. In principle, there are two ways how to establish the soundness of borrowing: proof theoretically by translating the obtained proof back to the original logic, or model theoretically by exhibiting a model translation between the two logics.

The model theoretical argument is usually much easier to carry out than the proof theoretical argument, but it is always semi-formal. Furthermore, often the translated proof goal is discharged using an automated prover that does not even return a proof that could be verified, let alone translated back. However, in practice researchers consider borrowing proofs as relatively reliable even though a formal soundness proof is missing (e.g., in [BPTF07] and [Bro06]). In such situations, the soundness of borrowing can be given as an informal proof and a specific application of borrowing as a formal proof step. This is the starting point for the investigation of the concept of heterogeneous proof terms, which combine formal and informal reasoning and logic translations.

Prover traces are the output produced by automated provers: Many of these return structured information, from which — in principle — a formal proof can be constructed. But in practice it is often far too expensive to do that. However, it is possible to treat the output or the log file as an informal proof, which is particularly interesting if the used axioms are marked up. In the latter case the MMT dependency analysis could use this information, e.g., for the management of change.

In the last two examples, flexible trust levels would mean that users can configure which logic translations and which automated provers they consider trustworthy.

6.5.9 Abstractions

Finally, we mention an important concept that is not fully understood yet. Consider a theory ZFC for set theory, in which the natural numbers are defined, and assume we want to interpret the natural numbers in some other theory, e.g., a theory `ring0` for rings with characteristic 0. There could be a theory `Nat`, which imports ZFC and declares constants for 0, `succ`, and each of the Peano axioms. All of these would be defined constants, including those for the Peano axioms, which are theorems in set theory. Intuitively, there is a view from `Nat` to `ring0` which maps 0 to 0 and `succ` to the function adding 1.

But currently this is impossible to express. The symbols of `Nat` are defined in terms of ZFC, and they cannot be instantiated by a view. To interpret them in `ring0`, we would need a view from ZFC to `ring0`, which is not possible. What is needed here is to somehow "undefine" the natural numbers in a way that forgets their definition in terms of ZFC but keeps their properties. In particular it should only undefine the Peano axioms, and not every property proved about the natural numbers. This should result in a new theory, from which a view into `ring0` can be given.

Another possibility to express the relation between `Nat` and `ring0` is to introduce a new theory that defines the natural numbers axiomatically and then give views from it into `Nat` and `ring0`. This is related to the hiding of ASL ([SW83]).

A similar problem arises when giving a view from the real numbers R of HOL Light ([Har96]) to the real numbers R' of Isabelle/HOL ([NPW02]). Both use type theoretical foundations (in fact, essentially the same one), in which the real numbers are defined. But no view can map R to R' because the two systems use different definitions of the real numbers. Here, we would want to translate all constants of R to their analogues in R' irrespective of their definitions. (This problem was addressed in [OS06], but no formal notion of soundness was used because the translated theorems were reproved in Isabelle.)

Somewhat different but still related is the feature of reflection. Some languages are able to represent themselves. For example, assume a theory DTT for dependent type theory that declares untyped, undefined constants, e.g., for `type` and `lambda`. Then there is a theory DTT_r with meta-theory DTT that declares the same constants, but using DTT-terms as types. For example, DTT_r would declare a symbol `type` : DTT_r?../`type` := $\bot$. To express reflection, we would have to exhibit a view from DTT_r to DTT. This view would have to map DTT_r?`type` to DTT?`type` and thus conflate DTT_r?`type` and DTT_r?../`type`; but then it cannot preserve typing anymore (except for degenerate cases).

We contend that finding a simple solution that covers such applications will provide valuable insights into the nature of translations.

Acknowledgments The work presented here is the result of a collaboration with Michael Kohlhase and together with Sect. 7 core of a joint paper ([RK08]). We collaborated with Elena Agapie on problems related to Sect. 6.5.6 ([Aga08]).

Chapter 7

Representing Foundations and Logics in MMT

It is clear from the definitions that most formal systems can be represented in MMT. As examples, we give a type theoretical and a set theoretical foundation using DTT and ZFC in Sect. 7.1 and 7.2. Then in Sect. 7.3, we go one step further and represent the logical framework $\mathbb{L}$ of Sect. 5 in MMT. This constitutes the climax of this text where type and set theoretical foundations and model and proof theoretical logical frameworks are unified in MMT as a foundation-independent meta-framework. We conclude the formal investigation of Part II in Sect. 7.4.

7.1 DTT as a Foundation

We give a foundation for dependent type theory. For some appropriate URI g, we use the document below to describe DTT. The given symbols give the necessary declarations for the Edinburgh Logical Framework, LF ([HHP93, Pfe01]). The horizontal dots indicate further symbols depending on the specific variant of DTT chosen; the same treatment can be applied to all subsystems of DTT as defined in Sect. 4.

$$g := \{\texttt{DTT} := \{\texttt{type}, \texttt{kind}, \texttt{lambda}, \texttt{Pi}, \texttt{oftype}, \ldots\}\}$$

Thus, the theory $DTT := g?\texttt{DTT}$ only declares untyped, undefined symbols for the basic operations of DTT. Application is not needed because we can express it using the application of MMT. `oftype` is needed to ascribe types to variables in binders. The document g should be made accessible at a reasonable location, e.g., by picking `http://cds.omdoc.org/logical_frameworks/dtt.omdoc` for g and answering requests for that URL with an MMT-aware database. Using the extensions of Sect. 6.5.7, the semantics of these symbols can be attached in natural language.

Then we give a foundation Φ^{DTT} for the set $\{DTT\}$ of theory names. It is defined as follows.

- $(\Phi^{DTT})^{\Lambda,i}(\omega \equiv \omega')$ according to the $\alpha\beta\eta$-equality of DTT.
- $(\Phi^{DTT})^{\Lambda,i}(\omega : \omega')$ according to:

$\overline{\omega}\backslash\overline{\omega'}$	$\bot$	ω'	$\top$
$\bot$	-	$\vdash^{DTT} \overline{\omega'} : \texttt{type}$ or $\vdash^{DTT} \overline{\omega'} : \texttt{kind}$	-
ω	-	$\vdash^{DTT} \overline{\omega} : \overline{\omega'}$	-
$\top$	+	+	+

  This means that $\bot$ type-checks against those terms that are well-formed types and kinds in DTT. This permits to declare typed or kinded constants in theories. $\top$ type-checks against every term as required for regular foundations. And for the remaining terms, typing is reduced to the typing and kinding judgment of DTT.

Here, the judgments $\vdash^{DTT} \omega \equiv \omega'$ and $\vdash^{DTT} \omega : \omega'$ hold iff the corresponding judgments for the used variant of dependent type theory hold for closed expressions. These judgments are the ones given in Sect. 4 with a small modification of the rules:

- $@(\omega_1, \omega_2)$ is treated as $\omega_1\ \omega_2$.
- $\beta(i/\texttt{lambda}, \alpha(x, i/\texttt{oftype} \mapsto \omega), \omega')$ is treated as $\lambda_{x:\omega}\, \omega'$.
- $\beta(i/\texttt{Pi}, \alpha(x, i/\texttt{oftype} \mapsto \omega), \omega')$ is treated as $\Pi_{x:\omega}\, \omega'$.
- Similar adjustments for the other term, type family, and kind constructors present in the specific variant of DTT used.
- The rules T_a of Fig. 4.4 and t_c of Fig. 4.5 (which are the only syntax formation rules that use the signature Σ) are replaced with

$$\frac{\Lambda^T(\underline{c}) = (\tau, \_) \quad \vdash \Gamma\ \texttt{Ctx}}{\Gamma \vdash T?\underline{c} : \overline{\tau}}\ .$$

These modifications make the judgments $\vdash^{DTT} \omega \equiv \omega'$ and $\vdash^{DTT} \omega : \omega'$ independent of a particular signature because all name lookups of constants are handled via the foundation-independent MMT-lookup $\Lambda^T(-)$. Here Λ is the global state and T defines the current scope. Because no signature needs to be maintained by the foundation, the implementation of Φ^{DTT} is not only straightforward, but in fact easier than a stand-alone implementation. On the downside, no advanced features like the reconstruction of implicit arguments can be handled. Finally, we have:

Lemma 7.1. *Φ^{DTT} for full dependent type theory as introduced in Sect. 4 is regular. It is decidable if restricted to LF.*

Proof. Regularity is straightforward. Decidability is a basic result about LF (see, e.g., [HHP93]). □

7.2 ZFC as a Foundation

Now we sketch a foundation for set theory. We will not go into the details so that our presentation applies to a variety of set theories, e.g., Zermelo-Fraenkel set theory with choice.

Let ST be the URI of a theory for set theory. ST declares at least a constant `Set` for the collection of all sets, constants for the basic operations of first-order logic, implicit definitions, elementhood, and the axioms, and other constants for the basic notions of the specific set theory. A foundation Φ^{ST} for $\{ST\}$ is given by:

- $(\Phi^{ST})^{\Lambda,i}(\omega \equiv \omega')$ according to:

$\omega \backslash \omega'$	ω'	$\top$
ω	$\omega \doteq \omega'$ provable	-
$\top$	-	+

- $(\Phi^{ST})^{\Lambda,i}(\omega : \omega')$ according to:

$\omega \backslash \omega'$	$\bot$	ω'	$\top$
$\bot$	-	-	-
ω	-	at least if ω is a set and $\omega' = ST?\texttt{Set}$	-
$\top$	+	+	+

Here the equality of Φ^{ST} is defined with reference to provability in the logic of the set theory, e.g., FOL with implicit definitions. Of course, this foundation is not decidable (unless the chosen set theory is inconsistent). As to the typing relation, there is a variety of possibilities how to

use some typing system in set theory (e.g., ZFC could be defined as in Ex. 5.38), and we do not intend to go into the details here. But the important thing it that all expressions denoting sets have type $ST?\texttt{Set}$. Therefore, any theory with meta-theory ST can introduce constants of type $ST?\texttt{Set}$ that have some set as a definiens, but it cannot introduce undefined constants. This is how set theory is usually developed as a large conservative extension of a small (possibly inconsistent) theory. Details can be found in any textbook on axiomatic set theory (e.g., [HJ84]).

Having a foundation for set theory permits to integrate the OpenMath standard content dictionaries ([BCC+04]) into Mmt. This is non-trivial because Mmt focuses on logical theories, which usually have uninterpreted constants, whereas the standard content dictionaries provide declarations (with types and definitions in natural language) for the constants occurring in math curricula of schools and undergraduate university courses. The latter is (more or less explicitly) based on some set theory. So we can say: The OpenMath standard content dictionaries can be understood as Mmt-theories with an appropriate meta-theory for set theory.

7.3 Representing a Logical Framework in Mmt

In the previous sections, we introduced theories DTT and ST and Mmt foundations for them. Building on that, we can now represent the logic encodings in $\mathbb{L}$ given in Sect. 5 in Mmt. In short, logics and logical theories are represented as Mmt theories with meta-theory DTT, and models are represented as Mmt theory morphisms into ST.

The following descriptions are visualized in Fig. 7.1. Using DTT, we define the theory

$$\texttt{log} \overset{DTT}{:=} \{o : \texttt{type} := \bot,\ \texttt{true} : o \to \texttt{type} := \bot\}$$

defined in some document g such that $Log := g?\texttt{log}$. Here we use LF syntax instead of the more verbose Mmt syntax to represent the types. Then a logic is represented as a theory L with meta-theory DTT along with a morphism l from Log to L. This corresponds to an $\mathbb{L}$ signature $(\Sigma; \Delta; R; \underline{o}; \underline{true})$ where Σ and Δ are given by L, and $\underline{o}$ and $\underline{true}$ are given by l. To present R, we need the extension of Mmt with role assignments that was already announced in Sect. 6.5.3.

Theories of a logic (L, l) are represented as Mmt theories T with meta-theory L. And theory morphisms from an (L, l)-theory S to an (L, l)-theory T are represented as Mmt morphisms μ from S to T. The sorts, terms, judgments, and proofs are explicitly represented by constants declared in L, possibly enriched by further constants in T. Then the Mmt terms over T (typed according to the foundation Φ^{DTT}) give the sentences, judgments, and proofs over T. In particular, $Log?\texttt{true}^{l \bullet T?..}$ represents the truth judgment in T. Sentence, judgment, and proof translation along μ is given by morphism application in Mmt.

Models of the theories S and T are represented as Mmt theory morphisms into ST. Here different variants of set theories can be chosen for ST depending on the set theory in which the models are expressed, for example Tarski-Grothendieck set theory for models described in the Mizar system ([Tar38, Bou64, TB85]). More generally, arbitrary theories can be used to express the models, e.g., type theoretical foundations such as in Isabelle/HOL ([NPW02]). Model reduction along μ of a T-model I' to an S-model I is simply morphism composition in Mmt.

Satisfaction of a T-formula ω in a model I' is represented by the equality in ST (as defined by the foundation Φ^{ST}). If the term $@(Log?\texttt{true}^{l \bullet T?..}, \omega)^{I'}$ is equal to the empty set, then ω is false.

Similarly, logic comorphisms from (L, l) to (L', l') can be represented by an Mmt morphism ν from L to L' (which does not necessarily satisfy $l \bullet \nu = l'$ according to Def. 5.9). In simple cases, the translation of T to (L', l') is a theory T' with meta-theory L' whose body only consists of an import from T, namely $\underline{i} : T \overset{\nu \bullet T'?..}{:=} \{\cdot\}$. Then the inter-logic sentence translation from T to T' and model reduction from T' to T are induced by the morphism $T'?\underline{i}$ just like in the

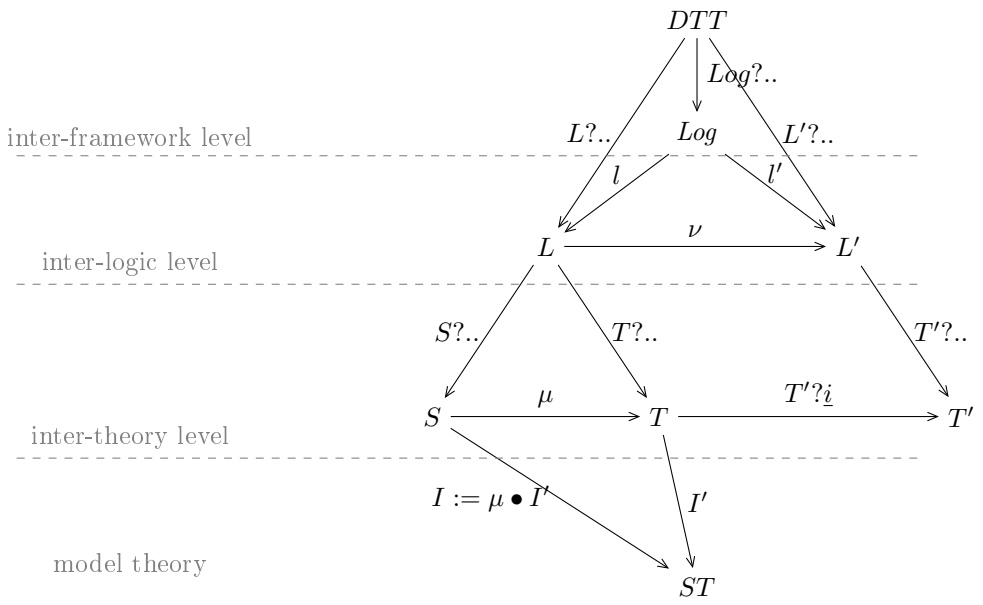


Figure 7.1: Logics in MMT

inter-theory case. It is also possible to compose the inter-theory translation μ and the inter-logic translation $T'?\underline{i}$.

Thus, we represent logical theories as pairs of a theory and its meta-theory and translations between them as pairs of a morphism and a meta-morphism. This is very similar to the use of Grothendieck institutions ([Dia02, Mos05]) in the context of institutions. Our approach is less expressive regarding the kinds of inter-logic translations; on the other hand, it extends elegantly to the case of inter-framework translations.

Example 7.2. The translation from ML to FOL of Ex. 5.37 can be directly represented in this way. L and L' are the MMT representations of the theories for ML and FOL given in Fig. 5.3 and 5.2. And ν is the MMT representation of the morphism given in Fig. 5.9. The assignment of T' and $T'?\underline{i}$ to T corresponds to the assignment of $\Phi(T)$ and m_T in the logic comorphism modification given in Ex. 5.37.

There are some limitations to our approach. It remains open for now how to represent in MMT the model morphisms and the translation of (L, l)-theory morphisms to (L', l)-theory morphisms (for which a pushout should be used). Furthermore, more complex logic comorphisms cannot be represented easily: The translation from ML to FOL has the special property that the constants of T are in bijection with those of $\Phi(T)$. But this need not be the case in general. For example, in the translation from DFOL of Ex. 5.30 to FOL given in [Soj08], $\Phi(T)$ contains one constant for every constant of T plus a variety of other constants defined by clauses of the form "For all constants $\underline{c}$ of type τ of T, $\Phi(T)$ contains ....". Even after introducing complex functors as outlined at the end of Sect. 6.5.6, this cannot be represented.

And finally, there is no way to specify which MMT-theories with meta-theory L should be considered logical theories. And a similar argument holds for the models. For example, if (L, l) represents FOL, then only function and predicate symbols should be declared in T. But currently every well-formed declaration is allowed. It is an interesting direction of future research to limit the theories T by declaration patterns. These patterns could then also be used when specifying translations thus solving both problems at the same time.

7.4 Conclusion

In Sect. 6, we introduced MMT, a module system for logical knowledge. Its most characteristic features are that it represents logical frameworks, logics, logical theories, and their translations uniformly as theories and theory morphisms in a way that we call the **logics-as-theories** approach, and that it is generic in the underlying non-modular language in a way that we call **foundation-independence**.

The logics-as-theories approach yields a two-dimensional structure of theories mediated by theory morphisms. Along one dimension, theories represent the levels of logical reasoning: logical frameworks, logics, and theories. Along the other dimension, theories are interconnected by theory morphisms that represent instantiations and translations of the module system. This provides an efficient and simple way of maintaining logical knowledge in highly modularized form.

At the same time the module system is conservative over the underlying non-modular language. This foundation-independence is realized by relegating the information about typing and equality, which captures the semantics of a language represented in MMT, to extraneous components. This creates an interface that encapsulates on the one side the intelligence-heavy formalizations of typing and equality of specific foundations, and on the other side the global state that is maintained and manipulated by low-intelligence knowledge management systems. Since MMT can relegate the semantics to foundations, the design becomes very extensible while still retaining a formal semantics, and we outlined some future extensions that can make MMT significantly more expressive.

These two features make MMT highly flexible and expressive. Many modular languages from distinct areas of mathematics and computer science can be represented naturally in MMT. At the same time, the syntax of MMT is rather lean: It is geared towards capturing the structural properties of the theories and their translations as well as their aggregation in documents and libraries. Many advanced features that would otherwise complicate the language — such as recursion, pattern matching, or implicit definitions — can be added when needed by using appropriate meta-theories or meta-meta-theories.

In Sect. 7, we showed how both set/model and type/proof theoretical logical reasoning in general as well as logic encodings in the specific logical framework $\mathbb{L}$ can be represented elegantly in MMT. While there are still some limitations (e.g., model morphisms and the signature translation in complex logic comorphisms), MMT goes a long way in representing logics and logic translations. In particular, MMT captures the most important aspects for logical reasoning: propositions and model and proof theory, and the translations of propositions, models, and proofs on all levels of logical reasoning.

An important consequence of the design of MMT is that it becomes efficient to implement and maintain a web-scalable architecture for MMT. This scalability lets MMT utilize the full potential that module systems have over non-modular developments. And the extent to which this is possible with MMT is one of the main contributions of this work. We will look at this in more detail in Sect. 8.

Acknowledgments The work presented here is the result of a collaboration with Michael Kohlhase and together with Sect. 6 core of a joint paper ([RK08]). A very early version appeared as [Rab07].

Chapter 8

Towards a Web-Scale Infrastructure for Logical Knowledge

The module system presented in Sect. 6 is an integral part of the upcoming OMDoc 2 document format, which will supersede OMDoc 1.2 ([Koh06]). Following practice in MathML ([ABC+03]), OMDoc 2 will consist of a **strict** and a **pragmatic** language. Pragmatic OMDoc will be of similar flavor as OMDoc 1.2 and will be (partially) elaborated into strict OMDoc. For logical knowledge, mainly the strict variant is important.

Mmt constitutes the formal abstract syntax corresponding to the strict OMDoc 2 language. And in this section, we sketch our vision for the concrete XML-based infrastructure for Mmt. Sect. 8.1 gives the RelaxNG grammar ([CM01]) for the XML syntax and defines the XML encoding of Mmt in it. Sect. 8.2, 8.3, and 8.4 present the design of the three most promising applications of generic logical knowledge management services to Mmt: persistent storage, presentation, and management of change.

8.1 Strict OMDoc 2

8.1.1 XML Syntax

A somewhat simplified version of our proposed strict OMDoc 2 syntax is given by the RelaxNG ([CM01]) grammar listed in Fig. 8.1. It already accounts for the addition of unnamed imports (see Sect. 6.5.4) and semantic roles (see Sect. 6.5.3). The grammar for objects is omitted because it is the same as for OpenMath.

The XML grammar mostly follows the abstract grammar. Documents are `omdoc` elements with a theory graph, i.e., `theory` and `view` elements as children. The children of theories are `constant` and `structure`. And the children of views and structures are `maps` (assignment to a constant or structure). We also allow for `imports` children of theories that represent unnamed import declarations.

Both terms and morphisms are represented by OpenMath elements (`mterm` and `mmorph` in the XML grammar). And all names of theories are URIs. The grammar does not account for the well-formedness of objects and names. In particular, well-formedness (see Sect. 6.3) is not checked at this level.

Formally, we define an encoding function $E(-)$ that maps Mmt-expressions to sequences of XML elements. The precise definition of $E(-)$ is given in Fig. 8.3 and 8.4 where we assume that the following namespace bindings are in effect:

```
xmlns="http://www.omdoc.org/ns/omdoc"
xmlns:om="http://www.openmath.org/OpenMath"
```

```
default namespace omdoc = "http://www.omdoc.org/ns/omdoc"

#useful abbreviations
from.attrib = attribute from {ModuleName}
to.attrib = attribute to {ModuleName}
name.attrib = attribute name {xsd:string}
ModuleName = URI

start = omdoc

#document level
omdoc = element omdoc {theorygraph}
theorygraph = theory* & view*

#module level
theory = element theory {name.attrib, metatheory?, theorybody}
metatheory = attribute metatheory {ModuleName}
theorybody = object* & structure* & imports*

view = element view {name.attrib, from.attrib, to.attrib, linkbody}
metamorphism = element metamorphism {mmorph}
linkbody = (metamorphism?, conass* & strass*) | element definition {mmorph}

imports = element imports {from.attrib}

#symbol level
constant = element constant {name.attrib, attribute semrole {semrole}?, type?, definition?}
semrole = "term" | "sort" | "proof" | "judgment"
type = element type {mterm}
definition = element definition {mterm}

structure = element structure {name.attrib, from.attrib, linkbody}

conass = element conass {name.attrib, mterm}
strass = element strass {name.attrib, mmorph}

#object level
mterm = mobj
mmorph = mobj

#include external scheme to define syntax for objects
mobj = grammar {include "openmath3.rnc"}
```

Figure 8.1: RelaxNG Scheme

Library	$E(Doc_1, \ldots, Doc_n)$	$E(Doc_1)\ \ldots\ E(Doc_n)$
Document	$E(g := \{Mod_1, \ldots, Mod_n\})$	<**omdoc**>$E(Mod_1)\ldots E(Mod_n)$</**omdoc**>
Theory	$E(\underline{T} \stackrel{[M]}{:=} \{Sym_1, \ldots, Sym_n\})$	<**theory** name="$\underline{T}$" [metatheory="$E(M)$"]> $E(Sym_1)\ldots E(Sym_n)$ </**theory**>
View	$E(\underline{v} : S \to T \stackrel{[\mu]}{:=} \{\sigma\})$	<**view** name="$\underline{v}$" from="$E(S)$" to="$E(T)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**view**>
	$E(\underline{i} : S \to T := \mu)$	<**view** name="$\underline{i}$" from="$E(S)$" to="$E(T)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**definition**> </**view**>

Figure 8.3: XML Encoding of Document and Module Level

And we assume the following content dictionary with cdbase `http://cds.omdoc.org/omdoc/mmt.omdoc`, which is itself given as an OMDoc theory:

```
<omdoc>
  <theory name="mmt">
    <constant name="hidden"/>
    <constant name="identity"/>
    <constant name="composition"/>
  </theory>
</omdoc>
```

We abbreviate the `OMS` elements referring to these symbols by *OMDoc*(`identity`) etc.

The encoding of the structural levels is straightforward. The encoding of objects is a generalization of the XML encoding of OpenMath objects. We use the `OMS` element of OpenMath to refer to symbols, theories, views, and structures. The symbol with name *OMDoc*(`hidden`) is used to encode the special term $\top$. To encode morphisms, *OMDoc*(`identity`) takes a theory as an argument and returns a morphism. *OMDoc*(`composition`) takes a list of structures and views as arguments and returns their (left-associative) diagram order composition. Morphism application is encoded by reusing the `OMA` element from OpenMath.

The encoding of names is giving separately in Fig. 8.2 on the right. There are two different ways to encode names. In `OMS` elements, triples $(g, \underline{m}, \underline{s})$ of document, module, and symbol name are used, albeit with more fitting attribute names. These triples correspond to the $(cdbase, cd, name)$ triples of the OpenMath standard ([BCC+04]). We also use them to refer to module level names by omitting the *name* attribute. When names occur in attribute values, their URIs are used.

`OMS`-triple	$E(g?\underline{m}?\underline{s})$	base="g" module="$\underline{m}$" name="$\underline{s}$"
	$E(g?\underline{m})$	base="g" module="$\underline{m}$"
URI	$E(g?\underline{m}?\underline{s})$	$g?\underline{m}?\underline{s}$
	$E(g?\underline{m})$	$g?\underline{m}$

Figure 8.2: XML Encoding of Names

8.1.2 Relative Names

In practice it is very inconvenient to always give qualified names. Therefore, we define relative references as a relaxation of the syntax that is elaborated into the official syntax.

A **relative reference** consists of three optional components: a document reference g, a module reference m and a symbol reference s. We write relative references as triples (g, m, s) where we write $\bot$ if a component is omitted. g must be a URI reference as defined in RFC 3986 ([BLFM05]) but without query or fragment. m and s must be unqualified names, i.e., slash-separated non-empty sequences of names. Furthermore, m and s may optionally start

Constant	$E(\underline{c} : [\tau] := [\delta])$	<**constant** name="$\underline{c}$"> [<**type**> <**OMOBJ**>$E(\tau)$</**OMOBJ**> </**type**>] [<**definition**> <**OMOBJ**>$E(\delta)$</**OMOBJ**> </**definition**>] </**constant**>
Structure	$E(\underline{i} : S \stackrel{[\mu]}{:=} \{\sigma\})$	<**structure** name="$\underline{i}$" from="$E(S)$"> [<**metamorphism**>$E(\mu)$</**metamorphism**>] $E(\sigma)$ </**structure**>
	$E(\underline{i} : S := \mu)$	<**structure** name="$\underline{i}$" from="$E(S)$"> <**definition**> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**definition**> </**structure**>
Assignments	$E(Ass_1, \ldots, Ass_n)$	$E(Ass_1) \ldots E(Ass_n)$
	$E(\underline{c} \mapsto \omega)$	<**conass** name="$E(\underline{c})$"> <**OMOBJ**>$E(\omega)$</**OMOBJ**> </**conass**>
	$E(\underline{i} \mapsto \mu)$	<**strass** name="$E(\underline{i})$"> <**OMOBJ**>$E(\mu)$</**OMOBJ**> </**strass**>
Term	$E(c)$	<**om:OMS** $E(c)$/>
	$E(x)$	<**om:OMV** name="x"/>
	$E(\top)$	*OMDoc*(`hidden`)
	$E(\omega^{\mu})$	<**om:OMA**>$E(\mu)$ $E(\omega)$</**om:OMA**>
	$E(@(\omega_1, \ldots, \omega_n))$	<**om:OMA**>$E(\omega_1)$ ... $E(\omega_n)$</**om:OMA**>
	$E(\beta(\omega_1, x_1, \ldots, x_n, \omega_2))$	<**om:OMBIND**> $E(\omega_1)$ <**om:OMBVAR**> $E(x_1)$ ... $E(x_n)$ </**om:OMBVAR**> $E(\omega_2)$ </**om:OMBIND**>
	$E(\alpha(\omega_1, \omega_2 \mapsto \omega_3))$	<**om:OMATTR**> <**om:OMATP**>$E(\omega_2)$ $E(\omega_3)$</**om:OMATP**> $E(\omega_1)$ </**om:OMATTR**>
Morphism	$E(id_T)$	<**om:OMA**> *OMDoc*(`identity`) <**om:OMS** $E(T)$/> </**om:OMA**>
	$E(\mu_1 \bullet \ldots \bullet \mu_n)$	<**om:OMA**> *OMDoc*(`composition`) $E(\mu_1) \ldots E(\mu_n)$ </**om:OMA**>
	$E(i)$	<**om:OMS** $E(i)$/>
	$E(v)$	<**om:OMS** $E(v)$/>

Figure 8.4: XML Encoding of Symbol and Object Level

with a slash, which is used to distinguish absolute module and symbol references from relative ones.

An **absolute reference**, which serves as the base of the resolution, is an MMT-name G, $G?\underline{M}$, or $G?\underline{M}?\underline{S}$. Then the resolution of relative references is a partial function that takes a relative reference $R = (g, m, s)$ and an absolute reference B as input and returns an MMT-name $resolve(B, R)$ as output. It is defined as follows:

- If $g \neq \bot$, then possible starting slashes of m and s are ignored and
  - if $R = (g, m, s)$: $resolve(B, R) = (G + g)?m?s$,
  - if $R = (g, m, \bot)$: $resolve(B, R) = (G + g)?m$,
  - if $R = (g, \bot, \bot)$: $resolve(B, R) = G + g$,

  where $G + g$ denotes the resolution of the URI reference g relative to the URI G as defined in RFC 3986 ([BLFM05]).
- If $g = \bot$ and $m \neq \bot$, then a possible starting slash of s is ignored and
  - if $R = (\bot, m, s)$: $resolve(B, R) = G?M + m?s$,
  - if $R = (\bot, m, \bot)$: $resolve(B, R) = G?M + m$,

  where $M + m$ resolves m relative to M: If M is not defined or if m starts with a slash, $M + m$ is m with a possible starting slash removed; otherwise, it is M/m.
- If $g = m = \bot$ and M is defined, then $resolve(B, R) = G?M?S + s$, where $S + s$ is defined like $M + m$ above.
- $resolve(B, R)$ is undefined otherwise.

Relative references can also be encoded as URIs: The triple (g, m, s) is encoded as $g?m?s$. If components are omitted, they are encoded as the empty string. Trailing (but not leading) ? characters can be dropped. For example,

- $(g, m, \bot)$ is encoded as $g?m$,
- $(\bot, /m, s)$ is encoded as $?/m?s$,
- $(\bot, \bot, s)$ is encoded as $??s$,

This encoding can be parsed back uniquely by splitting a URI into up to three components around the separator ?.

8.2 A Smart Logical Database

An MMT library Λ can be physically represented by a file system or a web server where every document corresponds to one file. An MMT-aware database can provide the name lookup from Sect. 6.2.2 as a service and permit to retrieve the XML encoding of the lookup of an arbitrary name. In particular, an MMT-library can be implemented as a restful web server ([Fie00]). Such a server would answer HTTP GET requests with the result of the corresponding lookup. This is made precise in Fig. 8.5, where we also use URIs of the form $g?\underline{v}?\underline{c}$ to retrieve the lookup in a view $g?\underline{v}$. We will look at PUT, POST, and DELETE requests in Sect. 8.4.

Among the advanced services that a smart database could offer are search, ABox extraction, presentation, flattening, and document aggregation.

For GET requests, the retrieval from a smart database is possible by hashing URIs. But for object level retrievals, more complex algorithms must be employed. To **search** for terms

URI	Lookup		XML element of encoding
g	$\Lambda(g)$		`omdoc`
$g?\underline{T}$	$\underline{T} \stackrel{[M]}{:=} \{\vartheta\}$	if $\Lambda(g?\underline{T}) = ([M], \vartheta)$	`theory`
$g?\underline{v}$	$\underline{v} : S \to T \stackrel{[\mu]}{:=} \{B\}$	if $\Lambda(g?\underline{v}) = (S, T, [\mu], B)$	`view`
$g?\underline{T}?\underline{c}$	$\underline{c} : \tau := \delta$	if $\Lambda^T(\underline{c}) = (\tau, \delta)$	`constant`
$g?\underline{T}?\underline{i}$	$\underline{i} : S \stackrel{[\mu]}{:=} \{B\}$	if $\Lambda(g?\underline{T}?\underline{i}) = (S, T, [\mu], B)$	`structure`
$g?\underline{v}?\underline{c}$	$\Lambda^m(\underline{c})$		`conass`
$g?\underline{v}?\underline{i}$	$\Lambda^m(\underline{i})$		`strass`

Figure 8.5: Answers to HTTP GET Requests

that conform to a certain pattern (both by unification or generalization), the substitution tree indexing of ([KŞ06]) can be applied, possibly in a way optimized for MMT. There are numerous applications of such queries, e.g., the search for applicable lemmas during theorem proving.

The retrieval of morphisms between two theories can be reduced to a path search in the theory graph. More generally, one is interested in retrieving all such morphisms that agree with a given partial morphism. The latter is a crucial research question in inter-theory reasoning: When building a view from S to T, where S contains $\underline{i} : R := \{\sigma\}$, one is interested in finding all morphisms μ from R to T that agree with a certain partial morphism m as in the judgment $\Lambda \rhd_T S?\underline{i} \mid^m \mu : R$. Those are available for assignments $\underline{i} \mapsto \mu$, which are needed when trying to find a view. This corresponds to finding decompositions of global theorem links in the development graph calculus [AHMS99]).

A service for **ABox extraction** would accept as input a theory, document, or directory in the database. Then it would compute and return the assertional box for it according to an appropriate MMT document ontology in some description logic ([BCM+03]). The output format would be, e.g., RDF triples ([RDF04]). Such a service is important for the smart navigation through a library. In an OMDOC ontology, individuals are represented by URIs, and the ontology contains unary predicates for documents, theories, views, modules, constants, structures, symbols, assignments to constants, assignments to structures, assignments, and links, and the following binary relations between URIs X and Y:

- X is a module declared in document Y,
- theory X has meta-theory M (transitive),
- theory X has unnamed import from Y (transitive),
- X is a symbol declared in theory Y,
- X is an assignment declared in link Y,
- link X has domain Y,
- link X has codomain Y,
- symbol X arises by importing symbol Y,
- symbol X occurs in the declaration of symbol Y (transitive).

These binary relations are the starting point of a dependency calculus used to axiomatize a semantic dependency ordering between the URIs (see Sect. 8.4).

A **presentation** service (see Sect. 8.3) retrieves a document (fragment) and filters it through a presentation engine that applies mathematical notations and other syntax transformations.

The expected output format of the presentation can be selected by additional parameters, e.g., the content type of the GET request.

A **flattening** service filters a retrieved theory through an implementation of the flattening from Sect. 6.4. Such a service could also respond to the request of a structure with the appropriate translation of the imported theory. Similarly, a **document aggregation** service also retrieves all knowledge items that are referenced by a theory. But instead of using them to flatten the requested theory, the theories and views are recombined into a new self-contained document.

8.3 Presenting Logical Knowledge on the Web

The presentation of documents is conceptually more complex than one might expect at first. In principle, the presentation process is a recursive function that transforms the content-oriented representation of the document into human- or machine-readable formats. To permit a flexible presentation, the presentation process must be parametric in the specific cases of this recursion so that users can influence the presentation. We call these cases **notations**. OMDoc 1.2 already allowed the user to specify notations, and the syntax for notations will be completely redesigned in OMDoc 2.

The question how to choose a specific notation is non-trivial. Some of the most important problems are:

- A few notations common in mathematics require non-compositional presentation functions, e.g., the notation $\sin^2 x$ for $(\sin x)^2$.
- Different target formats may require different notations, e.g., `&bull;` for HTML and `\bullet` for Latex. But, some notations are independent of the target format, e.g., associativity of $\bullet$.
- The presentation must generate brackets according to operator precedences. And the placement of brackets must itself be flexible, e.g., $f(x)$ or $(f\ x)$.
- Notations must be chosen according to user preferences including the inference of notations according to the system's knowledge about the user, e.g., C_n^k or $\binom{n}{k}$ for the binomial coefficient.
- Depending on the definition of notations, imported notations may have to be translated.
- There may be conflicting notations for the same expression, e.g., notations may be provided by the presented document itself (possibly via imports) or by the user or system requesting the presentation.
- Notations may have to be adapted dynamically when viewing (as opposed to statically when generating) the presentation, e.g., users could be asked to select a notation when the system was unable to decide among conflicting notations.
- Parts of the presentation must be elidable, i.e., users must be able to switch certain objects on or off, e.g., redundant brackets or implicit arguments.
- Different parts of a document may require different notations, e.g., when dropping brackets after proving associativity.
- Some functions are flexary in the sense that they take a list of flexible length of arguments, e.g., the associative binary operator $\bullet$ of Mmt is treated as a flexary operator in the XML encoding to avoid nested applications.

- Users may wish to specify notations for all theories with a certain meta-theory, e.g., $(f\ x_1\ \ldots\ x_n)$ for all applications of constants f declared in a theory with meta-theory DTT (and for arbitrary n).

- There are various complex notations such as ellipses $x_1, \ldots, x_n$ (possibly multi-dimensionally) and Andrews' dot $\forall x \bullet F \wedge G = \forall x(F \wedge G)$.

These problems were investigated in two different ways in [KLR07] and [KMR08] and prototype implementations were developed for both. Building on these experiences, we will develop a presentation system along the following basic design choices.

- The presentation algorithm takes as input a set of notations — the **extensional presentation context** — from which the notations are chosen.

- Notations consist of two parts:

  - The **pattern** part consists of three values:

    * The **for**-value is a URI that gives a document, theory, view, or symbol name. The notation applies to all MMT entities whose name starts with this URI, i.e., notations for theories apply to all symbols of the theory. More specific for-values take precedence.
    * The **role**-value is a reference to a production of the MMT syntax. The notation applies to all MMT expressions with this toplevel production.
    * The **icontext**-value is a key-value list that is used to further restrict the applicability of the notation intensionally. Possible keys are the output format, and the ID of the user requesting the presentation. The presentation algorithm takes such a list of key-value pairs — called the **intensional presentation context** — as an additional input, and only notations with a matching icontext are in scope.

  - The **rendering** part contains the desired output element in the syntax of the respective output format. This part may recursively call the presentation algorithm on the components of the presented object. For example, a notation for a constant declaration can recurse into name, type, and definition. The syntax will be mainly declarative. Only very limited operations will be permitted, e.g., testing whether a definition is present or not. Non-compositional translations will only be added later.

  - The rendering part may alternatively contain a declarative description of the presentation using key-value pairs with the keys "operator symbol", "fixity", "left bracket", "right bracket", "bracket style", "separator", "number of implicit arguments". Omitted key-value pairs are inherited from notations with a more general for-value.

- Every MMT document or theory may provide both two further extensional and two further intensional presentation contexts. One of each is a default context that has lower priority than the one passed as input to the presentation algorithm; and the other one has a higher priority and cannot be overridden by the user.

- In every step, the presentation algorithm first computes the extensional context and selects from it a set of applicable notations. Then the intensional context is used to select a notation among those. If ambiguities remain, a heuristic is used, and the user may choose the notation when viewing the document.

- Input and output precedences are used to permit mixfix notations with dynamic bracket placement.

- Notations are not intelligent. In particular, they are not translated along imports. If references are needed within the rendering part, they are relative to the for-value so that they do not require translation.

- The rendering part may classify certain parts of the output as elidable with an integer elidability level. Output fragments with positive elidability levels may be switched on and off when viewing the document by providing elision thresholds. Elidable parts are grouped, typical groups are brackets, type ascriptions to variables, implicit arguments, and qualified symbol names.

8.4 Management of Change

In Sect. 8.2, we described HTTP GET requests sent to an MMT-aware server. More generally, we can provide a restful interface ([Fie00]) to an MMT-aware database by defining PUT, POST (i.e., update) and DELETE requests. Such requests are the basic changes that users can apply to a library. The study of these changes is closely connected to a non-obvious but crucial property of the inference system for MMT-well-formedness: The rules can be understood as preconditions for special PUT requests.

For example, the rule *Doc* can be understood as a PUT request putting the expression γ at URI g. Adding a theory $\underline{T} := \{\vartheta\}$ to the document g via the rule *Mod* can be seen as a PUT request at URI $g?\underline{T}$. Similarly, adding views, symbols, and assignments can be understood. These PUT requests are special in that they always add at the end of the library, e.g., a constant can only be added at the end of a theory occurring at the end of the document occurring at the end of the library. Such PUT requests are always harmless in that they cannot affect parts of the already validated library. For example, adding a constant to a theory in the middle of a library may invalidate proofs that do not cover the new case.

Another challenge in this context is that, e.g., adding an import from S to a theory T in the middle of a library is ill-formed if S occurs after T; but it may be possible to employ semantically equivalent reorderings that move S in front of T so that the import becomes well-formed. If libraries are considered modulo semantic equivalence (and they should be), then checking the preconditions of such PUT requests is significantly harder than for the special PUT requests acting at the end.

The solution to these problems is to introduce transactions and dependencies. A **transaction** is a sequence of PUT, GET, DELETE, and POST requests. In particular, a library can be seen as a sequence of PUT requests: There is one PUT request for every declaration in the order in which they occur in the library. Then the semantic equivalence of reorderings of declarations can be stated as a commutativity property of PUT requests. The view of libraries as transactions has the advantage that it can be directly generalized to GET, DELETE, and POST requests using techniques known from databases. And the **dependency** is a binary relation between URIs such that "B depends on A" expresses that changes to A may affect the well-formedness of B (see also Sect. 8.2).

Then libraries can be manipulated by applying transactions to them. This provides a very powerful interface language both for humans and for software systems: All communication between systems can be expressed in terms of transactions. This provides the foundation of effective management of change because the amount of data to be communicated and processed can be restricted to transactions and thus to changes. The central open research questions in this situation are the following:

- When are two transactions equivalent in the sense that have the same effect on any library (up to semantic equivalence)? For example, reordering GET requests does not change the effect of a transaction, and neither does dropping duplicate DELETE requests.

- When is the application of a transaction to a well-formed library well-formed? Since the library is typically big and the transaction small, the naive solution of validating the resulting library is inefficient. For example, for PUT or DELETE requests at the end of the library, this can be answered much more easily.
- If the answer to the previous question is "no", which parts of the library or of the transaction would have to be changed so that it is "yes"? The effect of every non-GET request must potentially be propagated along the dependency relation.

8.5 Conclusion

The goals of Part III were threefold. Firstly, in Sect. 1.3.1, we demanded a scalable infrastructure for the logical framework based on $\mathbb{L}$ we developed in Part II. MMT is the core of this infrastructure. As seen in Sect. 7.3, we can represent all levels of logical reasoning in MMT, both in terms of proof theory and of model theory. MMT is neutral with respect to the set/type theory distinction making it applicable to problems from both domains.

Of course, this apparent greatness has a downside: The coupling between MMT and $\mathbb{L}$ is rather loose. MMT libraries must be type-checked by implementations in which the knowledge about the foundation is hard-coded as we saw in Sect. 7.1 and 7.2. And as we remarked at the end of Sect. 7.3, not all features of a logic encoding in $\mathbb{L}$ can be specified in MMT.

However, this is not necessarily a disadvantage: While MMT may seem as a complex language looking at the inference system in Sect. 6.3, the complexity stems from the consequent use of fully formal definitions. The actual input syntax given in Sect. 6.2 is remarkably simple as witnessed by the straightforward XML grammar we gave in Sect. 8.1. This simplicity is crucial for scalability: Scalability means that algorithms that handle huge input sizes cannot be only be designed but also implemented and maintained efficiently. And here MMT is extremely strong: The simplicity of the language and the strict separation between the underlying foundation and the MMT concepts on top of it yield a variety of foundation-independent results. These results include the maintenance of libraries and the lookup functions (Sect. 6.2.2, Sect. 8.2), the concept of semantic equivalence (Sect. 6.4), and the yet-to-be-developed management of change (Sect. 8.4). These foundation-independent results significantly lower the bar for implementations, both in terms of person years and in terms of qualification levels of the implementers.

Secondly, in Sect. 1.3.2.1, we demanded a knowledge management infrastructure that employs logical methods. While MMT is only a first step in this direction, the relatively simple integration of informal elements (Sect. 6.5.7, 6.5.8) yields a promising way to build the manipulation of informal documents both in mathematics and in software engineering on a formally rigorous base. Here the combination of the fully formal structuring concepts of MMT and the foundation independence, which scales to informal foundations, is an important factor: It provides the leeway to degrade gracefully; and in particular MMT can combine machine-understandability with informal reasoning better than OMDoc 1.2.

And thirdly, in Sect. 1.3.2.2, we demanded a knowledge management infrastructure for logical knowledge. Here MMT as a meta-meta-logical framework is at its best. In Sect. 8, we sketched the design of a variety of services that reach or go beyond the state of the art of comparable services as they are implemented for any single logical system. Using MMT, these services are not only possible, but in some respects generic implementations based on MMT are in fact easier: This is because MMT separates the highly difficult and foundation-dependent logical sphere and the relatively simpler foundation-independent knowledge management sphere.

Furthermore, while MMT is not a universal logic such as $\mathbb{L}$, it can be used as an interface language using translations as outlined in Sect. 6.3.4.2. The combination of formal semantics and foundation-independence enable MMT to represent a wide variety of languages. And

this is possible in a way that preserves modular structure: Most of the concepts described in Sect. 1.1.3.2 can be expressed naturally. On the other hand — and that is non-trivial — MMT is relatively easy to interpret: Translations out of MMT are much easier than translations out of any other of the described module systems. Thus, MMT can serve as a standardized interface language for modular systems: expressive enough to make translations into it easy, and simple enough to make translations out or it easy. In particular, two students educated in systems A and B, respectively, can effectively implement translations from A to MMT and from MMT to B, whereas they might be unable to implement a direct translation together.

Finally, however complex some aspects of MMT may be, understanding MMT is arguably easier than understanding the intricacies of logical systems, let alone their implementations. That creates a scarcity of resources for research groups maintaining logical systems that, together with the work-intensive maintenance of compatibility between releases, creates a high threshold for the addition of knowledge management services. Here a strong pragmatic advantage of MMT is that MMT services can be efficiently implemented by students. For example, in the imminent future, we will implement the software framework for the services presented in Sect. 8, which in particular includes the interfaces between them. And after an initial high workload, incremental improvements can be delegated to student projects.

Acknowledgments While the designs outlined in Sect. 8 represent the author's own understandings and conceptualizations, the general research area intersects strongly with that of the KWARC group at Jacobs University Bremen. Therefore, the presented material is connected to joint work and discussions with the other members of the group and their independent research results in a way that can hardly be resolved. In particular, Michael Kohlhase had many of the original ideas or participated in shaping them. Vyacheslav Zholudev recently started designing an OMDoc-aware database ([Zho08]). Andrei Ioniţă implemented a prototype of an MMT-aware database. Ioan Şucan, Constantin Jucovschi, and Ştefan Anca have implemented a MathML-aware search engine ([KŞ06]). Christoph Lange has worked on an ontology for OMDoc 1.2. We collaborated with Michael Kohlhase, Christoph Lange, Christine Müller, and Normen Müller for work on presentation, which was published as [KLR07] and [KMR08]. Finally, Normen Müller has reimplemented the SVN client in a way that can support MMT-aware management of change ([MK08]).

Bibliography

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY Tool. *Software and System Modeling*, 4:32–54, 2005. 28

[ABC+03] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 2.0 (second edition). Technical report, World Wide Web Consortium, 2003. See http://www.w3.org/TR/MathML2. 24, 29, 30, 165

[ABI+96] P. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996. 17

[AC01] D. Aspinall and A. Compagnoni. Subtyping Dependent Types. *Information and Computation*, 266(1–2):273–309, 2001. 113

[Aga08] E. Agapie. Representing Functors in a Web-Scalable Module System, 2008. Bachelor's thesis, Jacobs University Bremen. 2, 156, 158

[AHMP92] A. Avron, F. Honsell, I. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992. 27, 99

[AHMP98] B. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998. 27, 99, 115

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999. 20, 26, 143, 170

[AHMS02] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002. 21, 27, 150

[All87] S. Allen. A Non-Type-Theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the Second Annual IEEE Symp. on Logic in Computer Science, LICS 1987*, pages 215–221. IEEE Computer Society Press, 1987. 72

[AR08] S. Awodey and F. Rabe. Kripke Semantics for Martin-Löf Type Theory. To be submitted, see http://kwarc.eecs.iu-bremen.de/frabe/Research/LamKrip.pdf, 2008. 2, 98

[ArX94] `arXiv.org` e-print archive, 1994. `http://www.arxiv.org`. 23

[Awo00] S. Awodey. Topological representation of the lambda-calculus. *Mathematical Structures in Computer Science*, 10(1):81–96, 2000. 92

[Awo06] S. Awodey. *Category Theory*. Oxford University Press, 2006. 55

[Bar91] H. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991. 13

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. 15, 17, 20, 27, 30

[BCC+04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See `http://www.openmath.org/standard/om20`. 24, 30, 120, 125, 161, 167

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, pages 252–255. Springer, 1997. 17

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003. 17, 170

[BdP00] G. Bierman and V. de Paiva. On an Intuitionistic Modal Logic. *Studia Logica*, 65:383–416, 2000. 15, 64

[Ber37] P. Bernays, 1937. Seven papers between 1937 and 1954 in the Journal of Symbolic Logic. 13

[Béz05] J. Béziau, editor. *Logica Universalis*. Birkhäuser Verlag, 2005. 15

[BF85] J. Barwise and S. Feferman, editors. *Model-Theoretic Logics*. Springer-Verlag, 1985. 16

[BLFM05] Tim Berners-Lee, Roy. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force, 2005. 126, 167, 169

[BM99] C. Butz and I. Moerdijk. Topological representation of sheaf cohomology of sites. *Compositio Mathematica*, 2(118):217–233, 1999. 92

[Bou64] N. Bourbaki. Univers. In *Séminaire de Géométrie Algébrique du Bois Marie - Théorie des topos et cohomologie étale des schémas*, pages 185–Ũ217. Springer, 1964. 13, 17, 161

[Bou68] N. Bourbaki. *Theory of Sets*. Elements of Mathematics. Springer, 1968. 17, 148

[Bou74] N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974. 17, 148

[BPTF07] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. The LEO-II Project. In *Automated Reasoning Workshop*, 2007. 17, 23, 157

[Bro07] L. Brouwer. *Over de grondslagen der wiskunde*. PhD thesis, Universiteit van Amsterdam, 1907. English title: On the Foundations of Mathematics. 13, 15

[Bro06] C. Brown. Combining Type Theory and Untyped Set Theory. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning*, pages 205–219. Springer, 2006. 23, 157

[CAB+86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. 15, 20, 23, 113

[Can83] G. Cantor. Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen. *Mathematische Annalen*, 1883. 12

[Car86] J. Cartmell. Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32:209–243, 1986. 71

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996. 21, 26

[CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958. 15, 40, 64, 122

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988. 13, 15, 22, 27

[CH00] R. Constable and J. Hickey. Nuprl's Class Theory and Its Applications. In F. Bauer and R. Steinbruggen, editors, *Foundations of Secure Computation*, pages 91–115. IOS Press, 2000. 22

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940. 13, 15, 21, 23, 27, 72

[CM97] M. Cerioli and J. Meseguer. May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science*, 173:311–347, 1997. 22, 53, 67

[CM01] J. Clark and M. Makoto. RELAX NG Specification, 2001. By the Organization for the Advancement of Structured Information Standards (OASIS). `http://relaxng.org/`. 165

[Coe05] C. Sacerdoti Coen. Explanation in Natural Language of $\bar{\lambda}\mu\tilde{\mu}$-Terms. In M. Kohlhase, editor, *Mathematical Knowledge Management*, pages 234–249. Springer, 2005. 157

[CoF04] CoFI (The Common Framework Initiative). *Casl Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004. 20, 26, 122

[Com98] ISO/IEC Standard 15408. *Common Criteria for Information Technology Security Evaluation*, 1998. See `http://www.commoncriteriaportal.org/`. 28

[Coq08] Coq library, 2008. `http://coq.inria.fr/library-eng.html`. 27

[CP02] I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002. 28

[CS03] K. Claessen and N. Sorensson. New techniques that improve MACE-style finite model finding. In *CADE-19 Workshop on Model Computation - Principles, Algorithms, Applications*, 2003. 17

[Cur89] P. Curien. Alpha-Conversion, Conditions on Variables and Categorical Logic. *Studia Logica*, 48(3):319–360, 1989. 71

[dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970. 15

[Dia02] Razvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402, 2002. 59, 162

[Dia06] R. Diaconescu. Proof systems for institutional logic. *Journal of Logic and Computation*, 16(3):339–357, 2006. 63, 68

[Dia08] R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser, 2008. 26, 55

[Far00] W. Farmer. An Infrastructure for Intertheory Reasoning. In D. McAllester, editor, *Conference on Automated Deduction*, pages 115–131. Springer, 2000. 17

[Fef69] S. Feferman. Set-theoretical foundations of category theory. In S. Mac Lane, editor, *Reports of the Midwest Category Seminar III*, pages 201–247. Springer, 1969. 55

[Fef05] S. Feferman. Predicativity. In S. Shapiro, editor, *The Oxford Handbook of Philosophy of Mathematics and Logic*, pages 590–624. Oxford University Press, 2005. 13

[FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992. 17

[FGT93] W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993. 20

[Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 131, 169, 173

[Fra22] A. Fraenkel. The notion of 'definite' and the independence of the axiom of choice. 1922. 13

[Fre79] G. Frege. *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. 1879. 11

[Fre84] G. Frege. *Die Grundlagen der Arithmetik, Eine logisch-mathematische Untersuchung über den Begriff der Zahl*. 1884. English title: Foundations of Arithmetic, A Logico-Mathematical Enquiry into the Concept of Number. 13

[Fri75] H. Friedman. Equality Between Functionals. In R. Parikh, editor, *Logic Colloquium*, pages 22–37. Springer, 1975. 72

[FS88] J. Fiadeiro and A. Sernadas. Structuring theories on consequence. In *Recent Trends in Data Type Specification*, pages 44–72. Springer, 1988. 26

[GB86] J. Goguen and R. Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Workshop on Category Theory and Computer Programming*, pages 313–333. Springer, 1986. 26

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992. 15, 16, 21, 22, 26, 55, 63, 66

[Gen34] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Z.*, 39, 1934. English title: Investigations into Logical Deduction. 39

[Gir71] J. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971. 13

[Gir87] J. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987. 15, 64

[GJJ96] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996. 20

[GMdP+07] J. Goguen, T. Mossakowski, V. de Paiva, F. Rabe, and L. Schröder. An Institutional View on Categorical Logic. *International Journal of Software and Informatics*, 1(1), 2007. 2, 16, 64, 99

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English title: On Formally Undecidable Propositions Of Principia Mathematica And Related Systems. 14

[Göd40] K. Gödel. The Consistency of Continuum Hypothesis. *Annals of Mathematics Studies*, 3:33–101, 1940. 13

[Gor88] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988. 17, 113

[GR02] J. A. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002. 63

[GR04] J. Goguen and G. Rosu. Composing Hidden Information Modules over Inclusive Institutions. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, pages 96–123. Springer, 2004. 122

[GWM+93] J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993. 20, 26, 155

[Hal03] T. Hales. The flyspeck project, 2003. See `http://code.google.com/p/flyspeck/`. 17

[Hal05a] T. Hales. A proof of the the Kepler conjecture. *Annals of Mathematics*, 162:1065–1185, 2005. 17

[Hal05b] T. Hales. The jordan curve theorem in HOL light, 2005. See `http://www.math.pitt.edu/~thales/`. 17

[Har96] J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996. 17, 23, 158

[HC96] G. Hughes and M. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996. 50

[Hen50] L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15(2):81–91, 1950. 72

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993. 13, 15, 26, 27, 55, 99, 159, 160

[Hil00] D. Hilbert. Mathematische Probleme. *Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, pages 253–297, 1900. 13

[Hil26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–90, 1926. 13

[HJ84] K. Hrbacek and T. Jech. *Introduction to Set Theory*. Marcel Dekker Inc., New York, 1984. 161

[Hof94] M. Hofmann. On the Interpretation of Type Theory in Locally Cartesian Closed Categories. In *CSL*, pages 427–441. Springer, 1994. 71, 81

[Hor98] I. Horrocks. The FaCT System. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX*, pages 307–312. Springer, 1998. 17

[Hor08] L. Horsten. Philosophy of Mathematics. In E. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Stanford, 2008. 13

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980. 15, 40, 64, 122

[HS00] U. Hustadt and R. Schmidt. MSPASS: Modal Reasoning by Translation and First-Order Resolution. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, pages 67–71, 2000. 23

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994. 26, 27, 99

[HW06] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES conference*, pages 160–174. Springer, 2006. 22

[Isa08] Isabelle library, 2008. `http://isabelle.in.tum.de/dist/library/index.html`. 27

[Jac90] B. Jacobs. *Categorical Type Theory*. PhD thesis, Catholic University of the Netherlands, 1990. 71

[Jav04] Javadoc Tool, 2004. Part of the Java 2 SDK, see `http://java.sun.com/j2se/javadoc/`. 31

[JM93] B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, pages 209–29, 1993. 23

[Joh02] P. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford Science Publications, 2002. 77

[KLR07] M. Kohlhase, C. Lange, and F. Rabe. Presenting Mathematical Content with Flexible Elisions. In *Proceedings of the OpenMath/JEM workshop*, 2007. 2, 172, 175

[KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier and J. Campbell and J. Rubio and V. Sorge and M. Suzuki and F. Wiedijk, editor, *Mathematical Knowledge Management*, volume 5144 of *Lecture Notes in Computer Science*, pages 504–519, 2008. 2, 172, 175

[Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006. 25, 29, 119, 165

[Kri63] S. Kripke. Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–Ũ96, 1963. 16, 48

[Kri65] S. Kripke. Semantical Analysis of Intuitionistic Logic I. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1965. 72

[KŞ06] M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006. 170, 175

[KWP99] F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*. Springer, 1999. 22

[Lan69] S. Mac Lane. One universe as a foundation for category theory. In S. Mac Lane, editor, *Reports of the Midwest Category Seminar III*, pages 192–200. Springer, 1969. 55

[Lan98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998. 42, 55, 77

[Law63] F. Lawvere. *Functional Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963. 64, 96

[Law69] W. Lawvere. Adjointness in Foundations. *Dialectica*, 23(3–4):281–296, 1969. 97

[Lew18] C. Lewis. *A Survey of Symbolic Logic*. University of California Press, 1918. 15

[Lip92] James Lipton. Kripke Semantics for Dependent Type Theory and Realizability Interpretations. In J. Myers and M. O'Donnell, editors, *Constructivity in Computer Science, Summer Symposium*, pages 22–32. Springer, 1992. 72

[LM92] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic*. Lecture Notes in Mathematics. Springer, 1992. 77

[LS86] J. Lambek and P. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986. 16, 26, 64

[Łuk20] J. Łukasiewicz. O logice trojwartosciowej. *Ruch Filozoficny*, 5:170–171, 1920. English title: On three-valued logic. 15

[MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program*, 67(1–2):114–145, 2006. 20, 122

[McL06] S. McLaughlin. An Interpretation of Isabelle/HOL in HOL Light. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006. 23

[Mes89] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989. 16, 26, 63, 68, 69

[MGDT05] T. Mossakowski, J. Goguen, R. Diaconescu, and A. Tarlecki. What is a logic? In J. Béziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser Verlag, 2005. 16, 26, 63, 68

[MK08] N. Müller and M. Kohlhase. Fine-Granular Version Control & Redundancy Resolution, 2008. To be submitted. 150, 175

[ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*. North-Holland, 1974. 13, 15, 27, 71

[ML96] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10, 1996. 15, 27, 39

[MM91] J. Mitchell and E. Moggi. Kripke-style Models for Typed Lambda Calculus. *Annals of Pure and Applied Logic*, 51(1–2):99–124, 1991. 72

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007. 20, 22, 27, 113

[Mos99] T. Mossakowski. Specifications in an Arbitrary Institution with Symbols. In D. Bert, C. Choppy, and P. Mosses, editors, *Workshop on Recent Trends in Algebraic Development Techniques*, pages 252–270. Springer, 1999. 26

[Mos05] T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set, 2005. Habilitation thesis, see `http://www.informatik.uni-bremen.de/~till/`. 21, 22, 59, 162

[MRR03] P. Murray-Rust and H. Rzepa. Chemical Markup, XML and the Worldwide Web. Part 4. CML Schema. *Journal of Chemical Information and Computer Sciences*, 43(3):757–772, 2003. 28

[MS89] J. Mitchell and P. Scott. Typed lambda calculus and cartesian closed categories. In *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 301–316. Amer. Math. Society, 1989. 72

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997. 20

[MTP97] T. Mossakowski, A. Tarlecki, and W. Pawlowski. Combining and Representing Logical Systems. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, pages 177–196. Springer, 1997. 23, 26, 113

[Nip02] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *TYPES conference*, pages 259–278. Springer, 2002. 17, 27

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. 17, 23, 158, 161

[NSM01] P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001. 23, 27, 113

[Odl95] A. Odlyzko. Tragic loss or good riddance? The impending demise of traditional scholarly journals. *International Journal of Human-Computer Studies*, 42:71–122, 1995. 23

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992. 15, 17, 20

[OS97] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, SRI International, 1997. 20

[OS06] S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006. 23, 158

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. 15, 20, 27, 31

[Pea89] G. Peano. The principles of arithmetic, presented by a new method. 1889. 11

[Pei85] C. Peirce. On the Algebra of Logic: A Contribution to the Philosophy of Notation. *American Journal of Mathematics*, 7:180–202, 1885. 11

[Pet07] M. Petria. An Institutional Version of Gödel's Completeness Theorem. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO*, pages 409–424. Springer, 2007. 114

[Pfe00] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000. 27, 99, 106

[Pfe01] F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001. 15, 27, 55, 73, 75, 99, 159

[Pit00] A. Pitts. Categorical Logic. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000. 71, 82

[Pri02] G. Priest. Paraconsistent logic. In D. Gabbay, editor, *Handbook of Philosophical Logic*. Kluwer, 2002. 15

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999. 15, 27, 106, 114

[PS08] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008. To appear. 28, 113, 115

[PSK+03] F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. http://www.logosphere.org/. 23, 113

[PSS02] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002. 17

[Rab06] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2006. 2, 99, 109

[Rab07] F. Rabe. OMDoc Theory Graphs Revisited. In *Proceedings of the* OpenMath/*JEM workshop*, 2007. 2, 163

[RDF04] RDF Core Working Group of the W3C. Resource Description Framework Specification, 2004. http://www.w3.org/RDF/. 170

[Rey74] J. Reynolds. Towards a Theory of Type Structure. In *Paris Colloq. on Programming*, pages 408–425. Springer, 1974. 13

[RK08] F. Rabe and M. Kohlhase. A Web-Scalable Module System for Mathematical Theories. To be submitted, see http://kwarc.info/frabe/Research/mmt.pdf, 2008. 2, 158, 163

[Rob50] A. Robinson. On the application of symbolic logic to algebra. In *Proceedings of the International Congress of Mathematicians*, pages 686–694. American Mathematical Society, 1950. 16

[Rus01] B. Russll. Recent work in the philosophy of mathematics. *International Monthly*, 1901. 12

[Rus08] B. Russell. Mathematical Logic as Based on the Theory of Types. *American Journal of Mathematics*, 30:222–262, 1908. 13

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002. 17

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, pages 370–375. Springer, 2001. 17

[See84] R. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Cambridge Philos. Soc.*, 95:33–48, 1984. 15, 71, 73, 92, 97

[Sim95] A. Simpson. Categorical completeness results for the simply-typed lambda-calculus. In M. Dezani-Ciancaglini and G. Plotkin, editor, *Typed Lambda Calculi and Applications*, pages 414–427, 1995. 72

[Soj08] K. Sojakova. Translating Dependently-Typed Logic to First-Order Logic, 2008. Bachelor's thesis, Jacobs University Bremen. 2, 162

[Sol95] Ron Solomon. On finite simple groups and their classification. *Notices of the AMS*, pages 231–239, February 1995. 24

[SS04] C. Schürmann and M. Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004. 23, 99, 113

[SST92] D. Sannella, S. Sokolowski, and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited. *Acta Informatica*, 29(8):689–736, 1992. 26

[ST88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988. 20, 26, 122

[Str91] T. Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991. 71

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theoryational*, pages 413–427. Springer, 1983. 20, 158

[Tar33] A. Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. *Prace Towarzystwa Naukowego Warszawskiego Wydzial III Nauk Matematyczno-Fizycznych*, 34, 1933. English title: The concept of truth in the languages of the deductive sciences. 16

[Tar38] A. Tarski. Über Unerreichbare Kardinalzahlen. *Fundamenta Mathematicae*, 30:176–183, 1938. 13, 17, 161

[Tar96] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag, 1996. 23, 26, 59, 63, 67, 113

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985. 17, 23, 30, 31, 161

[TV56] A. Tarski and R. Vaught. Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102, 1956. 16

[Urb03] J. Urban. Translating Mizar for First Order Theorem Provers. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Mathematical Knowledge Management*, pages 203–215. Springer, 2003. 23, 30

[vH67] J. van Heijenoort. *From Frege To Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard Univ. Press, 1967. 12

[Vir96] R. Virga. Higher-order superposition for dependent types. In H. Ganzinger, editor, *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 123–137. Springer, 1996. 102

[vN25] J. von Neumann. Eine Axiomatisierung der Mengenlehre. *Journal für die reine und angewandte Mathematik*, 154:219–240, 1925. 13

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Conference on Automated Deduction*, pages 275–279. Springer, 2002. 17

[WCPW02] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. 28, 115

[Wey18] H. Weyl. *Das Kontinuum, Kritische Untersuchungen über die Grundlagen der Analysis*. 1918. English title: The Continuum: A Critical Examination of the Foundation of Analysis. 13

[Wie03] F. Wiedijk. Comparing mathematical provers. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Proceedings of Mathematical Knowledge Management*, pages 188–202. Springer, 2003. 17

[WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1913. 13

[Zad65] L. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965. 15

[Zal08] E. Zalta. The stanford encyclopedia of philosophy, 2008. `http://plato.stanford.edu/`. 12

[ZBM31] Zentralblatt MATH, 1931. `http://www.zentralblatt-math.org`. 23

[Zer08] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I. 13

[Zho08] V. Zholudev. Towards Distributed Mathematical Knowledge Management, 2008. PhD research proposal, Jacobs University Bremen. 175