Mathematical Knowledge Management Across Formal Libraries

Mathematisches Wissensmanagement Über Formale Bibliotheksgrenzen Hinaus

Der Technischen Fakultät der Friedrich-Alexander-Universität Erlangen-Nürnberg zur Erlangung des Doktorgrades

Doktor-Ingenieur

vorgelegt von

Dennis Müller

aus

Garmisch-Partenkirchen

Als Dissertation genehmigt von der Technischen Fakultät der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	_
Vorsitzender der Promotionsorgans:	Prof. Dr.-Ing. Reinhard Lerch
Gutachter:	Prof. Dr. Michael Kohlhase
	assoz. Prof. Dr. Claudio Sacerdoti Coen

Abstract

This dissertation is concerned with the problem of integrating formal libraries: There is a plurality of theorem prover systems and related software with different strengths and weaknesses, that are fundamentally incompatible. This is due to different input languages, IDEs, library management facilities etc., but also due to fundamental theoretical aspects; primarily the reliance on different logical foundations, such as set theories, type theories, and additional primitive features (e.g. subtyping, specific module systems,...).

This thesis describes the approach towards solving this problem pursued at our research group. Specifically, it is divided into three main parts:

1. I demonstrate how we can develop modular logical frameworks within the Mmt system – a foundation-agnostic framework for formal knowledge management based on the OMDoc/Mmt language. This allows for specifying almost arbitrarily complicated logics, type theories and related systems. An important aspect of such a framework is the ability to include potentially complex and computationally expensive features (such as subtyping mechanisms or record types) on demand only.

2. We can use this framework to formalize the logical foundation – or, if nonexistent, at least the foundational ontology – of a formal system, which we can use in turn to translate the associated *libraries* to OMDoc/Mmt. I demonstrate this exemplary on the theorem prover PVS, the computer algebra systems Sage and GAP, and the LMFDB database.

3. This enables accessing the core concepts and libraries of various systems from within the Mmt system, and hence implement knowledge management services generically, acting on multiple libraries at once. I present a few such services:

   (a) Curating *alignments* (roughly: two symbols S, T are called *aligned* if they represent the same platonic mathematical concept,

   (b) a generic method for *translating* formal expressions between systems and libraries; using known alignments, theory morphisms and (if necessary) programmatic tactics,

   (c) an algorithm for automatically finding morphisms between theories (modulo alignments, if between theories from different libraries) and

   (d) *Theory Intersections* as a method to refactor given theories using theory morphisms.

Zusammenfassung

Diese Doktorarbeit widmet sich dem Integrationsproblem formaler Bibliotheken: Es gibt eine Vielzahl an Theorembeweisern und verwandter formaler Software mit unterschiedlichen Stärken und Schwächen, die grundlegend inkompatibel sind. Dies liegt sowohl an unterschiedlichen Eingabesprachen, IDEs, Bibliotheksmanagement etc., als auch grundlegenden theoretischen Aspekten; primär die Verwendung unterschiedlicher logischer Fundamente, wie Mengentheorien, Typentheorien, und zusätzlichen primitiven Features (z.B. Subtyping, spezifische Modulsysteme,...).

Die Arbeit beschreibt den von unserer Forschungsgruppe verfolgten Ansatz, dieses Problem anzugehen. Konkret ist sie in drei zentrale Abschnitte geteilt:

1. Ich zeige, wie wir innerhalb des MMT Systems – einem fundament-agnostischen Framework für formales Wissensmanagement mit zugrundeliegender Sprache OMDoc/MMT – logische Frameworks modular entwickeln können, was es uns erlaubt nahezu beliebig komplexe Logiken, Typentheorien und ähnliche fundamentale Systeme darin zu spezifizieren. Wichtig ist hierbei, dass die potentiell komplexen Features die wir dafür auf Ebene des logischen Frameworks brauchen (wie Subtyping-Mechanismen oder record types) auch nur dort aktiv sind, wo wir sie explizit brauchen.

2. Dieses Framework können wir nun nutzen um die logischen Fundamente – oder, falls nicht vorhanden, zumindest die zugrundeliegende Ontologie – eines formalen Systems zu formalisieren, und auf Basis dessen die zugehörigen Bibliotheken nach OMDoc/MMT zu übersetzen, was ich am Beispiel des Beweissystems PVS, den Computeralgebrasystemen Sage und GAP, und der LMFDB Datenbank demonstriere.

3. Dies innerlaubt es uns, *innerhalb des* MMT *Systems* auf die zugrundeliegenden Konzepte verschiedener Systeme und deren Bibliotheken zuzugreifen, und entsprechende Services zu implementieren die auf mehreren verschiedenen Bibliotheken agieren. Entsprechend stelle ich einige solcher Features vor:

   (a) Das kuratieren von *Alignments* (Grob: zwei symbole S, T heißen *aligned*, wenn sie das selbe abstrakte/platonische mathematische Konzept repräsentieren),

   (b) eine generische Methode, formale Ausdrücke zwischen Systemen und Bibliotheken zu *übersetzen*; unter Verwendung bekannter Alignments, Theoriemorphismen und (notfalls) programmatischen Methoden,

   (c) einen Algorithmus, der Morphismen zwischen Theorien findet (modulo Alignments, falls zwischen unterschiedlichen Bibliotheken) und

   (d) *Theory Intersections* als eine Methode, gefundene Theoriemorphismen auszunutzen um gegebene Theorien zu refaktorisieren.

I am deeply indebted to the following people, who all contributed in some way to my success in – and/or well-being during – the writing of this thesis:

First and foremost my parents, for their unconditional support in all my endeavours, no matter how seemingly misguided.

My wonderful girlfriend Tina, for her unrestrained love and affection, and for happily playing *Tropico* next to me when I need to get work done while being with her.

My band mates in *Vroudenspil*, for providing me with a desperately needed outlet outside of my profession, besides being genuinely lovely friends to have. # *ShamelessPlug*

Many researchers I had the pleasure to meet, and who $\langle$ *any* $x \in \mathcal{P}(\{$*inspired, taught, advised, humoured, supported, entertained, just generally have been extremely nice to*$\}) \backslash \{\varnothing\} \rangle$ me in various ways; including but not limited to: Jacques Carette, Merlin Carl, Claudio Sacerdoti Coen, Paul-Olivier Dehaye, Bill Farmer, Michael Junk, Cezary Kaliszyk, Peter Koepke, Andrea Kohlhase, Thomas Koprucki, Markus Pfeiffer, Natarajan Shankar, Nicolas Thiéry, Makarius Wenzel, ...

My colleagues at KWARC, for contributing to establishing a productive, and yet unusually comfortable and fun work environment: Katja Berčič, Jonas Betzendahl, Deyan Ginev, Mihnea Iancu, Constantin Jucovschi, Theresa Pollinger, Max Rapp, Frederik Schaefer, Tom Wiesing. In particular Florian Rabe, who acted as a second advisor, especially in technical matters, and helped me literally every step of the way.

And last but not least Michael Kohlhase, for being an exceptionally passionate, caring and inspiring advisor, and for avoiding without exception all negative tropes about PhD advisors that make up the central plots of countless horror stories that traumatized post grads tell each other huddled around bonfires made from nth revisions of (ultimately failed) thesis drafts.

“You don’t have to have a dream. Americans on talent shows always talk about their dreams. Fine, if you have something you’ve always wanted to do, dreamed of, like, in your heart, go for it. After all, it’s something to do with your time, chasing a dream. And if it’s a big enough one, it’ll take you most of your life to achieve, so by the time you get to it and are staring into the abyss of the meaninglessness of your achievement you’ll be almost dead, so it won’t matter.
I never really had one of these dreams, and so I advocate passionate dedication to the pursuit of short-term goals. Be micro-ambitious. Put your head down and work with pride on whatever is in front of you. You never know where you might end up. Just be aware, the next worthy pursuit will probably appear in your periphery, which is why you should be careful of long-term dreams. If you focus too far in front of you, you won’t see the shiny thing out the corner of your eye.”

“Don’t seek happiness. Happiness is like an orgasm: If you think about it too much it goes away. Keep busy and aim to make someone else happy, and you might find you get some as a side effect.”

“Be hard on your opinions. A famous bon mot asserts that opinions are like arseholes in that everyone has one. There is great wisdom in this, but I would add that opinions differ significantly from arseholes in that yours should be constantly and thoroughly examined.”

“Remember it’s all luck. You are lucky to be here. You are incalculably lucky to be born, and incredibly lucky to be brought up by a nice family that helped you get educated and encouraged you to go to uni.
Or, if you were born into a horrible family, that’s unlucky and you have my sympathy, but you are still lucky. Lucky that you happen to be made of the sort of DNA that went on to make the sort of brain which, when placed in a horrible childhood environment, would make decisions that meant you ended up eventually graduating uni. Well done, you, for dragging yourself up by your shoelaces. But you were lucky. You didn’t create the bit of you that dragged you up. They’re not even your shoelaces.
I suppose I worked hard to achieve whatever dubious achievements I’ve achieved, but I didn’t make the bit of me that works hard, any more than I made the bit of me that ate too many burgers instead of attending lectures when I was [at uni].
Understanding that you can’t *truly* take credit for your successes, nor *truly* blame others for their failures, will humble you and make you more compassionate.
Empathy is intuitive, but it is also something you can work on intellectually.”

- Tim Minchin, Graduation Speech, University of Western Australia, 2013[1]

[1]http://www.news.uwa.edu.au/201309176069/alumni/tim-minchin-stars-uwa-graduation-ceremony

Contents

Disclaimer:

PhD students are in many universities – among them Jacobs University Bremen, where I started my PhD – given a choice between submitting a *cumulative* dissertation or a *monography*. The former simply consists of previous publications by the student and a summary, whereas the latter is a self-contained document.

I deliberately decided against the first option, because even though that would have been a lot less work, I thought it a worthy endeavour to present my research as a coherent, unifying document that bundles the results, both of existing publications and previously unpublished research, in the context of an overarching narrative suggestive of a bigger picture, research goals beyond the individual publications and an implied direction for future work.

Consequently however, a non-negligible fraction of the writing and many of the results in this thesis have been adapted, and sometimes taken verbatim, from previously published papers I coauthored. The reason being that in writing a paper, one is already forced to think deeply about how to present the subject manner in a coherent and precise manner. Additionally, these papers have been written in close collaboration with much more experienced authors than me, so without a contextual reason to adapt our collaborative writing to the context of this thesis, any attempt of mine to improve on them would be futile and guaranteed to result in an unnecessarily less well-suited presentation; primarily to the detriment of the reader.

This applies mainly to selected

- introductory paragraphs,
- state-of-the-art descriptions,
- descriptions of systems not developed at KWARC,
- detailled formal descriptions of algorithms or rule systems that were previously published, and
- work done by collaborators outside of our work group.

All chapters describing previously published work are prefaced with a disclaimer such as this one, describing (if possible) which parts have been adapted from which publications with which coauthors, and explicitly describing my personal contributions for evaluation.

All of those publications are listed in a separate section of the bibliography (Section 18).

Part I

Introduction

Chapter 1

Motivation

In the last decades, the formalization of mathematical knowledge (and the verification and automation of formal proofs) has become of ever increasing interest. Formal methods nowadays are not just used by computer scientists to verify software and hardware as well as in program synthesis, but – due to problems such as Kepler's conjecture [Hal+15a], the classification theorem for finite simple groups [Sol95], etc. – are also becoming increasingly popular among mathematicians in general.

By now, there is a vast plurality of formal systems and corresponding libraries to choose from. However, almost all of these are non-interoperable, because they are based on differing, mutually incompatible logical foundations (e.g. set theories, higher-order logic, variants of type theory etc.), library formats, library structures, and much work is spent developing the respective basic libraries in each system.

Moreover, since a library in one such system is not reusable in another system, developers are forced to spend an enormous amount of time and energy developing basic library organization features such as distribution, browsing/search or change management for each library format; all of which binds resources that could be used to improve core functionality and library contents instead.

One reason for the incompatibility is the widespread usage of the *homogeneous method*[1], which fixes some logical foundation with all primitive notions (e.g. types, axioms, inference rules) and uses conservative extensions of this foundation to allow for modeling some specific domain knowledge. While homogeneous reasoning is conveniently implementable and verifiable, it implies that a lot of work is needed just to model the basic domains of discourse necessary for mathematics, such as real numbers, algebraic theories, etc. Moreover, the resulting formalizations are actually less valuable to mathematicians, since they are intimately dependent on (and framed in terms of) the underlying logic, making it virtually impossible to move the results between different foundations or to abstract from them.

In contrast, mathematical practice favors the *heterogeneous method*, as exemplified by the works of Bourbaki [Bou64]. In heterogeneous reasoning, theories are used to introduce new primitive notions and the truth of some proposition is relative to a theory (as opposed to absolute with respect to some foundation). This method is closely related to the "little theories" paradigm [FGT92a] which prefers to state each theorem in the weakest possible theory, thus optimizing reusability of mathematical results. Even though in theory, all of mathematics can be reduced to first principles – most prominently first-order logic with (some extension of)

[1]The terminology *homogeneous/heterogeneous* is due to Florian Rabe, unpublished.

ZFC – it is usually carried out in a highly abstract setting that hides the foundation, often rendering it irrelevant and thus allowing it to be ignored.

Correspondingly, there is an inconvenient discrepancy between the currently existing formal libraries and the way (informal) mathematics is usually done in practice. Furthermore, the available formal knowledge is distributed among dozens of different mutually incompatible formal library systems with considerable overlap between them.

Additionally, [Car+19] recently introduced the *five aspects of big math systems* – aspects of doing mathematics that can and should be supported by software, collectively referred to as *tetrapod*, as in Figure 1.1. These aspects are

1. *inference* as supported by interactive and automated proof systems,
2. *computation* as supported e.g. by computer algebra systems,
3. *tabulation* as e.g. supported by databases of mathematical objects (finite groups, graphs, etc.) and their properties,
4. *narration* as supported e.g. by LaTeX and other text formating software with strong support for mathematical expressions, and finally
5. *organization* of formal knowledge.

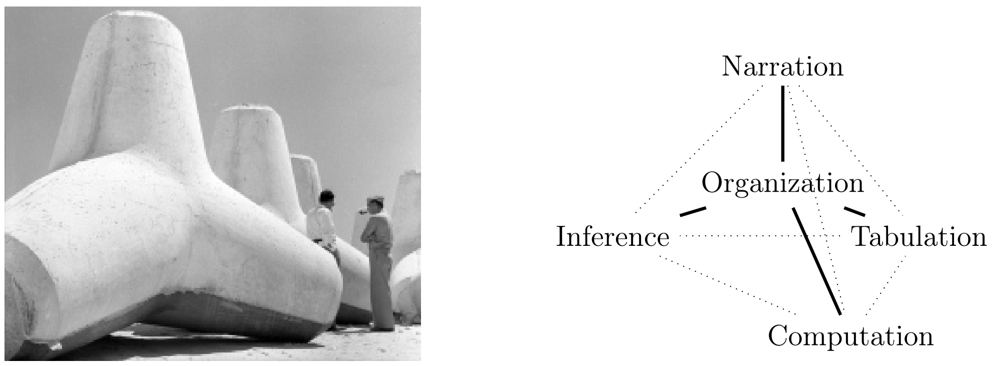


Figure 1.1: Tetrapods: Real and Conceptual

Most software systems focus on one of those tetrapod aspects, few on two, and none strongly support all of them. That is despite the fact that all five of these aspects are intrinsic and important parts of doing real-world mathematics, and it is correspondingly vital to support all of them in an interconnected workflow.

What we should consequently aim for is a *universal archiving solution for all formal knowledge*, that

1. does not depend on a specific logical foundation,
2. can import libraries from different formal systems, allowing one to choose which system to use for generating new content,
3. allows for abstracting from and transferring results between differing foundations and library structures,

4. supports all five aspects of big math systems.

The aim of this thesis is to describe our approach to achieving such an archiving solution by integrating the libraries of existing software systems, focusing on the *inference*, *computation*, *tabulation* and *organization* aspects of the tetrapod, and to demonstrate its feasibility.

Chapter 2

State of the Art

The attempt to systematically formalize both mathematical knowledge and its semantics goes back at least to the seminal work by Bertrand Russell and Albert North Whitehead [WR13]. In the 1950s and 1960s, computer systems were added to the tool chest for this endeavour, shifting the focus to designing foundations that combine both machine-friendliness and human readability. This enabled *automated theorem proving*, thanks to ideas going back to Allen Newell, Herbert A. Simon, and Martin Davis. This has been most succesful for first-order logic and related systems. For more expressive languages, the *verification* of human-written formal proofs has been the more successful approach, going back to John McCarthy, Nicolaas Govert de Bruijn, Robin Milner, and Per Martin-Löf. Modern proof assistants usually combine both approaches, generally verifying user input interactively and employing automation for routine proof steps whenever possible.

Since developing sophisticated proof systems requires both a lot of practical work and a high level of theoretical knowledge, most popular proof assistants have invested a large amount of time and energy into building their systems. Consequently, formalization within one such system pays off mostly at large scales, calling for a community effort to extend the corresponding formal libraries. In practice, however, the availability of many different and mutually incompatible systems – each with their own advantages and disadvantages – has instead driven towards ever increasing specialization within the formal mathematics community, resulting in lots of different libraries with considerable (and from a heterogeneous point of view unnecessary) overlap of actual content.

A more extensive summary of the state of the art and the scientific context of this work can be found in [KR16b].

2.1 Foundations

Overview The notion of a *foundation* goes back to the *foundational crisis of mathematics* at the beginning of the 20th century. Resulting from a general confusion regarding the ontological status of informally defined objects like infinitesimals in real analysis, non-euclidean geometries and sets as objects of study in their own right, as well as debates over the validity of certain non-constructive proof techniques of ever growing abstraction and intricacy, the idea developed to find a formal, logical basis that fixes both an ontology as well as an unambiguous notion of what constitutes a valid proof.

By now, it is generally accepted within the mathematical community, that the answer to this problem is the combination of first-order logic and some system of set theory, usually considered as (possibly extensions of) Zermelo-Fraenkel set theory with the axiom of choice (ZFC). However, already going back to *Principia Mathematica* by Russell and Whitehead (which can be seen as an early variant of type theory), alternative foundations have been around.

There's an inherent trade-off in every such foundation with respect to their complexity and expressiveness. Theoretically, a foundation should be simple with very few primitive notions, to make reasoning *about* the foundation more convenient and establish trust in its consistency. On the other hand, since a foundation should ultimately establish a framework for all of mathematics, it should be as expressive as possible to allow mathematicians to talk about all of the desired objects and realms in terms of the foundation. This trade-off naturally lead to a large diversity of different foundations, which nowadays are used in different proof assistants.

All of these systems fix one specific foundation as a basis for their specification language, usually variants of either constructive type theory, higher-order logic or (as implicitely used in mathematics) first-order set theory. The constructive type theories are mostly based on Martin-Löf type theory [ML74] or the calculus of constructions [CH88] and make use of the Curry-Howard correspondence [CF58; How80] to treat propositions as types (and proofs as λ-terms). Systems include Nuprl [Con+86], Agda [Nor05], Coq [Tea03], and Matita [Asp+06a]. The second group of systems go back to Church's higher-order logic [Chu40] and include HOL4 [HOL4], ProofPower [Art], Isabelle/HOL [NPW02], and HOL Light [Har96].

Since type theories and higher-order logics are more conveniently machine implementable, systems using set theories are noticably rarer. These are e.g. Mizar [Miz], Isabelle/ZF [PC93], and Metamath [MeMa].

The foundation of the PVS system (see Section 10.1) includes a variant of higher-order logic, but has been specifically designed to have a set theoretic semantics. The IMPS system [FGT93] is based on a variant of higher-order logic with partial functions. The foundation of ACL2 [KMM00] is an untyped language based on Lisp.

Heterogeneous Reasoning As mentioned in the previous section, even though all of mathematics is assumed to be reducible to some foundation, the way mathematics is usually practiced is according to the **heterogeneous method**, in which all foundational aspects are "hidden" and left implicit, unless necessary in the respective context. One major advantage of this method, in which theories are used to introduce new primitive notions, is that (in connection with the *little theories* approach) it allows for reusing mathematical results and moving them along *theory morphisms* (i.e. truth-preserving maps) between theories. This approach has been applied successfully in software engineering and algebraic specification, where formal module systems are used to build large theories out of little ones, e.g., in SML [Mil+97] and ASL [SW83].

To accomodate for this more convenient style of reasoning, most formal systems have specific features that introduce some form of heterogeneity either *explicitely* or *implicitely*. Explicit features include e.g. "locales" in Isabelle, "parametric theories" in PVS, "modules" in Coq, or "structures" in Mizar. The IMPS system [FGT93] is somewhat unique in so far, as it was designed specifically with heterogeneous reasoning in mind.

To use the heterogeneous method implicitly, the user introduces a new formal construct (such as real numbers) by defining them in terms of existing ones (e.g. the usual construction via equivalence classes on Cauchy sequences) and proving the actually relevant (i.e. construction independent) theorems about them (e.g. field axioms, topological completeness). Finally, the user can rely on the proven properties alone, without ever needing to expand the concrete definition originally implemented, which in the process is rendered irrelevant for all practical purposes.

To enable implicit heterogeneity, the system has to provide corresponding definition principles. Examples include type definitions in the HOL systems, provably terminating functions in Coq or Isabelle/HOL, or provably well-defined indirect definitions in Mizar. However, it should be noted, that such implicit heterogeneous definitions are usually internally expanded into conservative extensions of the foundation. Alternatively, several data types provide an additional way to use heterogeneity implicitely. These include (Co)inductive types and record types e.g, as done by Mizar structures and with Coq records in the Mathematical Components project [MC]. This has the additional advantage, that it allows for computation within the foundation.

The Incompatibility Problem The homogeneous method (in combination with implicit heterogeneity) has an obvious advantage: Since the foundation is fixed and can not be extended by new axioms, the trusted code base remains fixed as well. Furthermore, is allows integrating computational methods e.g. to reason about equality, without having to tediously instantiate and apply the corresponding axioms. However, it has the major disadvantage of making reuse of formalized knowledge difficult, if not impossible. Since the techniques for implicit heterogeneity are elaborated on the basis of the foundation, the actual heterogeneous structure of some fragment of implemented knowledge is difficult to identify. Furthermore, since different systems offer different techniques to introduce implicit heterogeneity, the construction method used in one system can often not be easily translated into another system. Their elaborated implementations themselves are again framed in terms of the foundation (or an intrinsic part thereof), and are consequently useless for a system based on a differing foundation.

Even though a fixed foundation is therefore reasonable for an individual formal system, if we envision a *universal library of mathematics* – a major goal of formal mathematics going back at least to the QED project and manifesto [Qed] of 1994 – this is the wrong approach. Furthermore, different areas of mathematics favor different foundations (such as category theory, specifically in algebra), as do different communities dealing with formalizing mathematics, whether due to technical reasons or simply familiarity. Thus, formal libraries would profit from **foundational pluralism**, i.e., the ability to support multiple foundations in a single universal library.

2.2 Formal Libraries

Overview Usually, implemented formal systems have some notion of a *library*, a collection of formalizations (usually just the corresponding source files) that can be handled in specific ways – imported and used when creating new content, collectively exported, etc. Most systems are distributed with some base library providing the most useful and ubiquitous settings of interest, such as Booleans or number spaces, and often maintain additional, community-created libraries with more advanced contents.

The Isabelle and the Mizar groups maintain one centralized library each – the "Archive of Formal Proofs" [AFP] and the "Mizar Mathematical Library" [MizLib], respectively. The Coq group relies on a package manager on top of distributed Git repositories instead of a central archive. These libraries contain individual formalizations with relatively few interdependencies. Other libraries are generated and maintained by communities apart from the developers of the system, however, these are still often valuable in their own right, such as Tom Hales's formalizations in HOL Light for the Kepler conjecture [Hal+15a] and Georges Gonthier's work in Coq for the recently proved Feit-Thompson theorem [Gon+13]. John Harrison's formalizations in his HOL Light system [Har96] and the NASA PVS library [PVS] have a similar flavor although they were not motivated by a single theorem but by a specific application domain. The latter is one of the biggest decentralized libraries, whose maintenance is disconnected from that of the system.

As mentioned, most of these systems are based on the homogeneous method. However, there are some libraries that are intrinsically heterogeneous, such as the IMPS library [FGT], the LATIN logic library [Cod+11] developed at KWARC (see Section 3.1) and the TPTP library [Sut09] of challenge problems for automated theorem provers. Unfortunately, none of these enjoy the level of interpretation, deduction, and computation support developed for individual fixed foundations.

The OpenTheory format [Hur09] offers some support for heterogeneity in order to allow moving theorems between systems for higher-order logic (specifically HOL Light, HOL4, and ProofPower). It provides a generic representation format for proofs within higher-order logic that makes the dependency relation (i.e., the operators and theorems used by a theorem) explicit. The OpenTheory library comprises several theories that have been obtained by manually refactoring exports from HOL systems.

Library Integration There are two problems concerning library integration. The first is, given a single library, **refactoring** its contents to increase modularity. This results in a "more heterogeneous" set of theories, thus making it easier to reuse and blowing up the corresponding theory graph to make shared or inherited theories and results more visible and explicit. An attempt at giving a formal calculus for theory refactoring has recently been published [AH15].

The second, and for this thesis more relevant, problem is to **integrate** two or more given libraries *with each other*, so that knowledge formalized in one of them can be reused in, translated to or identified with contents of the others. In the best case, two libraries might even be **merged** into a single library with ideally no redundant content.

No strong tool support is available for any of these facets. The state-of-the-art for refactoring a single library is manual ad hoc work by experts, maybe supported by simple search tools (often text-based). Also, the widespread use of the homogeneous method makes integrating and merging libraries in different systems extremely difficult, since usually basic concepts in one foundation cannot be directly translated to corresponding concepts in the other [KRSC11].

This is despite the large need for more integrated and easily reusable large libraries. For example, in Tom Hales's Flyspeck project [Hal+15a], his proof of the Kepler conjecture is formalized in HOL Light. But it relies on results achieved using Isabelle's reflection mechanism, which cannot be easily recreated in HOL Light. And this is an integration problem between two tools based on the same root logic.

Library Translations There are two ways to take on library integration. Firstly, one can try to translate the contents of one formal system directly into another system; I will call this a *library bridge*. This requires intimate knowledge of both systems and their respective foundations used, and is necessarily somewhat ad-hoc. A small number of library bridges have been realized, typically in special situations. [KW10] translates from HOL Light [Har96] to Coq [Tea03] and [OS06a] to Isabelle/HOL. Both translations benefit from the well-developed HOL Light export and the simplicity of the HOL Light foundation. [KS10] translates from Isabelle/HOL [NPW02] to Isabelle/ZF [PC93]. Here import and export are aided by the use of a logical framework to represent the logics. The Coq library has been imported into Matita [Asp+06a] once, aided by the fact that both use very similar foundations. The OpenTheory format [Hur09] facilitates sharing between HOL-based systems but has not been used extensively.

The second way is to use a more general *logical framework* (see Section 2.3) which provides some way to specify the respective foundations, and integrate the libraries under consideration directly into that framework. Then the framework can serve as a uniform intermediate data structure, via which other systems can import the integrated libraries. This approach will be extensively described in this thesis, using the logical framework LF [HHP93a] and making the libraries available to knowledge management services. Another example is the Dedukti system [BCH12], which imports, e.g., Coq and HOL Light into a similar logical framework, namely LF extended with rewriting.

Dedukti underlies the recent *Logipedia* project [DT], aiming to build an encyclopedia of formal proofs in simple type theory (originally developed in Matita), which can be exported to other systems.

Again, the prevalence of the homogeneous method constitutes a major problem here. Even with implicit heterogeneity, the fact that e.g. theories such as the real numbers are still modeled as conservative extensions of a fixed foundation using some intricate construction principle (cauchy sequences, Dedekind cuts) means, that using different definitions can make it impossible to align a theory in one library to the corresponding theory in a different library, even though from a mathematical point of view they are "the same" (i.e. isomorphic). And even if the same abstract construction principle is used in two libraries, their implementations in terms of the underlying foundation can be different enough to make it difficult to identify the two resulting theories.

Very little work exists to address this problem. In [OS06a], some support for library integration was present: Defined identifiers could be mapped to arbitrary identifiers ignoring their definition. No semantic analysis was needed because the translated proofs were rechecked by the importing system anyway. This approach was revisited and improved in [KK13a], which systematically aligned the concepts of the basic HOL Light library with their Isabelle/HOL counterparts and proved the equivalence in Isabelle/HOL. The approach was further improved in [GK14a] by using machine learning to identify large sets of further alignments.

The OpenTheory format [Hur09] provides representational primitives that, while not explicitly using theories, effectively permit heterogeneous developments in HOL. The bottleneck here is manually refactoring the existing homogeneous libraries to make use of heterogeneity.

A partial solution aimed at overcoming the integration problem was sketched in [RKS11].

2.3 Logical Frameworks

Over the last 20 years, formal systems have added an additional metal level through the introduction of logical frameworks. These provide tools to specify logical systems themselves in a formal way. An overview of the current state of art is given by [Pfe01].

An example for one such framework is LF [HHP93a] (see Section 4.3). Logical frameworks introduce the possibility to additionally reason *about* logics, as e.g. in Twelf [PS99], which is the currently most mature implementation of LF. Twelf has been used as a basis for the LATIN library (see Section 3.1). Also, since in logical frameworks logics are themselves represented as theories, they allow for defining logical systems heterogeneously, by building them up in a modular way.

Dedukti [BCH12] implements LF modulo rewriting. By supplying rewrite rules (whose confluence Dedukti assumes) in addition to an LF theory, users can give more elegant logic encodings. Moreover, rewriting can be used to integrate computation into the logical framework. A number of logic libraries have been exported to Dedukti, which is envisioned as a universal proof checker. Isabelle [Isa] implements intuitionistic higher-order logic, which (if seen as a pure type system with propositions-as-types) is rather similar to LF. Despite being logic-independent, most of the proof support in Isabelle is optimized for individual logics defined in Isabelle, most importantly Isabelle/HOL and Isabelle/ZF.

λProlog – in its most mature implementation ELPI [Dun+15] – extends the logic programming paradigm with higher-order functionalities. As such, it allows higher-order abstract syntax approaches and can be (and is) used as a logical framework as well.

Unfortunately, logical frameworks are not an efficient alternative to the prevailing homogeneous formal systems. State of the art proof assistents rely heavily on the specific peculiarities of their underlying foundation to provide efficient proof search techniques, which would be impossible to implement at the high level of generality that logical frameworks provide, at least without introducing cosiderable overhead and thus reducing efficiency.

Another problem is that specific concepts used by some logic (noticably record types, subtyping principles, inductive definitions, etc.) may be difficult to realize in a logical framework in such a way, that type checking can be done effectively, since the necessary information for doing so can not always easily be lifted to the rather general level on which the type checker operates (see Part II).

Chapter 3

Context and Contribution

This thesis is part of two ongoing research projects - OAF (Open Archive of Formalizations)[1] and ODK (OpenDreamKit)[2]. The goals of OAF in particular largely coincide with the objectives of this thesis, specifically regarding applications for theorem prover libraries; ODK entails work packages extending the same goals to other formal systems, such as computer algebra systems and mathematical databases. Our approach to realizing these projects largely make use of the Math-in-the-Middle architecture (MitM).

Before we describe the precise objectives and contribution of this work, we will go over the above and related projects, as to establish the necessary context.

3.1 LATIN

The LATIN project [Cod+11] was a DFG funded project running from 2009 to 2012 under the principal investigators Michael Kohlhase, Florian Rabe and Till Mossakowski. Its aim has been to build a heterogeneous, highly integrated library of formalizations of logics and related languages as well as translations between them. It uses Mmt (see Section 4.1) as a framework, with the logical framework LF (see Section 4.3) as a meta-theory for the individual logics.

True to the general Mmt philosophy, all the integrated theories are built up in a modular way and include propositional, first-order, sorted first-order, common, higher-order, modal, description, and linear logics. Type theoretical features, which can be freely combined with logical features, include the λ-cube, product and union types, as well as base types like booleans or natural numbers. In many cases alternative formalizations are given (and related to each other), e.g., Curry- and Church-style typing, or Andrews and Prawitz-style higher-order logic. The logic **morphisms** include the relativization translations from modal, description, and sorted first-order logic to unsorted first-order logic, the negative translation from classical to intuitionistic logic, and the translation from first to sorted first- and higher-order logic.

The left side of Figure 3.1 shows a fragment of the LATIN atlas, focusing on first-order logic (FOL) being built on top of propositional logic (PL), its translation to HOL and ultimately resulting in the foundations of Mizar, Isabelle/HOL and ZFC, as well as translations between them. The formalization of propositional logic includes its syntax as well as its proof and model theory, as shown on the right of Figure 3.1.

[1]Funded by DFG grant KO 2428/13-1

[2]Funded by EU Horizon 2020 grant 676541

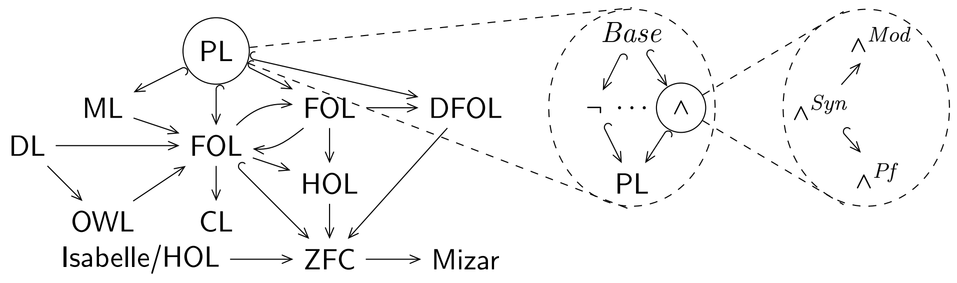


Figure 3.1: A Fragment of the LATIN Atlas

3.2 The OAF Project

The OAF project [OAF] is a DFG funded project running from 2015 to 2019 under the principal investigators Michael Kohlhase and Florian Rabe. The goal was to provide an **O**pen **A**rchive of **F**ormalizations: a universal archiving solution for formal libraries, corresponding library management services (such as distributing, browsing and searching library contents) and methods for integrating libraries in different formalisms in a unifying framework – namely OMDoc/Mmt (see Section 4.1), as in Figure 3.2 – to allow for sharing and translating content across them. We further want it to be scalable with respect to both the size of the knowledge base and the diversity of logical foundations.

Theoretically, the main prerequisite has been established in the LATIN project (see Section 3.1). However, whereas LATIN provides formalizations of basic logics, type theories and related systems, there still remains the problem of integrating the existing formal libraries of theorem prover systems. Consequently, there are two major objectives of the OAF project this thesis touches on:

Making existing libraries accessible to a unifying framework We want to be able to make available theorem prover libraries accessible to the Mmt system.

Libraries that have been imported into Mmt in the course of this project include HOL Light [KR14], Mizar [Ian+13], TPTP [Sut09], IMPS [Bet18], Coq [MRS], Isabelle (as-of-yet unpublished) and (as presented in detail in Chapter 10) PVS.

Refactoring and integrating libraries Once we have several libraries integrated into Mmt, we can implement generic knowledge management services for the integrated libraries, such as:

- **refactoring** the available libraries in such a way, that the heterogeneous part of some theory can be recognized as such and separated from the foundational aspects of its library, and
- **integrating** the libraries **with each other**, such that equivalent theories can be identified and contents can be reused and transferred between libraries.

Naturally, both aspects are heavily interrelated, since integrating libraries is easier after some suitable refactoring, and already aligned libraries can potentially be analyzed more easily and further refactored.

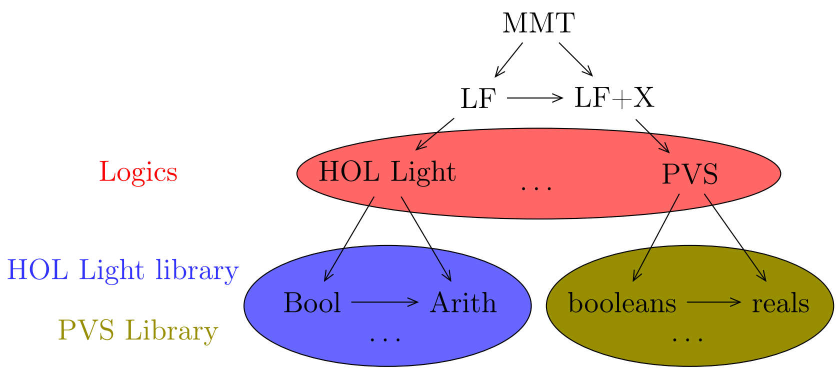


Figure 3.2: Representing Libraries in MMT

3.3 OpenDreamKit

Disclaimer:

The following two sections have been previously published as part of [Deh+16] with coauthors Paul-Olivier Dehaye, Mihnea Iancu, Michael Kohlhase, Alexander Konovalov, Samuel Lelièvre, Markus Pfeiffer, Florian Rabe, Nicolas M. Thiéry and Tom Wiesing.

Neither the theoretical results nor the majority of the writing can be attributed to me personally. These chapters should hence not be considered my contribution, and is mainly included because it is the best description of the OpenDreamKit project and the Math-in-the-Middle approach, which in turn represents a prime concrete application for the results of this thesis.

The Math-in-the-Middle *ontology* for formal mathematics however was largely developed and curated by me, and can thus be considered my contribution.

As with interactive theorem provers specifically, in the last decades we witnessed the emergence of a wide ecosystem of open-source tools to support research in pure mathematics in general. This ranges from specialized to general purpose computational tools such as GAP [Gro16], PARI/GP, LinBox, MPIR, Sage [Dev16], or Singular [SNG], via online databases like the LMFDB [LMF] or online services like Wikipedia, arXiv [Arx], to webpages like MathOverflow. A great opportunity is the rapid emergence of key technologies, in particular the Jupyter [Jup] (previously IPython) platform for interactive and exploratory computing which targets all areas of science.

This has proven the viability and power of collaborative open-source development models, by users and for users, even for delivering general purpose systems targeting large audiences such as researchers, teachers, engineers, amateurs, and others. Yet some critical long term investments, in particular on the technical side, are in order to boost the productivity and lower the entry barrier:

- Streamlining access, distribution, portability on a wide range of platforms, including High Performance Computers or cloud services.
- Improving user interfaces, in particular in the promising area of collaborative workspaces as those provided by CoCalc [CC] (previously called SageMathCloud).
- Lowering barriers between research communities and promoting dissemination. For

example make it easy for a specialist of scientific computing to use tools from pure mathematics, and vice versa.

- Bringing together the developer communities to promote tighter collaboration and symbiosis, accelerate joint development, and share best practices.
- Structure the development to outsource as much of it as possible to larger communities, and focus manpower on core specialities: the implementation of mathematical algorithms and databases.
- And last but not least: Promoting collaborations at all scales to further improve the productivity of researchers in pure mathematics and applications.

OpenDreamKit – "Open Digital Research Environment Toolkit for the Advancement of Mathematics" [ODK] – is a project funded under the European H2020 Infrastructure call [EI] on *Virtual Research Environments*, to work on many of these problems.

In practice, OpenDreamKit's work plan consists of several work packages: component architecture (modularity, packaging, distribution, deployment), user interfaces (Jupyter interactive notebook interfaces, 3D visualization, documentation tools), high performance mathematical computing (especially on multicore/parallel architectures), a study of social aspects of collaborative software development, and a package on data/knowledge/software-bases.

The latter package focuses on the identification and extension of ontologies and standards to facilitate safe and efficient storage, reuse, interoperation and sharing of rich mathematical data, whilst taking provenance and citability into account. This package is the most relevant regarding this thesis.

Its outcome will be a component architecture for semantically sound data archival and sharing, and integrate computational software and databases. The aim is to enable researchers to seamlessly manipulate mathematical objects across computational engines (e.g. switch algorithm implementations from one computer algebra system to another), front end interaction modes (database queries, notebooks, web, etc) and even backends (e.g. distributed vs. local).

3.4 The Math-in-the-Middle Approach and Library

The Math-in-the-Middle architecture is used heavily in the OpenDreamKit project. It centers around the Math-in-the-Middle *ontology*, an Mmt archive of formalized math[3] using all of the features introduced in Part II. In particular, this archive provides useful real-world examples for these features. The ontology serves as a mediator between individual systems involved in the OpenDreamKit project to facilitate knowledge exchange and remote procedure calls.

Additionally, the Math-in-the-Middle architecture can be used in other contexts. In particular, it facilitates across-library knowledge management services for formal libraries, as described in Part IV. From this perspective, while this section focuses on the role of the architecture in the OpenDreamKit project, the general ideas underlying this section can be generalized beyond the systems involved in OpenDreamKit.

Since we aim to make our components interoperable at a mathematical level, we have to establish a common meaning space that will allow us to share computation, visualization of the mathematical concepts, objects, and models between the respective systems. This mediation problem is well understood in information systems [Wie92], and has for instance been applied

[3]Available at `https://gl.mathhub.info/MitM/smglom` and `https://gl.mathhub.info/MitM/Foundation`

to natural language translation via a hub language [KW03]. Here, our hub is mathematics itself, and the vocabulary (or even language) admits further formalization that translates into direct gains in interoperability. For this reason, neither OpenMath [Bus+04] nor MathML [Aus+03] have the practical expressivity needed for our intended applications.

3.4.1 A Common Meaning Space for Interoperability

One problem is that the software systems in OpenDreamKit cover different mathematical concepts, and if there are overlaps, their models for them differ, and the implementing objects have different functionalities. This starts with simple naming issues (*e.g.* elliptic curves are named ec in the LMFDB, and as EllipticCurve in Sage), persists through the underlying data structures and in differing representations in the various tables of the LMFDB), and becomes virulent at the level of algorithms, their parameters, and domains of applicability.

To obtain a common meaning space for a VRE, we have the three well-known approaches in Figure 3.3.

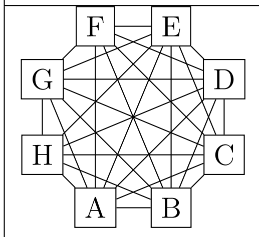
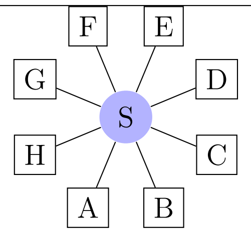
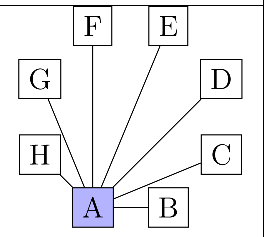
peer to peer	open standard	industry standard
		
$n^2/2$ translations symmetric	$2n$ translations symmetric	$2n-2$ translations asymmetric

Figure 3.3: Approaches for Many-Systems Interoperability

The first does not scale to a project with about a dozen systems, for the third there is no obvious contender in the OpenDreamKit ecosystem. Fortunately, we already have a "standard" for expressing the meaning of mathematical concepts – **mathematical vernacular**: the language of mathematical communication, and in fact all the concepts supported in the OpenDreamKit VRE are documented in mathematical vernacular in journal articles, manuals, etc. The obvious problem is that mathematical vernacular is too *i*) *ambiguous*: we need a human to understand structure, words, and symbols *ii*) *redundant*: every paper introduces slightly different notions.

Therefore we explore an approach where we **flexiformalize** (i.e. partially formalize; see [Koh13a]) mathematical vernacular to obtain a flexiformal ontology of mathematics that can serve as an open communication vocabulary. We call the approach the **Math-in-the-Middle** (MitM) Strategy for integration and the ontology the **MitM ontology**.

The descriptions in the MitM ontology must simultaneously be system-near to make interfacing easy for systems, and serve as an interoperability standard – *i.e.* be general and stable. If we have an ontology system that allows modular/structured ontologies, we can solve this apparent dilemma by introducing **interface theories** [KRSC11], *i.e.* ontology modules (the light purple circles in Figure 3.4) that are at the same time system-specific in their description of mathematical concepts – near the actual representation of the system – and part of the greater MitM ontology (depicted by the cloud in Figure 3.4) as they are connected to the core MitM ontology (the blue circle) by views we call **interface views**. The MitM approach stipulates that interface theories and interface views are maintained and released together with the respective systems, whereas the core MitM ontology represents the mathematical scope of the VRE and is maintained with it. In fact in many ways, the core MitM ontology is the conceptual essence of the mathematical VRE.

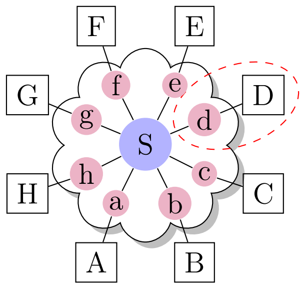


Figure 3.4: Interface Theories

Apart from OpenDreamKit, the MitM ontology is furthermore used in the FrameIT [RKM16] approach for developing serious games for math education, and the MaMoRed project ([Koh+17c],[Kop+18]) with the goal of building a modular library of computational models in physics and related STEM fields.

3.5 Objectives

The OAF and OpenDreamKit projects have a common aim, differing only in their domains of applicability – or aspects of the tetrapod (see Chapter 1): The integration of (libraries of) formal systems (*inference*) in the former, and mathematical software and databases (*computation/tabulation*) in the latter. The goal of this thesis is to outline an approach to solve this integration problem and to demonstrate its feasibility. This entails the following objectives:

O1 **Represent the logical foundations, libraries and ontologies** of theorem prover systems, computer algebra systems and other sources of mathematical knowledge desired for the OpenDreamKit project **in a unifying framework**.

O2 As lamented in Section 2.3, the logical frameworks currently available are not sufficient to formalize the foundations and distinct primitive features of real world theorem prover systems conveniently. Consequently, we need to **develop a modular, sufficiently powerful logical framework** in order to both achieve **O1**, as well as formalize the mathematical knowledge needed in the **Math-in-the-Middle library** as conveniently as possible.

O3 Having the libraries of various systems accessible from within a unifying framework, the goal is to **develop knowledge management service** on a generic level in order to systematically **integrate formal libraries**, facilitating knowledge and structure transfer and **translate expressions** between foundations and systems.

3.6 Contribution and Overview

In **Part II**, I present a modular logical framework (**O2**) that extends LF by various features and is used in the remainder of this work. In particular, I explain in detail how we can use MMT to develop such a framework and extend it by virtually arbitrary additional features via various means. Where issues arise (e.g. with advanced subtyping principles, see Chapter 7), I discuss these and suggest solutions.

Notably, unlike other fixed-foundation frameworks, we can do so without needing to express our principles in terms of a more primitive logic. Instead, we can use the conveniences of an extensive API providing all of the necessary and most common abstractions in a modern, production-ready programming language (Scala) and all the infrastructure that comes with it – such as modern IDEs with static type checking and debugging functionality, extensive library support, etc.

In **Part III** I demonstrate the OAF (and partially ODK) approach to integrating libraries and ontologies into the MMT system (**O1**), using the theorem prover system PVS (Chapter 10), the computer algebra systems GAP and SAGE (Chapter 12), and the database LMFDB (Chapter 11) as representative examples. The formalization of the foundational logic of PVS in Section 10.2 in particular requires and makes use of the logical framework presented in Part II. Since that logical framework is easily extendable – and since PVS can be considered particularly challenging as evidenced by the fact that no earlier representations of the PVS foundation exist in other frameworks – the approach likely scales to virtually arbitrary systems and their foundations.

As a result, we have, for the first time, all symbols from both the underlying foundations and ontologies as well as the associated libraries of formal systems accessible from within the same framework. In particular, we preserve the original presentation and semantics without the need to build dedicated (and sometimes impossible or unsound) translations between foundations.

Finally, **Part IV** covers knowledge management services (**O3**) that can now be implemented *generically* and foundation-independently, and which are consequently available for any (pair of) systems integrated into MMT now or in the future. In particular, I present:

- a specification for *alignments* and an implementation for managing them in Chapter 14, as a basis for across-library services,
- a *viewfinder* for finding theory morphisms, and consequently new alignments, automatically in Chapter 16,
- a method for translating content between systems using alignments and theory morphisms in Chapter 15,
- and finally *theory intersections* as a method for library refactoring in Chapter 17.

The last four points particularly achieve the main goals of the OAF project, and ultimately enable the intended results of (our part of) the OPENDREAMKIT project. The translation method in particular allows for transfering virtually arbitrary formal content between libraries and foundations, thus facilitating e.g. remote procedure calls between systems – in other words, it allows for flexibly and fully integrating formal systems and their libraries.

As such, this thesis demonstrates the feasibility of our approach to library integration in enabling *proper* knowledge management *beyond* individual systems and libraries *within a unifying framework* and system, bringing us one step closer to a full tetrapodal system.

Consequently this work covers a broad spectrum, ranging from purely theoretical developments down to implementation details.

Much of the content of this thesis has been published in previous papers. These are listed separately in the bibliography (Section 18).

Chapter 4

Preliminaries

Throughout this thesis we will use the MMT language and system as a meta-logical framework. MMT is a highly flexible system and API using the foundation-independent OMDOC language for its backend. Within MMT, we can implement almost arbitrary logical frameworks; most prominently, it comes with an implementation of LF which we extend and use throughout this thesis.

Disclaimer:

> The text of the this chapter has been assembled and adapted from various previously published papers by (primarily) Florian Rabe, Michael Kohlhase and myself (all of which are referenced in this thesis), since they have emerged as the most suitable, concise introductions to the concepts described herein. The exposition has been thoroughly reworked for a uniform representation, but some text fragments of the original might remain. Original publications are cited where adequate.

4.1 OMDOC/MMT

OMDOC [Koh13b] is a representation language developed by Michael Kohlhase, that extends OPENMATH and MATHML and allows for providing general definitions of the syntax of both mathematical objects as well as mathematical documents. They make use of *content dictionaries*, which introduce primitive notions and their semantics. In particular, it can represent exports of formal system libraries and their documentation.

In the last ten years, (chiefly) Florian Rabe redeveloped the fragment of OMDOC pertaining to formal knowledge resulting in the **OMDOC/MMT language** [RK13a; HKR12a; Rab17b]. OMDOC/MMT greatly extends the expressivity, clarifies the representational primitives, and formally defines the semantics of this OMDOC fragment. It is designed to be foundation-independent and introduces several concepts to maximize modularity and to abstract from and mediate between different foundations, to reuse concepts, tools, and formalizations.

More concretely, the OMDOC/MMT language *integrates successful representational paradigms*

- the logics-as-theories representation from logical frameworks,
- theories and the reuse along theory morphisms from the heterogeneous method,
- the Curry-Howard correspondence from type theoretical foundations,

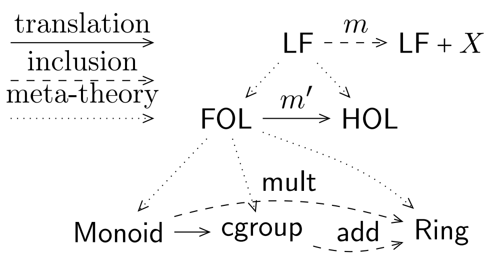


Figure 4.1: A Theory Graph with Meta-Theories and Theory Morphisms in Mmt

- URIs as globally unique logical identifiers from OpenMath,
- the standardized XML-based interchange syntax of OMDoc,

and makes them available in a single, coherent representational system for the first time. The combination of these features is based on a small set of carefully chosen, orthogonal primitives in order to obtain a simple and extensible language design.

OMDoc/Mmt offers vey few primitives, which have turned out to be sufficient for most practical settings (a more detailed grammar is given in Section 4.1.1). These are

1. *constants* with optional types and definitions,
2. types and definitions of constants are *objects*, which are syntax trees with binding, using previously defined constants as leaves,
3. *theories*, which are lists of constant declarations and
4. *theory morphisms*, that map declarations in a domain theory to expressions built up from declarations in a target theory.

Using these primitives, logical frameworks, logics and theories *within* some logic are all uniformly represented as Mmt theories, rendering all of those equally accessible, reusable and extendable. Constants, functions, symbols, theorems, axioms, proof rules etc. are all represented as constant declarations, and all terms which are built up from those are represented as objects.

Theory morphisms represent truth-preserving maps between theories. Examples include theory inclusions, translations/isomorphisms between (sub)theories and models/instantiations (by mapping axioms to theorems that hold within a model), as well as a particular theory inclusion called *meta-theory*, that relates a theory on some meta level to a theory on a higher level on which it depends. This includes the relation between some low level theory (such as the theory of groups) to its underlying foundation (such as first-order logic), and the latter's relation to the logical framework used to define it (e.g. LF).

All of this naturally gives us the notion of a *theory graph*, which relates theories (represented as nodes) via vertices representing theory morphisms (as in Figure 4.1), being right at the design core of the OMDoc/Mmt language.

Given this modular approach of OMDoc/Mmt, heterogeneity is made explicit in the sense, that even though foundations are present via meta-theories, only those aspects of the foundation that are used in the definition of a theory T are present in the theory itself, making it easy to abstract from the foundation and reuse the theory.

The OMDoc/Mmt language is used by the **Mmt system** [RK13a], which provides a powerful API to work with documents and libraries in the OMDoc/Mmt language, including a terminal to execute Mmt specific commands, a web server to display information about Mmt libraries (such as their theory graphs) and plugins for the text editors/IDEs jEdit and IntelliJ IDEA, that can be used to create, type check and compile documents into the OMDoc/Mmt language. The API is heavily customizable via plugins to e.g. add foundation specific type checking rules and import and translate documents from different formal systems.

All of this puts Mmt on a new meta level, which can be seen as the next step in a **progression towards more abstract formalisms** as indicated in the table below. In conventional mathematics (first column), domain knowledge is expressed directly in ad hoc notation. Logic (second column) provided a formal syntax and semantics for this notation. Logical frameworks (third column) provided a formal meta-logic in which to define this syntax and semantics. Now Mmt (fourth column) adds a meta-meta-level, at which we can design even the logical frameworks flexibly. (This meta-meta-level gives rise to the name Mmt with the last letter representing both the underlying theory and the practical tool.) That makes Mmt very robust against future language developments: We can, e.g., develop LF +X without any change to the Mmt infrastructure and can easily migrate all results obtained within LF.

Mathematics	Logic	Meta-Logic	Foundation-Independence
			Mmt
		logical framework	logical framework
	logic	logic	logic
domain knowledge	domain knowledge	domain knowledge	domain knowledge

4.1.1 Mmt Syntax

Intuitively, OMDoc/Mmt is a declarative language for theories and views over an arbitrary object language. For the purposes of this thesis, we will work with the (only slightly simplified) grammar given in Figure 4.2.

Thy ::=	$T[:T] = \{(\mathsf{Inc})^* \ (\mathsf{Const})^*\}$	Theory	Modules
View ::=	$T : T \rightarrow T = \{(\mathsf{Ass})^*\}$	View	
Const ::=	$C[\,:t][\,=t]$	Constant Declarations	Declarations
Inc ::=	`include` T	Includes	
Ass ::=	$C \mapsto t$	Assignments	
Γ ::=	$(x[:t][:=t])^*$	Variable Contexts	Objects
t ::=	$x \mid C \mid t\,[\Gamma]\,(t)$	Terms	

T represents a module name, C a constant name and x a variable name

Figure 4.2: The MMT Grammar

Modules and Declarations

In the simplest case, theories T are lists of **constant declarations** $c : E$, where E is an expression that may use the previously declared constants. Naturally, E must be subject to some type system (which MMT is also parametric in), for the purposes of this thesis primarily the logical framework LF (see Section 4.3) and extensions thereof.

MMT achieves language-independence through the use of **meta-theories**: every MMT-theory may designate a previously defined theory as its meta-theory. For example, when we represent the HOL Light library in MMT, we first write a theory L for the logical primitives of HOL Light. Then each theory in the HOL Light library is represented as a theory with L as its meta-theory. In fact, we usually go one step further: L itself is a theory, whose meta-theory is a logical framework such as LF. That allows L to concisely define the syntax and inference system of HOL Light.

Correspondingly, a **view** $V : T \to T'$ is a list of **assignments** $c \mapsto e'$ of T'-expressions e' to T-constants c. To be well-typed, V must preserve typing, i.e., we must have $\vdash_{T'} e' : \overline{V}(E)$. Here $\overline{V}$ is the homomorphic extension of V, i.e., the map of T-expressions to T'-expressions that substitutes every occurrence of a T-constant with the T'-expression assigned by V.

We call V **simple** if the expressions e' are always T'-*constants* rather than complex expressions. The type-preservation condition for an assignment $c \mapsto c'$ reduces to $\overline{V}(E) \mapsto E'$ where E and E' are the types of c and c'. We call V **partial** if it does not contain an assignment for every T-constant and **total** otherwise.

Importantly, we can then show generally at the MMT-level that if V is well-typed, then $\overline{V}$ preserves all typing and equality judgments over T. In particular, if we represent proofs as typed terms (see Section 4.3), views preserve the theoremhood of propositions. This property makes views so valuable for structuring, refactoring, and integrating large corpora.

Syntactically:

- Theories $T : T' = \{\texttt{include}\ T_1 \ldots \texttt{include}\ T_n \quad C_1 \ldots C_m\}$ have a module name T, an optional *meta-theory* T' and a body consisting of *includes* of other theories $T_1 \ldots T_n$ and a list of *constant declarations* $C_1 \ldots C_n$. A well-formed expression (see below) over T is allowed to only reference constants that are declared in T, T' or in one of the T_i included.

- Constant declarations $C[\,:t][\,=d]$ have a name C and two optional term components; a *type* ($[\,:t]$), and a *definition* ($[\,=d]$).

- Views $V : T_1 \to T_2 = \{\ldots\}$ have a module name V, a domain theory T_1, a codomain theory T_2 and a body consisting of assignments $C \mapsto t$.

Remark 4.1:
The module system is conservative: every theory can be elaborated (*flattened*) into one that only declares constants, by recursively replacing every `include` · statement by the list of constants declared in the included theory. We will occasionally reference the result of flattening a theory T as $T^\flat$.

Similarly, we can eliminate *defined* constants $C = d$ by replacing every occurence of the constant symbol C by its definition d (*definition expansion*).

We allow definitions in variable contexts for purely technical reasons, most notably in the context of record types (see Section 8.2). As a result, there is a notable similarity between variable contexts Γ and theories; in fact, we can think of flattened theories as named variable contexts.

Expressions

It remains to define the exact syntax of expressions. In the grammar in Figure 4.2, C refers to constants and x refers to bound variables.

Simple expressions are either references C to constants (of the meta-theory, an included theory or previously declared in the current theory) or to bound variables x. Complex expressions are of the form $o\,[x_1 : t_1, \dots, x_m : t_m]\,(a_1, \dots, a_n)$, where

- o is the operator that forms the complex expression,
- $x_i : t_i$ declares a variable of type t_i that is bound by o in subsequent variable declarations and in the arguments,
- a_i is an argument of o

The bound variable context may be empty, and we write $o\,(\vec{a})$ instead of $o\,[\cdot]\,(\vec{a})$. For example, the axiom $\forall x : \mathsf{set}, y : \mathsf{set}.\ \mathsf{finite}(x) \wedge y \subseteq x \Rightarrow \mathsf{finite}(y)$ would instead be written as

$$\forall\,[\mathtt{x} : \mathtt{set}, \mathtt{y} : \mathtt{set}]\,(\Rightarrow\,(\wedge\,(\mathtt{finite}\,(\mathtt{x}), \subseteq(\mathtt{y}, \mathtt{x})), \mathtt{finite}\,(\mathtt{y})))$$

URIs

To refer to constants (and modules), OMDoc/Mmt employs globally unique URIs, which are composed of a namespace, the name of the (containing) module and the name of a constant, separated by question marks.

Hence, Mmt URIs are triples of the form

$$\mathtt{NAMESPACE}\ ?\ \mathtt{MODULE}\ ?\ \mathtt{SYMBOL}$$

The namespace part is a URI that serves as a globally unique root identifier of a corpus, e.g. http://mathhub.info/MyLogic/MyLibrary. It is not necessary (although often useful) for namespaces to also be URLs, i.e., a reference to a physical location. But even if they are URLs, we do not specify what resource dereferencing should return. Note that because Mmt URIs use ? as a separator, `MODULE` ? `SYMBOL` is the query part of the URI, which makes it easy to implement dereferencing in practice.[1]

The module and symbol parts of an Mmt URI are logically meaningful names defined in the corpus: The module is the container (i.e. a theory, view or advanced structural feature) and the symbol is a name inside the module (of a constant, include or more advanced feature). Both module and symbol name may consist of multiple /-separated segments to allow for nested modules and qualified symbol names.

Mmt URIs allow arbitrary Unicode characters. However, ? and /, which are used as delimiters, as well as any character not legal in URIs must be escaped using the %-encoding (RFC 3986/7 standard).

By using URIs, namespaces have the great advantage of being guaranteed to be globally unique. This comes at the prize of being rather long. However, the CURIE standard can be used to introduce short prefixes.

Structures

Mmt offers an additional theory morphism that we will occasionally but rarely use in this thesis. A *structure* $S : T_1 \to T_2$ is a theory morphism that behaves like an include in that it

[1]For simplicity in the remaining part of the paper we will rarely use complete HTTP links, but rather use single keyword abbreviations.

makes all constants in T_1 accessible to T_2. Unlike includes, however, structures

1. can modify constants in the domain theory, e.g. by augmenting constants with new definientia, aliases and notations – provided, the typing preservation condition for theory morphisms is maintained, and

2. the symbols included via a structure are systematically renamed to be unique – a constant c in T_1 will be visible in T_2 as S/c. Consequently, unlike includes, structures are not idempotent.

Example 4.1:
Assume we have a theory Monoid (declaring a binary operation $\circ$ and a unit e), which is extended by a theory Group. We can specify a theory Ring by including two structures plus : Group→Ring and times : Monoid→Ring. Unlike with theory includes, the two resulting morphisms from Monoid to Ring (one directly, and one via Group) are *not* identified, giving us two distinct operations. Furthermore, plus and times can rename the operations to $+$ and $\cdot$ respectively, and rename the units to 0 and 1.

4.1.2 MathHub

MMT is used as a backend for the MathHub system [Ian+14; MH] – an online portal for mathematical documents. It is available on `mathhub.info` and hosts various MMT repositories, interfaces for browsing and semantic search [KŞ06], as well as several additional applications, such as the *semantic glossary for mathematics (SMGLoM)* [Gin+16].

Most of the specific OMDoc/MMT formalizations in this thesis are hosted on MathHub's GitLab instance (`gl.mathhub.info`) and linked accordingly.

4.2 The MMT API

The MMT system works on content represented in a fragment of OMDoc, a fact that is reflected by the internal datastructures corresponding to the syntax stated in Figure 4.2. All *expression-level* syntactic constructs are implemented as Scala case classes extending the class Term, and are named after the corresponding XML nodes of the underlying OMDoc.

MMT URIs (see Section 4.1.1) are implemented as objects of the class Path, which are assembled from individual path components of type LocalName. The relevant subtypes are DPath for namespaces, MPath for module URIs of the form (d : DPath) ? (ln : LocalName), and GlobalName for declaration URIs of the form (m : MPath) ? (ln : GlobalName).

Figure 4.3 gives a translation of the abstract syntax for expressions described above into the internal datastructures.

For a detailled and up-to-date description of the classes provided by the MMT API, I refer to the MMT documentation[2].

[2] `https://uniformal.github.io/doc/api/`

Abstract Syntax	Scala Syntax
Symbol References S	OMID(S : ContentPath)
Module References M	OMMOD(M : MPath) = OMID(M)
Constant References C	OMS(C : GlobalName) = OMID(C)
Variable References V	OMV(V : LocalName)
Applications $f(a_1, \ldots, a_n)$	OMA(f : Term, args : List[Term])
Binding Applications $f[\mathsf{con}](\mathsf{arg})$	OMBIND(f : Term, con : Context, arg: Term)
Context $G = \{\mathsf{vars}\}$	Context(vars : List[VarDecl])
Variable Declaration $v[: t][:= d]$	VarDecl(v : LocalName, t : Option[Term], d : Option[Term])

Figure 4.3: Internal Scala Datastructures for MMT Syntactical Constructors

4.2.1 Judgments

MMT has a component called *solver*, parametrized by rules, that checks **judgments** and infers implicit arguments (solving placeholder variables). The solver uses a bidirectional type system, i.e., we have two separate judgments for type *inference* and type *checking*. To check a typing judgment $t : T$ we have two possibilities:

1. find a type checking rule that is applicable to the judgment $t : T$. Type checking rules usually act on the head of the type T and recurse into the expression t, hence this can be seen as a top-down approach.

2. infer the (principal) type T' of t. Type inference rules usually act on the head of the expression t and require knowing the types of the subterms of t, hence this can be seen as a bottom-up approach. Afterwards, check whether T' is a subtype of T. If all subtyping rules are exhausted and failed (or none are applicable), check whether $T' = T$.

The MMT solver starts with a top-down approach and defers to the bottom-up approach if necessary. Equality checks are used for solving variables and by using additional *solution rules*. This approach is described in [Rab17a] in detail and is largely irrelevant for the purposes of this thesis, hence details are omitted. The details on *implementing* typing rules in MMT are treated by example in Chapter 5.

Similarly to typing, we have two equality judgments: one for checking equality of two given terms and one for reducing a term to another one (*simplification*). Our judgments are given in Figure 4.4.

For each of these rules, the solver assumes certain **pre/postconditions** to hold, which critically need to be preserved by the implemented rules. These are:

- $\Gamma \vdash U \; \mathtt{inh}$ and $\Gamma \vdash U \; \mathtt{univ}$ assume that Γ is a well-typed context.
- $\Gamma \vdash t \Leftarrow T$ assumes that T is well-typed, and implies that t is well-typed and T is inhabitable.
- $\Gamma \vdash t \Rightarrow T$ implies that both t and T are well-typed and that T is inhabitable.
- $\Gamma \vdash t_1 \rightsquigarrow t_2$ implies that t_2 is well-typed iff t_1 is (which puts additional burden on computation rules that are called on not-yet-type-checked terms).
- Equality and subtyping are only used for expressions that are assumed to be well-typed, i.e., $\Gamma \vdash t_1 \equiv t_2 : T$ implies $\Gamma \vdash t_i \Leftarrow T$, and $\Gamma \vdash T_1 <: T_2$ implies that T_i is a type/kind.

Judgment	Intuition
$\vdash \Gamma\ \mathtt{ctx}$	Γ is a well-formed context
$\Gamma \vdash T\ \mathtt{inh}$	T is *inhabitable* (may occur on the right side of a typing judgment)
$\Gamma \vdash U\ \mathtt{univ}$	U is a *universe* (inhabitable, and every T with $\Gamma \vdash T \Leftarrow U$ is inhabitable)
$\Gamma \vdash t \Leftarrow T$	t checks against inhabitable term T.
$\Gamma \vdash t \Rightarrow T$	type/kind of term t is inferred to be T
$\Gamma \vdash t_1 \equiv t_2 : T$	t_1 and t_2 are equal at type T
$\Gamma \vdash t_1 \rightsquigarrow t_2$	t_1 computes to t_2
$\Gamma \vdash T_1 <: T_2$	T_1 is a subtype of T_2

Figure 4.4: Judgments

Remark 4.2:
It is sufficient (and desirable) to consider subtyping to be an abbreviation: $\Gamma \vdash T_1 <: T_2$ iff for all t we have $\Gamma \vdash t \Leftarrow T_1$ implies $\Gamma \vdash t \Leftarrow T_2$. This allows for many subtyping rules to be derivable (usually) from typing rules.

Remark 4.3: Horizontal Subtyping and Equality
The equality judgment could alternatively be formulated as an untyped equality $t \equiv t'$. That would require some technical changes to the rules presented in this thesis, but would usually not be a huge difference. In our case, however, the use of typed equality is critical for the records introduced in Chapter 8; see Remark 8.2.

The rules given in Figure 4.5 capture the intended semantics of the judgments above and can be assumed to hold for any type system implemented in MMT.

$$\frac{}{\vdash \cdot\ \mathtt{ctx}} \qquad \frac{\vdash \Gamma\ \mathtt{ctx} \quad \Gamma \vdash T\ \mathtt{inh}}{\vdash \Gamma, x : T\ \mathtt{ctx}} \qquad \frac{\vdash \Gamma\ \mathtt{ctx} \quad \Gamma \vdash t \Leftarrow T}{\vdash \Gamma, x : T := t\ \mathtt{ctx}} \qquad \frac{\vdash \Gamma\ \mathtt{ctx} \quad \Gamma \vdash t \Leftarrow T}{\vdash \Gamma, x := t\ \mathtt{ctx}}$$

$$\frac{\vdash \Gamma\ \mathtt{ctx} \quad x : T[:= t] \in \Gamma}{\Gamma \vdash x \Rightarrow T} \qquad \frac{\vdash \Gamma\ \mathtt{ctx} \quad x[: T'] := t \in \Gamma \quad \Gamma \vdash x \Rightarrow T}{\Gamma \vdash x \equiv t : T}$$

$$\frac{\Gamma \vdash t \Rightarrow T' \quad \Gamma \vdash T' <: T}{\Gamma \vdash t \Leftarrow T} \qquad \frac{\Gamma \vdash t_1 \rightsquigarrow t_1' \quad \Gamma \vdash t_2 \rightsquigarrow t_2' \quad \Gamma \vdash t_1' \equiv t_2' : T}{\Gamma \vdash t_1 \equiv t_2 : T}$$

$$\frac{\Gamma \vdash U\ \mathtt{univ}}{\Gamma \vdash U\ \mathtt{inh}} \qquad \frac{\Gamma \vdash T \Rightarrow U \quad \Gamma \vdash U\ \mathtt{univ}}{\Gamma \vdash T\ \mathtt{inh}}$$

Figure 4.5: General Rules for MMT Judgments

The upper row contains the rules for contexts. Note that even though we allow the type T of a variable to be omitted (which will be helpful for records later), that is only allowed if a definiens t is present. (T must be inferable from t or otherwise known from the environment.)

The second row contains the rules for looking up the type and definition of variable.

The third row contains the bidirectionality rules, which algorithmically are the default rules that are applied when no type (resp. equality) checking rules are available: switch to type inference (resp. computation) and compare inferred and expected type (resp. the results).

The last row deals with universes and inhabitability.

Remark 4.4:
The inference rules as presented in this thesis always carry an algorithmic reading. To be precise, a rule of the form $\frac{P_1 \dots P_n}{C}$ can be implemented thusly:

To prove that the judgment C holds, check each of the premises $P_1 \dots P_n$ *in order.* Consequently, a premise should only contain variables or non-primitive symbols that 1) occur in the conclusion or 2) are introduced (either by type inference or simplification) in a previous premise. For example, the last rule in Figure 4.5 would still be adequate if we were to replace the premise $\Gamma \vdash T \Rightarrow U$ by the more general (and hence weaker) premise $\Gamma \vdash T \Leftarrow U$, making the inference rule itself stronger; however, it would be unclear how to implement such a rule efficiently, since there are potentially infinitely many candidates for U. The inference judgment $\Gamma \vdash T \Rightarrow U$ on the other hand unambiguously determines U as the principal type of T for the subsequent premise $\Gamma \vdash U$ $\texttt{univ}$. Both judgments are needed to solve unknown variables variables, however.

An equality premise (such as $T \equiv \prod_{x:A} B$) with as-of-yet unknown symbols (x, A, B) can algorithmically be interpreted as a pattern match; i.e. the premise that T can be simplified to have the specific syntactic form on the right hand side of the equation. This is often necessary for the inferred type of a term, as e.g. in the elimination rule for $\prod$ (Figure 4.8). Consequently, we introduce the notation $t \overset{\Rightarrow}{\equiv} E(\cdot)$ as abbreviation for the two consecutive premises $t \Rightarrow T$; $T \equiv E(\cdot)$.

4.3 Typing Rules and LF

LF [HHP93a] is a logical framework based on the dependently-typed lambda calculus $\lambda\Pi$. The **grammar** is given in Figure 4.6. The only deviation here is that we allow optional definitions in contexts, to make them syntactically equivalent to MMT contexts (see Remark 4.1). LF specifically as implemented in MMT is covered in more detail in [Rab17a]; the specific rules presented here are adapted from there.

Γ	::=	$\cdot \mid \Gamma, x[: T][:= T]$	contexts
T	::=	$x \mid \texttt{type} \mid \texttt{kind}$	variables and universes
	$\mid$	$\prod_{x:T'} T \mid T \to T' \mid \lambda_{x:T'}. T \mid T_1 T_2$	dependent function types

Figure 4.6: Grammar for Contexts and Expressions

The grammar in Figure 4.6 as well as the various grammars presented in Part II represent *object-level* languages *independent of* the MMT language. If implemented within MMT, it should not be seen as extending the MMT grammar from Section 4.1.1 rather than being embedded within it; for example, the well-formed expression $\lambda_{x:A}. t$ according to the LF grammar would be represented in OMDOC/MMT as the well-formed expression $\lambda [x : a] (t)$ according to the MMT grammar.

Note that the grammar for contexts aligns perfectly with the grammar for MMT contexts; hence we do not need to distinguish between the two apart from the well-formedness of the term components of the variables therein.

The universe rules of the framework are given in Figure 4.7 and are to be read in conjunction with the general rules in Figure 4.5.

$$\frac{\vdash \Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathtt{type}\ \mathtt{univ}} \qquad \frac{\vdash \Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathtt{kind}\ \mathtt{univ}} \qquad \frac{\vdash \Gamma\ \mathtt{ctx}}{\Gamma \vdash \mathtt{type} \Rightarrow \mathtt{kind}}$$

Figure 4.7: General Rules

Remark 4.5:
Usually, a new typing feature entails a type constructor C (in this case $\prod$), one or several term constructors $\mathtt{t}$ (in this case λ) and one or several elimination constructors $\mathtt{e}$ (in this case function application), and the rules governing these operators usually follow a specific pattern consisting of:

- A **formation rule** that specifies when $C(\cdot)$ is well-formed and what universe it belongs to.
- An **introduction rule** that allows to infer the C-type of an introductory form $\mathtt{t}(\cdot)$.
- An **elimination rule** that allows to infer the type of an elimination form $\mathtt{e}(\cdot)$.
- A **type checking rule** that tells us when a term checks against $C(\cdot)$. If every term is uniquely typed this rule is redundant (by virtue of the introduction rule), but it enables bidirectional type checking in an implementation.
- An **equality rule** that specifies when two introductory forms are equal.
- **computation rules** that (often) allow us to simplify nested elimination and introductory forms – i.e. terms of the form $\mathtt{e}(\mathtt{t}(\cdot))$ or $\mathtt{t}(\mathtt{e}(\cdot))$.
- A **subtyping rule**, usual either *covariance* (i.e. $C(A')<:C(A)$ iff $A'<:A$) or *contravariance* (i.e. $C(A')<:C(A)$ iff $A<:A'$). Since we consider subtyping an abbreviation (see Remark 4.2), this rule is usually derivable from the typing rule.

An additional rule – usually called the η-rule – that declares the introductory form to be *surjective* is often used in formal presentations. It could reasonably be called **representation rule**, since it tells us that e.g. every function can be represented as a λ-term. However, it often is a corollary of the equality and computation rules and hence does not need to be implemented as a separate rule.

In an implementation, it makes sense to add additional derivable inference, type checking, equality or computation rules, simply to speed up the type checking process or help with solving implicit arguments. For the latter, the MMT API offers an abstract class for SolutionRules, but we will largely ignore them in this thesis.

Unless otherwise specified, we assume the type constructor itself to be injective modulo α-renaming, i.e. two types $C(a), C(b)$ are equal iff $a = b$ and variables bound by C in $C(a)$ and $C(b)$ can be suitably renamed.

The specific rules for the dependent function types in LF are given in Figure 4.8. Note, that even though LF has precisely two universes `type` and `kind`, we still formulate the rules parametric in a generic universe U, to allow for flexible extending the number of universes later (see Chapter 6, particularly Section 6.4).

Definition 4.1:

For convenience, we introduce a notation for the larger of two inhabitable terms, which we will extend as needed. We start with the cases relevant to LF:

- For $A, B : \mathtt{type}$ or $A, B : \mathtt{kind}$, we let

$$\max\{A, B\} := \begin{cases} A & \text{if } B <: A, \\ B & \text{if } A <: B, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

- For $A : \mathtt{type}$, we let $\max\{A, \mathtt{type}\} := \mathtt{type}$.
- $\max\{\mathtt{type}, \mathtt{kind}\} := \mathtt{kind}$.
- In all other cases, we consider $\max\{\cdot\}$ to be undefined.

We consider $T \to T'$ an abbreviation for $\prod_{\_:T} T'$. We also write $T[x/_{T'}]$ for the usual capture-avoiding **substitution** of T' for x in T.

We can now show that the usual variance rule for function types and the η-rule are derivable. Even though plain LF has no notion of subtyping (in fact, every term has a unique type), extensions of LF may introduce subtyping principles, in which case this rule becomes important:

Theorem 4.1:

1. *The following subtyping rule is derivable:*

$$\frac{\Gamma \vdash A <: A' \quad \Gamma, x : A \vdash B' <: B}{\Gamma \vdash \prod\limits_{x:A'} B' <: \prod\limits_{x:A} B}$$

2. *The η-rule holds: For any f, A we have $\lambda_{x:A}.\, f(x) = f$.*

Proof. 1. Assume $\Gamma \vdash A <: A'$, $\Gamma \vdash B' <: B$. It suffices to show $\Gamma, f : \prod_{x:A'} B' \vdash f \Leftarrow \prod_{x:A} B$. By Figure 4.8 we can conclude this from $\Gamma, f : \prod_{x:A'} B', x : A \vdash f x \Leftarrow B$. Since $\Gamma \vdash A <: A'$ we have $\Gamma, x : A \vdash x \Leftarrow A'$ and consequently $\Gamma, f : \prod_{x:A'} B', x : A \vdash f x \Leftarrow B'$, so the claim follows by $\Gamma \vdash B' <: B$.

2. By the equality rule, we can show that given $f : \prod_{x:A} B$ we have $\Gamma, x : A \vdash (\lambda_{x:A}.\, f(x))(x) \equiv f(x) : \prod_{x:A} B$, which is precisely the conclusion of the computation rule.

For any U with $\Gamma \vdash U$ $\mathtt{univ}$:

Formation:

$$\frac{\Gamma \vdash A \Leftarrow \mathtt{type} \quad \Gamma, x : A \vdash B \Rightarrow U}{\Gamma \vdash \prod_{x:A} B \Rightarrow \max\{\, \mathtt{type}, U \,\}}$$

Introduction:

$$\frac{\Gamma, x : A \vdash N \Rightarrow B}{\Gamma \vdash \lambda_{x:A}.\, N \Rightarrow \prod_{x:A} B}$$

Elimination:

$$\frac{\Gamma \vdash F \Rrightarrow \prod_{x:A} B \quad \Gamma \vdash t \Leftarrow A}{\Gamma \vdash F t \Rightarrow B[\, x/_t \,]}$$

Type Checking:

$$\frac{\Gamma, x : A \vdash f x \Leftarrow B}{\Gamma \vdash f \Leftarrow \prod_{x:A} B}$$

Equality:

$$\frac{\Gamma, x : A \vdash f x \equiv g x : B}{\Gamma \vdash f \equiv g : \prod_{x:A} B}$$

Computation:

$$\frac{\Gamma \vdash a \Leftarrow A}{\Gamma \vdash (\lambda_{x:A}.\, t) a \rightsquigarrow t[\, x/_a \,]}$$

Figure 4.8: Rules for Dependent Function Types

$\square$

Moreover, we can show that every well-typed term t has a **principal type** T in the sense that (i) $\Gamma \vdash t \Leftarrow T$ and (ii) whenever $\Gamma \vdash t \Leftarrow T'$, then also $\Gamma \vdash T <: T'$. The principal type is exactly the one inferred by our rules (see Theorem 8.1). This is easily proven by induction on the grammar; however, as a result this property does not necessarily hold in the presence of other rules.

Remark 4.6: PLF

We can conveniently extend LF by *shallow polymorphism*, the resulting logical framework we call PLF (see [Rab17a]). It can be specified by a single additional rule:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma \vdash U \ \mathtt{univ} \quad \Gamma, x : A \vdash B \ \mathtt{inh}}{\Gamma \vdash \prod_{x:A} B \ \mathtt{inh}}$$

This rule elegantly allows for declaring shallow polymorphic functions (with type parameters only on the outside of the term), but since the polymorphic function types are merely inhabitable and not typed, they can only occur in well-typed expressions if the type arguments are instantiated via function application (the only $\prod$-rule that requires a function type to be typed by a universe is the formation rule, see Figure 4.8, hence type inference of polymorphic function types will fail, but of their applications will not).

For example, consider a type of groups over a fixed given type G. Using the above rule, we can specify this as a polymorphic operator $\mathtt{group} : \prod_{G:\mathtt{type}} \mathtt{type}$ In this case, expressions such as $G : \mathtt{group}(S)$ are perfectly valid, since `group` is well-typed. The *type* of `group` however is not well-typed; hence we can not e.g. quantify over all operators of type $\prod_{G:\mathtt{type}} \mathtt{type}$.

4.3.1 Judgments-as-Types

The lambda calculus described above makes LF a functioning logical framework, i.e. a language to specify the syntax and proof theory of various logics, type theories and related systems. This is achieved by using the **Judgments-as-Types** methodology, in which (as the name suggests) we assign types to the judgments of a given logic. We can interpret these types as the types of *proofs that the judgment holds*.

Example 4.2:
Consider classical propositional logic. We can formalize the syntax by introducing a new type $\mathsf{prop} : \mathtt{type}$ in LF, and declaring the logical connectives as constants with function types; e.g. $\mathsf{and} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$. Given A, B of type prop, $\mathsf{and}(A, B)$ is now a sytactically well-formed expression of type prop.

The only judgment in propositional logic is that some proposition is *true*, hence we introduce an operator $\mathrm{DED} : \mathsf{prop} \to \mathtt{type}$ assigning a type to each proposition. As mentioned, given some formula φ we can think of the type $\mathrm{DED}\ \varphi$ as the type of proofs of φ. By declaring a new constant of type $\mathrm{DED}\ \varphi$ we can axiomatically declare φ to be true, whereas to prove φ we need to construct a term of that type.

We can allow for the latter by formalizing the rules of any proof calculus as functions on DED -types. Consider as an example the conjunction introduction rule:

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}$$

Using DED , we can now declare this rule by introducing a function:

$$\mathsf{andI} : \prod_{A:\mathsf{prop}} \prod_{B:\mathsf{prop}} \mathrm{DED}\ A \to \mathrm{DED}\ B \to \mathrm{DED}\ \mathsf{and}(A, B)$$

It takes two propositions A, B as arguments, as well as (what we can think of as) proofs for both, and returns (what we can think of as) a proof for $A \wedge B$.

Remark 4.7:
The andI-rule in the last example suggests reading dependent function types in a different way: while we can think of the type as a function type, we can also read the expression as *"For all $A : \mathsf{prop}$ and for all $B : \mathsf{prop}$..."*. This reading is formally justified by the Curry-Howard correspondence between a type $\prod_{x:A} B$ and the universal quantifier $\forall x : A.\ B$.

Analogously, a simple function type $A \to B$ corresponds to the implication $A \Rightarrow B$.

Under the Curry-Howard correspondence, the introduction and elimination rules of a typing feature often mirror the respective inference rules in a natural deduction calculus.

4.3.2 Higher-Order Abstract Syntax

In this thesis we use the same notation for *application of* LF *functions f to arguments $\overline{a}$* as for the MMT primitive application of a term f to a list of terms $\overline{a}$. This is to avoid cumbersome notations for LF-applications like $f@\overline{a}$. However, it should be noted that technically (and internally) an application $f(a)$ of a function f on the level of LF-expressions is internally represented as $\mathsf{apply}\,(f,a)$ - the symbol apply provided by the LF-theory is applied (in the sense of an OMDoc/MMT-application) to the arguments f and a. In a naive implementation, this means that the head of the term $f(a)$ is not in fact the symbol f, but instead the symbol apply.

Furthermore, if we use LF to formalize another λ-calculus $\mathcal{C}$, we will need to provide constants for the λ- and apply-operators of $\mathcal{C}$. This naively means, that we need to reimplement many mechanisms which are already present in LF. Fortunately, we can use *higher-order abstract syntax* to have our new symbols inherit their core behaviors from the corresponding symbols in LF.

Example 4.3: λ-Calculus
We can formalize $\mathcal{C}$ by introducing constants

$$\mathcal{C}\_\mathsf{type} : \mathtt{type} \qquad \mathcal{C}\_\mathsf{expr} : \mathcal{C}\_\mathsf{type} \to \mathtt{type} \qquad \mathsf{funtype} : C\_\mathsf{type} \to C\_\mathsf{type} \to C\_\mathsf{type}$$

representing the types of $\mathcal{C}$, the expressions of type A in $\mathcal{C}$ and (simple) function types.

We can then formalize a λ-operator as an LF-function that "abuses" the LF-lambda as a binder for the variable of a λ-term; hence taking a function as argument. The $\mathcal{C}$-expression $\lambda x : A.t(x)$ can then be represented as $\mathcal{C}\_\mathsf{lambda}(\lambda_{x:\mathcal{C}\_\mathsf{expr}(A)}.\, t(x))$, and the application $f(a)$ as $\mathcal{C}\_\mathsf{apply}(f,a)$, which internally gets added a second layer, i.e.

$$\mathsf{apply}(\mathsf{apply}(\mathcal{C}\_\mathsf{apply}, f), a)$$

All of this makes it quite convenient to formalize type systems in LF. Internally, however, it is rather inconvenient to handle even simple function applications in $\mathcal{C}$, because of the nesting of apply-symbols on the various meta-levels involved.

For that reason, MMT offers functionality that allows to signify symbols belonging to a higher-order abstract syntax, and abstract them away in settings where we would rather have the term look like $f(a)$ than $\mathsf{apply}(\mathsf{apply}(\mathcal{C}\_\mathsf{apply}, f), a)$ – for example when implementing rules.

Example 4.4:
Let us focus on the higher-order abstract syntax rules for LF. The relevant symbols that the system needs to know about are LF's lambda and apply operations. Applications of apply to terms f and a are then internally simplied to applications of f to a directly, and terms of the form $f(\lambda_{x:A}.\, t)$ are simplified to binding applications $f\,[x:A]\,(t)$. The symbols are bundled in a simple case class HOAS, which is passed on to a class extending HOASNotation that takes care of the computational aspects. The case for LF hence looks like this:

```
object LFHOAS extends HOASNotation(HOAS(Apply.path, Lambda.path, OfType.path))
```

4.4 MMT/LF Surface Syntax

By and large, I will show relevant examples in MMT surface syntax directly whenever it is informative to do so. In this section I will consequently briefly introduce the structural syntax for MMT and the relevant notations for symbols provided by LF. An in-depth tutorial on using the MMT surface language to formalize mathematical content is available online.[3]

Especially noteworthy is MMT's usage of highly non-standard delimiters, namely higher unicode symbols. This is in order to allow for all standard delimiters used by other systems to be used in notations when implementing the foundations of these. A structural environment is delimited according to the level on which it is implemented (see Figure 4.2): Modules are delimited with the symbol ❚, declarations with ❙ and objects with ❘. The colors correspond to the ones used in the available MMT IDEs (jEdit and IntelliJ IDEA).

A theory named T with meta-theory M is given with the syntax theory T : M = < *content* >❚. If no namespace is provided, MMT will either take one provided by the meta information of the containing archive, or use a default namespace. A local namespace can be provided beforehand using the namespace command. Using the latter, the following will establish an empty theory with full URI http://mathhub.info/example?T:

```
namespace http://mathhub.info/example ❚
theory T = ❚
```

The theory providing LF has URI http://cds.omdoc.org/urtheories?LF; however, MMT provides the default CURIE abbreviation ur for its namespace - hence to create a new theory with LF as meta-theory, we can write theory T : ur:?LF = ❚.

A constant is declared by giving its name, followed by its *components* separated by the object delimiter ❘ in arbitrary order. A *component* is either a type (starting with :), a definiens (starting with =) or a notation (starting with #), and is delimited with the declaration delimiter ❙ – e.g. a constant c with type T and definiens t can be given either as c : T ❘= t ❙ or c = t ❘: T ❙. Includes are given with include ?T ❙

A notation is provided as a sequence of tokens and *argument markers* followed by a precedence. Argument markers are either a number literal n (denoting the nth argument of the constant), or V1 denoting a variable to be bound by the constant. If $V1$ is followed by a T, the system expects the variables provided in an application to be typed, infering their types if none are provided by a user. If an argument marker is followed by a token s followed by ellipses …, the system recognizes that a *sequence* of arguments is expected, the elements of which are delimited by s. Precedences are given as prec n for some (positive or negative) integer n – the higher the precedence, the stronger the notation binds. We will examine the notations provided by LF as an example:

Dependent function type $\prod_{x:A,y:B} C$ are written as {x:A,y:B}C – a notational practice inspired by the Twelf system. Consequently, the notation for the symbol Pi representing dependent function types has two argument markers: The first one being a comma-separated list of typed variables (represented as V1T,…) and the second one being a regular argument. Consequently, the theory ?LF declares Pi as Pi # { V1T,… } 2 prec −10000❙. It is given a deliberately extremely low precedence, since dependent function types most commonly occur on the outside of a term. Analogously, lambda is given the notation [V1T,…] 2 prec −10000.

[3] https://gl.mathhub.info/Tutorials/Mathematicians/blob/master/tutorial/mmt-math-tutorial.pdf

Simple function types have the notation `# 1→ ... prec −9990`. Finally, as usual in lambda calculi, function application has the notation `# 1%w... prec −10`, where `%w` represents arbitrary whitespace.

The syntax for views is rather analogous to theories. As a module, it is delimited with the module delimiter ❚. A view named `v` with domain `T1` and codomain `T2` is given via `view v : ?T1 -> ?T2 =` < *content* >❚. Assignments are considered declarations and use `=` as its symbol; i.e. an assignment $c \mapsto t$ is given as `c=t`.

Part II

Modular Design of Logical Frameworks

One crucial aspect of connecting libraries of mathematical knowledge to a universal framework – whether they come from theorem provers, computer algebra systems, databases or other systems – is to specify the ontological primitives of the system within the framework. Correspondingly, sufficiently expressive logical frameworks are needed to specify all the features such a system might offer.

A logical framework like LF [HHP93b], Dedukti [BCH12], or λ-Prolog [MN86] tends to admit very elegant definitions for a certain class of logics (such as first-order or higher-order logic, modal logics or classical type theories), but definitions can get awkward quickly if logics fall outside that fragment.

This often boils down to the question of shallow vs. deep encodings. The former represents a logic feature (e.g., subtyping) in terms of a corresponding framework feature, whereas the latter applies a logic encoding to remove the feature (e.g., encode subtyping in a logical framework without subtyping by using a subtyping predicate and coercion functions). Deep encodings have two disadvantages:

1. They destroy structure of the original formalization, often in a way that is not easily invertible and blow up the complexity of library translations.

2. They require the library translation to apply non-trivial and error-prone steps that become part of the trusted code base. In fact, many logical frameworks were specifically designed to allow for more logics to be defined elegantly, for example Dedukti.

Even if we ignore the proof theory (and thus the use of decision procedures) entirely, PVS (which we discuss more in-depth and integrate in Chapter 10) is particularly challenging in this regard, and hence serves as a good example for the challenges involved:

- The PVS typing relation is undecidable due to predicate subtyping: selecting a subtype of α by giving a predicate p as in $\{x \in \alpha \mid p(x)\}$. Thus, a shallow encoding is impossible in any framework with a decidable typing relation. Practical experience (discussed below) shows, that the most adequate solution is to design a new framework that allows for undecidable typing and then use a shallow encoding.

- PVS uses anonymous record types (like in SML) as a primitive feature. This includes record subtyping and a complex variant of record/product/function updates. A deep encoding of anonymous record types is extremely awkward: the simplest encoding would be to introduce a new named product type for every occurrence of a record type in the library. Even then it is virtually impossible to formalize an axiom like “two records are equal if they agree up to reordering of fields” elegantly in a declarative logical framework. Therefore, the most feasible option again is to design a new framework that has anonymous record types as a primitive.

- PVS uses several types of built-in literals, namely arbitrary-precision integers, rational numbers, and strings. Not every logical framework provides exactly the same set of built-in types and operations on them.

- PVS allows for (co)inductive types. While these are relatively well-understood by now, most logical frameworks do not support them. And even if they do, they are unlikely to match the idiosyncrasies of PVS such as automatically declaring a predicate subtype for every constructor. Again it is ultimately more promising to mimic PVS’s idiosyncrasies in the logical framework so that we can use a shallow encoding.

- PVS uses a module system that, while not terribly complex, does not align perfectly with the modularity primitives of existing logical frameworks. Concretely, theories are parametric, and a theory may import the same theory multiple times with different parameters, in which case any occurrence of an imported symbol is ambiguous. Simple deep encodings can duplicate the multiply-imported symbols or treat them as functions that are applied to the parameters. Both options seem feasible at first but ultimately do not scale well – already the PVS Prelude (the small library of PVS built-ins) causes difficulties. In Chapter 10, we will mimic PVS-style parametric imports in the logical framework as well to allow for a shallow encoding.

Florian Rabe has extensively investigated definitions of PVS in logical frameworks, going back more than 10 years when a first (unpublished) attempt to define PVS in LF was made by Schürmann as part of the Logosphere project [Pfe+03]. In the end, all of the above-mentioned difficulties pointed in the same direction: the logical framework must adapt to the complexity of PVS– any attempt to adapt PVS to an existing logical framework (by designing an appropriate deep encoding) is likely to be doomed. This negative result is in itself a notable contribution of [Koh+17b], where the PVS import described in Chapter 10 was first published. Since publication, we have found this to also hold for similarly complex object logics such as Coq.

Hence we need logical frameworks with advanced features. However, not all additional features should be active and available simultaneously all the time – the additional inference rules, even if in conjunction logically sound, can cause problems such as slowing down type checking[4], reserving notations, making the type system unnecessarily complex and potentially destroying adequacy of a library translation. Especially the first can happen easily in the presence of a *critical pair* of typing rules – two rules concluding the same judgement, but from different (and potentially expensive to check) premises. An example for such a critical pair on the *meta*-level is the two fundamental strategies in a bidirectional type checking algorithm (see Section 4.2.1).

What we actually need in practice is therefore a modular meta logical framework with a library of optional features that can be included on demand.

This library can be seen as an analog for LATIN (see Section 3.1) for logical frameworks. Where LATIN resulted in a modular library for logics using a fixed logical framework (namely LF), what we need to go beyond the "simple" logics in LATIN is a similarly modular library of logical frameworks, written in an appropriate meta-framework.

This is more difficult than it seems in that the rules for each feature must be implemented in such a way that they are *orthogonal* – they should be compatible (if possible) with each other and be minimally dependent on other features. A natural starting point of such a framework is the set of orthogonal features of Martin-Löf type theory [ML94], which are covered in Chapters 5 and 6. One of these – dependent function types – is already implemented by LF and described in Section 4.3.

Various extensions of LF have been proposed and designed before; [Pfe93] presents an extension of LF with refinement types (and *intersection types*, which we will cover in Section 7.3), [Sto+17] adds additional *union types* (essentially equivalent to coproducts, see Section 6.1). The grammatical framework GF [Ran04] is based on an extension of LF by record types.

[4]An example for such a feature is the naive implementation of declared subtyping in Section 7.2. In general, we try to avoid such implementations in this thesis.

Apart from the latter, all the mentioned extensions lack an implementation, and all of them have been designed with the additional feature as a primitive – in other words, the rules are intrinsically non-compositional and consequently not suited for a modular approach. The advantage of the primitive approach is that it allows for tightly controlling the interactions of different primitive typing features. This makes it relatively easy to ensure some desirable aspects of the typing system in the presence of several features at once, such as good performance or decidability. In fact, [Pfe93] makes decidability one of its central results. Our modular approach on the other hand makes it considerably more difficult to verify these kinds of properties. However, adding e.g. *predicate subtyping* to a logical framework – as is necessary for formalizing PVS for example (see Chapter 10) – renders the type system immediately undecidable anyway. This also holds for many other features that increase the *expressivity* of a logical framework, such as the model types presented in Chapter 8. Since expressivity is our primary concern, we consider losing decidability less of a problem in the context of this thesis. Similarly, in a tradeoff between modularity/expressivity and performance, we prioritize the former.

Conveniently, the modular implementation of the type checking component of MMT, and the abstractions provided in the MMT API, result in a very small difference between the theoretical design of a logical framework and its implementation in MMT. In fact, the set of formal rules as developed on paper corresponds closely to the set of implemented classes in Scala.

In this part, I will present various extensions of LF. In doing so, I will explain in detail how to implement new language features in MMT. For that reason, we will start with a relatively simple feature – namely $\sum$-types in Chapter 5 – as an introduction to the relevant classes of the MMT API. Chapter 6 covers additional simple typing features (primarily those of Martin-Löf type theory), culminating in a logical framework based on homotopy type theory [Uni13] as a case study in Section 6.5. Our approach to implementing logical frameworks in MMT is briefly described in [MR19], but essentially previously undocumented and hence covered in greater detail.

Continuing with more complicated features, Chapter 7 covers subtyping mechanisms in MMT, and Chapter 8 covers our implementation of record and model types in detail.

All the rule systems presented in this part follow a general pattern for typing rules presented in Remark 4.5. Also, note Remark 4.4 on how to read our rules algorithmically. The MMT API provides a dedicated class interface for each kind of rule – Figure 4.9 gives an overview of those covered in this part.

Most inference rules in Chapters 5 and 6 are taken from [Uni13] with minor adaptations to minimize dependencies on other typing features and be consistent in their presentation with the rest of this thesis. Where multiple design options exist, these are discussed explicitly – notably, in that case we do not decide on a specific rule system, but rather implement all of them as separate theories extending the same core rules, allowing to pick the most adequate implementation for a given logic formalization while reusing the same symbols across logics for the required features.

General Aspects	
MMT Judgments	Section 4.2.1
Symbol References	Section 5.2.1
Solver Judgments and Continuations	Section 5.2.2
Typing Rules	
Type Inference Rules	Section 5.2.3
Type Checking Rules	Section 5.2.4
Subtyping Rules	Section 5.2.7
Universe Rules	Section 6.4.1
Equality Rules	
Type Based Equality Rules	Section 5.2.5
Term Head Based Equality Rules	Section 6.1.2
Irrelevance Rules	Section 6.2.2
Computation Rules	Section 5.2.6
Other Features	
Generic Literals	Section 6.2.1
Structural Features	Section 6.3.2
Rule Generators / Change Listeners	Section 7.2

Figure 4.9: Table of MMT API Classes for Solver Rules

Chapter 5

Implementing Rules in MMT (e.g. $\sum$-Types)

In this chapter we will go over the implementation of $\sum$-types in MMT in detail, as a relatively simple example on how to extend the MMT solver with rules in a modular fashion. The main contribution of this chapter is the adaption of the typing rules governing $\sum$-types into the MMT framework and their implementation, as well as a detailed explanation of our methodology for constructiong a modular meta logical framework.

5.1 The Rules for $\sum$-Types

$\sum$-types are the dependent variant of cartesian products $A \times B$. The type $\sum_{x:A} B(x)$ is populated by pairs $\langle a , b \rangle$ with $a : A$ and $b : B(a)$, hence we can consider $A \times B$ as an abbreviation for $\sum_{\_:A} B$, where _ does not occur in B. The elimination forms are the two projections $\pi_\ell(\cdot)$ and $\pi_r(\cdot)$. $\sum$-types are conveniently simple, since they fall nicely into the pattern from Remark 4.5 and do not interact with $\prod$-types in any significant way.

Since $\sum$-types arise naturally as the dependent variant of Cartesian products, their history is difficult to reconstruct. To the best of my knowledge, they, along with most of the features in the subsequent chapter, were originally introduced as a consequence of the Curry-Howard correspondence (as e.g. in [ML94]), where the simple products $A \times B$ correspond to conjunctions $A \wedge B$ and dependent $\sum$-types $\sum_{x:A} B(x)$ correspond to the existential quantifier $\exists x : A.\, B(x)$.

The grammar for $\sum$-types is given in Figure 5.1.

Γ	$::=$	$\cdot \mid \Gamma, x[: T][:= T]$	contexts
T	$::=$	$x \mid \texttt{type} \mid \texttt{kind}$	variables and universes
	$\mid$	$\sum_{x:T} T \mid \langle T , T \rangle \mid \pi_\ell(T) \mid \pi_r(T)$	$\sum$-types

Figure 5.1: Grammar for $\sum$-Types

Remark 5.1:
$\sum$-types are often used in a manner similar to predicate subtypes, most notably in Coq [Coq15]: When using judgments-as-types (see Section 4.3.1), the type $\sum_{x:A} \text{DED } P(x)$ is populated by pairs $\langle a , p \rangle$, where p is a proof that $P(a)$ holds. If a system allows implicit

conversions, this means that we can convert an $a : A$ to have type $\sum_{x:A}$ DED $P(x)$ if and only if $P(a)$ holds (thus introducing a proof obligation), and trivially any pair $p : \sum_{x:A}$ DED $P(x)$ can be converted to type A by using π_ℓ.

The rules are correspondingly intuitive and given in Figure 5.2 (adapted from [Uni13]).

For any U with $\Gamma \vdash U$ univ:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma, x : A \vdash B \Rightarrow U'}{\Gamma \vdash \sum_{x:A} B \Rightarrow \max\{U, U'\}}$$

Introduction:

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash b \Rightarrow B}{\Gamma \vdash \langle a , b \rangle \Rightarrow \sum_{\_:A} B}$$

Elimination:

$$\frac{\Gamma \vdash p \Rrightarrow \sum_{x:A} B}{\Gamma \vdash \pi_\ell (p) \Rightarrow A} \qquad \frac{\Gamma \vdash p \Rrightarrow \sum_{x:A} B}{\Gamma \vdash \pi_r (p) \Rightarrow B\left[x/_{\pi_\ell(p)}\right]}$$

Type Checking:

$$\frac{\Gamma \vdash \pi_\ell (p) \Leftarrow A \quad \Gamma \vdash \pi_r (p) \Leftarrow B\left[x/_{\pi_\ell(p)}\right]}{\Gamma \vdash p \Leftarrow \sum_{x:A} B}$$

Equality:

$$\frac{\Gamma \vdash \pi_\ell (p) \equiv \pi_\ell (q) : A \quad \Gamma \vdash \pi_r (p) \equiv \pi_r (q) : B\left[x/_{\pi_\ell(p)}\right]}{\Gamma \vdash p \equiv q : \sum_{x:A} B}$$

Computation:

$$\frac{\Gamma \vdash b \Rightarrow B}{\Gamma \vdash \pi_\ell (\langle a , b \rangle) \rightsquigarrow a} \qquad \frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash \pi_r (\langle a , b \rangle) \rightsquigarrow b}$$

Figure 5.2: Rules for ∑-Types

Again, we can easily show the corresponding subtyping and η-rules:

Theorem 5.1:

The following subtyping rule is derivable:

$$\frac{\Gamma \vdash A' <: A \quad \Gamma, x : A' \vdash B' <: B}{\Gamma \vdash \sum_{x:A'} B' <: \sum_{x:A} B}$$

Furthermore, the η-rule (i.e. for any $p : \sum_{x:A} B$ we have $\langle \pi_\ell (p) , \pi_r (p) \rangle = p$) holds.

Proof. Analogous to Theorem 4.1 □

Remark 5.2:
$\sum$-Types can be naturally defined inductively, given a sufficiently strong induction principle that allows for defining polymorphic operators. In fact, this is how they are implemented in Coq.

Alternatively, they can be *almost* defined using $\prod$-types in `PLF` (see Remark 4.6) given a corresponding polymorphic equality $\doteq$; however,the computation rule for $\pi_r(\cdot)$ causes problems. Its formalization would be

$$\texttt{elim\_r} : \prod_{A:type} \prod_{B:A \to \texttt{type}} \prod_{a:A} \prod_{b:B(a)} \pi_r(\langle a , b \rangle) \doteq b$$

If $\doteq$ is a polymorphic equlity with implicit type argument, this is only well-typed if both sides of the equation (i.e. $\pi_r(\langle a , b \rangle)$ anf b) have the same type; however, b has type $B(a)$ whereas $\pi_r(\langle a , b \rangle)$ has by the elimination rule type $B(\pi_\ell(\langle a , b \rangle))$. To equate these two types, the system needs to be able to exploit the elimination rule for $\pi_\ell(\cdot)$ to rewrite $\pi_\ell(\langle a , b \rangle)$ as a, which needs a mechanism for generating computation rules from object-level equations. Such as mechanism has been implemented in Mmt, but consequently already goes beyond LF as a logical framework.

Before we look at the implementation, a couple of things should be noted:

1. Since we have two elimination forms, we have two elimination rules and two computation rules.

2. With $\sum$-types we lose the property that every term has a unique type: any pair $\langle a , b(a) \rangle$ with $b : B(a)$ has both type $A \times B(a)$ and $\sum_{x:A} B(x)$ (the former being the one that is inferred by the introduction rule). In general, neither of these two types is a subtype of the other one, so principality is lost here as well. However, a related property for sigma types is sometimes also called principality, namely that the inferred type of a term t is an *instance* of any other type of t, in the sense that if $t \Rightarrow A \times B$ (note that the inferred type of a pair is always non-dependent) and $t \Leftarrow \sum_{x:A'} B'$, then $A \times B <: A' \times B' \left[x /_{\pi_\ell(t)} \right]$.

3. The premises of the computation rules guarantee that the pair is well-typed, to satisfy the precondition for computation rules – if well-typedness has already been checked, the premises can be skipped.

Remark 5.3: Currying
In the presence of both $\sum$-types and function types, it makes sense to consider *currying*, i.e. the fact that we can think of a function $A \times B \to C$ as a function $A \to B \to C$ – or in the dependent case, a function $\prod_{p:\sum_{x:A} B(x)} C(p)$ as a function $\prod_{x:A} \prod_{y:B(x)} C(x,y)$ (and of course analogously $C : (\sum_{x:A} B(x)) \to U$ as $C : \prod_{x:A} B(x) \to U$).

It is debatable whether currying should be considered a judgment-level equality. If we want this to be the case, we can add two additional ComputationRules (see Section 5.2.6) that take care of $\prod$-types and λ-expressions with $\sum$-type arguments.

As mentioned in the introduction of this part, for sake of modularity we do not fix a specific rule set, but would rather implement such a rule in a separate theory to be included o excluded as appropriate for any specific use case. It should be noted however, that a currying rule has no notable disadvantages.

5.2 Implementing Rule Systems in Mmt

5.2.1 Mmt Symbols in Scala

Before implementing the rules themselves, we create an Mmt theory that contains the symbols we need, so that we can assign them proper notations determining their syntactic usage and refer to them in the implementations of the rules. In the case of $\sum$-types, this theory contains a symbol for the type constructor $\sum$, the introduction form $\langle\, \cdot\, ,\, \cdot\, \rangle$ and the two elimination forms $\pi_\ell\,(\,\cdot\,), \pi_r\,(\,\cdot\,)$. For convenience we add an additional symbol for simple products $A{\times}B$. The resulting theory is given in Listing 5.1.

Listing 5.1: The Symbols Theory[1]

```
namespace http://gl.mathhub.info/MMT/LFX/Sigma ❚

theory Symbols =
   Sigma         # ∑V1T,... . 2          prec −10000    ❙
   Product       # 1×...                 prec  −9990    ❙
   Tuple         # ⟨ 1,... ⟩             prec −10000    ❙
   Projl         # πl 1                  prec −900      ❙
   Projr         # πr 1                  prec −900      ❙
❚
```

Note that we specified the notations of `Sigma` , `Product` and `Tuple` all to be flexary – hence our rule will have to deconstruct a type $\sum_{x:A,y:B} C$ into the *actual* type $\sum_{x:a} \sum_{y:B} C$, and analogously a tuple $\langle a, b, c\rangle$ into the actual term $\langle\, a\, ,\, \langle\, b\, ,\, c\, \rangle\rangle$.

For convenience, we can implement helper objects in Scala, that provide apply and unapply methods to easily construct – and pattern match against – applications of our symbols. These can not only take care of the flexary notations, but also take care of `Product` being an abbreviation, which makes implementing our rules easier and more uniform.

Remark 5.4:
In Scala, if an object Foo has an apply-method it can be called by the name of the object directly; i.e. Foo(args) is an abbreviation for Foo.apply(args). An unapply-method on the other hand can be used to pattern match, i.e. if we write

```
t match {
        case Foo(args) =>
        ...
}
```

then upon encountering the case Foo(args) during runtime, the method Foo.unapply will be called on t and returns an Option[A] (for arbitrary type A). If the result is an instance of Some(ret), then in the pattern match above the case Foo(args) applies and args will be instantiated with ret. This makes Scala an extremely convenient language to use when wanting to handle complex terms of specific syntactic forms, as here.

Example 5.1:
The helper objects for Product and Sigma are given in Listing 5.2.

Listing 5.2: Helper Objects[2]

```
class LFSigmaSymbol(name:String) {
  val path = SigmaTypes.path ? name
```

[1]https://gl.mathhub.info/MMT/LFX/blob/master/source/Sigma.mmt

```
  val term = OMS(path)
}

object Product extends LFSigmaSymbol("Product") {
  def apply(t1 : Term, t2 : Term) = OMA(this.term,List(t1,t2))
  def apply(in: List[Term], out: Term) = if (in.isEmpty) out
        else OMA(this.term, in ::: List(out))
  def unapply(t : Term) : Option[(Term,Term)] = t match {
    case OMA(this.term, hd :: tl) if tl.nonEmpty =>
        Some((hd, apply(tl.init, tl.last)))
    case _ => None
  }
}
object Sigma extends LFSigmaSymbol("Sigma") {
  def apply(name : LocalName, tp : Term, body : Term) =
        OMBIND(this.term, OMV(name) % tp, body)
  def apply(con: Context, body : Term) = OMBIND(this.term, con, body)
  def unapply(t : Term) : Option[(LocalName,Term,Term)] = t match {
    case OMBIND(OMS(this.path),
        Context(VarDecl(n,None,Some(a),None,_), rest @ _*), s) =>
      val newScope = if (rest.isEmpty) s
        else apply(Context(rest:_*), s)
      Some(n,a,newScope)
    case OMA(Product.term,args) if args.length >= 2 =>
      val name = OMV.anonymous
      if (args.length > 2)
        Some((name, args.head, OMA(Product.term, args.tail)))
      else
        Some((name,args.head,args.tail.head))
    case _ => None
  }
}
```

The class LFSigmaSymbol merely provides the full path of the symbol under consideration (so we only need to provide the name) and the Term-object used to refer to the symbol in an expression. Note, that the unapply-method of Sigma has a case for Product.term, making sure that any application of the Product-symbol pattern matches against Sigma as well. This means that when implementing our rules, we only ever need to check whether a term is a $\sum$-type knowing that in doing so we cover the simple products as well. The analogous objects for pairs and projections are called Tuple and Proj1 and Proj2 respectively.

5.2.2 Solver Judgments

The solver checks a judgment $\Gamma\vdash J$ by collecting all rules included in the context Γ that correspond to the type of the judgment (an extension of the class Judgment), checking which of them are applicable and trying them in order of priority. These Judgments are:

- Equality(stack: Stack, tm1: Term, tm2: Term, tp: Option[Term]) checks the judgment $\Gamma\vdash$tm1$\equiv$tm2 :tp (where stack corresponds to the context Γ) or the untyped equality if tp is empty.

  The judgment is checked by iteratively simplifying both terms using ComputationRules (see Section 5.2.6) until they are syntactically equal or an EqualityRule (see Section 5.2.5) is applicable.

[2] https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/Sigma/Names.scala

- Typing(stack: Stack, tm: Term, tp: Term) checks the judgment $\Gamma \vdash$tm$\Leftarrow$tp by simplifying both terms until a CheckingRule (see Section 5.2.4) becomes applicable, or otherwise infers the principal type tp2 of tm using InferenceRules (see Section 5.2.3) and checking $\Gamma \vdash$tp2<:tp.

- Subtyping(stack: Stack, tp1: Term, tp2: Term) checks the judgment $\Gamma \vdash$tp1<:tp2 by first checking whether the two Terms are syntactically equal, if not simplifying both types until a SubtypingRule (see Section 5.2.7) becomes applicable, and if that fails, checking Equality(stack, tp1, tp2, None).

- Inhabitable(stack: Stack, wfo: Term) and Universe(stack: Stack, wfo: Term) finally correspond to the judgments $\Gamma \vdash$wfo `inh` and $\Gamma \vdash$wfo `univ`, respectively. They are checked using InhabitableRules and UniverseRules. In the case of inhabitability, if no InhabitableRule is applicable to wfo, its type tp is inferred and Universe(stack,tp) is checked.

To implement our rules, the MMT API provides the mentioned abstract classes for the different rule types. All of their extensions need to implement an apply-method with a particular signature. A Solver is passed on to the rule to allow for recursive checking and inference calls.

The most important methods a Solver provides for a rule are:

- check(j:Judgment) checks that the judgment j holds. This declares j to be a *necessary requirement* of the current judgment: if the judgment is disproved, an error is thrown and presented to the user.

- tryToCheckWithoutDelay(j:Judgment) like check, but does not throw an error if the check fails. Instead, the state of the solver is rolled back and variables solved during the check are reverted to their unsolved state.

- inferType(tm:Term) tries to infer the type of tm; returns an Option[Term].

- safeSimplifyUntil[A](tm:Term)(cond: Term => Option[A]) simplifies the term tm until it reaches a term t such that cond(t) returns Some(a). This is useful for enforcing a specific syntactic shape of a term (e.g. a λ-expression) and is therefore usually called with some unapply-method as argument.

- Additional syntactic sugar can be imported from info.kwarc.mmt.api.objects.Conversions._, most notably the notation x / t that creates a *substitution* of the variable with name x : LocalName and the term t : Term, and t ^? s that applies the (usual capture-avoiding) substitution s to t : Term.

5.2.3 Inference Rules

Formation, introduction and elimination rules are all trivial extensions of the class InferenceRule. They take as class argument the head symbol of the terms to which they apply (and the typing symbol used for the typing judgments) and its apply-method has the following signature:

```
def apply(solver: Solver)(tm: Term, covered: Boolean)
        (implicit stack: Stack, history: History): Option[Term]
```

where tm is the term whose type is to be inferred, and covered is a boolean flag that tells the rule whether the term tm has already been checked to be well-typed, and hence that the preconditions (see Section 4.2.1) are satisfied.

The implicit argument stack contains the context in which the type of tm is to be inferred. history is an object that holds textual information on the current state of the solver – if a check fails, the history is presented to the user as feedback.

An inference rule with head h is considered *applicable* to a term, if the head of the term is h. Additionally, a rule can override a method alternativeHeads:List[GlobalName] if it should be applicable to other heads as well, as in our case with `Sigma` and `Product` (see Example 5.3).

Example 5.2:
Listing 5.3 shows the Scala code for the introduction rule:

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash b \Rightarrow B}{\Gamma \vdash \langle a \,, b \rangle \Rightarrow \sum_{\_:A} B}$$

The rule itself has as parameters the MMT URI for the `Tuple`-symbol and the symbol for the typing operator (ultimately provided by LF). The apply-method

Line 7 deconstructs the term to be a Tuple(t1,t2), i.e. of the form $\langle t_1 \,, t_2 \rangle$,

Line 8 infers the type of t1 to be tpA (or returns None if this fails),

Line 11 infers the type of t2 to be tpB (or returns None if this fails),

Line 14 picks a fresh variable name xn not occuring in the current context and

Line 15 returns the type of the tuple term xn as Some(Sigma(xn,tpA,tpB)), i.e. the type $\sum_{\mathsf{xn:tpA}} \mathsf{tpB}$.

Listing 5.3: Introduction Rule[3]

```
1  /** Introduction: the type inference rule
2    * |-t1:A, |-t2:B ---> <t1,t2>:Sigma x:A.B **/
3  object TupleTerm extends IntroductionRule(Tuple.path, OfType.path) {
4    def apply(solver: Solver)(tm: Term, covered: Boolean)
5            (implicit stack: Stack, history: History) : Option[Term]
6      = tm match {
7        case Tuple(t1,t2) =>
8          val tpA = solver.inferType(t1)(stack, history).getOrElse(
9              return None
10         )
11         val tpB = solver.inferType(t2)(stack, history).getOrElse(
12             return None
13         )
14         val (xn,_) = Common.pickFresh(solver,"x")
15         Some(Sigma(xn,tpA,tpB))
16       case _ => None // should be impossible
17     }
18 }
```

[3] https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/Sigma/Rules.scala

5.2.4 Checking Rules

Similarly to inference rules, TypingRules (which trivially extend CheckingRule) take the head symbol of the *type* to which they apply as argument. They too have an overridable method alternativeHeads:List[GlobalName] if multiple head symbols are covered. Additionally, checking rules can *shadow* other rules (by overriding the method shadowedRules:List[Rule], in which case the shadowed ones are deactivated in the presence of the new rule. Furthermore, checking rules can be assigned a priority : Int with default value 0 to control in which order the solver attempts multiple applicable rules. The signature of their apply-method that needs to be implemented is

```
def apply(solver: Solver)(tm: Term, tp: Term)
        (implicit stack: Stack, history: History) : Option[Boolean]
```

Alternatively to returning a simple Some(Boolean) value, checking rules can also:

1. throw DelayJudgment(msg: String) – can be thrown if the result depends on an as-of-yet unsolved variable. In this case, the solver *delays* the current typing judgment until either all variables have been solved or no other judgment remains to be checked.

2. throw SwitchToInference – indicates that the solver should switch from top-down to a bottom-up approach using type inference to check this Judgment (see Section 4.2.1).

3. return None – indicating that the rule has determined to not be applicable to this judgment after all. This is useful when a quick, syntactical applicability check is insufficient to determine whether this rule can be used to check the given judgment.

Example 5.3:
Listing 5.4 shows the implementation of the type checking rule for ∑-types:

$$\frac{\Gamma\vdash\pi_\ell(p)\Leftarrow A \quad \Gamma\vdash\pi_r(p)\Leftarrow B\left[x/_{\pi_\ell(p)}\right]}{\Gamma\vdash p\Leftarrow\sum_{x:A} B}$$

Line 4 adds the `Product` symbol to the list of heads this rule is applicable to.

Line 8 deconstructs the type to be a Sigma(x,tpA,tpB), i.e. of the form $\sum_{\mathsf{x:tpA}} \mathsf{tpB}$. Note that by virtue of Sigma's unapply method (see Listing 5.2), this takes care of applications of the `Product` symbol as well.

Line 9 instructs the solver to verify the judgment that Proj1(tm) (i.e. $\pi_\ell(\mathsf{tm})$) has type tpA,

Line 10 instructs the solver to verify the judgment that Proj2(tm) (i.e. $\pi_r(\mathsf{tm})$) has type tpB ^? (x/Proj1(tm)) (i.e. $\mathsf{tpB}\left[x/_{\pi_\ell(\mathsf{tm})}\right]$). Since Solver.check returns true if and only if the checked judgment can be verified (and throws an error otherwise) this line can serve as the return statement.

Listing 5.4: Type Checking Rule[4]

```
1  /** type-checking: the type checking rule
2    * pi1(t):A|-pi2(t):B(pi1(t)) ---> t : Sigma x:A.B **/
3  object SigmaType extends TypingRule(Sigma.path) {
4    override def alternativeHeads:List[GlobalName] = List(Product.path)
5
6    def apply(solver:Solver)(tm:Term, tp:Term)
```

```
        (implicit stack: Stack, history: History) : Option[Boolean] = tp match {
      case Sigma(x,tpA,tpB) =>
        solver.check(Typing(stack,Proj1(tm),tpA))
        Some(solver.check(Typing(stack,Proj2(tm),tpB ^? (x / Proj1(tm)))))
      case _ => None // should be impossible
    }
}
```

5.2.5 Equality Rules

The MMT API provides three classes for equality rules: TypeBasedEqualityRule, TermBasedEqualityRule and TermHeadBasedEqualityRule.

- TypeBasedEqualityRules correspond to typed equalities and use the head symbol of the *type* of two terms tm1 and tm2 to determine applicability. Obviously, for these rules to fire, the types of the two terms have to be known to the solver and be equal.

  They take as class parameters the head symbol of the governing type, as well as a (possibly empty) list of application-symbols, in case that the type constructor is applied using a higher-order abstract syntax. This can be used to implement equality rules for e.g. symbols from a logic or type theory which is itself implemented in a logical framework.

  Additionally, an instance of this class has to implement an applicableToTerm-method. This is convenient in case where the rule requires the two terms to have a specific form.

  The signatures of the methods to implement are:

  ```
  def applicableToTerm(tm: Term): Boolean
  def apply(solver: Solver)(tm1: Term, tm2: Term, tp: Term)
          (implicit stack: Stack, history: History): Option[Boolean]
  ```

  The applicableToTerm-method will be called on both terms and the rules is considered applicable if *either* returns true.

  The apply should return Some(true) or Some(false) if the equality of the two terms tm1 and tm2 is provable or disprovable, and None if the solver should proceed trying other equality rules.

- TermBasedEqualityRules are the most general equality rules. Their apply method takes an optional argument for the type of the two terms if known, but the solver determines applicability by calling an applicable-method that needs to be implemented. The signatures of the methods to implement are thus:

  ```
  def applicable(tm1: Term, tm2: Term): Boolean
  def apply(check: CheckingCallback)(tm1: Term, tm2: Term, tp: Option[Term])
          (implicit stack: Stack, history: History): Option[Boolean]
  ```

  The applicable-method should fail quickly; preferentially it should only check whether the terms tm1 and tm2 have a specific syntactic form.

  The apply-method only takes a CheckingCallback, which is a superclass of Solver with slightly limited functionality.

[4] Ibid.

- TermHeadBasedEqualityRule finally is a simple extension of TermBasedEqualityRule that takes two head symbols (for tm1 and tm2) as arguments and implements an applicable method that uses the head symbols.

Example 5.4:
In the case of ∑-types, we have a typed equality rule (for pairs at their ∑-types):

$$\frac{\Gamma \vdash \pi_\ell(p) \equiv \pi_\ell(q) : A \quad \Gamma \vdash \pi_r(p) \equiv \pi_r(q) : B\left[x/_{\pi_\ell(p)}\right]}{\Gamma \vdash p \equiv q : \sum_{x:A} B}$$

Hence we choose a TypeBasedEqualityRule, which is given in Listing 5.5.

Line 4 adds the `Product` symbol to the list of heads this rule is applicable to.

Line 5 determines that the rule is applicable to any pair of terms.

Line 10 deconstructs the type to be a Sigma(x,tpA,tpB), i.e. of the form $\sum_{\mathsf{x:tpA}} \mathsf{tpB}$.

Line 11 checks that the projections π_ℓ of the two terms are equal under the type tpA.

Line 12 checks that the two terms Proj2(tm1) and Proj2(tm2) are equal under the type $\mathsf{tpB}\left[\mathsf{x}/_{\pi_\ell(\mathsf{tm1})}\right]$ and returns the result of that check.

Listing 5.5: Equality Rule[5]

```
1  /** equality-checking: the rule
2    * |- pi1(t1) = pi1(t2) : A , |- pi2(t1) = pi2(t2) : B(pi1(t)) ---> t1 = t2 : Sigma x:A. B **/
3  object TupleEquality extends TypeBasedEqualityRule(Nil,Sigma.path) {
4    override def alternativeHeads:List[GlobalName] = List(Product.path)
5    override def applicableToTerm(tm: Term): Boolean = true
6
7    def apply(solver: Solver)(tm1: Term, tm2: Term, tp: Term)
8            (implicit stack: Stack, history: History): Option[Boolean]
9      = tp match {
10       case Sigma(x, tpA, tpB) =>
11         solver.check(Equality(stack, Proj1(tm1), Proj1(tm2), Some(tpA)))
12         Some(solver.check(Equality(stack, Proj2(tm1), Proj2(tm2),
13           Some(tpB ^? (x / Proj1(tm1))))))
14       case _ => None // should be impossible
15     }
16 }
```

5.2.6 Computation Rules

Computation rules implement the class ComputationRule, which takes as class parameter the head symbol of the term to decide applicability. This is only a default implementation, though – the governing method `def applicable(tm : Term): Boolean` can be overridden. The apply-method has the following signature:

```
def apply(check: CheckingCallback)(tm: Term, covered: Boolean)
        (implicit stack: Stack, history: History): Simplifiability
```

[5] Ibid.

It returns an object of type Simplifiability that instructs the solver (or other CheckingCallback instance) on how to recurse into a term. The latter has by and large three possible instances:

1. Simplify(result: Term): The rule can simplify the given term to the new term result.
2. RecurseOnly(positions: List[Int]): This rule can *not* simplify the term right now, but if the subterms at the positions given in positions can be, this rule might become applicable.
3. Simplifiability.NoRecurse: This rule can *not* simplify the term at all (as long as the head does not change). This is equivalent to RecurseOnly(Nil).

The solver can use the information returned by a ComputationRule to strategically recurse only into those subterms that might cause a rule to become applicable, without unnecessarily simplifying subexpressions or expanding definitions.

Example 5.5:
In the case of $\sum$-types, we have the following computation rules:

$$\frac{\Gamma \vdash b \Rightarrow B}{\Gamma \vdash \pi_\ell\,(\langle a\,,\,b\rangle) \rightsquigarrow a} \qquad \frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash \pi_r\,(\langle a\,,\,b\rangle) \rightsquigarrow b}$$

The implementation of the first one is presented in Listing 5.6. The second one is completely analogous.

Line 5 is the intended case for the rule – a projection π_ℓ applied to a pair. In that case, the rule can simplify by returning the first component of the pair a.

Line 6 To satisfy the precondition, we call inferType(b,false) if tm has not been checked to be well-typed (i.e. covered is false).

Line 8 is the case where we have a projection π_ℓ, but (by exclusion) no pair construct as its argument. In that case, the rule *might* become applicable if we manage to simplify the first (and only) argument of the projection such that it *becomes* an actual pair, so we return RecurseOnly(List(1)) to instruct the solver accordingly.

Line 9 is the default case, where tm is not a projection. By virtue of the class parameter Proj1.path, this case should never happen, but the general information to return (Simplifiability.NoRecurse) would be that this rule is never applicable to the term given, unless the head changes (to an actual projection).

Listing 5.6: Computation Rule[6]

```
1  /** computation: the beta−reduction rule Proj1(<a,b>) = a **/
2  object Projection1Beta extends ComputationRule(Proj1.path) {
3    def apply(solver: CheckingCallback)(tm: Term, covered: Boolean)
4            (implicit stack: Stack, history: History) = tm match {
5      case Proj1(Tuple(a,b)) =>
6            if (!covered) solver.inferType(b,false)
7            Simplify(a)
8      case Proj1(_) => RecurseOnly(List(1))
9      case _ => Simplifiability.NoRecurse // should be impossible
10   }
11 }
```

5.2.7 Subtyping Rules

Finally, to implement subtyping rules MMT offers a generic class SubtypingRule, and conveniently a more specific class VarianceRule for the case where we want to declare an operator to be covariant or contravariant in its arguments. The former needs to implement an applicable(tp1 : Term, tp2 : Term)-method, the latter takes a head symbol as class argument that determines the applicability of the rule. In both cases, the apply-method has the signature

```
def apply(solver: Solver)(tp1: Term, tp2: Term)
        (implicit stack: Stack, history: History) : Option[Boolean]
```

Example 5.6:
In the case of $\sum$-types, we have the derivable subtyping rule given in Theorem 5.1:

$$\frac{\Gamma \vdash A' <: A \quad \Gamma, x : A' \vdash B' <: B}{\Gamma \vdash \sum_{x:A'} B' <: \sum_{x:A} B}$$

The implementation of this rule is presented in Listing 5.7.

Line 7 deconstructs both types into $\sum$-types.

Line 8 instructs the solver to check that a1 is a subtype of a2.

Line 9 picks a new variable name xn not occuring in the current context.

Lines 10–13 instruct the solver to check that in the context extended by xn:a1 (stack ++ xn%a1), b1 ^? (x1/OMV(xn)) is a subtype of b2 ^? (x2/OMV(xn)) (i.e the original variable names are replaced by the one introduced in Line 9).

Listing 5.7: Subtyping Rule[7]

```
1  /** The variance rule
2     * |-A1<:A2, x:A1|-B1<:B2 ---> Sigma x:A1.B1 <: Sigma x:A2.B2 **/
3  object SigmaSubtype extends VarianceRule(Sigma.path) {
4    def apply(solver: Solver)(tp1: Term, tp2: Term)
5           (implicit stack: Stack, history: History) : Option[Boolean]
6     = (tp1,tp2) match {
7      case (Sigma(x1,a1,b1),Sigma(x2,a2,b2)) =>
8        solver.check(Subtyping(stack,a1,a2))
9        val (xn,_) = Common.pickFresh(solver,x1)
10       Some(solver.check(Subtyping(stack ++ xn%a1,
11         b1 ^? (x1/OMV(xn)),
12         b2 ^? (x2/OMV(xn))
13       )))
14     case _ => None // should be impossible
15   }
16 }
```

After implementing the rules in Scala, we can import them into an MMT theory. Including the latter into any theory will then activate those rules. It makes sense to declare the symbols and rules in separate theories, so we can reuse the same symbols with different sets of typing rules,

[6]Ibid.
[7]Ibid.

as in Listing 5.8. Ultimately, we implement four theories in total: for the symbols (`Symbols`), the rules (`Rules`), a theory that combines symbols and rules in the context of the basic typing rules (see Figure 4.7) from LF (`TypedSigma`), and finally one that adds symbols and rules on top of the full LF theory with dependent function types (`LFSigma`). We will roughly follow this naming convention throught this part; the various theories for e.g. the symbols are disambiguated by the namespaces (in this case `http://gl.mathhub.info/MMT/LFX/Sigma`).

Listing 5.8: The Rules Theory[8]

```
import rules  scala://Sigma.LFX.mmt.kwarc.info❚

theory Rules =
  rule  rules ?SigmaTerm          ❘
  rule  rules ?TupleTerm          ❘
  rule  rules ?Projection1Term    ❘
  rule  rules ?Projection2Term    ❘
  rule  rules ?SigmaType          ❘
  rule  rules ?TupleEquality      ❘
  rule  rules ?Projection1Beta    ❘
  rule  rules ?Projection2Beta    ❘
  rule  rules ?SigmaSubtype       ❘
❚

theory TypedSigma : ur:?TermsTypesKinds =
  include ?Symbols        ❘
  include ?Rules          ❘
❚

theory LFSigma =
  include ?TypedSigma             ❘
  include ur:?LF                  ❘
❚
```

The import statement in the first line of Listing 5.8 introduces an abbreviation for the namespace `scala://Sigma.LFX.mmt.kwarc.info`, which by virtue of the `scala`-scheme determines the fully qualified class path of the scala objects referenced by the subsequently declared rule constants.

5.3 Using ∑-Types

The theory in Listing 5.9 from the Math-in-the-Middle library serves as an example; it uses ∑-types to implement the product of two vector spaces, by defining the corresponding operations on the product space – since the latter is a natural occurence of Cartesian products in mathematics, ∑-types are naturally ideal for formalizing products of spaces (analogous e.g. product topologies, categories etc.). In fact, thanks to ∑-types,the definitions for the universes, operations, units and inverses here conforms quite naturally to the usual informal presentation of product spaces.

Listing 5.9: Product Spaces using Sigma Types[9]

```
theory Productspace : base:?Logic =
        include ?Vectorspace ❘
        include  base:?ProductTypes ❘

        product_universe : {F : field } vectorspace F → vectorspace F → type ❘
```

[8]https://gl.mathhub.info/MMT/LFX/blob/master/source/Sigma.mmt

```
            = [F,v1,v2]  v1.universe  × v2.universe  |  # 2 x_ 1 3|
    product_op : {F : field , v1: vectorspace F, v2: vectorspace F}
      (v1 x_ F v2) → (v1 x_ F v2) → (v1 x_ F v2) |
            = [F,v1,v2] [a,b] ⟨ ((v1.op) (πl a) (πl b)), ((v2.op) (πr a) (πr b))⟩ | # 2 xop_ 1 3 |
    product_unit : {F : field , v1 : vectorspace F, v2 : vectorspace F} v1 x_ F v2 |
            = [F,v1,v2] ⟨ (v1.unit), (v2.unit) ⟩ | # product_unit 1 2 3 |
    product_inverse : {F : field , v1 : vectorspace F, v2 : vectorspace F}
      (v1 x_ F v2) → (v1 x_ F v2) |
            = [F,v1,v2] [x] ⟨ (v1.inverse ) πl x, (v2.inverse ) πr x ⟩ | # product_inverse 1 2 3 |
    product_scalarmult : {F : field , v1: vectorspace F, v2: vectorspace F}
      F.universe → (v1 x_ F v2) → (v1 x_ F v2) |
            = [F,v1,v2][ α,v] ⟨ ((v1.scalarmult ) α (πl v)), ((v2.scalarmult ) α (πr v)) ⟩ |
            # product_scalarmult 1 2 3 |
```

[9]https://gl.mathhub.info/MitM/smglom/blob/master/source/algebra/modulsvectors.mmt

Chapter 6

LFX

LFX [LFX] is an Mmt archive that contains various features as extensions of LF. Two such extensions are already part of the urtheories archive [OMU], namely PLF (see Remark 4.6) and LFS [HKR14], that adds flexary operators and sequence arguments.

Many of the features in this chapter are primitives in homotopy type theory, and are hence treated in some detail in [Uni13]. The contribution of this chapter consists, as the previous chapter, in adapting the rules for the typing features covered here to the Mmt framework and their implementation. Together and correspondingly modularly implemented, these features yield a flexible, modular meta logical framework following the LATIN approach.

6.1 Coproducts

Coproducts $A \oplus B$ can be seen as *disjoint union types* – their introduction form is a simple embedding, the elimination a case distinction on the constituent types A and B. In particular, coproducts give us a relatively simply implementable mechanism for pattern matching.

Under propositions-as-types, coproducts are the type-level correspondants to logical disjunction. Additionally, they arise rather naturally as the (cateogory theoretical) dual notion to simple Cartesian products.

To be more precise:

- For every $a : A$ and every type B, there is an element $(a \hookrightarrow_{\ell} B) : A \oplus B$ and an element $(B_{r}\!\hookleftarrow a) : B \oplus A$
- For $c_1, c_2 : C$ and $c : A \oplus B$, we have the elimination form $(\mathbf{match}_c\{ x \Longrightarrow_C c_1 \mid c_2 \})$ of type $C[x/_c]$ (where $x : A$ may occur in c_1, $x : B$ may occur in c_2 and $x : A \oplus B$ may occur in C – the details regarding instantiating C are apparent from the rules in Figure 6.2)
- We have $(\mathbf{match}_{a \hookrightarrow_{\ell} B}\{ x \Longrightarrow_C c_1 \mid c_2 \}) = c_1[x/_a] : C[x/_{a \hookrightarrow_{\ell} B}]$ and $(\mathbf{match}_{A_{r}\hookleftarrow b}\{ x \Longrightarrow_C c_1 \mid c_2 \}) = c_2[x/_b] : C[x/_{A_{r}\hookleftarrow b}]$.

The grammar for coproducts is given in Figure 6.1

Remark 6.1: Function Addition
The elimination form basically gives us functions on $\oplus$-types via case distinction, each case of which we can think of as a lambda expression. Hence it makes sense to add the following abbreviation:

$$\begin{array}{llll}
\Gamma & ::= & \cdot \mid \Gamma, x[: T][:= T] & \text{contexts} \\
T & ::= & x \mid \mathtt{type} \mid \mathtt{kind} & \text{variables and universes} \\
 & \mid & T \bigoplus T \mid T \hookrightarrow_\ell T \mid T_r \hookleftarrow T \mid \mathbf{match}_T \{ x \Longrightarrow_T T \mid T \} & \bigoplus\text{-types}
\end{array}$$

Figure 6.1: Grammar for Coproduct-Types

Definition 6.1:

For $f : \prod_{x:A} C(x \hookrightarrow_\ell B)$ and $g : \prod_{x:B} C(A_r \hookleftarrow x)$, let:

$$f \oplus g \;:=\; \lambda_{y:A\oplus B}.\, \mathbf{match}_y \{ x \Longrightarrow_{C(x)} f(x) \mid g(x) \} \quad : \prod_{x:A\bigoplus B} C(x)$$

(In the independent case $f : A \to C$ and $g : B \to C$, we get $f \oplus g : A \bigoplus B \to C$)

The implementation consists of a single computation rule for terms $f \oplus g$.

Notably, [Uni13] uses lambda-expressions in the elimination form of $\bigoplus$-types directly. While this is equivalent to our treatment in the presence of dependent function types, it makes $\bigoplus$-types entirely dependent on function types, thus breaking modularity.

The basic rule system is given in Figure 6.2, but can be extended for a more desirable behavior in presence of subtyping.

For any U with $\Gamma \vdash U$ $\mathtt{univ}$:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma \vdash B \Rightarrow U'}{\Gamma \vdash A \bigoplus B \Rightarrow \max\{U, U'\}}$$

Introduction:

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash B \Rightarrow U}{\Gamma \vdash a \hookrightarrow_\ell B \Rightarrow A \bigoplus B} \qquad \frac{\Gamma \vdash b \Rightarrow B \quad \Gamma \vdash A \Rightarrow U}{\Gamma \vdash A_r \hookleftarrow b \Rightarrow A \bigoplus B}$$

Elimination:

$$\frac{\Gamma, x : A \vdash C \Rightarrow U \quad \Gamma \vdash p \Rrightarrow A \bigoplus B \quad \Gamma, x : A \vdash c_1 \Leftarrow C[x/_{x \hookrightarrow_\ell B}] \quad \Gamma, x : B \vdash c_2 \Leftarrow C[x/_{A_r \hookleftarrow x}]}{\Gamma \vdash \mathbf{match}_p \{ x \Longrightarrow_C c_1 \mid c_2 \} \Rightarrow C[x/_p]}$$

Computation:

$$\frac{\Gamma \vdash B \Rightarrow U \quad \Gamma, x : B \vdash c_2 \Leftarrow C[x/_{A_r \hookleftarrow x}]}{\Gamma \vdash \mathbf{match}_{a \hookrightarrow_\ell B} \{ x \Longrightarrow_C c_1 \mid c_2 \} \rightsquigarrow c_1[x/_a]} \qquad \frac{\Gamma \vdash A \Rightarrow U \quad \Gamma, x : A \vdash c_1 \Leftarrow C[x/_{x \hookrightarrow_\ell B}]}{\Gamma \vdash \mathbf{match}_{A_r \hookleftarrow b} \{ x \Longrightarrow_C c_1 \mid c_2 \} \rightsquigarrow c_2[x/_b]}$$

Figure 6.2: Rules for Coproducts

In a categorial setting, coproducts are *dual* to certesian products as covered in Chapter 5. As a result, the roles of introduction and elimination forms in our rule pattern are somewhat reversed. Consequently, there are a couple of things to note about our rules:

- Under the Curry-Howard correspondence, coproducts $A \bigoplus B$ correspond to disjunctions

$A \vee B$.

- This time, we have two introduction forms and one elimination form.
- The premises in the computation rules again guarantee that the preconditions for computation rules are satisfied. If the injection and **match** expressions are known to be well-typed, the premises can be skipped.
- Before, the type checking rules always governed generic expressions of the newly introduced type – more precisely: a term type checks against the type constructor, if all applications of the elimination forms to the term have the expected type. This works well for dependent functions (by applying the function to a generic variable and checking against the codomain) and $\sum$-types (by checking the two projections against the expected types). In the case of coproducts this would translate to *"a term p checks against $A \bigoplus B$ if all pattern matches with cases A, B are applicable to p"*. This rule would look like this:

$$\frac{\text{For } C = \mathbf{match}_x\{\, y \Longrightarrow_U A \mid B \,\}: \quad \Gamma \vdash \mathbf{match}_p\{\, x \Longrightarrow_C x \mid x \,\} \Leftarrow C[\, x/_p \,]}{\Gamma \vdash p \Leftarrow A \bigoplus B}$$

  The crucial premise here is the second one, since to evaluate the **match**-expression, we will need to infer the type of p. Hence this typing rule would in practice reduce to checking whether the inferred type of p is a subtype of $A \bigoplus B$, something the solver checks as a default anyway.

- A representation (or η) rule in the sense that *"every element of a $\bigoplus$-type can be represented as an introduction form"* does not exist. In general, a term $p : A \bigoplus B$ can not be coerced to be an introductory form, since while *morally* p is *either* an injection from A *or* from B, it is impossible to determine which without p already being an injection.
- Note that the rules given here seem to break extensionality for functions: Assume $f : A \bigoplus B \to C$. Then $\lambda_{x:A \bigoplus B}.\ \mathbf{match}_x\{\, y \Longrightarrow_C f(y \hookrightarrow_\ell B) \mid f(A_r \hookleftarrow y) \,\}$ is not equal to f, even though for any *particular* introduction form for $A \bigoplus B$ both functions agree.

  It is thus debatable whether "equal for any particular introduction form" should imply "equal everywhere". If we answer this question positively we imply axiom K [HS02], which is equivalent to term irrelevance for equality types[1] and hence incompatible with e.g. HoTT (see Section 6.5) and similar constructive systems. Hence, if we want to add an equality rule to preserve extensionality, we should do so in a separate theory and exclude it when formalizing constructive logics and type theories.

6.1.1 Subtyping Behavior of Coproducts

We have previously used the type checking rules as a justification for derivable subtyping rules. Since a type checking rule for arbitrary elements of $A \bigoplus B$ does not exist (in a meaningful way), we can instead introduce a subtyping rule and use that to derive type checking rules for the introductory forms. The obvious guiding criterion to use is that any pattern match on $A \bigoplus B$ needs to be applicable to any $x : S$ for $S <: A \bigoplus B$. This implies the following covariance rule:

$$\frac{\Gamma \vdash A' <: A \quad \Gamma \vdash B' <: B}{\Gamma \vdash A' \bigoplus B' <: A \bigoplus B}$$

[1] Many thanks to Claudio Sacerdoti Coen for pointing this out to me.

from which we can derive the following typing rules:

$$\frac{\Gamma\vdash a\Leftarrow A \quad \Gamma\vdash B'<:B}{\Gamma\vdash a\hookrightarrow_\ell B'\Leftarrow A\bigoplus B} \qquad \frac{\Gamma\vdash b\Leftarrow B \quad \Gamma\vdash A'<:A}{\Gamma\vdash A'{}_r\hookleftarrow b\Leftarrow A\bigoplus B}$$

This is especially useful in the presence of an empty type $\varnothing$ (see Section 6.2), where $a\hookrightarrow_\ell\varnothing$ now type checks against any $A\oplus\_$ (analogously for ${}_r\hookleftarrow$). However, we also need to add an additional equality rule then, since all left injections $a\hookrightarrow_\ell B_i$ should be equal (at some $\oplus$-type). Hence:

$$\frac{\Gamma\vdash a_1\equiv a_2 :A}{\Gamma\vdash a_1\hookrightarrow_\ell B_1\equiv a_2\hookrightarrow_\ell B_2 :A\bigoplus B} \qquad \frac{\Gamma\vdash b_1\equiv b_2 :B}{\Gamma\vdash A_1{}_r\hookleftarrow b_1\equiv A_2{}_r\hookleftarrow b_2 :A\bigoplus B}$$

6.1.2 Implementation

Given the rules in Figure 6.2 and the examples in Chapter 5, an implementation of Coproducts is rather straight-forward.

Listing 6.1 and Listing 6.2 show excerpts of the associated MMT theory and a toy example for testing the implementation. Notably, both $\oplus$ and **match** are implemented with flexary notations; a helper object (similar to $\sum$-types, see Chapter Chapter 5) takes care of deconstructing expressions with multiple arguments into binary applications of the symbol.

Notably, coproducts are not very useful in and of themselves; however, in conjunction with a unit type they are sufficient to construct finite types (see Section 6.2), and are used to construct W-types (see Section 6.3).

Remark 6.2:
The notation of coproducts as an "additive" construct $A\oplus B$ is suggestive of a more general correspondance:

Thinking of coproducts as sums, we can think about the result of iteratively taking the coproduct of a fixed type B over some index type A; i.e. $\underbrace{B\oplus B\oplus\ldots\oplus B}_{|A|\text{ times}}$. Naturally, an element of this type consists of some element $b:A$ and its *index* $a:A$; i.e. we can identify this type with the type $A\times B$. Analogously, taking an indexed coproduct over a type *family* $B(a)$ we can think of the type $\bigoplus_{a:A} B(a)$ as the dependent sigma type $\sum_{a:A} B(a)$. In other words: A product is an iterated sum, reminiscent of arithmetics.

Analogously, we can think of a dependent function type $\prod_{a:A} B(a)$ as the iterated product $B(a_1)\times B(a_2)\times\ldots$, an element of which consists of exactly one element $b:B(a)$ for each $a:A$, which we can naturally interpret as a set of pairs, i.e. a function.

Note that this is merely an intuitive correspondence, but neither relevant nor helpful from an implementation perspective.

6.2 Finite Types

Finite types[3] are types with a fixed, finite number of elements. Their predetermined number of elements can be exploited using case distinction. Their usefulness is immediately obvious

[2]https://gl.mathhub.info/MMT/LFX/tree/master/source

[3]also called *enumeration types*, however, this usually refers to (a priori mutually disjoint) types with finitely many *uniquely named* elements.

Theories for Coproducts[2]

Listing 6.1: Symbols and Rules (Coproducts.mmt)

```
theory Symbols =
  Coprod # 1⊕. . . prec −10000❙
  inl  # 1 ↪l 2 prec −5 ❙
  inr  # 2 r↩ 1 prec −5 ❙
  coprodmatch
    # 2 match V1 . 3|. . . to 4
      prec −9000❙
❚

theory FuncaddSymbol =
  Addfunc # 1⊕. . . prec 0❙
❚

theory SimpleRules =
   rule  rules ?CoprodTerm ❙
   rule  rules ?inlTerm ❙
   rule  rules ?inrTerm ❙
   rule  rules ?MatchTerm ❙
   rule  rules ?MatchComp ❙
   rule  rules ?MatchEquality ❙
   rule  rules ?CoprodSubType ❙
❚

theory TypedCoproduct
    :  ur:? TermsTypesKinds =
  include  ?Symbols ❙
  include  ?SimpleRules ❙
❚


theory FuncaddRules =
  rule  rules ?AddFuncComp ❙
❚

theory LFCoprod =
  include  ?FuncaddSymbol ❙
  include  ?FuncaddRules ❙
  include  ?TypedCoproduct ❙
  include  ur:? LF ❙
❚
```

Listing 6.2: Example Theory Testing Coproducts (Test.mmt)

```
theory coprodtest
    :  LFX/Coproducts?LFCoprod =
  A: type ❙
  B: type ❙
  a: A ❙
  b: B ❙
  C:type ❙
  t:  A⊕B ❙

  inla :  A⊕B ❘= (a ↪l B) ❙
  inlb :  A⊕B ❘= (A r↩ b) ❙

  // shouldfail  :  A⊕B ❘= (B r↩ a) ❙

  f: A → C ❙
  g: B → C ❙
  addfuncA: C ❘  = (f ⊕ g)  t ❙
  // shouldfail2   :  C ❘  = (f ⊕ g)  a ❙

  Pred :  C → type ❙
  p:  Pred (t  match x .
    f  x |  g x  to  C) ❙
  q:  Pred (t  match y .
    f  y |  g y  to  C) ❘
    = p ❙

  refleq :  {c} Pred c ❙
  claim2:  Pred ((a ↪l B) match x .
    f  x |  g x  to  C) ❘
    = refleq  (f  a) ❙
  typecheck:  (A ⊕B) → C ❘
    = (f ⊕ g) ❙
  claim:  Pred ((f  ⊕ g)  (a ↪l B)) ❘
    = refleq  (f  a) ❙
❚
```

when trying to formalize e.g. finite groups and functions between them, but they are also ubiquitous in e.g. object-oriented programming languages. Naturally, it is sufficient to have for each natural number n one type with n elements, since all finite types with the same number of elements are isomorphic. Therefore we will focus on this approach, since it conveniently introduces names for the members of finite types and spares us the trouble of having to label them.

Starting with the empty type $\varnothing$, we face the question how to formally declare that this type has no elements. Since we can always declare a new constant `null : ∅` we need to make sure that the existence of this constant leads to a contradiction. We can do so by allowing to construct a term of *any type* given an element of $\varnothing$, which (using Curry-Howard or judgments-as-types) corresponds to the principle *ex falso quodlibet*. There are three ways to go about this:

1. Declare a new primitive $\hookrightarrow_A e$ that given $e : \varnothing$ and $A : U$ has type A,
2. use function types and axiomatically declare $\hookrightarrow_A \cdot$ to have type $\varnothing \to A$, or
3. declare $\varnothing$ to be a subtype of any other type.

The obvious advantage of the second approach is that it allows using $\hookrightarrow_A \cdot$ like an actual typed function. The disadvantage is that using this approach requires function types to be present, hence reducing modularity, and the advantage is negligable.

The third approach has the advantage that it obviates the need for an elimination form $\hookrightarrow_{\cdot} \cdot$ completely – a postulated element $e : \varnothing$ already gives us an element of every other type, namely itself.

Given an additional singleton type $\textsc{Unit}$ with a single element $\star$, we can use the coproducts implemented in Section 6.1 to construct the remaining finite types. This way, we can use the elimination form for coproducts to pattern match an element of a finite type, obviating the need for a dedicated elimination form. There are two ways to do this:

1. We simply define $\mathbf{0} := \varnothing$, $\mathbf{1} := \textsc{Unit}$, $\mathbf{2} := \mathbf{1} \oplus \mathbf{1}$, $\mathbf{3} := \mathbf{1} \oplus \mathbf{2}$, etc. The rules for coproducts guarantee, that $\mathbf{2}$ has exactly two elements, namely $\underline{0} := \star \hookrightarrow_\ell \mathbf{1}$ and $\underline{1} := \mathbf{1}\,{}_r\!\hookleftarrow \star$. The elements of $\mathbf{3}$ then are $\underline{0} := \star \hookrightarrow_\ell \mathbf{2}$, $\underline{1} := \mathbf{1}\,{}_r\!\hookleftarrow (\star \hookrightarrow_\ell \mathbf{1})$ and $\underline{2} := \mathbf{1}\,{}_r\!\hookleftarrow (\mathbf{1}\,{}_r\!\hookleftarrow \star)$. Notably the terms $\underline{1} : \mathbf{2}$ and $\underline{1} : \mathbf{3}$ are distinct – $\mathbf{2}$ is not a subtype of $\mathbf{3}$.
2. If we define $\varnothing$ via subtyping, we can alternatively define $\mathbf{0} := \varnothing$ or $\mathbf{0} := \varnothing \oplus \varnothing$, $\mathbf{1} := \textsc{Unit} \oplus \mathbf{0}$, $\mathbf{2} := \textsc{Unit} \oplus \mathbf{1}$, $\mathbf{3} := \textsc{Unit} \oplus \mathbf{2}$, etc. This way we can use the subtyping rules for coproducts to guarantee that $\mathbf{i} <: \mathbf{j}$ for $i < j$:

$$\begin{array}{lll}
\underline{0} := \star \hookrightarrow_\ell \varnothing & : \mathbf{1} := \textsc{Unit} \oplus \mathbf{0} & <: \textsc{Unit} \oplus N \\
\underline{1} := \textsc{Unit}\,{}_r\!\hookleftarrow (\star \hookrightarrow_\ell \varnothing) & : \mathbf{2} := \textsc{Unit} \oplus \mathbf{1} & <: \textsc{Unit} \oplus (\textsc{Unit} \oplus N) \\
\underline{2} := \textsc{Unit}\,{}_r\!\hookleftarrow (\textsc{Unit}\,{}_r\!\hookleftarrow (\star \hookrightarrow_\ell \varnothing)) & : \mathbf{3} := \textsc{Unit} \oplus \mathbf{2} & <: \textsc{Unit} \oplus (\textsc{Unit} \oplus (\textsc{Unit} \oplus N))
\end{array}$$

   and $\underline{i}$ is the same expression at all numerical types. In particular, we can easily define the successor function as $\mathtt{succ}(n) := \textsc{Unit}\,{}_r\!\hookleftarrow n$.

Conveniently, defining $\varnothing$ to be a subtype of any other type allows for implementing both variants.

In the presence of function types, we can also define $\textsc{Unit}$ as $\varnothing \to \varnothing$, knowing that the only possible inhabitant of this type is the function $\star = \lambda_{x:\varnothing}.\ x = \lambda_{x:\varnothing}.\ \hookrightarrow_{\varnothing} x$. However, even using this definition, we still have to provide a rule for the solver to be able to exploit this knowledge (and to make sure that $\hookrightarrow_{\varnothing} x$ is the identity).

All of this leaves us with several options listed in Figure 6.3 and the basic rules in Figure 6.4.

	$\varnothing$	$\textsc{Unit}$	**higher finite types**	**elimination**
finite types	primitive $\hookrightarrow .\cdot$	primitive	primitive	dedicated case construct
finite types $+ \oplus$	primitive $\hookrightarrow .\cdot$	primitive	definable using $\oplus$	**match**-terms
finite types $+ \prod$	$\hookrightarrow_A \cdot : \varnothing \to A$	$\textsc{Unit} := \varnothing \to \varnothing$ $\star := \lambda_{x:\varnothing}.\ x$	primitive	dedicated case construct
finite types $+ \oplus + \prod$	$\hookrightarrow_A \cdot : \varnothing \to A$	$\textsc{Unit} := \varnothing \to \varnothing$ $\star := \lambda_{x:\varnothing}.\ x$	definable using $\oplus$	**match**-terms

(column $\varnothing$ can be replaced by a subtyping rule)

Figure 6.3: Possible Implementations for Finite Types

For any U with $\Gamma \vdash U$ `univ`:

Formation:

$$\frac{}{\Gamma \vdash \varnothing \Rightarrow \texttt{type}} \qquad \frac{}{\Gamma \vdash \textsc{Unit} \Rightarrow \texttt{type}}$$

Introduction:

$$\frac{}{\Gamma \vdash \star \Rightarrow \textsc{Unit}}$$

Elimination:

$$\frac{\Gamma \vdash e \Leftarrow \varnothing \quad \Gamma \vdash A \Rightarrow U}{\Gamma \vdash \hookrightarrow_A e \Rightarrow A}$$

Equality:

$$\frac{}{\Gamma \vdash a \equiv b : \textsc{Unit}}$$

Computation (Optional):

$$\frac{\Gamma \vdash A \Rightarrow U}{\Gamma \vdash \hookrightarrow_A e \rightsquigarrow e}$$

Subtyping (Optional):

$$\frac{\Gamma \vdash A \Rightarrow U}{\Gamma \vdash \varnothing <: A}$$

Figure 6.4: Basic Rules for Finite Types

Of course, constructing enumeration types and their elements by nesting $\oplus$-types and injections is simple to formally specify and implement, but inconvenient to use in practice. Furthermore, while it is possible to build up enumeration types using $\oplus$ even in the absence of coproducts by adapting their rules, the most general and user-friendly way to implement finite types is via a primitive type constructor on actual integers.

We can do so using a type NAT of number literals, which we discuss further in Section 6.2.1: We implement a type constructor $\mathsf{enum}(\,i\,)$ for the type $\mathbf{i}$, and analogously a constructor $\mathsf{case}_j(\,i\,)$ for $\underline{i} : \mathbf{j}$. In this case, it is desirable that $\mathsf{enum}(\,x\,)$ is a valid type even if x is a variable or otherwise undetermined and $\mathsf{case}_j(\,x\,)$ checks against type $\mathsf{enum}(\,j\,)$ iff $x < j$. If x and j are not definite number literals, this implies proving a judgment $\Gamma \vdash x < j$, which requires corresponding proof rules for the ordering on natural numbers and a way for the user (and the solver) to construct (and find) such proofs.

If we want to consider enumeration types as subtypes of each other, the argument j in $\mathsf{case}_j(\,i\,)$ becomes unnecessary, so we can write $\mathsf{case}(\,i\,)$ instead.

Remark 6.3:
In service of clarity, I will not introduce object-level judgments here, since they require corresponding proof rules which are not really relevant for this chapter. Suffice it to say that MMT offers a method Solver.prove(tm : Term), that returns true if and only if the Solver manages to find *any* element p of type tm. This method can cover premises of the form $\Gamma \vdash x < y$ for any choice of object-level judgments (and their proofs). Of course this immediately makes type checking undecidable, and the power of the rule system ultimately depends on the power of the internal prover, which in the case of MMT is rather weak.

Figure 6.5 gives a set of rules for these enum-types, where $\mathbf{n}$ represents the (or any variant of, in the case of subtyping) *defined* finite type corresponding to the (definite) number literal n, and $\underline{n}$ represents an element of such a type (again, for a definite number n). Which of the optional rules to implement depends on which of the implementation approaches discussed above we choose.

To summarize: Finite types give us various ways of implementing their different aspects, depending on which additional typing features are available in any context and how we want them to behave with respect to subtyping. Note however, that implementing all constructors as primitives still allows us to reuse the *symbols* already implemented for other typing features: A primitive elimination form for finite types can reuse the **match**-symbol from the `Symbols` theory for coproducts, and we can reuse the $\oplus$-symbol to construct higher finite types even in the absence of the rules governing coproducts. Even the rules governing these symbols can be reused with minimal adaptation (to restrict the types that are valid arguments to finite types).

This allows for developing library content with minimal dependencies, allowing for adding new features (e.g. coproducts or function types) as needed without reimplementing content that was developed without these features. The additional features only provide new definitions for the already implemented formalizations without changing their semantics significantly. Consequently, we will omit the rules for the constructs already covered in the previous sections in the basic rules listed in Figure 6.4.

6.2.1 Generic Literals in MMT

To implement enum and case, we need to be able to explicitly use numbers and have them be syntactically valid and semantically meaningful MMT terms – i.e. we want them to be *literals*. The MMT API offers a class RealizedType for *generic literals*, which combine a *syntactic type* (an inhabitable MMT term) with a SemanticType. A SemanticType represents an actual Scala class holding the values of our literals, as well as a lexer used to parse them in surface syntax.

Formation:

$$\frac{\Gamma \vdash i \Leftarrow \textsc{Nat}}{\Gamma \vdash \mathsf{enum}(\,i\,) \Rightarrow \mathsf{type}}$$

Introduction:

$$\frac{\Gamma \vdash i \Leftarrow \textsc{Nat} \quad \Gamma \vdash j \Leftarrow \textsc{Nat} \quad \Gamma \vdash i < j}{\Gamma \vdash \mathsf{case}_j(\,i\,) \Rightarrow \mathsf{enum}(\,j\,)} \qquad \left(\frac{\Gamma \vdash i \Leftarrow \textsc{Nat}}{\Gamma \vdash \mathsf{case}(\,i\,) \Rightarrow \mathsf{enum}(\,i+1\,)}\right)$$

Elimination:

(pattern match rule)

Type Checking (Optional):

$$\frac{\Gamma \vdash i < j}{\Gamma \vdash \mathsf{case}(\,i\,) \Leftarrow \mathsf{enum}(\,j\,)}$$

Equality: Computation (Optional):

$$\frac{}{\Gamma \vdash \mathsf{enum}(\,n\,) \rightsquigarrow \mathbf{n}} \qquad \frac{}{\Gamma \vdash \mathsf{case}(\,n\,) \rightsquigarrow \underline{n}}$$

Subtyping (Optional):

$$\frac{\Gamma \vdash i < j}{\Gamma \vdash \mathsf{enum}(\,i\,) <: \mathsf{enum}(\,j\,)}$$

Figure 6.5: Convenience Rules for Finite Types

Additionally, the MMT API already implements SemanticTypes for the most prevalent literals and their corresponding Scala implementations, namely BigInt (unrestricted integers), String, Double (floating point numbers), Boolean etc. In our specific case, we can use StandardNat extends SemanticType, which uses BigInt values restricted to positive numbers.

MMT's urtheories library [OMU] also already has a theory `NatSymbols` with a type `NAT` that we want to use as a syntactic type. To couple the two, we import a rule in an MMT theory, which we either implement in Scala (using the class RealizedType), or even more conveniently by import the *parametric* rule Realize with the name of the SemanticType and the syntactic type as parameters:

```
import rules  scala://lf.mmt.kwarc.info❚
import uom scala://uom.api.mmt.kwarc.info❚

theory NatLiteralsOnly  =
  include  ?NatSymbols❘
  rule  rules:?Realize  NAT uom:?StandardNat❘
❚
```

Hence, by importing the above theory we can use integer literals which will be assigned the principal type `NatSymbols?NAT`.

6.2.2 Implementation and Irrelevance Rules

One noteworthy aspect of the rules in Figure 6.4 is that the equality rule tells us that *all* terms of a specific type are equal – in other words, the type UNIT has (at most, i.e. in this case) exactly one element. While mathematically both of these statements are logically equivalent, they are intrinsically different from the point of view of an implementation.

In an implementation, to leverage the first statement we need two syntactically distinct terms that can then be shown to be equal. Hence this rule will only ever become relevant in presence of two distinct terms of the same type UNIT. The second statement however tells us that whenever we need *any* term of type UNIT, e.g. to solve a variable of type UNIT, there is exactly one element that we can use.

There is a dedicated rule class for these situations, namely TypeBasedSolutionRules. These not only extend TypeBasedEqualityRules, but also tell the system that *which* term of the given type is used is *irrelevant*, and more specifically, how a variable of this type should be solved. This is particularly relevant in the presence of judgments-as-types and proof arguments, where usually the specific term (i.e. proof) of a given judgment type J is irrelevant, as long as there is *any* such term (this is called **proof irrelevance**), in which case the solution rule can activate the prover.

Example 6.1:
The solution rule for the UNIT type is given in Listing 6.3.

Line 2 makes this rule applicable also to Pi-terms. This is to guarantee compatibility when defining UNIT as the type $\varnothing \to \varnothing = \prod_{\_ : \varnothing} \varnothing$.

Line 5 checks that tp is the UNIT type. The Unit.unapply-method also takes care of the case $\varnothing \to \varnothing$. If tp *is* the unit type, we return $\star$ as a solution.

Listing 6.3: The solution rule for UNIT [4]

```
1 object UnitIrrelevance extends TypeBasedSolutionRule(Nil,Unit.path) {
2   override def alternativeHeads: List[GlobalName] = List(Pi.path)
3   override def solve(solver: Solver)(tp: Term)
4       (implicit stack: Stack, history: History): Option[Term] = tp match {
5     case Unit(true) => Some(Unit.elemtm)
6     case _ => None
7   }
8 }
```

Listing 6.4 shows the MMT theories associated with all the above options for finite types. Note their modular development, offering theories that mix in basic LF, or coproducts, or both, with or without subtyping. Figure 6.6 shows the corresponding development graph with the symbol theories at the bottom.

Notably, finite types allow us to implement finite functions in an exploitably computational manner. As an example, Listing 6.5 shows a formalization of the finite group $\mathbb{Z}/2\mathbb{Z}$ and its group operation in such a manner, that the solver manages to compute $0 \circ 0 = 0$ from the definition of the operation. While the definition of op is syntactically ugly, one could imagine using a structural feature (see Section 6.3) specifically to introduce finite functions.

[4] https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/FiniteTypes/Rules.scala

[5] https://gl.mathhub.info/MMT/LFX/tree/master/source - note that the theory development used in the actual implementation is only a fragment of the more modular version described here.

Theories for Coproducts[5]

Listing 6.4: Symbols and Rules (FiniteTypes.mmt)

```
namespace http://mathhub.info/MMT/LFX/Finite ❚
import rules  scala : //FiniteTypes. LFX.mmt.kwarc.info❚

theory  Symbols =
  emptyType # ∅❙
  emptyFun # 1 ∅f 2 prec −900 ❙
  UNIT ❙
  unite  # ⋆❙
❚

theory  TypedRules =
  include  ?Symbols ❙
  include  ur:? Typed ❙
  include  ur:? Kinded ❙
  rule  rules ?EmptyTypeFormation ❙
  rule  rules ?UnitTypeFormation ❙
  rule  rules ?UnitIntro  ❙
  rule  rules ?EmptyElim ❙
  rule  rules ?UnitIrrelevance   ❙
❚

theory  SubtypedRules =
  include  ?TypedRules ❙
  rule  rules ?EmptySub ❙
  rule  rules ?EmptyFunComputation ❙
❚

theory  CoprodRules =
  include  ?TypedRules ❙
  include  ../ Coproducts?TypedCoproduct ❙
❚


theory  LFFiniteBase =
  include  ur:? LF ❙
  include  ?TypedRules ❙
  rule  rules ?UnitComputation ❙
  rule  rules ?UnitElemComputation ❙
  rule  rules ?EmptyFunComputation ❙
❚

theory  LFFiniteCoprod =
  include  ?LFFiniteBase ❙
  include  ?CoprodRules ❙
❚


theory  EnumSymbols =
  ENUM # ENUM 1 ❙
  CASE # CASE 1 ❙
❚

theory  EnumRules =
  include  ur:? NatLiteralsOnly  ❙
  include  ?EnumSymbols ❙
  rule  rules ?EnumFormation ❙
  rule  rules ?CaseIntroduction ❙
  rule  rules ?CaseTyping ❙
❚

theory  EnumSubtyped =
  include  ?EnumRules ❙
  include  ?SubtypedRules ❙
  rule  rules ?EnumSubtyping ❙
❚


theory  EnumCoprodRules =
  include  ?EnumRules ❙
  include  ?CoprodRules ❙
  rule  rules ?EnumComputation ❙
  rule  rules ?CaseComputation ❙
❚

theory  EnumLF =
  include  ?LFFiniteBase ❙
  include  ?EnumRules ❙
  include  ur:? Ded ❙
  include  ur:? NatRels ❙
❚

theory  EnumLFCoprod =
  include  ?EnumCoprodRules ❙
  include  ?EnumLF ❙
❚

theory  EnumLFSubtyped =
  include  ?EnumSubtyped ❙
  include  ?EnumLF ❙
❚

theory  EnumLFCoprodSubtyped =
  include  ?EnumLFCoprod ❙
  include  ?EnumLFSubtyped ❙
❚
```

Listing 6.5: Finite Functions Example (test.mmt)

```
theory  finitetest   :  ?EnumLFCoprodSubtyped =
          Zmod2Z = ENUM 2 ❙
          O :  Zmod2Z ❘= CASE 0 ❙
          op :  Zmod2Z → Zmod2Z → Zmod2Z ❘
           = [a, b]  a  match x.
              b |  (b  match y.
                  a |  O | y
              to  Zmod2Z) | x
            to  Zmod2Z ❘# 1 ∘ 2 ❙

           P2 :  Zmod2Z → type ❙

           claim  :  P2 O ❙
           claim2  :  P2 (O ∘ O) ❘  = claim ❙
❚
```

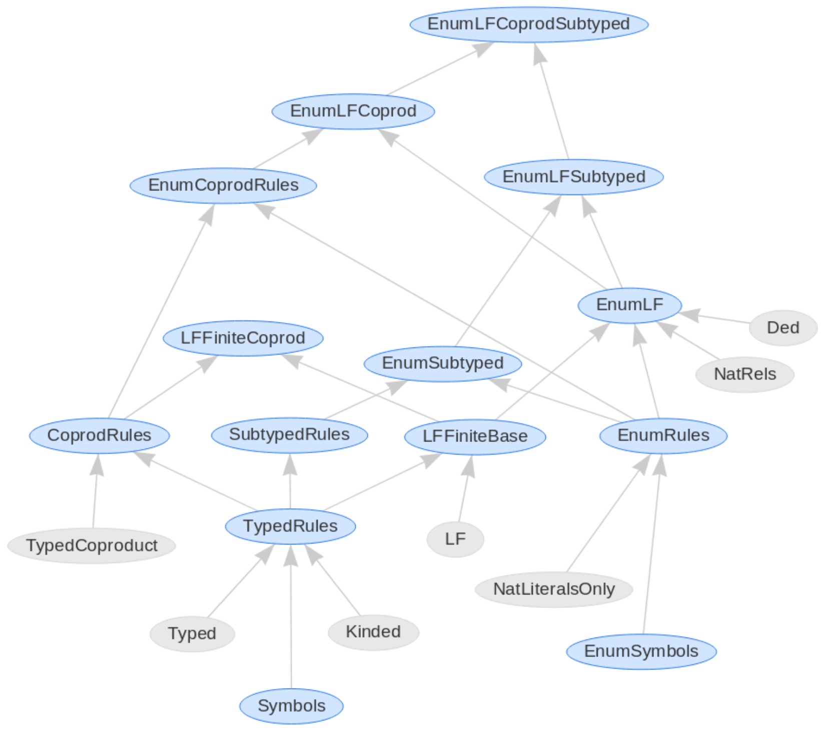


Figure 6.6: A Modular Development Graph for Finite Types

6.3 W-Types

W-Types were introduced in [ML94] to represent wellorderings, but were shown to be able to represent many other inductively defined sets in Martin-Löf style type theories as well (see [Dyb97]). The central idea is to define an inductive type from the number of constructors n of the type and their arities a_n with respect to the type that is being defined; the introduction and elimination forms then assure that the resulting type represents an initial algebra.

Ignoring the category theory involved, how to represent inductive types as W-types is best explained by example:

Example 6.2:

- Natural numbers have a constant constructor $0 : \mathbb{N}$ (of arity 0) and a unary constructor $S : \mathbb{N} \to \mathbb{N}$ (of arity 1). We can encode the number of constructors as the type $\mathbf{2}$ and the arities as the types $\mathbf{0}$ and $\mathbf{1}$.

  Their W-type would thus be $\mathbb{N} := \mathsf{W}_{x:\mathbf{2}}\, \mathbf{match}_x \{ y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1} \}$.

- Lists over a type A have a constant constructor $\mathsf{Nil}_A : \mathsf{List}_A$, and *for each* $a : A$ a unary constructor $\mathsf{append}_a : \mathsf{List}_A \to \mathsf{List}_\mathsf{A}$. We encode the number of constructors as the type $\textsc{Unit} \oplus A$ (i.e. each case append_a is considered a single unary constructor).

  Their W-type would thus be $\mathsf{List}_A := \mathsf{W}_{x:\textsc{Unit}\oplus A}\, \mathbf{match}_x \{ y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1} \}$.

- A binary tree over a type A is either a leaf $a : A$ (constant constructor), or a node consisting of an element $a : A$ and two subtrees (one constructor of arity 2 for every $a : A$).

Their W-type would thus be $\mathsf{Tree}_A := \mathsf{W}_{x:A\oplus A}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{2}\,\}$.

As the examples show, constructors that depend on an element of another type A are considered separate constructors for each $a : A$.

Remark 6.4:
Since the number and arities of constructors in a W-type are almost always finite but need encoding as types, W-types (at least as described here) are almost useless without finite types. Moreover, dependent inductive types such as lists over a type A additionally require coproducts.

However, note that our W-types do not strictly require coproducts or finite types a priori – they are merely a lot less useful without these features.

As introduction form for a W-type $W := \mathsf{W}_{x:A}\, B(x)$, we have a supremum-operator $\mathrm{sup}_c\{\, x \Longrightarrow a(x)\,\}$, where $c : A$ tells us the constructor case, $x : B(c)$ is a variable that represents the inductive parameters needed, and $a(x) : W$ represents the arguments for the constructor. How this works is again best understood by example:

Example 6.3:

- For natural numbers $\mathbb{N} := \mathsf{W}_{x:\mathbf{2}}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1}\,\}$ we get:

$$0 := \mathrm{sup}_{\underline{0}}\{\, x \Longrightarrow \hookrightarrow_{\mathbb{N}} x\,\} \qquad \mathsf{succ}(n) := \mathrm{sup}_{\underline{1}}\{\, x \Longrightarrow n\,\}$$

- For lists $\mathsf{List}_A := \mathsf{W}_{x:\textsc{Unit}\oplus A}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1}\,\}$ we get:

$$\mathsf{Nil}_A := \mathrm{sup}_{\star \hookrightarrow_\ell A}\{\, x \Longrightarrow \hookrightarrow_{\mathsf{List}_A} x\,\} \qquad \mathsf{append}_a(\ell) := \mathrm{sup}_{\textsc{Unit}_r \hookleftarrow a}\{\, x \Longrightarrow \ell\,\}$$

- For trees $\mathsf{Tree}_A := \mathsf{W}_{x:A\oplus A}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{2}\,\}$ we get:

$$\mathsf{Leaf}(a) := \mathrm{sup}_{a \hookrightarrow_\ell A}\{\, x \Longrightarrow \hookrightarrow_{\mathsf{Tree}_A} x\,\}$$

$$\mathsf{Node}(a, s, t) := \mathrm{sup}_{A_r \hookleftarrow a}\{\, x \Longrightarrow \mathbf{match}_x\{\, y \Longrightarrow_{\mathsf{Tree}_A} s \mid t\,\}\,\}$$

We can think of the argument $a(x)$ in $\mathrm{sup}_c\{\, x \Longrightarrow a(x)\,\}$ for a type $W := \mathsf{W}_{x:A}\, B(x)$ as a function that assigns each parameter position of type W of a constructor (such as the two subtrees in the last example) to the corresponding argument. In the constant cases, x has type $\varnothing$ and we need to construct an element of W from it, hence in these cases the argument is always $\hookrightarrow_W x$.

Remark 6.5:
Note that the body of a sup-expression is basically a λ-expression. Indeed, given function types we can instead write sup simply as $\mathrm{sup}_c(f)$ for a function $f : B(c) \to \mathsf{W}_{x:A}\, B(x)$.

Elimination of W-Types is cumbersome and technical, so it is worth going into some detail. To eliminate an element of a type $W := \mathsf{W}_{c:A}\, B$, we want to define a function f out of W with target type $C(w)$ (for $w \in W$) via well-founded recursion. For every constructor case $c : A$, we have an arity of $B(c)$, so we would have to define f on c and the $B(c)$ many (recursive) arguments of type W. The latter can be encoded (or thought of) as a function $g_c : B(c) \to W$. Additionally, we need to be able to use the recursive values of the very function f we are defining *on* the recursive arguments encoded in g_c. These we can similarly think of encoded as a function $h_{c,g_c} : \prod_{b:B(c)} C(g_c(b))$

Hence, to be able to define f we need an element $c : A$, a way to obtain $g_c(y) : W$ for any $y : B(c)$ for the recursive arguments, and a way to obtain $h(b) : C(g_c(b))$ for any $b : B(c)$. The easiest way to do so is to have the elimination constructor bind variables for these, some of them representing (dependent) functions. This requires function types, of course. In the absence of function types, the rules governing them can be added without making the symbols associated (lambda-abstraction, function type constructors etc.) available to the user. Alternatively, one could implement dedicated syntactic constructs that basically mirror the application of functions in the particular context of recursive definitions on W-types, which is even more cumbersome, but theoretically straight-forward. I will hence for simplicities sake assume function types to be present in the remainder of this Section.

Consequently, we introduce an elimination operator $\mathsf{rec}(\,w\,)\{(\,c,g,h\,)\Longrightarrow_{C(w)} e(c,g,h)\,\}$, with the defining equation:

$$\begin{aligned}&\mathsf{rec}(\,\mathrm{sup}_d\{\,x\Longrightarrow a(x)\,\}\,)\{(\,c,g,h\,)\Longrightarrow_{C(\mathrm{sup}_d\{\,x\Longrightarrow a(x)\,\})} e(c,g,h)\,\}\\ &= e\left(d, \lambda_{x:B(d)}.\, a(x), \lambda_{y:B(d)}.\, \mathsf{rec}(\,a(y)\,)\{(\,c,g,h\,)\Longrightarrow_{C(\mathrm{sup}_d\{\,x\Longrightarrow a(x)\,\})} e(c,g,h)\,\}\right),\end{aligned}$$

where e corresponds to the inductive definition of our function, c to the constructor case of the input, g to the parameters of the constructor, and h to the recursive calls on these parameters.

Example 6.4:

- For natural numbers $\mathbb{N} := \mathsf{W}_{x:\mathbf{2}}\, \mathbf{match}_x\{\,y\Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1}\,\}$, let us consider the factorial function defined via induction, i.e. we need to encode the two cases $0! := 1$ and $\mathsf{succ}(n)! := \mathsf{succ}(n) \cdot n!$, where we obtain n from the bound variable g and $n!$ from the variable h (assuming we have a multiplication already). In the case where $n = \mathsf{succ}(m)$, we have $c = \underline{1}$ and $B(c) = \mathbf{1}$, hence we can only apply g and h to $\star$ and obtain $g(\star) = m$ and $h(\star) = m!$. So:

  $$n! := \mathsf{rec}(\,n\,)\{(\,c,g,h\,)\Longrightarrow_{\mathbb{N}} \mathbf{match}_c\{\,x\Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}(g(\star)) \cdot h(\star)\,\}\,\}$$

  By the above definition, we have

  $$2 = \mathrm{sup}_{\underline{1}}\{\,x\Longrightarrow 1\,\} = \mathrm{sup}_{\underline{1}}\{\,x\Longrightarrow \mathrm{sup}_{\underline{1}}\{\,x\Longrightarrow 0\,\}\,\} = \mathrm{sup}_{\underline{1}}\{\,x\Longrightarrow \mathrm{sup}_{\underline{1}}\{\,x\Longrightarrow \mathrm{sup}_{\underline{0}}\{\,x\Longrightarrow \hookrightarrow_{\mathbb{N}} x\,\}\,\}\,\}$$

  hence using the defining equation for rec, we get

$$\begin{aligned}
2! &= \left(\mathrm{sup}_{\underline{1}}\{\, x \Longrightarrow 1 \,\}\right)! \\
&= \mathsf{rec}\left(\mathrm{sup}_{\underline{1}}\{\, x \Longrightarrow 1 \,\}\right)\{(\ c, g, h\ ) \Longrightarrow_{\mathbb{N}} \mathbf{match}_c\{\, x \Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}(g(\star)) \cdot h(\star) \,\}\,\} \\
&= \mathbf{match}_{\underline{1}}\{\, x \Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}((\lambda_{y:\mathbf{1}}.\ 1)(\star)) \cdot (\lambda_{y:\mathbf{1}}.\ 1!)(\star) \,\} \\
&= \mathsf{succ}(1) \cdot 1! \\
&= \mathsf{succ}(1) \cdot \mathsf{rec}(\,1\,)\{(\ c, g, h\ ) \Longrightarrow_{\mathbb{N}} \mathbf{match}_c\{\, x \Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}(g(\star)) \cdot h(\star) \,\}\,\} \\
&= \mathsf{succ}(1) \cdot \mathsf{rec}\left(\mathrm{sup}_{\underline{1}}\{\, x \Longrightarrow 0 \,\}\right)\{(\ c, g, h\ ) \Longrightarrow_{\mathbb{N}} \mathbf{match}_c\{\, x \Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}(g(\star)) \cdot h(\star) \,\}\,\} \\
&= \mathsf{succ}(1) \cdot \mathbf{match}_{\underline{1}}\{\, x \Longrightarrow_{\mathbb{N}} 1 \mid \mathsf{succ}((\lambda_{y:\mathbf{1}}.\ 0)(\star)) \cdot (\lambda_{y:\mathbf{1}}.\ 0!)(\star) \,\} \\
&= \mathsf{succ}(1) \cdot \mathsf{succ}(0) \cdot 0!
\end{aligned}$$

- For lists $\mathsf{List}_A := \mathsf{W}_{x:\textsc{Unit}\oplus A}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1} \,\}$, let us consider concatenation of lists $\ell_1 ::: \ell$ for a fixed list ℓ. In the case where $\ell_1 = \mathsf{Nil}_A$, we return ℓ, in the case $\ell_1 = \mathsf{append}_a(r)$, the a is encoded in $c = \textsc{Unit}_r \hookleftarrow a$ and we return $\mathsf{append}_a(r ::: \ell)$. Hence:

$$\ell_1 ::: \ell := \mathsf{rec}(\,\ell_1\,)\{(\ c, g, h\ ) \Longrightarrow_{\mathsf{List}_A} \mathbf{match}_c\{\, x \Longrightarrow_{\mathsf{List}_A} \ell \mid \mathsf{append}_x(h(\star)) \,\}\,\}$$

- For trees $\mathsf{Tree}_A := \mathsf{W}_{x:A\oplus A}\, \mathbf{match}_x\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{2} \,\}$, we can define the topological order: $\mathcal{O}(\mathsf{Leaf}(a)) := \mathsf{append}_a(\mathsf{Nil}_A)$ and $\mathcal{O}(\mathsf{Node}(a, s, t)) := \mathsf{append}_a(\mathcal{O}(s) ::: \mathcal{O}(t))$. Hence:

$$\mathcal{O}(t) := \mathsf{rec}(\,t\,)\{(\ c, g, h\ ) \Longrightarrow_{\mathsf{List}_A} \mathbf{match}_c\{\, x \Longrightarrow_{\mathsf{List}_A} \mathsf{append}_x(\mathsf{Nil}_A) \mid \mathsf{append}_x(h(\underline{0}) ::: h(\underline{1})) \,\}\,\}$$

The resulting grammar is given in Figure 6.7, the rules are given in Figure 6.8. Again there are several things to note:

- W-types do not correspond to a logical connective or quantifier under the Curry-Howard correspondence; however, the rec-elimination (using either Curry-Howard or judgments-as-types) enables proofs by well-founded induction on W-types.

- In an implementation, sup needs to carry the governing W-type in order for the introduction rule to be able to infer the type of a W-expression. Since it is usually clear which the intended type is, we will continue to omit it outside of the rules.

- The computation rule's premises can again be omitted if the expression is already known to be well-typed.

$$\begin{array}{lll|l}
\Gamma & ::= & \cdot \mid \Gamma, x[: T][:= T] & \text{contexts} \\
T & ::= & x \mid \mathtt{type} \mid \mathtt{kind} & \text{variables and universes} \\
 & \mid & \mathsf{W}_{x:T}\, T \mid \mathrm{sup}_T\{\, x \Longrightarrow_T T \,\} \mid \mathsf{rec}(\,T\,)\{(\ x, x, x\ ) \Longrightarrow_T T \,\} & \mathsf{W}\text{-types}
\end{array}$$

Figure 6.7: Grammar for Coproduct-Types

For any U with $\Gamma \vdash U$ `univ`:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma, x : A \vdash B \Rightarrow U'}{\Gamma \vdash \mathsf{W}_{x:A} B \Rightarrow \max\{\, U, U' \,\}}$$

Introduction:

$$\frac{\Gamma \vdash c \Leftarrow A \quad \Gamma, x : B[\, x/_{c} \,] \vdash a \Leftarrow \mathsf{W}_{x:A} B}{\Gamma \vdash \mathrm{sup}_{c}\left\{\, x \Longrightarrow_{\mathsf{W}_{x:A}\, B} a \,\right\} \Rightarrow \mathsf{W}_{x:A} B}$$

Elimination:

For $C' := C\left[\, c/_{\mathrm{sup}_{c}\{\, x \Longrightarrow_{\mathsf{W}_{x:A}\, B}\, g(x) \,\}} \,\right]$:

$$\frac{\Gamma \vdash w \overset{\Rightarrow}{=} \mathsf{W}_{x:A} B \quad \Gamma, c : \mathsf{W}_{x:A} B \vdash C \Rightarrow U \quad \Gamma, c : A, g : B[\, x/_{c} \,] \to \mathsf{W}_{x:A} B, h : \prod_{y:B[\, x/_{c} \,]} C[\, c/_{g(y)} \,] \vdash e \Leftarrow C'}{\Gamma \vdash \mathsf{rec}(\, w \,)\{(\, c, g, h \,) \Longrightarrow_{C} e \,\} \Rightarrow C[\, c/_{w} \,]}$$

Computation:

For $d := \mathrm{sup}_{c'}\{\, x \Longrightarrow_{W} a \,\}$, $g' := \lambda_{x:B[\, x/_{c'} \,]}.\, a$, and $h' := \lambda_{y:B[\, x/_{c'} \,]}.\, \mathsf{rec}(\, a[\, x/_{y} \,] \,)\{(\, c, g, h \,) \Longrightarrow_{C[\, c/_{d} \,]} e \,\}$:

$$\frac{\Gamma \vdash d \Rightarrow W' \quad \Gamma, w : W' \vdash \mathsf{rec}(\, w \,)\{(\, c, g, h \,) \Longrightarrow_{C} e \,\} \Rightarrow C[\, c/_{w} \,]}{\Gamma \vdash \mathsf{rec}(\, d \,)\{(\, c, g, h \,) \Longrightarrow_{C} e \,\} \rightsquigarrow e[\, c/_{c'} \,][\, g/_{g'} \,][\, h/_{h'} \,]}$$

Figure 6.8: Rules for W-Types

Remark 6.6: Equality and Subtyping

There are several subtyping behaviors imaginable for W-types:

- For inductive types with *named* constructors, one would usually want the names to matter, such that e.g. the type of natural numbers and the type of lists over the UNIT-type are distinct. Since both are represented as the type $\mathsf{W}_{x:\mathbf{2}}\, \mathbf{match}_{x}\{\, y \Longrightarrow_{\mathtt{type}} \mathbf{0} \mid \mathbf{1} \,\}$, this is impossible by virtue of W-types being anonymous.

  The intended behavior could be recovered by having W-types carry an additional label for each constructor.

- One might want any distinct W-types to be unrelated. This is the behavior in the absence of any additional rules.

- One might want an inductive type A to be equal to a type B iff they have the same number of constructors with the same arities *irrespective of the order* – e.g. $\mathsf{W}_{x:A \oplus B}\, \mathbf{match}_{x}\{\, y \Longrightarrow_{\mathtt{type}} C_1 \mid C_2 \,\}$ is the same as $\mathsf{W}_{x:B \oplus A}\, \mathbf{match}_{x}\{\, y \Longrightarrow_{\mathtt{type}} C_2 \mid C_1 \,\}$. This makes sense intuitively and would be easily representable as an equality rule, but every call of the rule would be computationally expensive, since it would have to check equality on any possible permutation of the constructors and their arities as a premise.

- Finally, one might want an inductive type W_1 to be a subtype of W_2 iff all constructor cases of W_1 are also constructors of W_2 – i.e. if there is an embedding from W_1 to W_2.

This can be ill-defined, if W_2 has more than one constructor case with the same base type and arity (i.e. there is no *unique* embedding). Consider e.g. $W_2 = \mathsf{W}_{x:A\oplus A}\,\mathbf{0}$ and $W_1 = \mathsf{W}_{x:A}\,\mathbf{0}$ – in this situation it is unclear which of the two (constant) constructors of W_2 the element of W_1 should correspond to.

This behavior can again be achieved by having W-types carry labels for the constructors, in which case the ambiguity can be resolved by demanding that the labels match for intended subtypes.

6.3.1 Implementation

From an implementation perspective, W-types are conveniently simple and fall neatly into our pattern of rules. Consequently, implementing them in MMT is straight-forward; the corresponding MMT-theory is presented in Listing 6.6.

Listing 6.6: Theories for W-types[6]

```
theory Symbols =
  wtype # W V1T . 2 prec −10000 ❙
  sup # sup 2 , V1 ⟹ 3 to 4 prec −8000 ❙
  rec # rec 4 , V1 , V2 , V3 ⟹ 5 to 6 prec −8000 ❙
❚

theory Rules =
   rule rules ?WTypeFormation ❙
   rule rules ?SupIntroduction ❙
   rule rules ?WEquality ❙
   rule rules ?RecElimination ❙
   rule rules ?RecComputation ❙
❚
```

From a user perspective however, W-types are incredibly inconvenient and unintuitive, counter to our goal of being as close to mathematical practice as possible. Listing 6.7 shows as an example the natural numbers with inductively defined addition on them.

Listing 6.7: Natural Numbers on W-types[7]

```
theory Wtest : LFX/WTypes?LFW =

  beta : ENUM 2 → type ❘
    = [x] x match y. ∅| UNIT | ∅ to type ❙
  ℕ : type ❘ = W x:ENUM 2 . (beta x) ❙

  zeroN : ℕ       ❘ = sup (CASE 0) , x ⟹ (x ∅f ℕ) to ℕ ❙
  S : ℕ → ℕ       ❘ = [x : ℕ] sup (CASE 1) , y ⟹ x to ℕ ❙

  plusN : {n:ℕ,m:ℕ} ℕ❘ # 1 + 2 prec 5 ❘
    = [n][ m] (rec m,c,g, h ⟹ (c match (ENUM 2), x. n | S (h x) | x to ℕ) to ℕ) ❙

  // test : 0+0=0 ❙
  Eq : ℕ → type ❙
  eq : Eq (zeroN) ❙

  claim : Eq (plusN zeroN zeroN) ❘ = eq❙
❚
```

[6]https://gl.mathhub.info/MMT/LFX/blob/master/source/WTypes.mmt

[7]https://gl.mathhub.info/MMT/LFX/blob/master/source/test.mmt

6.3.2 Structural Features

We can remedy this inconvenience by providing a **structural feature** that offers a more convenient and intuitive syntax and *elaborates* into W-types, giving users a way to specify inductive types without having to use the cumbersome syntax offered by W-types – or having to know about them at all. The goal is to formalize natural numbers more akin to this:

```
induct Nats =
  Nat  : type    | # ℕ   |
  Zero : ℕ       | # O   |
  Succ : ℕ → ℕ   | # S 1 |
|
```

A structural feature is an extension of the class StructuralFeature. Firstly, it instructs the parser how to parse the *header* of an instance of this feature (e.g. `induct [name] =`). For this, the structural feature has to provide a notation. The header is followed by an optional body of a theory (e.g. containing the declarations `Nat`, `Zero` and `Succ`). The parser wraps any application of the header notation into an object of class DerivedDeclaration, holding the components provided in the header and the optional body.

Secondly, after parsing, this DerivedDeclaration is passed on to the StructuralFeature, which returns an Elaboration, holding a list of MMT declarations computed from the DerivedDeclaration. The details on implementing derived declarations and structural features are sketched in [Ian17] and thus omitted here.

For inductive types, we want two structural features: One that elaborates into a W-Type and its constructors, and one for the elimination form.[8] The keywords for these features are set as induct and def respectively, which allows us to implement natural numbers and addition in surface syntax as in Listing 6.8.

Listing 6.8: W-Types with Structural Features[9]

```
induct Nats () | =
  Nat : type | # ℕ |
  Zero : ℕ | # O |
  Succ : ℕ → ℕ | # S 1 |
|

Ntest : ℕ | = O |
Ntest2 : ℕ | = S (S Ntest) |

def addition (n : ℕ) | =
  add : ℕ → ℕ | # 1 + 2 |
  Zero = n |
  Succ = [m] S (add m) |
|
P : ℕ → type |
pzero : P O |
Ptwo : P (S (S O)) |
Ntest3 : ℕ | = Ntest + Ntest2 |
claim : P (O + O) | = pzero |
claim2 : P((S O) + (S O)) | = Ptwo |
```

[8]The full implementation of these is omitted here and can be found at https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/WTypes/Features.scala

[9]Ibid.

All declarations in an `induct` or `def`-environment are bundled into the body of a theory, which is itself wrapped into a DerivedDeclaration *d*. This declaration is then passed on to the StructuralFeature's elaborate-method, which in the case of `induct` computes from them the appropriate W-type (e.g. for ℕ) and the constructors (e.g. `Zero` and `Succ`), and in the case of `def` the corresponding function defined by a rec-expression. We can pass additional parameters to the structural feature in the header of an `induct` or `def` environment, to allow for e.g. polymorphic W-Types or inductive functions with arity > 1 (as in `addition`), as shown with lists and binary trees in Listing 6.9.

Listing 6.9: Lists and Trees [10]

```
induct Lists (B:type) | =
  List : type | # List 1 |
  Nil : List | # Nil 1 |
  Cons : B → List → List | # 2 ~> 3|
|

def Concatenation (B : type, ls : List B) | =
  conc : List B → List B | # 2 ++ 3|
  Nil = ls |
  Cons = [b:B,l: List B] b ~> (conc l) |
|

induct Trees (B : type) | =
  Tree : type | # Tr 1 |
  Leaf : B → Tree | # Lf 2|
  Node : B → Tree → Tree → Tree |# Nd 2 3 4 |
|

def topoSort (B : type) | =
  sort : Tree B → List B | # topsort 2|
  Leaf = [b] b ~> (Nil B) |
  Node = [b,s,t] b ~> ((sort s) ++ (sort t)) |
|

A : type |
a1 : A |
a2 : A |
a3 : A |
test1 : Tr A | = Lf a1 |
test2 : Tr A | = Nd a2 test1 test1 |
test3 : List A | = topsort test2 |
```

Remark 6.7: Structural Features for Induction
While we use W-types here as a target for elaborating our structural features, we are in no way required to do so. Colin Rothgang recently developed a similar feature (as well as many other structural features, e.g. for equivalence relations and quotients)[a], which elaborates into undefined constants for the inductive principles instead. While this has the disadvantage that the solver can not natively exploit the generated induction principles automatically, it has the advantage that the resulting elaboration works with plain LF and can represent more advanced induction principles such as mutually recursive types, which can not be represented using W-types alone.

Again, we are able to marry the two approaches quite easily while reusing the same structural feature and hence syntactic representation. For example, the feature rule can check for the presence of the W-type rules in the current context and decide accordingly

[10] Ibid.

whether to elaborate into W-types or primitive LF constants.

[a]https://github.com/UniFormal/MMT/tree/master/src/mmt-lf/src/info/kwarc/mmt/lf/structuralfeatures

6.4 Cumulative Universe Hierarchies

Plain LF offers two universes `type` and `kind`. With respect to their ontological status, they can be thought of as analogous to the distinction between *sets* and *proper classes*. However, often the two universes are somewhat restrictive – `PLF` is already an extension of this two-tiered hierarchy, but even the shallow polymorphism enabled by it is often not enough, e.g. when wanting to quantify over all *groups* (each of which has a base type), or sets (for some correspondance between sets and types), or when formalizing categories while trying to avoid a deep embedding.[11]

In those situations, it can be more convenient to have infinitely many universes $\mathcal{U}_i$ for each $i \in \mathbb{N}$.[12] By declaring $\mathcal{U}_i \Leftarrow \mathcal{U}_j$ and $\mathcal{U}_i <: \mathcal{U}_j$ whenever $i < j$, we get an infinite cumulative hierarchy of universes analogous (in a set-theoretic model) to higher levels of the Von-Neumann hierarchy indexed by large cardinals.

This is the approach taken by homotopy type theory [Uni13], Mizar [Miz] via Grothendieck universes, and most notably Coq [Tea03], where the precise universe in which a declaration lives is unspecified by the user and computed by the system by solving appropriate constraints on the full current context (*floating universes*).

The basic rules are straight-forward and given in Figure 6.9, and as with enumeration types (Section 6.2), we use number literals NAT and postulate some (potentially judgments-as-types based) judgment $i < j$.

Universe Rule:

$$\frac{\Gamma \vdash i \Leftarrow \text{NAT}}{\Gamma \vdash \mathcal{U}_i \ \texttt{univ}}$$

Formation:

$$\frac{\Gamma \vdash i \Leftarrow \text{NAT}}{\Gamma \vdash \mathcal{U}_i \Rightarrow \mathcal{U}_{i+1}}$$

Subtyping (Optional):

$$\frac{\Gamma \vdash i < j}{\Gamma \vdash \mathcal{U}_i <: \mathcal{U}_j}$$

Figure 6.9: Rules for Universes

To be "backwards compatible" with everything implemented in LF alone, we can redefine `type` $:= \mathcal{U}_0$ and `kind` $:= \mathcal{U}_1$. Even though `type`$<:$`kind` does not hold in LF, this additional

[11]These kinds of formalizations are enabled by using e.g. record types with `type`-valued fields, see Chapter 8

[12]According to folklore, all mathematics can be done with at most 7th-order universes, but this number is seemingly arbitrary and in practice it is just as easy to drop the restriction to a fixed finite number of universes entirely.

subtyping judgment does not impact plain LF content.[13] However, in order for $\prod$-types to be usable for types living in higher universes, we need to slightly modify the formation rule for $\prod$:

$$\frac{\Gamma \vdash A \Leftarrow U \quad \Gamma, x : A \vdash B \Rightarrow U'}{\Gamma \vdash \prod_{x:A} B \Rightarrow \max\{U, U'\}}$$

Note, that for $U = \texttt{type}$, this subsumes the original rule. Since we can declare our new formation rule to shadow the old one, this does not break modularity. In addition, we augment Definition 4.1 by new cases:

Definition 6.2:

- For $A = \mathcal{U}_i$ and $B : \mathcal{U}_j$ we let $\max\{A, B\} = \mathcal{U}_{\max(i,j)}$
- For $A, B : \mathcal{U}_i$, we let

$$\max\{A, B\} := \begin{cases} A & \text{if } B <: A, \\ B & \text{if } A <: B, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Remark 6.8: Polymorphism

One interesting aspect to consider is which universe $\textsc{Nat}$ should live in. In Section 6.2.1, we determined $\textsc{Nat}$ to be a `type`, hence $\textsc{Nat} \Leftarrow \mathcal{U}_0$. Additionally, our formation rule for $\prod$ was:

$$\frac{\Gamma \vdash A \Leftarrow U \quad \Gamma, x : A \vdash B \Rightarrow U'}{\Gamma \vdash \prod_{x:A} B \Rightarrow \max\{U, U'\}}$$

Consider a type $T = \prod_{n:\textsc{Nat}} \prod_{U:\mathcal{U}_n} B$. Is this type valid?

The answer is no, since the universe that T lives in would be $\max\{\mathcal{U}_0, \mathcal{U}_n\}$, which can not be computed since $\mathcal{U}_n$ depends on a free variable. Hence our rules forbid quantification over all universes.

However, similar as in PLF (see Remark 4.6), we might want to allow *shallow polymorphism* – i.e. the ability to quantify over all universes on the *outside* of a $\prod$-type only. By declaring such a type to be inhabitable but untyped, we avoid paradoxes resulting from unrestricted quantification, but still allow for implementing polymorphic functions. The corresponding rule would hence be:

$$\frac{\Gamma, n : \textsc{Nat}, u : \mathcal{U}_n \vdash B \ \texttt{inh}}{\Gamma \vdash \prod_{n:\textsc{Nat}} \prod_{u:\mathcal{U}_n} B \ \texttt{inh}}$$

6.4.1 Implementation and Universe Rules

All the rules are again straight-forward, but here we encounter a *universe rule* for the first time. As with the other rules, we need to implement an apply method for them, the signature

[13] To be precise: Any judgment that holds in LF still holds in this new logical framework, but not the other way around.

of which is rather simple:

```
def apply(solver: Solver)(tm: Term)(implicit stack: Stack, history: History) : Boolean
```

and returns `true` if the term `tm` is declared to be a universe by this rule.

Example 6.5:
The universe rule for $\mathcal{U}$ is given in Listing 6.10.

Line 2 makes this rule applicable also to `type` and `kind`– the helper object TypeLevel's unapply-method takes care of the conversions `type` $\to \mathcal{U}_0$ and `kind` $\to \mathcal{U}_1$ for us.

Line 5 merely checks that the argument passed on to $\mathcal{U}$ is in fact of type NAT. Since that check returns a boolean, it can serve as the return value.

Listing 6.10: The Universe Rule for $\mathcal{U}$[14]

```
1  object TypeLevelUniverse extends UniverseRule(TypeLevel.path) {
2    override def alternativeHeads: List[GlobalName] = List(TypeLevel.path,Typed.ktype,Typed.kind)
3    def apply(solver: Solver)(tm: Term)(implicit stack: Stack, history: History) : Boolean = tm match {
4      case TypeLevel(i) =>
5        solver.check(Typing(stack,i,NatRules.NatLit.synType))
6      case _ => false
7    }
8  }
```

The corresponding theories for the symbols and rules are given in Listing 6.11. Notably, the theory TypedHierarchy which is to serve as a "standalone" theory of a cumulative type hierarchy without any typing features, needs to import several fundamental MMT theories that validate notions like *typing*, notations, module expressions etc. in the first place, as well as the theories containing the symbols for `type` and `kind` for compatibility with plain LF theories. Listing 6.12 shows a small example theory using a cumulative hierarchy. Additionally, universes are used all over the Math-in-the-Middle archive[15], notably in the algebraic theories and in categories.

Listing 6.11: Symbols and Rules[16]

```
theory Symbols =
   TypeLevel # 𝒰1 prec −1000 ❙
❚

theory Rules =
  rule rules ?TypeLevelUniverse ❙
  rule rules ?TypeLevelSubRule ❙
  rule rules ?LevelType ❙
❚

theory TypedHierarchy =
   include meta:?Errors❙
   include http://cds.omdoc.org/urtheories?ModExp❙
   include meta:?mmt❙
   include ur:?Typed ❙
   include ur:?Kinded ❙
   include ur:?NatLiteralsOnly ❙
   include ?Symbols ❙
   include ?Rules ❙
```

[14] https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/TypedHierarchy/Rules.scala

[15] https://gl.mathhub.info/MitM/smglom/tree/master/source

```
theory LFHierarchy =
   include ?TypedHierarchy |
   include ur:?LF |

   rule rules ?Polymorphism|
```

Listing 6.12: A Simple Theory for Universes[17]

```
theory HierarchyTest : LFX/TypedHierarchy?LFHierarchy =
        c : type |
        d : 𝓤1 | = c |
        f : 𝓤2 → 𝓤2 |
        test : 𝓤2 | = f c |
```

6.5 A Logical Framework Based on Homotopy Type Theory

Homotopy Type Theory [Uni13] (HoTT) is a foundation of mathematics built upon a type system that we can now define as *LF+Cumulative Universes+$\sum$-Types+$\bigoplus$-Type+Finite Types+W-Types*. It is based entirely around Propositions-as-Types, as summarized in Figure 6.10 – more precisely, the "propositional" part of homotopy type theory is an intuitionistic higher-order logic: Our proof rules can be easily observed to correspond to the usual introduction and elimination rules of a natural deduction calculus for intuitionistic logic. To get to a classical setting, it suffices to add a polymorphic constant tnd : $\prod_{i:\mathrm{NAT}} \prod_{U:\mathcal{U}_i} \prod_{A:U} A \bigoplus (A \to \varnothing)$ – which by Propositions-as-Types corresponds to the statement $A \vee \neg A$ for all "propositions" A.

Higher-Order Logic		Homotopy Type Theory	
Constant symbols	$c : \alpha$	constants	$c : \alpha$
Function symbols	$f : \alpha \to \beta$	functions	$f : \alpha \to \beta$
Predicate symbols	$P : \alpha \to \mathsf{Bool}$	functions	$\alpha \to \mathtt{type}$
Propositions	A	types	$A : \mathcal{U}_i$
Conjunctions	$A \wedge B$	Simple Products	$A \times B$
Implications	$A \Rightarrow B$	Simple Function Types	$A \to B$
Disjunctions	$A \vee B$	Coproducts	$A \bigoplus B$
Negations	$\neg A$	Function Types	$A \to \varnothing$
Universal Quantifier	$\forall x : A.\ P(x)$	Dependent Function Types	$\prod_{x:A} P$
Existential Quantifier	$\exists x : A.\ P(x)$	Dependent $\sum$-Types	$\sum_{x:A} P$
Proofs for A		Elements $p : A$	

Figure 6.10: Propositions as Types

Formalizing HoTT in a logical framework such as LF is prohibitively difficult; a previous formalization by Florian Rabe is available online and shows the limitations of such an attempt[18]:

[16] https://gl.mathhub.info/MMT/LFX/blob/master/source/TypedHierarchy.mmt
[17] https://gl.mathhub.info/MMT/LFX/blob/master/source/test.mmt
[18] https://gl.mathhub.info/MMT/examples/blob/master/source/hott.mmt

Even $\sum$-types can only be fully formalized in LF by adding simplification rules (see Remark 5.2) and $\mathbb{W}$-types are left out entirely. Furthermore, since HoTT is – like LF– based on a Martin-Löf type theory, a formalization *within* LF requires re-implementing that very type theory using a HOAS deep embedding, with all the associated drawbacks for working with and within the resulting formalization. This is contrasted by HoTT's aim to be a foundation of mathematics alongside and orthogonal to other such frameworks. Consequently, it is much more adequate to lift an implementation of Homotopy Type Theory to the level of logical frameworks itself. Additionally, this makes HoTT an attractive case study for a modular logical framework.

Homotopy Type Theory is a framework under active investigation and development, and hence frequently subject to change and subtle shifts regarding formal details. The description here follows the presentation in [Uni13], which does not necessarily reflect the current state of the art.

Accordingly, the following definitions and theorems are all taken from the same source and described in much more detail there. They are only contained here for clarification. In particular, we omit all proofs.

6.5.1 Identity Types

Notably, in the correpondences in Figure 6.10 we are yet missing (propositional) equalities. For these, we can add a simple *equality type* $a \doteq_A b : U$ whose semantics is provided by the intended propositions-as-types correspondence.

However, HoTT is somewhat peculiar here, since it does *not* consider propositional types to be *proof irrelevant* - *Equality* of elements $a \doteq_A b$ is considered a type with *arbitrarily many distinct elements*, and this notion of equality is distinct from the *judgmental* equality $a \equiv b : A$. In particular, the existence of a "proof" $p : a \doteq_A b$ does not imply $a \equiv b : A$. The reverse implication holds trivially by the introduction form, for which we use *reflexivity*: For every $a : A$, we have $\mathtt{refl}_a : a \doteq_A a$.

For elimination we use a mechanism called (in the context of HoTT) *path induction*: Given any $p : a \doteq_A b$ and $c(x) : C(x)$, we have $\mathtt{ind}_p\left\{ x, y, q \Longrightarrow_{C(x,y,q)} c(x) \right\} : C(a, b, p)$, where for any $x : A$ we demand that $c(x) : C(x, x, \mathtt{refl}_x)$ and will declare that $\mathtt{ind}_{\mathtt{refl}_a}\left\{ x, y, q \Longrightarrow_{C(x,y,q)} c(x) \right\} = c(a)$. This requires some deliberation:

The central idea is that given a proposition (or type) $C(x)$ depending on $x : A$, if we have a proof/element $c(a) : C(a)$ and a proof $p : a \doteq_A b$, then we should also have a proof/element $c(b) : C(b)$ (*congruence* of equality). This is what $\mathtt{ind}$ yields in the simple case where C only depends on x.

In homotopy type theory, this principle is generalized in that the type C may depend not just on x, but also on an element y considered propositionally (but not necessarily judgmentally) equal to x, as well as the *proof* $p : x \doteq_A y$ for that equality. The postulated judgmental equality $\mathtt{ind}_{\mathtt{refl}_a}\left\{ x, y, q \Longrightarrow_{C(x,y,q)} c(x) \right\} = c(a)$ can then be interpreted as the statement that the family of types $x \doteq_A y$ (for fixed A and variable x, y) is inductively defined by the elements of the form $\mathtt{refl}_a$, in the sense that any type family $C(x, y, p)$ for $x, y : A$ and $p : x \doteq_A y$ is determined by the cases of the form $C(x, x, \mathtt{refl}_x)$.

Note that this does *not* imply that the types $a \doteq_A b$ (for fixed elements $a, b : A$) are *themselves* inductively defined via the elements $\mathtt{refl}_a$– whenever a and b are not judgmentally equal, this type is not inhabited by a $\texttt{refl}\_$-term at all.

Remark 6.9: Based Path Induction

The above is often inconvenient in situations where we want to use *congruence* of equality – i.e. when we know $P(a)$ holds for a fixed, definite element $a : A$, and given $p : a \doteq_A b$ want to infer that $P(b)$ holds as well. The reason is that we need to be able to provide an element $c(x)$ which for *arbitrary* $x : A$ represents a witness for $P(x)$, which we don't have in those instances.

The way around this is to instead use path induction on lambda abstractions, which is a common enough situation to warrant a separate name: **Based Path Induction**. This is definable using general path induction in the following way:

Let $T_C = \prod_{x:A} a \doteq_A x \to \mathcal{U}$ and $T_D = \prod_{z:A} x \doteq_A z \to \mathcal{U}$:

$$\begin{array}{lll} \mathsf{based\_ind} & : & \prod_{a:A} \prod_{C:T_C} \prod_{c:C(a,\texttt{refl}_a)} \prod_{b:A} \prod_{p:a \doteq_A b} C(b,p) \\ & := & \lambda_{a:A}.\ \lambda_{C:T_C}.\ \lambda_{c:C(a,\texttt{refl}_a)}.\ \lambda_{b:A}.\ \lambda_{p:a \doteq_A b}. \\ & & \left(\texttt{ind}_p\left\{x,y,q \Longrightarrow_{\prod_{D:T_D} D(x,\texttt{refl}_x) \to D(y,q)} \lambda_{D,d}.\ d\right\}\right)(C,c) \end{array}$$

Example 6.6:

1. We can use path induction to prove symmetry of equality, i.e. a function of type

$$\prod_{a:A} \prod_{b:A} a \doteq_A b \to b \doteq_A a$$

for any type A.

Given the argument $p : a \doteq_A b$, we have $\texttt{ind}_p\left\{x,y,q \Longrightarrow_{C(x,y,q)} c(x)\right\} : C(a,b,p)$, so we need to choose $C(x,y,q) = y \doteq_A x$ for $C(a,b,p)$ to be the type $b \doteq_A a$.

The term $c(x)$ will have to check against $C(x,x,\texttt{refl}_x)$, so we can choose $c(x) := \texttt{refl}_x$.

Hence, we get $\texttt{ind}_p\left\{x,y,q \Longrightarrow_{y \doteq_A x} \texttt{refl}_x\right\} : b \doteq_B a$, so our function is

$$\mathsf{symm} := \lambda_{a:A}.\ \lambda_{b:A}.\ \lambda_{p:a \doteq_A b}.\ \texttt{ind}_p\left\{x,y,q \Longrightarrow_{y \doteq_A x} \texttt{refl}_x\right\}$$

2. We can use based path induction to prove congruence, i.e. a function of type

$$\prod_{a:A} \prod_{b:A} \prod_{P:A \to \mathcal{U}} a \doteq_A b \to P(a) \to P(b)$$

simply by applying the function $\mathsf{based\_ind}$ in Remark 6.9:

$$\mathsf{cong} := \lambda_{a,b,P,p,pa}.\ \mathsf{based\_ind}(a, (\lambda_{x,q}.\ P(x)), pa, b, p)$$

The rules for identitiy types are given in Figure 6.11. Note that if we wanted to identify propositional equality $a \doteq_A b$ and judgmental equality $a \equiv b : A$, we could add a corresponding equality rule.

6.5.2 Equivalences and Univalence

The principal idea behind homotopy type theory is to interpret propositional equality topologically: Given two "points" $a, b : A$, the type $a \doteq_A b$ is interpreted as the type of all continuous paths between a and b. This makes sense of the multitude of possible "proofs" of $a \doteq_A b$, since

For any U with $\Gamma \vdash U$ `univ`:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma \vdash a \Leftarrow A \quad \Gamma \vdash b \Leftarrow A}{\Gamma \vdash a \doteq_A b \Rightarrow U}$$

Introduction:

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash A \Rightarrow U}{\Gamma \vdash \mathtt{refl}_a \Rightarrow a \doteq_A a}$$

Elimination:

$$\frac{\Gamma \vdash p \Rrightarrow a \doteq_A b \quad \Gamma, x : A, y : A, q : x \doteq_A y \vdash C \Rightarrow U \quad \Gamma, x : A \vdash c \Leftarrow C[\, y/_x \,][\, q/_{\mathtt{refl}_x} \,]}{\mathtt{ind}_p\{\, x, y, q \Longrightarrow_C c \,\} \Rightarrow C[\, x/_a \,][\, y/_b \,][\, q/_p \,]}$$

Computation:

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma, x : A, y : A, q : x \doteq_A y \vdash C \Rightarrow U \quad \Gamma, x : A \vdash c \Leftarrow C[\, y/_x \,][\, q/_{\mathtt{refl}_x} \,]}{\Gamma \vdash \mathtt{ind}_{\mathtt{refl}_a}\{\, x, y, q \Longrightarrow_C c \,\} \rightsquigarrow c[\, x/_a \,]}$$

Figure 6.11: Rules for Identity Types

not all continuous paths between points are equal. Two such paths $p, q : a \doteq_A b$ can be equivalent though, in the sense that there can be a continuous transformation from p to q - i.e. there can be a path $r : p \doteq_{(a \doteq_A b)} q$ - making r a homotopy.

Definition 6.3:

Let $f, g : \prod_{x:A} B(x)$. A **homotopy** from f to g is a function of type

$$f \sim g := \prod_{x:A} f(x) \doteq_{B(x)} g(x)$$

Theorem 6.1:

Under propositions-as-types, homotopy is an equivalence relation, i.e.there are functions of the following types:

$$\prod_{f:\prod_{x:A} B(x)} f \sim f$$

$$\prod_{f,g:\prod_{x:A} B(x)} (f \sim g) \to (g \sim f)$$

$$\prod_{f,g,h:\prod_{x:A} B(x)} (f \sim g) \to (g \sim h) \to (f \sim h)$$

We can now use homotopies to define two types as equivalent in the category theoretical manner: A function is an *equivalence* (isomorphism) if it is invertible (up to homotopy), and two types are *equivalent* (isomorphic) if there is an equivalence between them:

Definition 6.4:

1. We call a function $f : A \to B$ an **equivalence** if the following type is inhabited:

$$\mathsf{isequiv}(f) := \left(\sum_{g:B \to A} f \circ g \sim (\lambda_{x:B}.\, x) \right) \times \left(\sum_{g:B \to A} g \circ f \sim (\lambda_{x:A}.\, x) \right)$$

   Note that under propositions-as-types, this corresponds to the proposition "there are left-inverse and right-inverse functions for f".

2. We call two types A, B **equivalent** if the following type is inhabited:

$$A \simeq B := \sum_{f:A \to B} \mathsf{isequiv}(f)$$

   Note that under propositions-as-types, this corresponds to "there is an equivalence $f : A \to B$".

Theorem 6.2:

As with homotopy, equivalence of types is an equivalence relation.

The final step that Homotopy Type Theory takes is to declare function extensionality (on equivalences) and postulating the *univalence axiom* that declares equality of types *itself* an equivalence.

Definition 6.5:

1. We define a polymorphic function:

$$\mathsf{happly} : \prod_{n:\mathrm{NAT}} \prod_{A:\mathcal{U}_n} \prod_{B:A \to \mathcal{U}_n} \prod_{f,g:\prod_{x:A} B(x)} f \doteq_{\prod_{x:A} B(x)} g \to \prod_{x:A} f(x) \doteq_{B(x)} g(x),$$

   which is definable using `ind`.

2. We define a polymorphic function

$$\mathsf{idtoequiv} : \prod_{n:\mathrm{NAT}} \prod_{A,B:\mathcal{U}_n} A \doteq_{\mathcal{U}_n} B \to A \simeq B,$$

   which is definable using `ind`.

Axiom 6.1: Function Extensionality

For any A, B, f, g, happly is an equivalence, i.e. we have a constant

$$\_ : \mathsf{isequiv}(\mathsf{happly}(n, A, B, f, g))$$

Axiom 6.2: Univalence

For any A, B, idtoequiv is an equivalence, i.e. we have a constant

$$\_ : \mathsf{isequiv}(\mathsf{idtoequiv}(n, A, B))$$

In particular, it follows that
$$(A \doteq_{\mathcal{U}_n} B) \simeq (A \simeq B)$$

6.5.3 Implementation

Implementing the rules for identity types as in Figure 6.11 is straight-forward[19]. All concepts and axioms of Homotopy Type Theory beyond our previously covered typing features can conveniently be specified in MMT syntax directly, as in Listing 6.13.

Listing 6.13: A Theory for HOTT [20]

```
theory Types : TypedHierarchy?LFHierarchy =
    include WTypes?Inductive ❙
    include Equality ?LFEquality ❙

    based_ind : {n : NAT, A :𝒰 n} {a : A, C : {x : A}a ≐x on A → 𝒰n, c : C a (refl a), b : A,
                p : a ≐ b on A} C b p ❘
            = [n,A,a,C,c, b,p] (ind p.x, y, q ⟹ ([C : {z:A}x ≐z on A → 𝒰n,d : C x (refl x)] d)
                to ({C:{z:A}x ≐ z on A → 𝒰n}C x (refl x) → C y q)) C c❙

    Bool : type ❘ = ENUM 2 ❙
    True : Bool ❘ = CASE 1 ❘# ⊤❙
    False : Bool ❘ = CASE 0 ❘# ⊥❙

    u_eqtype : {i : NAT}{A: 𝒰i} A → A → 𝒰i❘ = [i,A,a,b] a ≐ b on A ❘ # 3 ≐4 prec −499 ❙
    // to allow for types of equalities to be inferred rather than given explicitly ❙
❚

theory HOTT : ?Types =

    identity_fun : {i: NAT, A : 𝒰i } A → A ❘= [i , A]([x : A] x) ❘ # id 2 ❙

    homotopy : {i : NAT}{A : 𝒰i, B : A → 𝒰i} ({x: A} B x) → ({x:A} B x) → 𝒰i ❘
            = [i ][ A, B][f][ g] {x: A} (f x ≐ g x) ❘ # 4 ~5 ❙

    is_equiv : {i : NAT}{A:𝒰 i,B:𝒰 i} (A → B) → 𝒰i ❘
            = [i ][ A,B][f] (∑ g : B → A . ([x: B] f (g x)) ~ (id B)) ×
                (∑ h : B → A . ([x: A] h (f x)) ~ (id A)) ❘ # isequiv 4 ❙

    equivalence : {i : NAT}𝒰 i → 𝒰i → 𝒰i ❘
            = [i ][ A,B] ∑f : A → B . isequiv f ❘ # 2 ≃3 ❙

    univalence : {i : NAT}{A:𝒰 i,B:𝒰 i} equivalence (succ i ) (A ≃ B) (A ≐ B) ❙
❚
```

[19]The implementation can be found at https://gl.mathhub.info/MMT/LFX/blob/master/scala/info/kwarc/mmt/LFX/Equality/Rules.scala, the corresponding symbols in https://gl.mathhub.info/MMT/LFX/blob/master/source/Equality.mmt

[20]https://gl.mathhub.info/MMT/LFX/blob/master/source/HOTT.mmt

Remark 6.10: Higher Inductive Types

One noteworthy aspect in Homotopy Type Theory that is being actively researched is the notion of *higher inductive types* - types that are inductively defined via elements of equality types. The prototypical example is the unit circle:

Since in HoTT, elements of an equality type can be thought of as a (homotopical) continuous path between two points, one can imagine defining a unit circle as a single point $a : S_1$ with a *non-trivial* path from a to a; that is an element $p : a \doteq_{S_1} a$ distinct from $\mathtt{refl}_a$. Using higher inductive types, the unit circle (as a type) S_1 is thougt to be *inductively defined / freely generated* from the two declarations $a : S_1$ and $p : a \doteq_{S_1} a$.

Unfortunately, no formal specification of higher inductive types in general exists yet. However, once such a specification exists, one can use a structural feature as in Remark 6.7 (or even the *same* feature) to generate higher inductive types. In the mean time, such a feature can easily serve as a tool to test and experiment with various approaches for formalizing higher inductive types.

Chapter 7

Subtyping

The previously discussed subtyping rules were almost exclusively concerned with variance behavior of type constructors, i.e. rules of the form $C(a){<:}C(a')$ if (depending on the variance) $a{<:}a'$ or $a'{<:}a$. Naturally, in the absence of any additional subtyping rules, these variance rules ultimately reduce to equality rules, since the only way to prove $a{<:}a'$ (ignoring recursively calling variance rules) is to check $a{\equiv}a' :A$. However, some systems allow for primitive subtyping principles of various forms, e.g. by declaring $A{<:}B$ axiomatically for two primitive types A, B or by introducing predicate subtypes. Systems with subtyping mechanism include e.g. PVS [ORS92], IMPS [FGT93] and – in a certain sense – Coq [Coq15] and Matita [Asp+06a], as we will discuss later. Variance rules are accordingly important for the composite types to behave as intended in the presence of primitive subtyping principles.

Notably, subtyping in general breaks several often desirable properties of a type system, such as terms having a unique type and (potentially) decidability. Consequently, many systems decide to not support any subtyping. While this allows for a convenient implementation and formal analysis of a system, it often leads to awkward formalizations of content which can be implemented very naturally with subtypes. For example, numerical types (natural numbers, integers, real numbers) are often desirable to behave as subtypes. Similarly, types for algebraic structures yield a very natural subtyping hierarchy (e.g. all groups are monoids).

In this chapter, I will discuss various approaches to implementing subtyping principles in Mmt, how and why they are not fully implementable in the system, and speculate on how to improve Mmt to better support subtyping mechanisms.

7.1 Issues with Subtyping in Mmt

Mmt supports subtyping in the form of a binary subtyping judgment $A{<:}B$ and rules returning these judgments as conclusion. Notably however, a binary subtyping judgment is insufficient to exploit all deducable subtyping judgments without additional treatment or huge increases in computational cost. A simple example is *transitivity*:

Example 7.1:
Assume we have $\mathbb{N}, \mathbb{Z}, \mathbb{R} : \texttt{type}$ and subtyping rules proving $\mathbb{N}{<:}\mathbb{Z}$ and $\mathbb{Z}{<:}\mathbb{R}$. Consider a statement $\prod_{n:\mathbb{N}} \prod_{r:\mathbb{R}} n+r$, where "+" has type $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$. In order for this to be well-typed, the system has to prove $\Gamma, n : \mathbb{N} {\vdash} n {\Leftarrow} \mathbb{R}$, which in the absence of additional rules would default to checking $\mathbb{N}{<:}\mathbb{R}$. We know that this follows from $\mathbb{Z}$ being an intermediary type between $\mathbb{N}$ and $\mathbb{R}$, but in an implementation, this implies finding a seemingly arbitrary type C satisfying

two additional (binary) judgments, which is computationally expensive. Hence the solver can not algorithmically conclude the judgment from the above two subtyping rules alone.

This example alone is sufficient to demonstrate that merely checking binary subtyping judgments is not sufficient for covering subtyping mechanisms found in other systems.

One way this can be resolved is by keeping the full subtyping lattice for the current context in memory, and dynamically expanding it whenever the context is extended by additional types. Additionally, we can model subtyping premises as constraints (i.e. lower and upper bounds in the subtyping lattice), and introduce appropriate constraints when encountering unsolved type variables. These can then be solved in a manner similar to the method proposed in [KP93]. Notably, to be able to solve these constraints the full subtyping lattice still needs to be known. The feasibility, potential cost and benefits of this approach will need to be explored.

Additional problems arise whenever it becomes necessary to coerce a term to a refinement of its principal type:

Example 7.2:
Consider $n, m : \mathbb{N}$ odd numbers, $r_1 : \mathbb{R} = n/2$ and $r_2 : \mathbb{R} = m/2$, and $S : \mathbb{N} \to \mathbb{N}$.

We know then, that $S(r_1 + r_2)$ is a well-typed expression, since $r_1 + r_2$ can be coerced to have type $\mathbb{N}$, however, $r_1 + r_2$ will have the inferred type $\mathbb{R}$ from the types of $+$ and the arguments r_1, r_2 - from the point of view of the algorithm, the information that $r_1 + r_2$ is in fact a natural number is lost.

This problem can in principle be solved by introducing an object-level *typing*-predicate $a\$B$ with a rule of the form

$$\frac{\Gamma \vdash \_ \Leftarrow a\$B}{\Gamma \vdash a \Leftarrow B}.$$

By allowing arbitrary terms for a and B and quantification on the outside, this corresponds to *term declarations* [SS89]. Again, how to implement this efficiently and whether doing so is an effective solution remains to be determined.

Things get more difficult in the presence of predicate subtyping: $\langle x : A \mid P(x) \rangle$ – the subtype of A containing exactly the elements $x : A$ for which the predicate $P(x)$ holds. Correspondingly, checking $a : \langle x : A \mid P(x) \rangle$ entails *proving* the proposition $P(a)$, for which a strong prover is (if not required) strongly desirable, and the typing system is immediately undecidable. Additionally, the set of all types for a given term t is now a (for all practical purposes) infinite lattice induced by all propositions that hold for t and the implications between them.

However, for knowledge management purposes where fully checking the accuracy of content is not strictly necessary, even weak support for computationally difficult features is sufficient for most purposes, especially plain *representation*. Consequently, in this section we will discuss several subtyping mechanisms that, while not strongly supported by MMT, are still desirable as purely syntactical features with weak inference support to cover as many situations as possible.

7.2 Declared Subtypes and Rule Generators

At its most primitive, a user might want to axiomatically *declare* two types A, B to be subtypes of each other. Naively, one might want to implement this by introducing a new constructor $A <* B$ such that $A <* B$ is inhabitable whenever A and B are, and adding a simple subtyping rule:

$$\frac{\Gamma \vdash \_ : A <* B}{\Gamma \vdash A <: B}$$

The disadvantage of this naive approach is the existence of a subtyping rule that a) needs to be applicable for arbitrary types A, B and hence be potentially called every time a subtyping judgments is checked, and b) defers to the prover to find an element $p : A <* B$, which is computationally expensive and fails in the majority of cases anyway (e.g. when A, B are actually equal). Correspondingly, even though adequate, the mere presence of this rule will noticably slow down type checking in general.

To avoid this problem, we can instead implement a *change listener*, that generates a new subtyping rule for every constant of type $A <* B$ which is only applicable on the specific judgment $A <: B$ and always returns true if applicable. Additionally, this introduces the constraint on the user that they can only declare two types as subtypes by introducing a constant of the respective $<*$-type, which prohibits subtype judgments as arguments in dependent types. This allows the typing system (barring other factors) to stay decidable. The resulting inference rules are simple and listed in Figure 7.1.

Formation:

$$\frac{\Gamma \vdash A \ \mathtt{inh} \quad \Gamma \vdash B \ \mathtt{inh}}{\Gamma \vdash A <* B \ \mathtt{inh}}$$

Subtyping:
For each constant $c : A <* B$:

$$\frac{}{\Gamma \vdash A <: B}$$

Figure 7.1: Rules for Declared Subtypes

A change listener extends the class ChangeListener and needs to implement the following methods:

- onAdd(e: StructuralElement) is called whenever a new StructuralElement (e.g. Constants, Theorys) are added to MMT's library, as after parsing.
- onDelete(e: StructuralElement) is called whenever a StructuralElement is removed from MMT's memory, as during reparsing.
- onCheck(e: StructuralElement) is called when the StructuralElement has been checked to be well-typed.

Example 7.3:
The change listener generating subtyping rules is given in Listing 7.1.

Line 1	declares a new class SubtypeJudgRule for our generated SubtypingRules for the judgment tm1<:tm2. Its head is the path to the symbol $<*$.
Line 3	makes such a rule applicable for the above judgement only, in which case
Line 4/5	simply returns true for that judgement.

The actual ChangeListener starts in Line 8.

Line 12	Given a constant with path $m?c$ that generates a SubtypeJudgRule, the corresponding RuleConstant which declares the generated rule valid has path $m?\{c/$subtypeTag$\}$.
Line 16	attempts to map a constant c to its corresponding generated rule.
Lines 31–33	makes sure to remove the RuleConstant whenever the generating constant is removed.
Lines 35–47	contain the core method of the change listener. It acts whenever a new constant is added (Line 36) whose type is of the form $A <* B$ (Line 37), in which case a new SubtypeJudgRule is generated (Line 38), as well as a corresponding RuleConstant (Line 39), and the latter is added to the library (Line 42).

Listing 7.1: A Change Listener Generating Subtyping Rules [1]

```
1  class SubtypeJudgRule(val tm1 : Term, val tm2 : Term, val by : GlobalName) extends SubtypingRule {
2    val head = subtypeJudg.path
3    def applicable(tp1: Term, tp2: Term): Boolean = tp1.hasheq(tm1) && tp2.hasheq(tm2)
4    def apply(solver: Solver)(tp1: Term, tp2: Term)(implicit stack: Stack, history: History): Option[Boolean]
5     = Some(true)
6  }
7
8  class SubtypeGenerator extends ChangeListener {
9    override val logPrefix = "subtype-rule-gen"
10   protected val subtypeTag = "subtype_rule"
11
12   private def rulePath(r: SubtypeJudgRule) = r.by / subtypeTag
13
14   private def present(t: Term) = controller.presenter.asString(t)
15
16   private def getGeneratedRule(p: Path): Option[SubtypeJudgRule] = {
17     p match {
18       case p: GlobalName =>
19         controller.globalLookup.getO(p / subtypeTag) match {
20           case Some(r: RuleConstant) => r.df.map(df => df.asInstanceOf[SubtypeJudgRule])
21           case _ => None
22         }
23       case _ => None
24     }
25   }
26
27   override def onAdd(e: StructuralElement) {
28     onCheck(e)
29   }
30
```

```
  override def onDelete(e: StructuralElement) {
    getGeneratedRule(e.path).foreach { r => controller.delete(rulePath(r)) }
  }

  override def onCheck(e: StructuralElement): Unit = e match {
    case c: Constant if c.tpC.analyzed.isDefined => c.tp match {
      case Some(subtypeJudg(tm1,tm2)) =>
        val rule = new SubtypeJudgRule(tm1,tm2,c.path)
        val ruleConst = RuleConstant(c.home,c.name / subtypeTag,subtypeJudg(tm1,tm2),Some(rule))
        ruleConst.setOrigin(GeneratedBy(this))
        log(c.name + " ~~> " + present(tm1) + " <: " + present(tm2))
        controller add ruleConst
      case _ =>
    }
    case _ =>
  }
}
```

Remark 7.1:
A ChangeListener is an Extension that needs to be explicitly registered with the MMT system. As such it can not be activated by a rule <path>-statement. Instead, the command extension <fully−qualified−classpath> needs to be called before handling any content which depends on the ChangeListener. This is because change listeners have access to MMT's central components (backend, library, server, other extensions) and hence can change the global state of the system, which mere content should not be allowed to do.

For that reason, the above change listener is currently implemented as part of the ODK-plugin in the MMT code repository itself ([MMT]) rather than the LFX repository ([LFX]), where it arguably belongs.

7.3 Intersection Types

In [Pfe93], the authors present an extension of LF by refinement types (*sorts*) and intersection types, which introduce a weak subtyping principle while retaining decidability. The following canonical example illustrates the usefulness of these intersection types:

Example 7.4:
Assume a type $\mathbb{N}$ of natural numbers with two declared subtypes (*refinement types*) Even and Odd, representing even and odd natural numbers, respectively. Naturally, the successor function S has type $\mathbb{N} \to \mathbb{N}$, however, we also know that it will map odd numbers to even numbers and vice versa – hence S can also be implemented with the types $\mathsf{Odd} \to \mathsf{Even}$ and $\mathsf{Even} \to \mathsf{Odd}$, both of which are *refinements* of the type $\mathbb{N} \to \mathbb{N}$.

Intersection types allow to encode this additional typing information by giving S all three types at once, or more precisely: The intersection type of all three:

$$S : (\mathsf{Odd} \to \mathsf{Even}) \sqcap (\mathsf{Even} \to \mathsf{Odd}) \sqcap (\mathbb{N} \to \mathbb{N})$$

[1]https://github.com/UniFormal/MMT/blob/master/src/mmt-odk/src/info/kwarc/mmt/odk/Rules.scala

Remark 7.2: $\prod$-Elimination
As the above example suggests, intersection types are primarily useful for refining function types by encoding additional information on the return types based on the (refinement) types of the inputs. Note that this breaks the up until now valid assumptions that *every function has a principal* $\prod$*-type*. In fact, the basic rules for LF (to be precise: the elimination rule for function application) assume that the inferred type of every function is equal to a $\prod$-type.

[Pfe93] deals primarily with refinement types, which can be thought of as sorts of a specific (maximal) type. Two refinement types can only be compared if they refine the same maximal type. This allows for retaining decidability and keep the original elimination rule for function applications (under the additional rule that $(\prod_{x:A} B) \sqcap (\prod_{x:A} C) <: \prod_{x:A} B \sqcap C)$, but already excludes Example 7.4.

Thankfully, MMT allows for rules to *shadow* other rules – we can hence implement a new elimination rule for LF with intersection types which subsumes the primtive LF rule. In the presence of this new rule, the old rule is "deactivated", thus preserving modularity.

Remark 7.3: Inhabitants of Intersection Types
Another problem with intersection types in a logical framework is to inhabit them with λ-terms. Consider the following example:

```
        succ :  (Even → Odd) ⊓  (Odd → Even) ⊓  (Nat → Nat) | # S 1 |
        succ2 :  (Even → Even) ⊓ (Odd → Odd) ⊓ (Nat → Nat) | = [x]  S S x |
```

To check the definiens of succ2 against its type, the λ-bound variable x needs to be either explicitly typed, or its type must be inferred from the remainder of the definiens. In both cases we run into the problem that checking the λ-expression against the three distinct intersected function types requires typing x differently each time.

[Sto+17] attempts to solve this by introducing a dedicated introduction form $\langle a, b\rangle$ for intersection types $A \sqcap B$, where it is required that $a \Leftarrow A$, $b \Leftarrow B$ and a and b have the same *shape* (roughly meaning: the same syntax tree disregarding types of bounds variables). The idea being that the definiens for succ2 would have to be ⟨[x:Even] S S x, [x:Odd] S S x, [x:Nat] S S x⟩. While this solves the problem in theory, it is in practice inconvenient for users.

The basic rules for intersection types are given in Figure 7.2.

For any U with $\Gamma \vdash U$ univ:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma \vdash B \Rightarrow U'}{\Gamma \vdash A \sqcap B \Rightarrow \max\{U, U'\}}$$

Type Checking:

$$\frac{\Gamma \vdash a \Leftarrow A \quad \Gamma \vdash a \Leftarrow B}{\Gamma \vdash a \Leftarrow A \sqcap B}$$

Subtyping:

$$\frac{\Gamma \vdash A <: C}{\Gamma \vdash A \sqcap B <: C} \qquad \frac{\Gamma \vdash B <: C}{\Gamma \vdash A \sqcap B <: C} \qquad \frac{\Gamma \vdash C <: A \quad \Gamma \vdash C <: B}{\Gamma \vdash C <: A \sqcap B}$$

Figure 7.2: Rules for Intersection Types

Remark 7.4: Lambda-Expressions in Intersection Types
We can attempt to simplify the introduction form in Remark 7.3 for users by supplying additional rules. This is necessarily computationally expensive, since every conjunct in an intersection needs to be checked individually for each application of such a function, and hence not an ideal solution. Consider the following expression:

$$\begin{aligned} add2 : \ & \mathsf{E} \to \mathsf{E} \to \mathsf{E} \sqcap \mathsf{O} \to \mathsf{O} \to \mathsf{E} \sqcap \mathsf{E} \to \mathsf{O} \to \mathsf{O} \sqcap \mathsf{O} \to \mathsf{E} \to \mathsf{O} \sqcap \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ = \ & \langle \lambda_n .\ \lambda_m .\ n + m \rangle \end{aligned}$$

where E and O abbreviate Even and Odd, respectively. Then we can supply the $\langle\cdot\rangle$-Operator with rules that either duplicate the lambda-expression with different types for the bound variables (as in Remark 7.3), or alternatively type the variables n, m with appropriate $\bigoplus$-types and automatically insert $\cdot \hookrightarrow_\ell \cdot$ or $\cdot {}_r\!\hookleftarrow \cdot$ in any application of a $\langle\lambda\rangle$-expression. Either case would retain the formal correctness of the approach in [Sto+17] without forcing users to provide multiple lambda expressions of the same shape, and require merely a new inference and computation rule for LF-function applications.

A rudimentary form of intersection types has been implemented in MMT[2]. Notably, intersection types as described above still have the drawback that a user has to provide all the intersected types when declaring a new constant, without being able to refine afterwards. For example, the successor function as in Remark 7.3 would require the types Odd and Even to be declared beforehand, which is counter to the usual order of declarations when implementing natural numbers.

Term declarations as mentioned above could potentially be used to add additional refinement types to a constant after its declaration.

7.4 Predicate Subtypes

The type $\langle\, x : A \mid P(x) \,\rangle$ corresponds to the subtype of A containing exactly those elements $x : A$ for which the predicate $P(x)$ holds. In the presence of Judgments-as-Types, this corresponds to the type of elements $x : A$ for which the dependent type $P(x)$ is inhabited – i.e. checking a term t against the type $\langle\, x : A \mid P(x) \,\rangle$ entails checking $t : A$ and the existence of a witness $p : P(t)$.

Given an element $a : \langle\, x : A \mid P(x) \,\rangle$, we should be able to obtain a witness $p : P(a)$, hence we add an additional symbol $\texttt{predOf}\,(\, a \,) : P(a)$.

The remaining rules can be easily derived from this intended semantics:

Remark 7.5:
Note that the second subtyping rule is immediately derivable from the type checking rule. Furthermore, *either* subtyping rule implies $\langle\, x : A \mid P(x) \,\rangle {<:} \langle\, y : A \mid Q(y) \,\rangle$ iff for any $z : A$ the proposition $P(z)$ implies $Q(z)$ – i.e. if there is a function $f : \prod_{z:A} P(z) \to Q(z)$.

Remark 7.6:
As mentioned in Remark 5.1, predicate subtyping is on paper subsumed by $\sum$-types. The difference in an implementation is that $\sum$-types necessitate inserting coercions between the types A and $\sum_{x:A} P$; by pairing an element $a : A$ with a witness $p : P(a)$ in the one direction,

[2] https://gl.mathhub.info/MMT/LFX/blob/master/source/IntersectionTypes.mmt

For any U with $\Gamma \vdash U$ univ:

Formation:

$$\frac{\Gamma \vdash A \Rightarrow U \quad \Gamma, x : A \vdash P \Rightarrow U'}{\Gamma \vdash \langle x : A \mid P \rangle \Rightarrow \max\{U, U'\}}$$

Type Checking:

$$\frac{\Gamma \vdash a \Leftarrow A \quad \Gamma \vdash \_ \Leftarrow P[x/a]}{\Gamma \vdash a \Leftarrow \langle x : A \mid P \rangle}$$

Elimination:

$$\frac{\Gamma \vdash a \Rightarrow \langle x : A \mid P \rangle}{\Gamma \vdash \texttt{predOf}(a) \Rightarrow P[x/a]}$$

Subtyping:

$$\frac{\Gamma, x : A, p : P \vdash x \Leftarrow B}{\Gamma \vdash \langle x : A \mid P \rangle <: B} \qquad \frac{\Gamma \vdash B <: A \quad \Gamma, y : B \vdash \_ \Leftarrow P[x/y]}{\Gamma \vdash B <: \langle x : A \mid P \rangle}$$

Figure 7.3: Rules for Predicate Subtypes

and by the projection $\pi_\ell(\cdot)$ in the other. If these coercions can be inserted automatically and hence left implicit, we can define $\texttt{predOf}(a) := \pi_r(a)$ and all rules in Figure 7.3 are derivable for $\sum$-types. This approach is taken e.g. in Matita and enabled in Coq via the Program package[a], which is intended to mimic the subtyping behavior of PVS.

[a]https://coq.inria.fr/refman/addendum/program.html

The rudimentary implementation for predicate subtypes[3] in MMT based on these rules is used e.g. for our formalization of the PVS logic, described in Chapter 10.

[3]https://gl.mathhub.info/MMT/LFX/blob/master/source/Subtyping.mmt

Chapter 8

Record Types and Models

Disclaimer:

The contents of this chapter have been previously published as [MRK18] and [MRK] with coauthors Florian Rabe and Michael Kohlhase. Regarding individual contributions, both the research and the original writing have been done in close collaboration, hence a strict seperation of contributions among the authors is impossible in retrospect. The writing has been revised for this thesis.

Additionally, the implementation and formulation of the rules for record types are my contribution, under supervision (and with help) by Florian Rabe.

In the area of formal systems like type theories, logics, and specification and programming languages, various language features have been studied that allow for inheritance and modularity, e.g., theories, classes, contexts, and records. They all share the motivation of grouping a list of declarations into a new entity such as in $R = [\![x_1 : A_1, \ldots, x_n : A_n]\!]$. The basic intuition behind it is that R behaves like a product type whose values are of the form $\langle\!| x_1 : A_1 := a_1, \ldots, x_n : A_n := a_n |\!\rangle$. Such constructs are indispensable already for elementary applications such as defining the algebraic structure of Semilattices (as in Figure 8.1), which we will use as a running example.

$$\texttt{Semilattice} = \left\{ \begin{array}{ll} U & : \texttt{type} \\ \wedge & : U \to U \to U \\ \texttt{assoc} & : \mathrm{DED}\ \forall x, y, z : U.\ (x \wedge y) \wedge z \doteq x \wedge (y \wedge z) \\ \texttt{commutative} & : \ldots \\ \texttt{idempotent} & : \ldots \end{array} \right\}$$

Figure 8.1: A Grouping of Declarations for Semilattices

Many systems support **stratified grouping** (where the language is divided into a lower level for the base language and a higher level that introduces the grouping constructs) or **integrated grouping** (where the grouping construct is one out of many type-forming operations without distinguished ontological status), or both. The names of the grouping constructs vary between systems, and we will call them **theories** and **records** in this chapter. An overview of some representative examples is given in Figure 8.2.

	Name of feature	
System	stratified	integrated
ML	signature/module	record
C++	class	class, struct
Java	class	class
Idris [Bra13]	module	record
Coq [Coq15]	module	record
HOL Light [Har96]	ML signatures	records
Isabelle [Wen09]	theory, locale	record
Mizar [Miz]	article	structure
PVS [ORS92]	theory	record
OBJ [Gog+93]	theory	
FoCaLiZe [Har+12]	species	record

Figure 8.2: Stratified and Integrated Groupings in Various Systems

The two approaches have different advantages. Stratified grouping permits a separation of concerns between the core language and the module system. It also captures high-level structure well in a way that is easy to manage and discover in large libraries, closely related to the advantages of the little theories approach [FGT92a]. But integrated grouping allows applying base language operations (such as quantification or tactics) to the grouping constructs. For this reason, the (relatively simple) stratified Coq module system is disregarded in favor of records in major developments such as [Mat].

Allowing both features can lead to a duplication of work where the same hierarchy is formalized once using theories and once using records. A compromise solution is common in object-oriented programming languages, where classes behave very much like stratified grouping but are at the same time normal types of the type system. We call this **internalizing** the higher level features. While combining advantages of stratified and integrated grouping, internalizing is a very heavyweight type system feature: stratified grouping does not change the type system at all, and integrated grouping can be easily added to or removed from a type system, but internalization adds a very complex type system feature from the get-go. It has not been applied much to logics and similar formal systems: the only example we are aware of is the FoCaLiZe [Har+12] system.A much weaker form of internalization is used in OBJ and related systems based on stratified grouping: here theories may be used as (and only as) the types of parameters of parametric theories. Most similarly to our approach, OCaml's first-class modules internalize the theory (called *module type* in OCaml) M as the type `module` M; contrary to both OO-languages and our approach, this kind of internalization is in addition and unrelated to integrated grouping.

In any case, because theories usually allow for advanced declarations like imports, definitions, and notations, as well as extra-logical declarations, systematically internalizing theories requires a correspondingly expressive integrated grouping construct. Records with defined fields are comparatively rare; e.g., present in [Luo09] and OO-languages. Similarly, imports between record types and/or record terms are featured only sporadically, e.g., in Nuprl [Con+86], maybe even as an afterthought only.

Finally, note a related trade-off that is orthogonal to our development: even after choosing either a theory or a record to define grouping, many systems still offer a choice whether a declaration becomes a parameter or a field. See [SW11] for a discussion.

This chapter presents the first formal system that systematically internalizes theories into record types[1]. The central idea is to use an operator `Mod` that turns the theory T into the type `Mod(`T`)`, which behaves like a record type. We take special care not to naively compute this record type, which would not scale well to the common situations where theories with hundreds of declarations or more are used. Instead, we introduce record types that allow for defined fields and merging so that `Mod(`T`)` preserves the structure of T.

This approach combines the advantages of stratified and integrated grouping in a lightweight language feature that is orthogonal to and can be easily combined with other foundational language features. Concretely, it is realized as a module in the MMT framework. By combining our new modules with existing ones, we obtain many formal systems with internalized theories. In particular, our typing rules conform to the abstractions of MMT so that MMT's type reconstruction [Rab17a] is immediately applicable to our features.

8.1 Groupings in Formal Languages

Languages can differ substantially in the syntax and semantics of these constructs. Our interest here is in one difference in particular, which we call the difference between **stratified** and **integrated** grouping.

8.1.1 Analysis

With **stratified** grouping, the language is divided into a lower level for the base language and a higher level that introduces the grouping constructs. For example, the SML module system is stratified: it uses a simply typed λ-calculus at the lower level and *signatures* for the type-like and *structures* for the value-like grouping constructs at the higher level. Critically, the higher level constructs are not valid objects at the lower level: even though signatures behave similarly to types, they are not types of the base language. With **integrated** grouping, only one level exists: the grouping construct is one out of many type-forming operations of the base language with no distinguished ontological status. For example, SML also provides record types as a grouping construct that is integrated with the type system.

Stratified languages have the advantage that they can be designed in a way that yields a conservativity property: all higher level features can be seen as abbreviations that can be compiled into the base language. This corresponds to a typical historical progression where a simple base language is designed first and studied theoretically (e.g., the simply-typed λ-calculus) and grouping is added later when practical applications demand it. But they have the disadvantage that they tend towards a duplication of features: many operations of the lower level are also desirable at the higher level. For example, SML *functors* are essentially functions whose domain and codomain are signatures, a duplication of the function types that already exist in the base language. In logics, this problem is even more severe because quantification and equality (and eventually tactics, decision procedures etc.) quickly become desirable at the higher level as well, at which point a duplication of features tends to become infeasible. A well-known example of this trap is the stratified Coq module system (inspired by

[1]originally published as [MRK18]; for an extended report see [MRK].

SML), which practitioners often dismiss in favor of using record types, most importantly in the mathematical components project [Mat].

This may lead us to believe that record types are the way to go — but this is not ideal either. Record types usually do not support advanced declarations like imports, definitions, and notations, which are commonplace in stratified languages and indispensable in practice. Depending on the system, record types may also forbid some declarations such as type declarations (which would require a higher universe to hold the record type), dependencies between declarations (which would require dependent types), and axioms (which do not fit the record paradigm in systems that do not use a propositions-as-types design). And complex definition principles such as for inductive types and recursive functions are often placed into a stratified higher level just to handle their inherent difficulty. Moreover, stratified grouping has proved very appropriate for organizing extra-logical declarations such as prover instructions (e.g., tactics, rewrite rules, unification hints) examples, sectioning, comments, and metadata. While some systems use files as a simple, implicit higher level grouping construct, most systems use an explicit one. The exalted status of higher level grouping also often supports documentation and readability because it makes the large-scale structure of a development explicit and obvious. This is particularly helpful when formalizing software specifications or mathematical theories, whose structure naturally corresponds to those offered by higher-level grouping. In our work on integrating theorem prover libraries (see Part III), my colleagues at KWARC and me have experienced that this correspondence makes it much easier to compare and integrate different stratified formalizations of the same concepts.

For an in-depth discussion of stratified and integrated groupings in various languages, see [MRK].

8.2 Record Types with Defined Fields

We now introduce record types as an additional module of our framework. The basic intuition is that $[\![\Gamma]\!]$ and $\langle\!|\Gamma|\!\rangle$ construct record types and terms. We call a context **fully typed** resp. **defined** if all fields have a type resp. a definition. In $[\![\Gamma]\!]$, Γ must be fully typed and may additionally contain defined fields. In $\langle\!|\Gamma|\!\rangle$, Γ must be fully defined; the types are optional and usually omitted in practice.

Because we frequently need fully defined contexts, we introduce a notational convention for them: a context denoted by a *lower* case letters like γ is always fully defined. In contrast, a context denoted by an *upper* case letter like Γ may have any number of types or definitions.

We extend our grammar as in Figure 8.3.

$$\begin{array}{rcll}
\Gamma & ::= & \cdot \mid \Gamma, x[:T][:=T] & \\
T & ::= & x \mid \texttt{type} \mid \texttt{kind} & \\
 & \mid & [\![\Gamma]\!] \mid \langle\!|\Gamma|\!\rangle \mid T.x & \text{record types, terms, projections}
\end{array}$$

Figure 8.3: Grammar for Records

Remark 8.1: Field Names and Substitution in Records
Note that we use the same identifiers for variables in contexts and fields in records. This allows reusing results about contexts when reasoning about and implementing records. In particular, it immediately makes our records dependent, i.e., both in a record type and —

maybe surprisingly — in a record term every variable x may occur in subsequent fields. In some sense, this makes x bound in those fields. However, record types are critically different from $\sum$-types: we must be able to use x in record projections, i.e., x can *not* be subject to α-renaming.

As a consequence, capture-avoiding substitution is not always possible. This is a well-known problem that is usually remedied by allowing every record to declare a name for itself (e.g., the keyword `this` in many object-oriented languages), which is used to disambiguate between record fields and a variable in the surrounding context (or fields in a surrounding record). We gloss over this complication here and simply make substitution a partial function.

Before stating the rules, we introduce a few critical auxiliary definition:

Definition 8.1: Substituting in a Record

We extend substitution $t[x/_{t'}]$ to records:

- $[\![x_1 : T_1, \dots, x_n : T_n]\!][y/_t]$
$$= \begin{cases} [\![x_1 : T_1[y/_t], \dots, x_{i-1} : T_{i-1}[y/_t], x_i : T_i, \dots, x_n : T_n]\!] \text{if } y = x_i \\ [\![x_1 : (T_1[y/_t]), \dots, x_n : (T_n[y/_t])]\!] \text{else} \end{cases}$$
if none of the x_i are free in t. Otherwise the substitution is undefined.

- $\langle\!|x_1 := t_1, \dots, x_n := t_n|\!\rangle[y/_t] = \begin{cases} \langle\!|x_1 := t_1, \dots, x_n := t_n|\!\rangle \text{if } y \in \{x_1, \dots, x_n\} \\ \langle\!|x_1 := (t_1[y/_t]), \dots, x_n := (t_n[y/_t])|\!\rangle \text{else} \end{cases}$
if none of the x_i are free in t. Otherwise the substitution is undefined.

- $(r.x)[y/_t] = (r[y/_t]).x$.

Definition 8.2: Substituting with a Record

We write $t[r/_\Delta]$ for the result of substituting any occurrence of a variable x declared in Δ with $r.x$

In the special case where $r = \langle\!|\delta|\!\rangle$, we simply write $t[\delta]$ for $t[\langle\!|\delta|\!\rangle/_\delta]$, i.e., we have $t[x_1 := t_1, \dots, x_n := t_n] = t[x_n/_{t_n}] \dots [x_1/_{t_1}]$.

Our rules for records are given in Figure 8.4. We remark on a few subtleties below.

- The formation rule is partial in the sense that not every context defines a record type or kind. This is because the universe of a record type must be as high as the universe of any undefined field to avoid inconsistencies. If the presence of a hierarchy of universes (see Section 6.4), we can turn every context into a record type.

- The introduction rule infers the principal type of every record term. Because we allow record types with defined fields, this is the singleton type containing only that record term. This may seem awkward but does not present a problem in practice, where type checking is preferred over type inference anyway.

- The elimination rule is straightforward, but it is worth noting that it is entirely parallel to the second computation rule.[2]

[2]Note that it does not matter how the fields of the record are split into Δ_1 and Δ_2 as long as Δ_1 contains all fields that the declaration of x depends on.

Formation:

$$\frac{\vdash \Gamma,\Delta\ \mathtt{ctx} \qquad \Delta \text{ fully typed} \quad \max\{\Delta\} \in \{\mathtt{type},\mathtt{kind}\}}{\Gamma\vdash[\![\,\Delta\,]\!]\Rightarrow \max\{\Delta\}}$$

Introduction:

$$\frac{\vdash \Gamma,\Delta\ \mathtt{ctx} \quad \delta \text{ fully defined} \quad \Delta \text{ like } \delta \text{ but with all missing types inferred}}{\Gamma\vdash\langle\,\delta\,\rangle\Rightarrow[\![\,\Delta\,]\!]}$$

Elimination:

$$\frac{\Gamma\vdash r\Rightarrow[\![\,\Delta_1, x:T[:=t],\Delta_2\,]\!]}{\Gamma\vdash r.x\Rightarrow T[\,r/_{\Delta_1}\,]}$$

Type checking:

$$\frac{\overbrace{\Gamma\vdash r.x\Leftarrow T[\,r/_{\Delta}\,]}^{\text{For } x:T[:=t]\in\Delta} \quad \overbrace{\Gamma\vdash r.x\equiv t[\,r/_{\Delta}\,]:T[\,r/_{\Delta}\,]}^{\text{additionally for } x:T:=t\in\Delta} \quad \Gamma\vdash r\Rightarrow R}{\Gamma\vdash r\Leftarrow[\![\,\Delta\,]\!]}$$

Equality checking (extensionality):

$$\frac{\overbrace{\Gamma\vdash r_1.x\equiv r_2.x : T[\,r_1/_{\Delta}\,]}^{\text{For every } (x:T)\in\Delta}}{\Gamma\vdash r_1\equiv r_2 :[\![\,\Delta\,]\!]}$$

Computation:

$$\frac{\delta = (\delta_1, x[:T] := t, \delta_2) \quad \Gamma\vdash\langle\,\delta\,\rangle\Rightarrow R}{\Gamma\vdash\langle\,\delta\,\rangle.x\rightsquigarrow t[\delta_1]} \qquad \frac{\Gamma\vdash r\Rightarrow[\![\,\Delta_1, x:T:=t,\Delta_2\,]\!]}{\Gamma\vdash r.x\rightsquigarrow t[\,r/_{\Delta_1}\,]}$$

Figure 8.4: Rules for Records

- The type checking rule has a surprising premise that r must already be well-typed (against some type R). Semantically, this assumption is necessary because we only check the presence of the fields required by $[\![\,\Delta\,]\!]$ — without the extra assumption, typing errors in any additional fields that r might have could go undetected. In practice, we implement the rule with an optimization: If r is a variable or a function application, we can efficiently infer some type for it. Otherwise, if $r = \langle\,\delta\,\rangle$, some fields of δ have already been checked by the first premise, and we only need to check the remaining fields. The order of premises matters in this case: we want to first use type checking for all fields for which $[\![\,\Delta\,]\!]$ provides an expected type before resorting to type inference on the remaining fields.

- In the equality checking rule, note that we only have to check equality at undefined fields — the other fields are guaranteed to be equal by the assumption that r_1 and r_2 have type $[\![\,\Delta\,]\!]$.

- Like the type checking rule, the first computation rule needs the premise that r is well-typed to avoid reducing an ill-typed into a well-typed term. In practice, our framework implements computation with a boolean flag that tracks whether the term to be simplified can be assumed to be well-typed or not; in the former case, this assumption can be skipped.
- The second computation rule looks up the definition of a field in the type of the record. Both computation rules can be seen uniformly as definition lookup rules — in the first case the definition is given in the record, in the second case in its type.

Remark 8.2: Horizontal Subtyping and Equality

As mentioned in Remark 4.3, MMT's equality judgment could alternatively be formulated as an untyped equality $t \equiv t'$. For our presentation of record types, however, the use of typed equality is critical.

For example, consider record values $r_1 = \langle a := 1, b := 1 \rangle$ and $r_2 = \langle a := 1, b := 2 \rangle$ as well as record types $R = [\![a : nat]\!]$ and $S = [\![a : nat, b : nat]\!]$. Due to *horizontal subtyping*[a], we have $S <: R$ and thus both $r_i \Leftarrow S$ and $r_i \Leftarrow R$. This has the advantage that the function $S \to R$ that throws away the field b becomes the identity operation. Now our equality at record types behaves accordingly and checks only for the equality of those fields required by the type. Thus, $r_1 \equiv r_2 : R$ is true whereas $r_1 \equiv r_2 : S$ is false, i.e., the equality of two terms may depend on the type at which they are compared. While seemingly dangerous, this makes sense intuitively: r_1 can be replaced with r_2 in any context that expects an object of type R because in such a context the field b, where r_1 and r_2 differ, is inaccessible.

Of course, this treatment of equality precludes downcasts: an operation that casts the equal terms $r_1 : R$ and $r_2 : R$ into the corresponding unequal terms of type S would be inconsistent. But such downcasts are still possible (and valuable) at the meta-level. For example, a tactic $GroupSimp(G, x)$ that simplifies terms x in a group G can check if G is commutative and in that case apply more simplification operations.

We will sometimes omit a type in the formal presentation of rules if convenient. The implementation in MMT allows for equality rules to have an *optional* argument for the type at which they operate, hence both typed an untyped equalities can be treated by the system.

[a]Also called *structural subtyping*; e.g., $[\![x : T, y : S]\!]$ is a subtype of $[\![x : T]\!]$. In contrast, e.g. the straightforward covariance rule for record types is called *vertical* subtyping.

Example 8.1:

Figure 8.5 shows a record type of **Semilattices** (actually, this is a `kind` because it contains a `type` field) analogous to the grouping in Figure 8.1 (using the usual encoding of axioms via judgments-as-types and higher-order abstract syntax for first-order logic).

$$\texttt{Semilattice} := \left[\!\!\left[\begin{array}{ll} U & : \texttt{type} \\ \wedge & : U \to U \to U \\ \texttt{assoc} & : \text{DED}\ \forall x, y, z : U.\ (x \wedge y) \wedge z \doteq x \wedge (y \wedge z) \\ \texttt{commutative} & : \ldots \\ \texttt{idempotent} & : \ldots \end{array} \right]\!\!\right]$$

Figure 8.5: The (Record-)Kind of Semilattices

Then, given a record $r : \texttt{Semilattice}$, we can form the record projection $r.\wedge$, which has type $r.U \to r.U \to r.U$ and $r.\texttt{assoc}$ yields a proof that $r.\wedge$ is associative. The intersection on sets forms a semilattice so (assuming we have proofs $\cap-\texttt{assoc}$, $\cap\texttt{-comm}$, $\cap\texttt{-idem}$ with the corresponding types) we can give an instance of that type as

$$\texttt{interSL} : \texttt{Semilattice} := \langle U := \texttt{Set}, \wedge := \cap, \texttt{assoc} := \cap\texttt{-assoc}, \dots \rangle$$

Theorem 8.1: Principal Types

Our inference rules (in plain LF*) infer a principal type for each well-typed normal term.*

Proof. Let Γ be a fixed well-typed context. We need to show that for any normal expression t the inferred type is the most specific one, meaning if $\Gamma \vdash t \Rightarrow T$, then for any T' with $\Gamma \vdash t \Leftarrow T'$ we have $\Gamma \vdash T <: T'$.
If the only type checking rule applicable to a term t is an inference rule, then the only way for t to check against a type T' which is not the inferred type T is by first inferring T and then checking $\Gamma \vdash T <: T'$, so in these cases the claim follows by default.
By induction on the grammar:

$t = x$ and $x : T \in \Gamma$, then $\Gamma \vdash t \Rightarrow T$, which is principal by default.

$t = \texttt{type}$ then $\Gamma \vdash t \Rightarrow \texttt{kind}$, which is principal by default.

$t = \texttt{kind}$ is untyped.

$t = \prod_{x:A} B$ then $\Gamma \vdash t \Rightarrow U$, where $U \in \{\texttt{kind}, \texttt{type}\}$, both of which are principal by default.

$t = \lambda x : A.t'$ then for some $B : U$ we have $\Gamma, x : A \vdash t' \Rightarrow B$, which is principal by default.

$t = f a$ then $\Gamma \vdash t \Rightarrow B[x/a]$, where $\Gamma \vdash f \Rightarrow \prod_{x:A} B$ $\Gamma \vdash a \Rightarrow A'$ are both principal types and for well-typedness $\Gamma \vdash A' <: A$ needs to hold. Since t is normal, f does not simplify to a lambda-expression and $B[x/a]$ is principal by default.

$t = [\![\Delta]\!]$ then $\Gamma \vdash t \Rightarrow U$, where $U \in \{\texttt{kind}, \texttt{type}\}$, both of which are unique and hence principal.

$t = \langle \delta \rangle$ then $\Gamma \vdash r \Rightarrow [\![\Delta]\!]$, wheren Δ contains the exact same variables, but with all types inferred (and by induction hypothesis principal). For t to check against a type, it has to have the form $[\![\Delta']\!]$, hence we need to show that if $\Gamma \vdash t \Leftarrow [\![\Delta']\!]$, then $\Gamma, r : [\![\Delta]\!] \vdash r \Leftarrow [\![\Delta']\!]$.
Consider the type checking rule for records in Figure 8.4 and let $x : T[:= d] \in \Delta'$. Since $\Gamma \vdash t \Leftarrow [\![\Delta']\!]$, we have $\Gamma \vdash r.x \Leftarrow T$ (and if x is defined in Δ' also $\Gamma \vdash t.x \equiv d : T$) and since Δ is inferred from t we have $x : T' := d$ in Δ, where by hypothesis T' is the principal type of d and hence $\Gamma \vdash T' <: T$.
As a result, $\Gamma, r : [\![\Delta]\!] \vdash r.x \Rightarrow T'$, therefore $\Gamma, r : [\![\Delta]\!] \vdash r.x \Leftarrow T$ and $\Gamma, r : [\![\Delta]\!] \vdash r.x \equiv d : T$ and hence $\Gamma, r : [\![\Delta]\!] \vdash r \Leftarrow [\![\Delta']\!]$, which makes $[\![\Delta]\!]$ the principal type of t.

$t = r.x$ Since t is normal, we have $\Gamma \vdash r.x \Rightarrow T[r/_{\Delta_1}]$ for some $\Gamma \vdash r \Rightarrow [\![\Delta_1, x : T, \Delta_2]\!]$. Since the latter is by hypothesis the principal type of r, for t to typecheck against some T' it needs to be the case that r type checks against some $R = [\![\Delta_1', x : T', \Delta_2']\!]$ and $\Gamma \vdash T <: T'$ holds by the principal type of r.

□

In analogy to function types, we can derive the subtyping properties of record types. We introduce context subsumption and then combine horizontal and vertical subtyping in a single statement.

Definition 8.3: Context Subsumption

For two fully typed contexts Δ_i we write $\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2$ iff for every declaration $x : T[:= t]$ in Δ_1 there is a declaration $x : T'[:= t']$ in Δ_2 such that

- $\Gamma \vdash T' <: T$ and
- if t is present, then so is t' and $\Gamma \vdash t \equiv t' : T$

Intuitively, $\Delta_1 \hookrightarrow \Delta_2$ means that everything of Δ_1 is also in Δ_2. That yields:

Theorem 8.2: Record Subtyping

The following rule is derivable:

$$\frac{\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2}{\Gamma \vdash [\![\Delta_2]\!] <: [\![\Delta_1]\!]}$$

Proof. Assume $\Gamma \vdash \Delta_1 \hookrightarrow \Delta_2$. We need to show $\Gamma, r : [\![\Delta_2]\!] \vdash r \Leftarrow [\![\Delta_1]\!]$. By the type checking rule in Figure 8.4, for any $x : T[:= t] \in \Delta_1$, we need to show that $\Gamma, r : [\![\Delta_2]\!] \vdash r.x \Leftarrow T$ (and if applicable $\Gamma, r : [\![\Delta_2]\!] \vdash r.x \equiv t : T$).
By definition of $\Delta_1 \hookrightarrow \Delta_2$, since $x : T[:= t] \in \Delta_1$, we have $x : T'[:= t] \in \Delta_2$ and $\Gamma \vdash T' <: T$, and if x is defined in Δ_1 the required equality holds as well, so the type checking rule proves $\Gamma, r : [\![\Delta_2]\!] \vdash r \Leftarrow [\![\Delta_1]\!]$ and the claim follows.

□

Merging Records We introduce an advanced operation on records, which proves critical for both convenience and performance: Theories can easily become very large containing hundreds or even thousands of declarations. If we want to treat theories as record types, we need to be able to build big records from smaller ones without exploding them into long lists. Therefore, we introduce an explicit merge operator $+$ on both record types and terms.

In the grammar, this is a single production for terms:

$$T ::= T + T$$

The intended meaning of $+$ is given by the following definition:

Definition 8.4: Merging Contexts

Given a context Δ and a (not necessarily well-typed) context E, we define a partial function $\Delta \mp E$ as follows:

- $\cdot \mp E \;=\; E$
- If $\Delta = d, \Delta_0$ where d is a single declaration for a variable x:
  - if x is not declared in E: $(d, \Delta_0) \mp E \;=\; d,\; (\Delta_0 \mp E)$
  - if $E \;=\; E_0, e, E_1$ where e is a single declaration for a variable x:
    - if a variable in E_0 is also declared in Δ_0: $\Delta \mp E$ is undefined,
    - if d and e have unequal types or unequal definitions: $\Delta \mp E$ is undefined[3],
    - otherwise, $(d, \Delta_0) \mp (E_0, e, E_1) = E_0, m, (\Delta_0, E_1)$ where m arises by merging d and e.

Note that $\mp$ is an asymmetric operator: While Δ must be well-typed (relative to some ambient context), E may refer to the names of Δ and is therefore not necessarily well-typed on its own.

We do not define the semantics of $+$ via inference and checking rules. Instead, we give equality rules that directly expand $+$ into $\mp$ when possible:

$$\frac{\vdash \Gamma, (\Delta_1 \mp \Delta_2)\ \mathtt{ctx}}{\Gamma \vdash [\![\, \Delta_1 \,]\!] + [\![\, \Delta_2 \,]\!] \rightsquigarrow [\![\, \Delta_1 \mp \Delta_2 \,]\!]} \qquad \frac{\vdash \Gamma, (\delta_1 \mp \delta_2)\ \mathtt{ctx}}{\Gamma \vdash \langle\!\langle\, \delta_1 \,\rangle\!\rangle + \langle\!\langle\, \delta_2 \,\rangle\!\rangle \rightsquigarrow \langle\!\langle\, \delta_1 \mp \delta_2 \,\rangle\!\rangle}$$

$$\frac{\vdash \Gamma, (\Delta \mp \delta)\ \mathtt{ctx}}{\Gamma \vdash [\![\, \Delta \,]\!] + \langle\!\langle\, \delta \,\rangle\!\rangle \rightsquigarrow \langle\!\langle\, \Delta \mp \delta \,\rangle\!\rangle}$$

In implementations some straightforward optimizations are needed to verify the premises of these rules efficiently; we omit that here for simplicity. For example, merges of well-typed records with disjoint field names are always well-typed, but e.g., $[\![\, x : \mathit{nat} \,]\!] + [\![\, x : \mathit{bool} \,]\!]$ is not well-typed even though both arguments are.

In practice, we want to avoid using the computation rules for $+$ whenever possible. Therefore, we prove admissible rules (i.e., rules that can be added without changing the set of derivable judgments) that we use preferentially:

Theorem 8.3:

- *If R_1, R_2, and $R_1 + R_2$ are well-typed record types, then $R_1 + R_2$ is the greatest lower bound with respect to subtyping of R_1 and R_2. In particular, $\Gamma \vdash r \Leftarrow R_1 + R_2$ iff $\Gamma \vdash r \Leftarrow R_1$ and $\Gamma \vdash r \Leftarrow R_2$*
- *If $\Gamma \vdash r_i \Leftarrow R_i$ and $r_1 + r_2$ is well-typed, then $\Gamma \vdash r_1 + r_2 \Leftarrow R_1 + R_2$*

Proof.
- If $R_1 \;=\; [\![\, \Delta_1 \,]\!]$ and $R_2 \;=\; [\![\, \Delta_2 \,]\!]$ are well-typed record types, then $R_1 + R_2 \rightsquigarrow [\![\, \Delta_1 \mp \Delta_2 \,]\!]$. By definition of $\mp$, any $r : R_1 + R_2$ hence has all fields defined that are required by both $[\![\, \Delta_1 \,]\!]$ and $[\![\, \Delta_2 \,]\!]$ and the other way around.
- By the rules for $+$, for $r_1 = \langle\!\langle\, \delta_1 \,\rangle\!\rangle$ and $\langle\!\langle\, \delta_2 \,\rangle\!\rangle$ we get $r_1 + r_2 = \langle\!\langle\, \delta_1 \mp \delta_2 \,\rangle\!\rangle$ and if $R_1 = [\![\, \Delta_1 \,]\!]$ and $R_2 = [\![\, \Delta_2 \,]\!]$, then (since $\Gamma \vdash r_i \Leftarrow R_i$) we have $\Delta_1 \hookrightarrow \delta_1$ and $\Delta_2 \hookrightarrow \delta_2$.

[3] It is possible and important in practice to also define $\Delta \mp E$ when the types/definitions in d and e are provably equal. We omit that here for simplicity.

As can easily be verified, it follows that $\Delta_1 \mp \Delta_2 \hookrightarrow \delta_1 \mp \delta_2$, and hence $\langle\!| \delta_1 \mp \delta_2 |\!\rangle \Rightarrow [\![\delta_1 \mp \delta_2]\!] <: [\![\Delta_1 \mp \Delta_2]\!] = R_1 + R_2$.

□

Inspecting the type checking rule in Figure 8.4, we see that a record r of type $[\![\Delta]\!]$ must repeat all defined fields of Δ. This makes sense conceptually but would be a major inconvenience in practice. The merging operator solves this problem elegantly as we see in the following example:

Example 8.2:
Continuing our running example, we can now define a type of semilattices *with order* (and all associated axioms) as in Figure 8.6.

$$\texttt{SemilatticeOrder} := \texttt{Semilattice} + \left[\!\!\left[\begin{array}{lll} \leq & : & U \to U \to U := \lambda_{x,y:U}.\, x \doteq x \wedge y \\ \texttt{refl} & : & \mathrm{DED}\ \forall a : U.\ a \leq a := (\mathrm{proof}) \\ \ldots & & \end{array} \right]\!\!\right]$$

$$\texttt{interSLO} := \texttt{SemilatticeOrder} + \texttt{interSL}$$

Figure 8.6: Running Example

Now the explicit merging in the type `SemilatticeOrder` allows the projection $\texttt{interSLO}.\leq$, which is equal to $\lambda x, y : (\texttt{interSLO}.U)\ .\ (x \doteq x(\texttt{interSLO}.\wedge)y)$ and `interSLO.refl` yields a proof that this order is reflexive – without needing to define the order or prove the axiom anew for the specific instance `interSL`.

8.3 Internalizing Theories

For the purposes of this section, let Θ be the *global context* of available modules.

We can now add the internalization operator, for which everything so far was preparation. We add one production to the **grammar**:

$$T ::= \texttt{Mod}(\,X\,)$$

The intended meaning of $\texttt{Mod}(\,X\,)$ is that it turns a theory X into a record type and a morphism $X : P \to Q$ into a function $\texttt{Mod}(\,Q\,) \to \texttt{Mod}(\,P\,)$. For simplicity, we only state the rules for the case where all include declarations are at the beginning of theory/morphism:

$$\frac{P = \{\texttt{include}\ P_1, \ldots, \texttt{include}\ P_n, \Delta\}\ \text{in}\ \Theta \quad \Delta\ \text{flat} \quad \max\{\,P\,\}\ \text{defined}}{\Theta; \Gamma \vdash \texttt{Mod}(\,P\,) \rightsquigarrow \texttt{Mod}(\,P_1\,) + \ldots + \texttt{Mod}(\,P_n\,) + [\![\Delta]\!]}$$

$$\frac{V : P \to Q = \{\texttt{include}\ V_1, \ldots, \texttt{include}\ V_n, \delta\}\ \text{in}\ \Theta \quad \delta\ \text{flat}}{\Theta; \Gamma \vdash \texttt{Mod}(\,V\,) \rightsquigarrow \lambda r : \texttt{Mod}(\,Q\,).\ \texttt{Mod}(\,P\,) + (\texttt{Mod}(\,V_1\,)r) + \ldots + (\texttt{Mod}(\,V_n\,)r) + \langle\!| \delta[r] |\!\rangle}$$

where we use the following abbreviations:

- In the rule for theories, $\max\{\,P\,\}$ is the biggest universe occurring in any declaration transitively included into P, i.e., $\max\{\,P\,\} = \max\{\max\{\,P_1\,\}, \ldots, \max\{\,P_n\,\}, \max\{\,\Delta\,\}\}$ (undefined if any argument is).

- In the rule for morphisms, $\delta[r]$ is the result of substituting in δ every reference to a declaration of x in Q with $r.x$.

In the rule for morphisms, the occurrence of $\mathtt{Mod}(\,P\,)$ may appear redundant; but it is critical to (i) make sure all defined declarations of P are part of the record and (ii) provide the expected types for checking the declarations in δ.

Example 8.3:

Consider the theories in Figure 8.7. Applying $\mathtt{Mod}(\,\cdot\,)$ to these theories yields exactly the record types of the same name introduced in Section 8.2 (Figures 8.5 and 8.6), i.e., we have $\mathtt{interSL}{\Leftarrow}\mathtt{Mod}(\,\mathtt{Semilattice}\,)$ and $\mathtt{interSLO}{\Leftarrow}\mathtt{Mod}(\,\mathtt{SemilatticeOrder}\,)$. In particularly, $\mathtt{Mod}$ preserves the modular structure of the theory.

$$
\mathtt{theory}\ \mathtt{Semilattice} = \left\{\begin{array}{ll}
U & : \mathtt{type} \\
\wedge & : U \to U \to U \\
\mathtt{assoc} & : \mathrm{DED}\ \forall x,y,z:U.\ (x\wedge y)\wedge z \doteq x \wedge (y \wedge z) \\
\mathtt{commutative} & : \ldots \\
\mathtt{idempotent} & : \ldots
\end{array}\right\}
$$

$$
\mathtt{theory}\ \mathtt{SemilatticeOrder} = \left\{\begin{array}{ll}
\mathtt{include} & \mathtt{Semilattice} \\
\mathtt{order} & : U \to U \to U := \lambda_{x,y:U}.\ x \doteq x \wedge y \\
\mathtt{refl} & : \mathrm{DED}\ \forall a:U.\ a \leq a := \ (\text{proof}) \\
\ldots &
\end{array}\right\}
$$

Figure 8.7: A Theory of Semilattices

The basic properties of $\mathtt{Mod}(\,X\,)$ are collected in the following theorem:

Theorem 8.4: Functoriality

$\mathtt{Mod}(\cdot)$ *is a monotonic contravariant functor from the category of theories and morphisms ordered by inclusion to the category of types (of any universe) and functions ordered by subtyping. In particular,*

- *if P is a theory in Θ and* $\max P \in \{\mathtt{type}, \mathtt{kind}\}$, *then* $\Theta;\Gamma\vdash\mathtt{Mod}(\,P\,)\Leftarrow\max P$
- *if $V : P \to Q$ is a theory morphism in* $\Theta;\Gamma\vdash\mathtt{Mod}(\,V\,)\Leftarrow\mathtt{Mod}(\,Q\,)\to\mathtt{Mod}(\,P\,)$
- *if P is transitively included into Q, then* $\Theta;\Gamma\vdash\mathtt{Mod}(\,Q\,)<:\mathtt{Mod}(\,P\,)$

Proof. • Follows immediately by the computation rule for $\mathtt{Mod}(\,P\,)$.

- Follows immediately by the computation rule for $\mathtt{Mod}(\,V\,)$ and the type checking rule for λ.
- Follows immediately by the computation rule for $\mathtt{Mod}(\,P\,)$ and Theorem 8.3.

□

An immediate advantage of $\mathtt{Mod}(\,\cdot\,)$ is that we can now use the expression level to define expression-like theory level operations. As an example, we consider the **intersection** $P \cap P'$ of two theories, i.e., the theory that includes all theories included by both P and P'. Instead of defining it at the theory level, which would begin a slippery slope of adding more and more theory level operations, we can simply build it at the expression level:

$$P \cap P' := \mathtt{Mod}(\,Q_1\,)+\ldots+\mathtt{Mod}(\,Q_n\,)$$

where the Q_i are all theories included into both P and P'.[4]

Note that the computation rules for `Mod` are efficient in the sense that the structure of the theory level is preserved. In particular, we do not flatten theories and morphisms into flat contexts, which would be a huge blow-up for big theories.[5]

However, efficiently *creating* the internalization is not enough. $\texttt{Mod}(X)$ is defined via $+$, which is itself only an abbreviation whose expansion amounts to flattening. Therefore, we establish admissible rules that allow *working with* internalizations efficiently, i.e., without computing the expansion of $+$:

Theorem 8.5:

Fix well-typed Θ, Γ *and* $P = \{\texttt{include}\ P_1, \ldots, \texttt{include}\ P_n, \Delta\}$ *in* Θ. *Then the following rules are admissible:*

$$\frac{\overbrace{\Theta;\Gamma\vdash r\Leftarrow\texttt{Mod}(P_i)}^{1\leq i\leq n} \quad \overbrace{\Theta;\Gamma\vdash r.x\Leftarrow T[r/_P]}^{x:T\in\Delta} \quad \overbrace{\Theta;\Gamma\vdash r.x\equiv t[r/_P]:T[r/_P]}^{x:T:=t\in\Delta} \quad \Gamma\vdash r\Rightarrow R}{\Theta;\Gamma\vdash r\Leftarrow\texttt{Mod}(P)}$$

$$\frac{\overbrace{\Theta;\Gamma\vdash r_i\Leftarrow\texttt{Mod}(P_i)}^{1\leq i\leq n} \quad \overbrace{\Theta;\Gamma\vdash r_i\equiv r_j:P_i\cap P_j}^{1\leq i,j\leq n} \quad \Theta;\Gamma\vdash\langle\delta\rangle[r/_P]\Leftarrow[\![\Delta]\!] \quad \Gamma\vdash r\Rightarrow R}{\Theta;\Gamma\vdash \underbrace{\texttt{Mod}(P)+r_1+\ldots+r_n+\langle\delta\rangle}_{=:r}\Rightarrow\texttt{Mod}(P)}$$

where $[r/_P]$ *abbreviates the substitution that replaces every* x *declared in a theory transitively-included into* P *with* $r.x$.

The first rule in Theorem 8.5 uses the modular structure of P to check r at type $\texttt{Mod}(P)$. If r is of the form $\langle\delta\rangle$, this is no faster than flattening $\texttt{Mod}(P)$ all the way. But in the typical case where r is also formed modularly using a similar structure as P, this can be much faster. The second rule performs the corresponding type inference for an element of $\texttt{Mod}(P)$ that is formed following the modular structure of P. In both cases, the last premise is again only needed to make sure that r does not contain ill-typed fields not required by $\texttt{Mod}(P)$. Also note that if we think of $\texttt{Mod}(P)$ as a colimit and of elements of $\texttt{Mod}(P)$ as morphisms out of P, then the second rule corresponds to the construction of the universal morphisms out of the colimit.

Example 8.4:

We continue Example 8.3 and assume we have already checked $\texttt{interSL}\Leftarrow\texttt{Mod}(\texttt{Semilattice})$ (*).

We want to check $\texttt{interSL}+\langle\delta\rangle\Leftarrow\texttt{Mod}(\texttt{SemilatticeOrder})$. Applying the first rule of Theorem 8.5 reduces this to multiple premises, the first one of which is (*) and can thus be discharged without inspecting `interSL`.

[4]Note that because $P\cap P'$ depends on the syntactic structure of P and P', it only approximates the least upper bound of $\texttt{Mod}(P)$ and $\texttt{Mod}(P')$ and is not stable under, e.g., flattening of P and P'. But it can still be very useful in certain situations.

[5]The computation of $\max\{P\}$ may look like it requires flattening. But it is easy to compute and cache its value for every named theory.

[5]In practice, these substitutions are easy to implement without flattening r because we can cache for every theory which theories it includes and which names it declares.

Example 8.4 is still somewhat artificial because the involved theories are so small. But the effect pays off enormously on larger theories.

Additionally, we can explicitly allow views *within* theories into the current theory. Specifically, given a theory T, we allow a view

$$T = \{\dots, V : T' \to \cdot = \{\dots\}, \dots \quad \}$$

the codomain of which is the containing theory T (up to the point where V is declared). This view induces a view $T/V : T' \to T$ in the top-level context Θ, but importantly, within T (and its corresponding inner context Γ) every variable in V is defined via a valid term in Γ. Correspondingly, $\mathtt{Mod}(\,V\,)$ is – in context Γ – a constant function $\mathtt{Mod}(\,T\,) \to \mathtt{Mod}(\,T'\,)$ which we can consider as an element of $\mathtt{Mod}(\,T\,)$ directly.

This allows for conveniently building instances of $\mathtt{Mod}(\,\cdot\,)$ and all checking for well-formedness is reduced to structurally checking the view to be well-formed, effectively carrying over all efficiency advantages of structure checking and modular development of theories/views:

Theorem 8.6:

Let Θ, Γ be well-formed and $T/V : T' \to T = \{\dots\}$ in Θ, where Γ is the current context within theory T containing $V : T' \to \cdot = \{\dots\}$. The following rule is admissible:

$$\frac{}{\Theta; \Gamma \vdash \mathtt{Mod}(\,V\,) \Leftarrow \mathtt{Mod}\big(\,T'\,\big)}$$

Proof. We consider $\mathtt{Mod}(\,V\,)$ an abbreviation for $\mathtt{Mod}(\,T/V\,)(\langle\!|\,\cdot\,|\!\rangle)$. Since all definitions in $\mathtt{Mod}(\,T/V\,)$ are well-typed terms in context Γ, the record $\langle\!|\,\cdot\,|\!\rangle$ does not actually occur anywhere in the simplified application $\mathtt{Mod}(\,T/V\,)(\langle\!|\,\cdot\,|\!\rangle)$, which makes this expression well-typed. □

8.4 Implementation

The implementation of record types and the $\mathtt{Mod}(\,\cdot\,)$ can be found at [LFX]/scala/.../Records. They are used extensively in the *Math-in-the-Middle* archive (see Section 3.4).

For a particularly interesting example that occurs in MitM, consider the theories for modules and vector spaces (over some ring/field) given in Listing 8.1 to Listing 8.4, which elegantly follow informal mathematical practice. Going beyond the syntax introduced so far, these use *parametric* theories. Our implementation extends Mod to parametric theories as well, namely in such a way that e.g. $\mathtt{Mod}(\mathtt{module\_theory}) : \prod_{R:\mathtt{ring}} \mathtt{Mod}(\mathtt{module\_theory}(R))$ and correspondingly for fields. Thus, we obtain

$$\mathtt{vectorspace} = \lambda \mathtt{F} : \mathtt{field}.((\mathtt{module}\,\mathtt{F}) + \dots)$$

and, e.g., $\mathtt{vectorspace}(\mathbb{R}) <: \mathtt{module}(\mathbb{R})$, where `vectorspace`, `module`, `field` and `ring` are all $\mathtt{Mod}(\,\cdot\,)$-types.

Because of type-level parameters, this requires some kind of parametric polymorphism in the type system. For our approach, the shallow polymorphism module that is available in PLF (see Remark 4.6) is sufficient.

Listing 8.1: A Theory of Rings[6]

```
theory Ring : base:?Logic =

  theory ring_theory : base:?Logic =
    include sets:?SetStructures/setstruct_theory |
    structure addition : ?AbelianGroup/abeliangroup_theory =
      universe = 'sets:?SetStructures/setstruct_theory?universe | # Uadd |
      op @ rplus | # 1 + 2 prec 5|
      unit @ rzero | # O |
      inverse @ rminus | # − 1 |
    |
    structure multiplication : ?Monoid/monoid_theory =
      universe = 'sets:?SetStructures/setstruct_theory?universe | # Umult |
      op @ rtimes | # 1 · 2 prec 6|
      unit @ rone | # I |
    |
    unitset = ⟨ x: U | ⊢ x ≠ O ⟩ |
    axiom_distributive : ⊢ prop_distributive rplus rtimes |
    axiom_oneNeqZero : ⊢I ≠ O |
    axiom_leftUnital : ⊢ ∀[x : U] I · x ≐ x |
    axiom_rightUnital : ⊢ ∀[x : U] x · I ≐ x |
  |
  ring = Mod ring_theory |
|
```

Listing 8.2: A Theory of Fields [7]

```
theory Field : base:?Logic =
  include ?Ring |

  theory field_theory : base:?Logic =
    include ?Ring/ring_theory |
    inverse : unitset → unitset | # 1 ⁻¹ prec 24 |
    axiom_inverse : ⊢ ∀ [x : unitset ] x · (x ⁻¹) ≐ I |
    axiom_commutative : ⊢∀[x : U]∀[y] (x · y) ≐ (y · x) |
  |
  field : kind | = Mod field_theory |
|
```

Listing 8.3: A Theory of Modules [8]

```
theory Module : base:?Logic =
  include ?Ring |

  theory module_theory : base:?Logic > scalars : ring | =
    scaltp : type | = scalars.universe |
    include ?AbelianGroup/abeliangroup_theory |
    scalarmult : scaltp → U → U |# 1 · 2 |
    axiom_dist1 : ⊢ prop_dist1 scalarmult op|
    axiom_dist2 : ⊢ prop_dist2 scalarmult (scalars.rplus) op |
    axiom_associative_scalars : ⊢ prop_associative_scalars scalarmult (scalars.rtimes) |
    axiom_unit_scalars : ⊢ prop_unit_scalars scalarmult (scalars.rone) |
  |
  module = [R : ring] Mod module_theory R |
|
```

Listing 8.4: A Theory of Vector Spaces [9]

[6]https://gl.mathhub.info/MitM/smglom/blob/master/source/algebra/ringsfields.mmt
[7]Ibid.
[8]https://gl.mathhub.info/MitM/smglom/blob/master/source/algebra/modulsvectors.mmt

```
theory Vectorspace : base:?Logic =
  include ?Field ❙
  include ?Module ❙

  theory vectorspace_theory : base:?Logic > scalars : field ❘ =
    include ?Module/module_theory (scalars) ❙
  ❚
  vectorspace : field → kind ❘ = [F : field ] Mod vectorspace_theory F ❙
❚
```

[9] Ibid.

Chapter 9

Conclusion and Additional Features

The features presented in this part, collectively, yield a logical framework capable of representing the logical foundations of most formal systems encountered in practice. Our modular approach allows for picking and choosing the required features for a given formalization in a compositional manner, while avoiding the potential overhead of unnecessary rules for a given formalization.

Additional Features Notably, many formal systems offer additional features as "syntactic sugar", with an abbreviation-like semantics that can already be fully represented in the underlying logic, but which allow for formalizing content more conveniently. Examples include record and function updates in PVS (e.g. "r is defined as the record s, except for field a, which has value t instead"), *let*-operators that introduce new variables as abbreviations for complex expressions, or *sections* in Coq (where local variables are introduced as constants, references to which are lambda-abstracted away when closing a section).

In the interest of preserving the syntactic representation of an imported library as much as possible, we prefer adding these features in the formalization of a system's foundation. Due to their trivial semantics, these features rarely pose a challenge for implementation, and because they can be elaborated away (and usually are in the system itself during checking), they do not introduce hurdles for library integration.

We have implemented some of these convenience features, including a *let*-operator[1], an object-level substitution function[2], a primitive polymorphic type of *lists*[3], and Coq-like sections[4], the latter of which is very naturally implemented as a structural feature and used in our import of the Coq library [MRS]. Furthermore, Fulya Horozal previously implemented an extension of LF with *finite sequences* and *flexary* operators [Hor14].

Module-Level Extensions The Mod-operator described in Chapter 8 additionally uses Module-level expressions (namely references to theories), but is still a rule-based object-level typing feature of the underlying logic. However, it is also possible to extend MMT's module system itself in various ways. These are (regarding their implementation) rather recent additions to the MMT system and are not central to this thesis, however, they offer interesting extensions

[1]https://gl.mathhub.info/MMT/LFX/tree/master/scala/info/kwarc/mmt/LFX/datatypes

[2]ibid.

[3]ibid.

[4]https://github.com/UniFormal/MMT/blob/master/src/mmt-coq/src/info/kwarc/mmt/coq/Features.scala

for the goal of this part. In particular, module-level features are entirely orthogonal to the logic-level typing feature discussed in this part; hence they are intrinsically compatible with the modular development approach central to this thesis, and composable with all features herein:

- Structural features have by now been extended to the module level. Instead of elaborating into a list of declarations, they can instead elaborate into a list of modules.[5]
- *Implicit morphisms* can be used to treat arbitrary theory morphisms in a manner similar to plain includes; in particular they preserve the (fully qualified) names of the symbols in the source theory, whose semantics in the target theory is provided by the morphism. This approach has been published in [RM18], which also contains a novel description of MMT's module system and its semantics. While the details therein are largely irrelevant and unnecessarily elaborate for this thesis, it is highly recommended for readers interested in designing module systems.
- In this thesis, we treat all theories as named modules with explicitly declared contents. However, MMT treats module references as MMT terms, and *theory combinators* (such as unions or pushouts) allow for building complex terms representing anonymous theories. In particular, the theory combinators of *mathscheme* [CFO10] have been implemented in MMT in .

In addition to allowing the formalization of the foundations of formal systems, our modular logical framework is used as a basis for the Math-in-the-Middle library (see Section 3.4), which serves as a case study and explorative playground for evaluation and inspiration for additional features.

Additionally, the latter allows us to experiment with combining and unifying various different approaches and combinations of features, e.g. Mod-types for theories defined via theory combinators that are the codomains of implicit morphisms, or identifying natural numbers as a W-type with a declarative implementation using the Peano axioms, and a separate type populated by literals.

[5]As-of-yet unpublished

Part III

Translating Mathematical Libraries into a Universal Format

To integrate formal libraries from various sources and to implement generic knowledge management services, as is one of the objectives of this thesis (see Section 3.5), it is necessary to have a unifying framework and language in which these libraries are represented. Naturally, OMDoc/Mmt is (due to its generality and modularity) the prime candidate for such a framework. This Part covers the process of translating one library from an external system into the OMDoc/Mmt language.

The OAF project (see Section 3.2) in particular focuses on integrating libraries from theorem prover systems; however, the methodology involved generalizes easily to the purposes of the OpenDreamKit project (see Section 3.3), where the aim is to integrate databases of mathematical objects and their properties (e.g. finite groups), computer algebra systems and related systems. Together, they cover three aspects of the *tetrapod* (see Chapter 1), namely *inference*, *tabulation* and *computation*. Additionally, the Mmt system itself manages the *organization* aspect.

In all settings, the process involved in translating a library L from system S to OMDoc/Mmt entails the following steps:

1. Use a suitable logical framework (as discussed in Part II) to formalize the foundational ontology of S.

2. Obtain an export of the contents of L in a parsable format. In the case of theorem prover libraries, this usually requires collaboration with the developers of S, since the source files themselves are often only readable for the corresponding system and do not contain inferred information such as implicit arguments, resolved notations, relevant type-check conditions, etc.

3. Implement an importer that systematically translates the export from the previous step into OMDoc/Mmt, using the symbols in the formalization from step 1 to represent the foundational concepts of S.

There is a continuum of possible translations, ranging from perfect adequacy to mere representation. In the best case, we can fully type check the result of the translation and the precise semantics of the original content is preserved under translation. In the worst case, we obtain a representation of the symbols in the original library as untyped and undefined constants. Ideally we want the former, however, depending on the specifics of the system, this might be impossible to achieve without reimplementing the entire system within Mmt. For example, a system with predicate subtypes and a powerful automated prover will usually rely on the latter to type check contents in a manner that can only be reproduced with a similarly powerful prover. In the case of (untyped) computer algebra systems, it might be impossible to check whether a given expression in the language of the system is even well-formed without passing it back to the system itself and evaluating the result.

However, it should be noted that for many knowledge management services such as alignments (see Chapter 14) or (in a limited form) content translation (see Chapter 15), even mere representation is sufficient to obtain useful results. From that point of view, an adequate translation is not strictly necessary, but usually the usefulness of generic services strongly correlates with the amount of details and structure preserved in the translated library.

In this thesis we only consider the *statements* in a given formal library (declarations, definitions, theorems, axioms) – i.e. we do not cover *proofs* or specific implementations of methods in computer algebra system. Importing proofs poses additional challenges and requires an adequate proof language sufficiently generic to subsume the various ways proofs can be represented in formal systems, if they can be exported from the system at all. To the best of my knowledge, such a generic proof language that can serve as an adequate target for translations from theorem prover systems does not yet exist. All knowledge management services presented in this thesis consequently do not benefit from any knowledge about proofs, other than (potentially) the binary information whether a statement is proven at all and (possiby) dependency management.

The methodology described in this part has been applied to various systems and libraries by several people, resulting in OMDoc/Mmt imports for Mizar [Ian+13], HOL Light [KR14], TPTP [Sut09], IMPS [Bet18], Isabelle (as-of-yet unpublished), Coq [MRS] and PVS [Koh+17b]. As a representative example, the latter is covered in detail in Chapter 10. Moving on to less formal corpora, Chapter 11 demonstrates how to handle databases of mathematical objects, exemplary the LMFDB, and Chapter 12 covers computer algebra systems, using the GAP and Sage systems as examples.

Chapter 10

Integrating Theorem Prover Libraries - PVS

Disclaimer:

The contents of this chapter have been published in [Koh+17b] with coauthors Michael Kohlhase, Sam Owre and Florian Rabe.

Both the writing as well as the theoretical results were developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors. My contribution with regards to these can be assumed to be minor, although the writing has been reworked for this thesis.

The implementations (as described below) of the PVS-to-OMDoc translation, the formalization of the PVS foundation and the implementation of the logical framework used therein are my contribution.

10.1 Introduction and Preliminaries

PVS [ORS92] is a verification system, combining language expressiveness with automated tools. Its language is based on higher-order logic, and is strongly typed. It includes types and terms for concepts such as: numbers, records, tuples, functions, quantifiers, and recursive definitions. Full predicate subtypes are supported, which makes type checking undecidable; PVS generates type obligations (TCCs) as artefacts of type checking. For example, division is defined such that the second argument is nonzero, where nonzero is defined:

```
nonzero_real: TYPE = {r: real | r /= 0}
```

Note that functions in PVS are total; partiality is only supported via subtyping.

Beyond this, the PVS language has structural subtypes (i.e., a record that adds new fields to a given record), dependent types for records, tuples, and functions, recursive and co-recursive datatypes, inductive and co-inductive definitions, theory interpretations, and theories as parameters, conversions, and judgements that provide control over the generation of proof obligations. Specifications are given as collections of parameterized theories, which consist of declarations and formulas, and are organized by means of imports.

The PVS prover is interactive, but with a large amount of automation built in. It is closely integrated with the type checker, and features a combination of decision procedures, including BDDs, automatic simplification, rewriting, and induction. There are also rules for ground evaluation, random test case generation, model checking, and predicate abstraction. The prover may be extended with user-defined proof strategies.

PVS has been used as a platform for integration. It has a rich API, making it relatively easy to add new proof rules and integrate with other systems. Examples of this include the model checker, Duration Calculus, MONA, Maple, Ag, and Yices. The system is normally used through a customized Emacs interface, though it is possible to run it standalone (PVSio does this), and PVS features an XML-RPC server (developed independently of the work presented here) that will allow for more flexible interactions. PVS is open source, and is available at `http://pvs.csl.sri.com`.

As an example, Figure 10.1 gives a part of the PVS theory defining *equivalence closures* on a type `T` in its original syntax. PVS uses upper case for keywords and logical primitives; square brackets are used for types and round brackets for term arguments. The most important declarations in theories are

- includes of other theories, e.g., the binary subset predicate `subset?` and the type `equivalence` of equivalence relations on `T` are included from the theories `sets` and `relations` (These includes are redundant in the PVS prelude and added here for clarity.),
- typed identifiers, possibly with definitions such as `EquivClos`, and
- named theorems (here with omitted proof) such as `EquivClosSuperset`.

`VAR` declarations are one of several non-logical declarations: they only declare variable types, which can then be omitted later on; here `PRED[[T,T]]` abbreviates the type of binary relations on `T`.

```
EquivalenceClosure[T : TYPE] : THEORY
BEGIN
  IMPORTING sets, relations
  R: VAR PRED[[T, T]]
  x, y : VAR T
  EquivClos(R) : equivalence[T] =
    { (x, y) | FORALL(S : equivalence[T]) : subset?(R, S) IMPLIES S(x, y) }
  EquivClosSuperset : LEMMA
    subset?(R, EquivClos(R))
  ...
END EquivalenceClosure
```

Figure 10.1: An Excerpt From the PVS Prelude Library

10.2 Formalizing Foundations

To define the language of PVS in MMT, we carry out two steps.

Firstly, we choose a suitable logical framework by picking the necessary features presented in Part II. Notably, we include three features: anonymous record types (see Chapter 8), predicate subtypes (see Chapter 7), and imports of multiple instances of the same parametric theory. The latter is achieved by a structural feature (see Section 6.3) that elaborates into the lambda-abstracted declarations of the imported theory.

Then we use this logical framework to define the MMT theory for PVS. Listing 10.1 shows the most fundamental constants of this theory.

Listing 10.1: The Fundamental Higher-Order Logic of PVS [1]

```
tp : type |
expr : tp → type | # 1 prec −1 |
tpjudg : {A} expr A → tp → type | # 2 : 3 |

pvssigma : {A} (expr A → tp) → tp | # ∑2 |
tuple_tp : tp → tp → tp | # 1 × 2 | = [A][B] ∑[x: expr A]B |
tuple_expr : {A,B} expr A → expr B → expr A ×B | # < 3 , 4 > | ## ( 3 , 4 ) |
proj : {A,B} expr A → expr NatLit → expr B | # 3 _ 4 |

pvspi : {A} (expr A → tp) → tp | # Π2 |
fun_type : tp → tp → tp | = [A,B] Π[x: expr A] B | # 1 ⟹ 2 |
pvsapply : {A,f : expr A → tp} expr (Π f) → {a:expr A} expr (f a)
         | # 3 ( 4 ) prec −1000015 |
lambda : {A,f : expr A → tp} ({a:expr A}expr (f a)) → expr (Π f) | # λ3 |

bool : tp |
boolean : tp | = bool |

FALSE : expr bool |
TRUE: expr bool |
NOT @ ¬| : expr (bool ⟹ bool) |
...

NatLit : tp |
StringLit : tp |
RatLit : tp |
rule rules ?NatLiterals |
rule rules ?StringLiterals |
rule rules ?RationalLiterals |
...

setsub : {A} (expr (A ⟹ bool)) → tp |# ⟨1 | 2 ⟩ |
rule rules ?SetsubRule |

rectp : {' '} → tp | ## 1 |
recordexpr : {r : {' '} } {' '} → expr (rectp r) | ## 2 |
fieldapp # 1 .@ 2 | ## 1 . 2 |

recursor : {A} (expr A → expr A) → expr A | ## INDUCTIVE 2|
```

We begin with a definition of PVS's higher-order logic using only LF features. This includes dependent product and function types[2], classical booleans, and the usual formula constructors (see Listing 10.1). This is novel in how exactly it mirrors the syntax of PVS (e.g., PVS allows multiple aliases for primitive constants) but requires no special MMT features.

We declare three constants for the three types of built-in literals together with MMT rules for parsing and typing them. Using the new framework features, we give a shallow encoding of predicate subtyping (see Listing 10.2 for the new typing rule), a shallow definition of anonymous record types, as well as new declarations for PVS-style inductive and co-inductive types.

[1] https://gl.mathhub.info/PVS/Prelude/blob/master/source/pvs.mmt

[2] Contrary to typical dependently-typed languages, PVS does not allow declaring dependent *base* types, but predicate subtyping can be used to introduce types that depend on terms. Interestingly, this is neither weaker nor stronger than the dependent types in typical $\lambda\Pi$ calculi.

Listing 10.2: PVS-Style Predicate Subtyping in MMT and the Corresponding Rule [3]

```
object SetsubRule extends ComputationRule(PVSTheory.expr.path) {
  def apply(check: CheckingCallback)(tm: Term, covered: Boolean)
    (implicit stack: Stack, history: History): Option[Term]
   = tm match {
    case expr(PVSTheory.setsub(tp,prop)) =>
      Some(LFX.Subtyping.predsubtp(expr(tp),proof("internal_judgment",
        Lambda(doName,expr(tp),pvsapply(prop,OMV(doName),expr(tp),
                        bool.term)._1))))
    case _ => None
  }
}
```

10.3 Importing Libraries

The PVS library export required three separate developments:

Firstly, Sam Owre has extended PVS with an XML export. This is similar to the LaTeX extension in PVS, which is built on the Common Lisp Pretty Printing facility. The XML export was developed in parallel with a Relax NG specification for the PVS XML files. Because PVS allows overloading of names, infers theory parameters, and automatically adds conversions, the XML generation is driven from the internal type-checked abstract syntax, rather than the parse tree. Thus the generated XML contains the fully type-checked form of a PVS specification with all overloading disambiguated. Future work on this will include the generation of XML forms for the proof trees.

Figure 10.2 exemplary shows the (slightly simplified) XML representation of the PVS theory presented in Figure 10.1.

```
<theory place="6049␣0␣6075␣22">
 <id>EquivalenceClosure</id>
 <const-decl place="6057␣2␣6058␣75">
   <id>EquivClos</id>
   <arg-formals>
     <binding place="6057␣12␣6057␣13">
       <id>R</id>
       <type-name>
         <id>PRED</id>
         <actuals>
           <tuple-type>
             <type-name><id>T</id></type-name>
             <type-name><id>T</id></type-name>
           </tuple-type>
         </actuals>
       </type-name>
     </binding>
   </arg-formals>
   <type-name place="6057␣17␣6057␣31">
     <id>equivalence</id>
     <actuals>
       ...
```

Figure 10.2: A Simplified Part of the Function EquivalenceClosure/EquivClos in XML [4]

[3]Ibid. and https://github.com/UniFormal/MMT/blob/master/src/mmt-pvs/src/info/kwarc/mmt/pvs/Plugin.scala

Secondly, Florian Rabe documented the XML schema used by PVS as a set of case classes in Scala and wrote a generic XML parser in Scala that generates a schema-specific parser from such a set of inductive types (see Figure 10.3 for part of the specification). That way any change to the inductive types automatically changes the parser. While seemingly a minor implementation detail, this was critical for feasibility because the XML schema changed frequently along the way.

```
case class const_decl(
    named: ChainedDecl,
    arg_formals: List[bindings],
    tp: DeclaredType,
    _def: Option[Expr]
) extends Decl
```

Figure 10.3: The Scala-Specification of PVS Constant Declarations for XML Parsing

Thirdly, I wrote an MMT plugin that parses the XML files generated by PVS and systematically translates their content into OMDoc/MMT, using the symbols from my formalization of the PVS logic. This includes creating various generic indexes that can be used later for searching the content.

All processing steps preserve source references, i.e., URLs that point to a location (file and line/column) in a source file (the quatruples of numbers at `place=` in Figure 10.2 and `<link rel="...?sourceRef"` in Figure 10.4).

```
<omdoc>
 <theory name="EquivalenceClosure"
  base="http://pvs.csl.sri.com/prelude"
  meta="http://pvs.csl.sri.com/?PVS">
  <constant name="EquivClos">
   <type>
    <om:OMOBJ>
     <om:OMA>
      <om:OMS base="http://cds.omdoc.org/urtheories" module="LambdaPi" name="apply"/>
      <om:OMS base="http://pvs.csl.sri.com/" module="PVS" name="expr"/>
      <om:OMA>
       <om:OMS base="http://cds.omdoc.org/urtheories" module="LambdaPi" name="apply"/>
       <om:OMS base="http://pvs.csl.sri.com/" module="PVS" name="pvspi"/>
        ...
        <metadata>
         <link rel="http://cds.omdoc.org/mmt?metadata?sourceRef"
           resource="prelude/pvsxml/EquivalenceClosure.xml#-1.6057.17:-1.6057.31"/>
        </metadata>
      </om:OMA>
```

Figure 10.4: A Part of the Function EquivalenceClosure/EquivClos in OMDoc [5]

The table in Figure 10.5 gives an overview of the sizes of the involved libraries and the run times[6] of the conversion steps. We note that the XML encoding considerably increases the size of representations. This is due to two effects: the internal, disambiguated form contains significantly more information than the user syntax (e.g. theory parameter instances

[4]https://gl.mathhub.info/PVS/Prelude/blob/master/source/prelude/pvsxml/EquivalenceClosure.xml

[5]https://gl.mathhub.info/PVS/Prelude/blob/master/content/http..pvs.csl.sri.com/prelude/$Equivalence$Closure.omdoc

[6]all numbers measured on standard laptops

	PVS source		PVS → XML		XML → OMDoc	
	size/gz	check time	result size/gz	run time	result size/gz	run time
Prelude	189.7/46.6kB	33s	23.5/.67MB	11s	83.3/1.6MB	3m41s
NASA Lib	1.9/.426MB	23m25s	387.2/8.9MB	3m11s	2.5/.04GB	58m56s

Figure 10.5: File Sizes of the PVS Import at Various Stages

and reconstructed types), and XML as a machine-oriented format is naturally more verbose. Furthermore, OMDoc uses OpenMath for term structures, which again increases file size. While in practice the file sizes are no problem for the Mmt tools presented here, the analogous import for the Isabelle libraries due to Makarius Wenzel[7] has prompted us to compress the OMDoc files by default, significantly reducing their sizes. The Mmt system can read the compressed files directly without need for a user to extract them manually.

[7] As of yet unpublished

Chapter 11

Integrating External Databases (LMFDB)

Having covered importing fully formal theorem prover libraries, we will now turn our attention to systems, where the semantics of the imported ontology is less precise. Specifically, we will look at our methodologies to integrate external databases of mathematical objects (exemplarily the LMFDB) and computer algebra systems (examplarily the GAP and SAGE systems). In other words, the *tabulation* and *computation* aspects of the tetrapod (see Chapter 1).

Integrating these into the MMT system was done as part of the OPENDREAMKIT project (see Section 3.3) with the aim to facilitate data exchange and remote procedure calls between the systems, using the Math-in-the-Middle approach (MitM, see Section 3.4).

Disclaimer:

Parts of the contents of this and the following chapter have been published in [Deh+16] with coauthors Paul-Olivier Dehaye, Mihnea Iancu, Michael Kohlhase, Alexander Konovalov, Samuel Lelièvre, Markus Pfeiffer, Florian Rabe, Nicolas M. Thiéry and Tom Wiesing.

Both the writing as well as the theoretical results were developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors. With respect to the writing itself, this applies only to the introductory sections and descriptions of the systems involved, and my contribution with regards to these can be assumed to be minor. However, they provide important context for the methodologies described.

My contribution in this and the following chapter consists of

- A detailed description of the workflows involved in importing the system ontologies.
- Implementing the importers for SAGE and GAP.
- Implementing the schema theories for the LMFDB.
- Minor parts in the implementation of the infrastructure connecting the schema theories to the LMFDB backend.

11.1 LMFDB Knowledge and Interoperability

11.1.1 Introduction

The *L-functions and modular forms database* is a project involving dozens of mathematicians who assemble computational data about L-functions, modular forms, and related number theoretic objects. The main output of the project is a website, hosted at http://www.lmfdb.org, that presents this data so that it can serve as a reference for research efforts, and is accessible for postgraduate students. The mathematical concepts underlying the LMFDB are complex and varied, and a large amount of effort has been focused on how to relay knowledge, such as mathematical definitions and their relationships, to data and software. For this purpose, the LMFDB has developed so-called *knowls*, which are a technical solution to present LaTeX-encoded information interactively, heavily exploiting the concept of transclusion. The end result is a very modular and highly interlinked set of definitions in mathematical vernacular which can be easily anchored in vastly different contexts, such as an interface to a database, to browsable data, or as constituents of an encyclopedia [Lmfa].

The LMFDB code is primarily written in Python, with some reliance on Sage for the business logic. The backend uses the database system PostgreSQL. Again, due to the complexity of the objects considered, many idiosyncratic encodings are used for the data. This makes the whole data management lifecycle particularly tricky, and dependent on different select groups of individuals for each component.

As the LMFDB spans the whole "vertical" workflow, from writing software, to producing new data, up to presenting this new knowledge, it is a perfect test case for a large scale case study of the MitM approach. Conversely, a semantic layer would be beneficial to integrating its activities across data, knowledge and software.

11.2 Integrating the LMFDB with Math-in-the-Middle

Among the components of the LMFDB, elliptic curves stand out as a well-documented data set, and a source of best practices for other areas. We have generated MitM interface theories for LMFDB elliptic curves by (manually) refactoring and flexiformalizing the LaTeX source of knowls into sTeX (see Listing 11.1 for an excerpt), which can be converted into flexiformal OMDoc/MMT automatically. The MMT system can already type-check the definitions, avoiding circularity and ensuring some level of consistency in their scope and make it browsable through MathHub.info.

Listing 11.1: sTeX Flexiformalization of an LMFDB Knowl (original: [Lmfb])

```
\begin{mhmodnl}{minimal−Weierstrass−model}{en}
   A \defi{minimal} \trefii{Weierstrass}{model} is one for which
   $\absolutevalue\wediscriminantOp$ is minimal among all Weierstrass models
   for the same curve. For elliptic curves over $\RationalNumbers$, minimal
   models exist, and there is a unique minimal model which satisfies the
   additional constraints $\minset{\livar{a}1,\livar{a}3}{\set{0,1}}$, and
   $\inset{\livar{a}2}{\set{−1,0,1}}$.
   This is defined as the reduced minimal Weierstrass model of the elliptic curve.
  \end{definition}
\end{mhmodnl}
```

The first step consisted of translating these informal definitions into progressively more exhaustive MMT formalizations of mathematical concepts (see Listing 11.2) as part of the

MitM library. The two representations are coordinated via the theory and symbol names – we can see the sTEX representation as a human-oriented documentation of the MMT equivalent.

Listing 11.2: MMT Formalization of Elliptic Curves[1]

```
theory Elliptic_curve : ?Base =
  include ?Weierstrass_model |

  elliptic_curve : type |
  base_field : elliptic_curve → field |
  weierstrass_model_construction : weierstrass_model → elliptic_curve |
  minimal_Weierstrass_model : elliptic_curve → weierstrass_model |
  construction_surjective : {A} ⊢weierstrass_model_construction (minimal_Weierstrass_model A) ≐A |
|
```

This gives us the necessary formal content to more precisely specify the semantics of the contents of the LMFDB elliptic curves database. The entries in this database represent elliptic curves and contain fields such as their Cremona-label, that uniquely identifies a curve, its 2-adic generators, conductor, degree, etc.

The core methodology to connect a database to the MitM infrastructure is described in detail in [WKR17]. It relies on the following strategy:

- The contents of a database are specified by a *schema theory* in MMT, whose constants specify the columns of the database and their semantics (via metadata), and whose URI specifies the URL at which the database can be queried.

  Listing 11.3: The Schema Theory for Elliptic Curves[2]

```
theory ec_curves : lmfdb:/omf?EllipticCurves =
  meta /?Metadata?implements elliptic_curve|
  meta /?Metadata?constructor from_record|
  meta /?Metadata?key "label"|

  2adic_gens: List (matrix int_lit 2 2)
    | meta /?Metadata?codec standardList (standardMatrix standardInt 2 2)
    | link /?Metadata?implements MitM:/smglom/elliptic_curves?Rouse_classification?rouse_generators |
  ...
  conductor: pos_lit | meta /?Metadata?codec standardPos
    | link /?Metadata?implements MitM:/smglom/elliptic_curves?Conductor?conductor|
  degree : pos_lit | meta /?Metadata?codec standardPos
    | link /?Metadata?implements MitM:/smglom/elliptic_curves?Modular_degree?modular_degree|
|
```

- A database corresponds to a *virtual theory* in MMT– a theory whose declarations are generated on demand by querying the database.

- An entry in the database corresponds to a constant in its virtual theory, whose name is generated from the `key`-metadatum in the schema theory, whose type is specified by the `implements`-metadatum (e.g. `elliptic_curve` obtained from the formalization above), and whose definiens is a dedicated constuctor specified by the `constructor`-metadatum (e.g. `from_record`) applied to a *record expression* (see Chapter 8).

- This record is obtained by generating a field for each constant in the schema theory, and applying *codecs* to the query result obtained from the database, that compute

[1] https://gl.mathhub.info/MitM/smglom/blob/master/source/elliptic_curves/minimal-Weierstrass-model.omf.mmt

[2] https://gl.mathhub.info/ODK/lmfdb/blob/master/source/schemas/elliptic_curves.mmt

OMDoc/Mmt objects of the required type. For example, `standardInt` converts a JSON number to an integer literal of type `int_lit`, and `standardMatrix` converts an array of arrays to a MitM matrix.

- A new computation rule (see Part II) guarantees that the relevant functions (specified by the `implements`-metadatum) formalized in OMDoc/Mmt applied to the constants obtained simplify to the field in the record corresponding to the function. For example, the function `Conductor?conductor` applied to an expression `from_record(⟨..., conductor := 5, ...⟩)` simplifies to `⟨..., conductor := 5, ...⟩.conductor` and hence ultimately to 5.

This methodology is generic and extends beyond the motivating databases in the LMFDB. [BKR] describes how to add the necessary infrastructure to cover additional databases and generate e.g. query interfaces and other services directly from schema theories.

Listing 11.4 shows how we can use the resulting constants in practice. The included theory `db?ec_curves` is the virtual theory generated by the schema theory in Listing 11.3. The constant `11a1` is computed as described above from the entry in the database with label `11a1` when it is first referenced. Applying the `conductor` function to this constant yields an `int_lit` with the value obtained from querying the database.

Listing 11.4: Using Virtual Theories[3]

```
include db?ec_curves❙
// get elliptic  curve 11a1 from LMFDB ❙
mycurve: elliptic_curve ❘= 'db?ec_curves?11a1❙
// let c be its conductor (as stored in the LMFDB) ❙
c: int_lit ❘ = conductor mycurve❙ // = 11 ❙
```

As a result, we obtain access to all the objects in the databases from within the Mmt system. The objects have globally unique Mmt URIs, and all properties and values for a given object that are stored in the database are obtainable using the Mmt syntax. In other words: The database behaves just like any other collection of Mmt theories and can consequently be used for any of the applications described in this thesis; most importantly those described in Part IV and those relevant for the OpenDreamKit project.

[3]https://gl.mathhub.info/ODK/lmfdb/blob/master/source/schemas/tutorial_example.mmt

Chapter 12

Integrating Generic Ontologies

12.1 Distributed Collaboration with GAP/Sage

Another aspect of interoperability in a mathematical virtual research environment – as is the goal of OpenDreamKit – is the possibility of distributed multisystem computations, where e.g. a given system may decide to delegate certain subcomputations or reasoning tasks to other systems.

There are already a variety of peer-to-peer interfaces between systems in the OpenDreamKit project (see Figure 3.3), which are based on the *handle paradigm*; for example Sage includes, among others, interfaces for GAP, Singular, and PARI. In this paradigm, when a system A delegates a calculation to a system B, the result r of the calculation is not converted to a native A object; instead B just returns a *handle* h (or reference) to the object r. Later, A can run further calculations with r by passing it as argument to functions or methods implemented by B. Some advantages of this approach are that we can avoid the overhead of back and forth conversions between A and B, and that we can manipulate objects of B from A, even if they have no native representation in A.

The next desirable feature is for the handle h to behave in A as if it was a native A object; in other words, one wants to adapt the API satisfied by r in B to match the API for the same kind of objects in A. For example, the method call `h.cardinality()` on a Sage handle `h` to a GAP object `G` should trigger in GAP the corresponding function call `Size(G)`.

This can be implemented using the classical *adapter pattern*, mapping calls to Sage's method to corresponding GAP methods. *Adapter classes* have already been implemented for certain types of objects, like Sage's `PermutationGroup` or `MatrixGroup`. However, this implementation lacks modularity: for example, if `h` is a handle to a mere set `S`, Sage cannot use the *adapter method* that maps `h.cardinality()` to `Size(S)`, because this adapter method is only available in the above two adapter classes.

To get around this problem, Nicolas M. Thiéry and others have worked on a more semantic integration, where adapter methods are made aware of the type hierarchies of the respective systems, and defined at the highest available level of generality, as in Listing 12.1.

Listing 12.1: A Semantic Adapter Method in Sage

```
class Sets: # Everything generic about sets in Sage
    class GAP: # Adapter methods relevant to Sets in the Sage−Gap interface
        class ParentMethods: # Adapter methods for sets
            def cardinality(self): # The adapter for the cardinality method
```

```
                return self.gap().Size().sage()
        class ElementMethods: # Adapter methods for set elements
            ...
        class MorphismMethods: # Adapter methods for set morphisms
            ...
class Groups: # Everything generic about groups in Sage
    # This automatically includes features defined at a more general level
```

This peer-to-peer approach however does not scale up to a dozen systems. This is where the Math-in-the-Middle paradigm comes to the rescue. With it, the task is reduced to building interface theories and interface views into the core MitM ontology in such a way that the adapter pattern can be made generic in terms of the MitM ontology structure, without relying on the concrete structure of the respective type systems. Then the adapter methods for each peer-to-peer interface can be automatically generated. In our example the adapter method for `cardinality` can be constructed automatically as soon as the MitM interface views link the `cardinality` function in the SAGE interface theory on Sets with the `Size` function in the corresponding interface theory for GAP.

Exporting the GAP Knowledge: Type System Documentation The GAP type system encodes a wealth of mathematical knowledge, which can influence method selection. For example establishing that a group is nilpotent will allow for more efficient methods to be run for finding its centre. The main difference between SAGE and GAP lies in the method selection process. In SAGE the operations implemented for an object and the axioms they satisfy are specified by its class which, together with its super classes, groups syntactically all the methods applicable in this context. In GAP, this information is instead specified by the truth-values of a collection of independent **filters**, while the context of applicability is specified independently for each method (the details are discussed later). Breuer and Linton describe the GAP type system in [BL] and the GAP documentation [Gap] also contains extensive information on the types themselves.

GAP allows some introspection of this knowledge after the system is loaded: the values of those attributes and properties that are unknown on creation, can be computed on demand, and stored for later reuse.

As a first step in generating interface theories for the MitM ontology, GAP developers have implemented tools to access mathematical knowledge encoded in GAP, such as introspection inside a running GAP session, export to JSON to import to MMT, and export as a graph for visualization and exploration. The JSON output of the GAP object system with default packages is currently around 11 Megabytes, and represents a knowledge graph with 540 vertices, 759 edges and 8 connected components, (see Figures 12.1,12.2). If all packages are loaded, this graph expands to 1616 vertices, 2178 edges and 17 connected components.

There is, however, another source of knowledge in the GAP universe: the documentation, which is provided in the GAPDoc format [LN12]. Besides the main manuals, GAPDoc is adopted by 97 out of the 130 packages currently redistributed with GAP. Conventionally GAPDoc is used to build text, PDF and HTML versions of the manual from a common source given in XML. The reference manual has almost 1400 pages and the packages add hundreds more.

The GAPDoc sources classify documentation by the type of the documented object (function, operation, attribute, property, etc.) and index them by system name. In this sense they are synchronized with the type system (which *e.g.* has the types of the functions) and can be combined into flexiformal OMDoc/MMT interface theories, just like the ones for LMFDB in

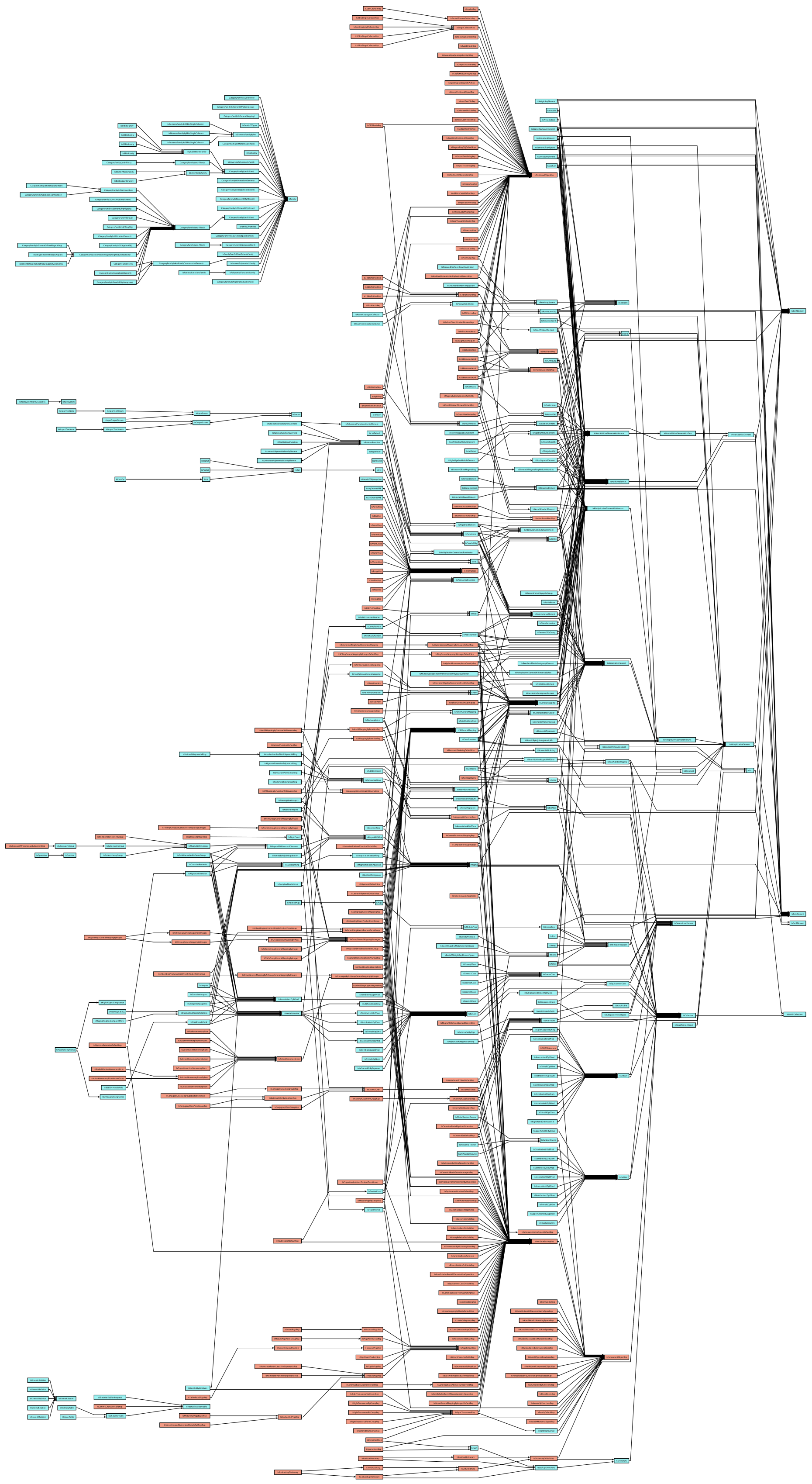


Figure 12.1: The GAP Knowledge Graph.

Section 11.1. This conversion is currently under development and will lead to a significant increase of the scope of the MitM ontology.

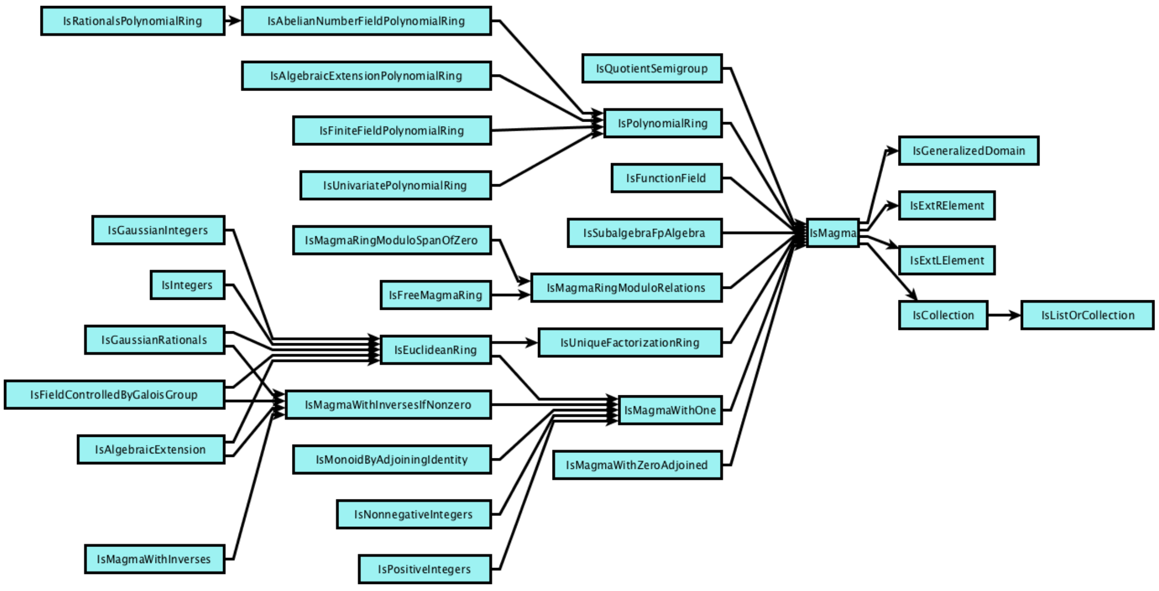


Figure 12.2: The GAP Knowledge Graph (fragment).

As a side-effect of this work, Markus Pfeiffer discovered quite a few inconsistencies in the GAP documentation, which came from a semi-automated conversion of GAP manuals from the TEX-based manuals used in GAP 4.4.12 and earlier. In response, he implemented a consistency checker for the GAP documentation, which extracts type annotations from the documented GAP objects and compares them with their actual types. It immediately reported almost 400 inconsistencies out of 3674 manual entries, 75% of which have been eliminated in a subsequent cleanup.

Semantics in the SAGE Category System The SAGE library includes 40k functions and allows for manipulating thousands of different kinds of objects. In any large system it is critical to tame code bloat by

i) identifying the core concepts describing common behavior among the objects;

ii) implementing generic operations that apply on all objects having a given behavior, with appropriate specializations when performance calls for it;

iii) designing or choosing a process for selecting the best implementation available when calling an operation on some objects.

Following mathematical tradition and the precedent of the AXIOM, FRICAS, or MUPAD systems, SAGE has developed a category-theory-inspired "category system", and found a way to implement it on top of the underlying PYTHON object system [Dev16; SC]. In short, a **category** specifies the available **operations** and the **axioms** they satisfy. This category system models taxonomic knowledge from mathematics explicitly and uses it to support genericity, control the method selection process, structure the code and documentation, enforce consistency, and provide generic tests.

To generate interface theories from the SAGE category system, Nicolas Thiéry et al are experimenting with a system of annotations in the SAGE source files. Consider for instance

```
@semantic(mmt="sets")
class Sets:
    class ParentMethods:
        @semantic(mmt="card?card", gap="Size")
        @abstractmethod
        def cardinality(self):
            "Return␣the␣cardinality␣of␣``self``"
```

Figure 12.3: An Annotated Category in SAGE

the situtation in Figure 12.3 where we have annotated the `Sets()` category in SAGE with `@semantic` lines that state correspondences to other interface theories. From these the SAGE-to-MMT exporter can generate the respective interface theories and views.

In ongoing experiments, variants of the annotations are tested for annotating existing categories without touching their source files and providing the signature or the corresponding method names in other systems when this information has not yet been formalized elsewhere.

12.2 Representing the GAP Ontology in OMDoc/Mmt

To facilitate the kind of integration demanded by the OPENDREAMKIT project (such as remote procedure calls), we need appropriate representations of the functionalities offered by the targeted systems. In particular, we need to represent the underlying ontologies of the systems.

In the case of GAP, this ontology is based on the notion of *filters*, as described above: Every object belongs to several categories and satisfies a list of filters, which can be thought of as predicates (e.g. `IsGroup` and `Abelian`, jointly specifying that the object is an abelian group). The way a user interacts with the system is by calling *operations* (e.g. the order of a group) on an object, which has to satisfy certain filters in order for the operation to be applicable. Notably however, each operation is implemented by possibly multiple *methods*: specific algorithms for computing an operation. The system uses the known filters of an object to determine which specific method to use for a specific operation. For example,the order of a group might be easier to compute if the group is known to be abelian, hence the system would choose a more appropriate method to compute the order of an abelian group than for a group that is not (known to be) abelian. Notably, the system chooses the method automatically; a user only interacts with the system by calling operations.

This notion of filters effectively yields a soft typing system, which we formalize as in Listing 12.2.

Listing 12.2: The GAP Foundational Ontology in MMT [1]

```
theory Types : ur:?PLF =
        object  :  type ❘
        category  :  type ❘

        booleans :  type ❘
        integers  :  type ❘
        floats   :  type ❘
        rule  rules ?Booleans ❘
        rule  rules ?Integers  ❘
        rule  rules ?Floats  ❘

        gapbool :  booleans → object ❘
```

```
13        gapint : integers → object ❘
14        gapfloat : floats → object ❘
15
16        filter  = object → type ❘
17        hastp : object → filter → type ❘ = [o][f] f o ❘ # 1 $ 2 ❘
18
19        filter_and : filter → filter → filter ❘ # 1 and 2 ❘
20        filter_and_hasFilter1 : {x : object , f : filter , g : filter } x $ f → x $ g → x $ (f and g) ❘
21        filter_and_hasFilter2 : {x : object , f : filter , g : filter } x $ (f and g) → x $ f ❘
22        filter_and_hasFilter3 : {x : object , f : filter , g : filter } x $ (f and g) → x $ g ❘
23
24        catFilter : category → filter ❘
25        propertyFilter : (object → object) → filter ❘ = [p] [o] ded (p o) ❘
26
27        CategoryCollection : filter → category ❘
28        Set : filter → (object → object) ❘
29  ❚
```

The most primitive notions in the GAP ontology are objects and categories, which we represent as LF types. Naturally, GAP uses literals for booleans, integers and real numbers; consequently we introduce new types for these and populate them with literals as well (lines 5–10).To convert our literals to GAP objects, we introduce conversion functions (lines 12–14). These GAP native literals do not occur in the JSON-export; they are needed for most services implemented in the OpenDreamKit project however. For example, they are needed for translating even simple commands such as `2 + 2` into the GAP ontology to allow for remote procedure calls.

We formalize filters as functions `object → type` and effectively use judgments-as-types for the judgment `o$f` representing "object o has filter f". The filter_and-operator corresponds to a conjunction of filters. catFilter allows to build the corresponding filter to a given category, conversely, `CategoryCollection` allows to form the category of all objects satisfying a given filter. An operation f is considered a *property* if it returns a boolean value. In this case we can form the corresponding filter `propertyFilter(f)` that an object satisfies iff the operation f returns *true* on that object.

Using this formalization, we can systematically translate the contents of the JSON-export of the GAP ontology mentioned above into OMDoc/Mmt declarations.

The export contains a JSON object for each category and operation. For categories, it additionally contains the list of implied filters, for example, the category `IsAbelianNumberFieldPolynomialRing` implies filters such as `IsCollection` and `IsDuplicateFree`. Naturally, a category C is translated to a constant of type `category`. For each implied Filter F, we generate an additional constant of type $\prod_{x:\mathsf{object}} x\$\mathsf{catFilter}(C) \to x\$F$.

Unfortunately, the current JSON export does not yet contain any information regarding the required filters of an operation, or the filters of the return object. In the course of the work on the export described above, attempts are being made to make those accessible as well. For now, all operations are correspondingly simply typed as $\mathsf{object}^n \to \mathsf{object}$, depending on the arity.

While more detailled information regarding the filters would be desirable for documentation purposes, the above import methodology is sufficient to translate GAP objects, operations,

[1]https://gl.mathhub.info/ODK/GAP/blob/master/source/types.mmt

and applications of operations to objects into well-formed OMDoc/Mmt expressions.[2]

12.3 Representing the Sage Ontology in OMDoc/Mmt

Having covered the GAP ontology already, extending the same methodology to a smiliar system as Sage is in many respects straight forward. Notably however, whereas GAP implements its own fundamental notions for even simple concepts like integers, Sage is a python library, and as such operates on many of python's datatypes directly. Consequently, we start with a formalization of the python datatypes relevant for our purposes:

Listing 12.3: Python Datatypes in Mmt [3]

```
theory Python : ur:?PLF =
  // types|
  python_type : type |
  Str    : python_type|
  Int    : python_type|
  Float  : python_type|
  Bool   : python_type|
  Tuple  : python_type|
  List   : python_type|
  Dict   : python_type|
  None   : python_type|
  Fun    : python_type|
  Type   : python_type|

  // object constructors |
  python_obj : type |
  false  : python_obj|
  true   : python_obj|
  tuple  : python_obj|
  list   : python_obj|
  // dict takes arguments of the form tuple(key,value) |
  dict   : python_obj|
  // dot takes a and a string f and returns a.f|
  dot    : python_obj|
  none   : python_obj|
|
```

Note that since python uses a dynamic type system, we can not assign a fixed type to a given python object. This is reflected by our ontology, which specifies even the *constructors* for the various types as having type `python_object` rather than some dependent type on `python_type`. As with GAP, these datatypes are not needed for the import of the ontology, but rather for services that operate on expressions within the ontology.

For the ontology itself, we use – as with GAP– a generic type `object`.

Listing 12.4: The Sage Foundational Ontology in Mmt [4]

```
theory Types : ur:?PLF =
  object : type |
  prop : type |
  ded : prop → type | ## ⊢1 |
  structural  : type |
  structureof  : structural  → type | ## ≤1 |
|
```

[2]The resulting ontology is available at https://gl.mathhub.info/ODK/GAP
[3]https://gl.mathhub.info/ODK/Sage/blob/master/source/sage.mmt

```
theory Sage : ?Python =
  include ?Types ❘
  sage_type : type ❘
  sageMMT : sage_type → type ❘
  sagepy     : sage_type → python_type❘
❙
```

As mentioned above, knowledge in SAGE– and hence in its JSON-export – is organized in *categories* (e.g. `Ring`). A category satisfies certain *axioms/properties* (which after translating live in the type `prop` in Listing 12.4) and implements certain *structures* (or *signatures*, which will live in `structural`) – e.g. the category of rational polynomials implements the signature of a ring, and satisfies (among others) the axioms of a commutative ring.

Each category provides certain methods; namely *element methods* that operate on the elements in an object of the category (e.g. addition or multiplication in a ring), *parent methods* that operate on an object of the category directly (e.g. its characteristic or degree), and *morphism methods* that operate on the respective morphisms of the category.

We translate categories directly to OMDOC/MMT theories. The JSON-export currently provides no further information on neither the axioms nor the structures. Correspondingly, we treat them as atomic objects of the respective type. We collect all axioms and structures occuring during the import and store them in separate generated theories `Axioms` and `Structures` , which are included in all the theories corresponding to categories. If a category C statisfies an axiom A or implements a structure S, we add to the theory constants of types $\vdash A$ or $\leq S$, respectively.

As with GAP, we currently have no information about the input or output types (or categories) of methods; hence we are left with translating those to functions $\mathsf{object}^n \to \mathsf{object}$, depending on their arity. Again, this yields a sufficient formalization of the functionalities provided by SAGE to facilitate the goals of the OPENDREAMKIT project.[5]

[4]Ibid.

[5]The full import of the SAGE import can be found at https://gl.mathhub.info/ODK/Sage

Chapter 13

Conclusion

We have seen how, using the logical frameworks developed in Part II, we can represent libraries (in the broadest sense) of formal systems in OMDoc/Mmt, including both fully formal libraries as obtained by theorem prover systems, as well as the ontologies of less formal systems such as computer algebra systems and databases of mathematical objects.

In Section 13.1 below, we will look at some suggested applications enabled by representing a *single* (fully formal) library in OMDoc/Mmt; however, in the light of the title of this thesis, we are much more interested in applications enabled *across* libraries. Consequently, the important result of this part is less the ability to represent a single library, rather than having a universal framework in which *multiple* libraries/ontologies of various systems can be (and are) represented, as a prerequisite for the knowledge management service described in Part IV. As such, the intended goals of the OpenDreamKit project are much more in line with the aims of this thesis, even though the systems involved therein (covered in Chapters 11 and 12) are less formal than the fully formal – and hence for knowledge management purposes more interesting – libraries discussed in Chapter 10.

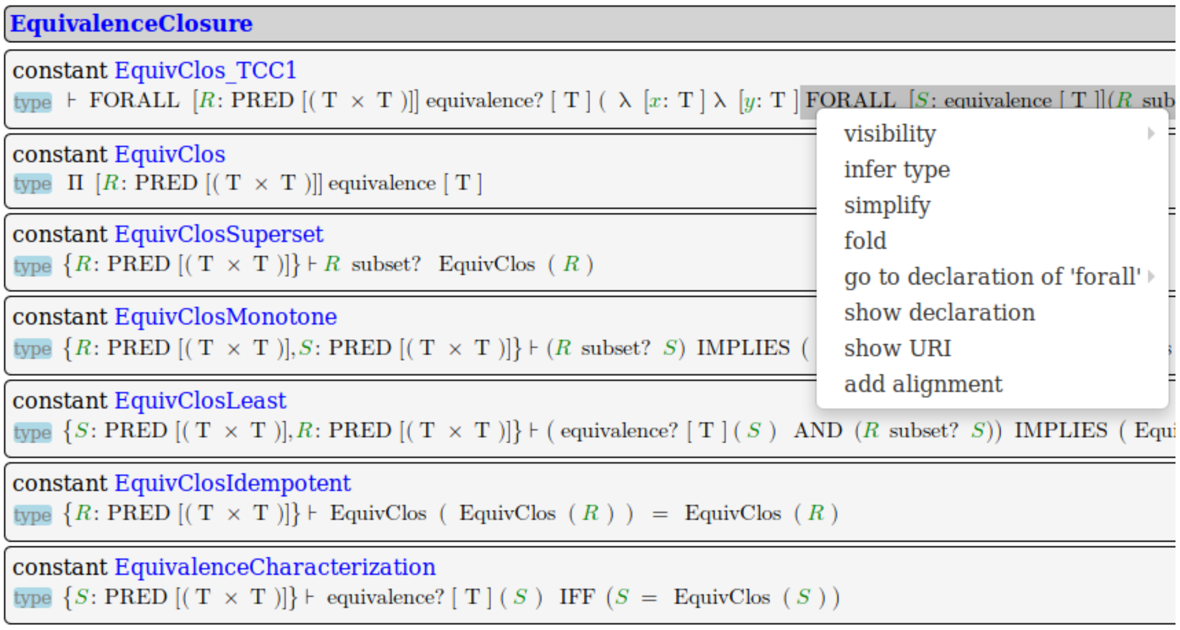
13.1 Enabled Applications

With the OMDoc/Mmt translation of formal libraries, the originating systems gain access to library management facilities implemented at the OMDoc/Mmt level.[1] There are two ways to exploit this: publishing the converted libraries on a dedicated server, like the MathHub system [MH], or running the OMDoc/Mmt toolstack locally. Both options offer similar functionality, the main difference is the intended audience: the first option is for outside users who want to access the libraries, and the latter is for users who develop new content or refactor the library.

MathHub (see Section 4.1.2) bundles a GitLab-based repository manager with Mmt and various periphery systems into a common, web-based user interface. We commit the exported libraries – exemplarily from PVS– as OMDoc/Mmt files into a repository to make these available via the *i*) MathHub user interface, *ii*) Mmt presentation web server, *iii*) Mmt web services, and *iv*) the MathWebSearch daemon. All of these components give the user different ways of interacting with the system and content. Below we explore three examples that are directly useful for PVS users, but the services apply similarly to users of other systems.

[1] The description of services in this Section has been adapted from [Koh+17b] and [Con+]

The local workflow installs OMDoc/Mmt tools on the same machine as PVS. In that case, users are able to browse the current version of the available PVS libraries including all experimental or private theories that are part of the current development. This also enables PVS to use OMDoc/Mmt services as background tools that remain transparent to the PVS user.



EquivalenceClosure

constant EquivClos_TCC1
type ⊢ FORALL [R: PRED [(T × T)]] equivalence? [T] (λ [x: T] λ [y: T] FORALL [S: equivalence [T]](R sub

constant EquivClos
type Π [R: PRED [(T × T)]] equivalence [T]

constant EquivClosSuperset
type {R: PRED [(T × T)]} ⊢ R subset? EquivClos (R)

constant EquivClosMonotone
type {R: PRED [(T × T)], S: PRED [(T × T)]} ⊢ (R subset? S) IMPLIES (

constant EquivClosLeast
type {S: PRED [(T × T)], R: PRED [(T × T)]} ⊢ (equivalence? [T] (S) AND (R subset? S)) IMPLIES (Equi

constant EquivClosIdempotent
type {R: PRED [(T × T)]} ⊢ EquivClos (EquivClos (R)) = EquivClos (R)

constant EquivalenceCharacterization
type {S: PRED [(T × T)]} ⊢ equivalence? [T] (S) IFF (S = EquivClos (S))

Figure 13.1: The PVS Prelude in the Mmt Browser

In both workflows, OMDoc/Mmt-based periphery systems become available to the PVS user that are either not provided by the PVS tools or in a much restricted way. We will go over the three most important ones in detail.

13.1.1 Browsing and Interaction

The transformed PVS content can be browsed interactively in the document-oriented MathHub presentation pages (theories as active documents) and in the Mmt web browser (see Figure 13.1). Both allow interaction with the PVS content via a generic Javascript-based interface. This provides buttons to toggle the visibility of parts computed by PVS – e.g. omitted types and definitions – at the declaration level. The right-click menu shown in Figure 13.1 is specific to the selected sub-formula (highlighted in gray); here we have eight applicable interactions which range from inferring the subformula type via definition lookup to management actions such as registering an alignment to concepts in other libraries. New interactions can be added as they become available in the MMT system.

The Mmt instance in the local workflow provides the additional feature of inter-process communication between PVS and Mmt as a new menu item: the action *navigate to this declaration in connected systems*. Florian Rabe implemented a listener for this action that forwards the command to PVS via an XML-RPC call at the default PVS port. Correspondingly, Sam Owre implemented an experimental handler in the PVS server that opens the corresponding file in the PVS emacs system and navigates to the relevant line – unfortunately, due to time constraints this functionality has so far not been fully developed and integrated in an official PVS release.

13.1.2 Graph Viewer

MathHub includes a theory graph viewer developed by Marcel Rupprecht that allows interactive, web-based exploration of the OMDoc/Mmt theory graphs [RKM17]. It builds on the `visjs` JavaScript visualization library [VJS], which uses the HTML5 canvas to layout and interact with graphs client-side in the browser.

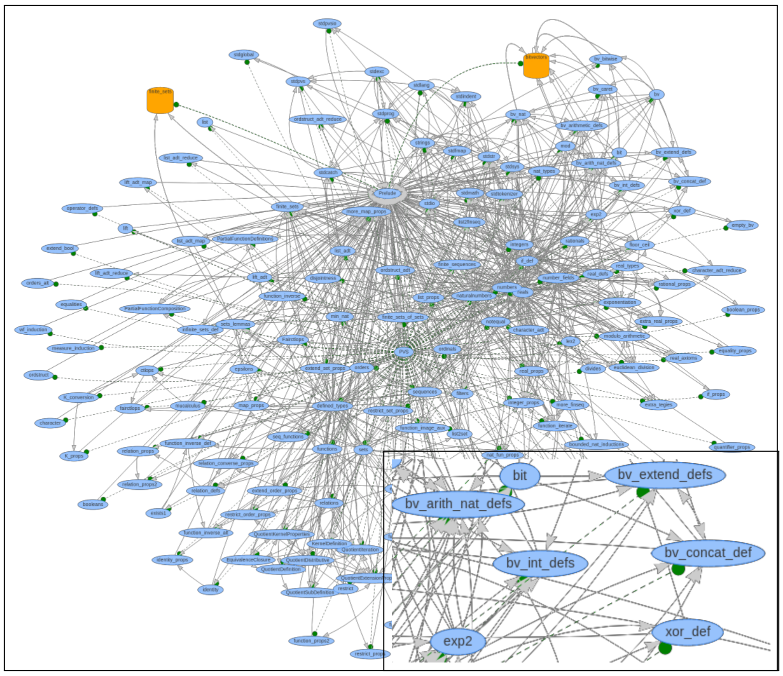


Figure 13.2: The Basic PVS Libraries in the MathHub Theory Graph Viewer

PVS libraries make heavy use of theories as a structuring mechanism, which makes a graph viewer for PVS particularly attractive. Figure 13.2 shows the full graph in a central-gravity layout induced by the PVS prelude, where we have (manually) clustered the subgraphs for bit vectors and finite sets (the orange barrel-shaped nodes). The lower right corner shows a zoomed-in fragment.

The theory graph allows dragging nodes around to fine-tune the layout. Hovering over a node or edge triggers a preview of the theory. All nodes support the same context menu actions in the graph viewer as the corresponding theories do in the browser above. Thus, it is possible to select a theory in the graph viewer and then navigate to it in the browser or (if run locally) in the PVS system.

13.1.3 Search

MathWebSearch [KŞ06] is an OMDoc/Mmt-level formula search engine that uses query variables for subterms and first-order unification as the query language. It is developed independently, but Mmt includes a plugin for generating MathWebSearch index files using its content MathML interface. Thus, any library available to MMT can be indexed and searched via MathWebSearch. Moreover, MMT includes a frontend for MathWebSearch so that search queries can be supplied in any format that MMT can understand, e.g., the XML format produced by PVS.

```
<mws:query limitmin="0" answsize="1000" totalreq="yes"
  output="xml" xmlns:m="http://www.w3.org/1998/Math/MathML"
  xmlns:mws="http://www.mathweb.org/mws/ns">
 <mws:expr>
  <apply>
   <csymbol>http://cds.omdoc.org/urtheories?LambdaPi?apply</csymbol>
   <csymbol>http://pvs.csl.sri.com/?PVS?pvsapply</csymbol>
   <mws:qvar xmlns:mws="http://www.mathweb.org/mws/ns">I1</mws:qvar>
   <mws:qvar xmlns:mws="http://www.mathweb.org/mws/ns">I2</mws:qvar>
   <csymbol>http://pvs.csl.sri.com/prelude?EquivalenceClosure?EquivClos</csymbol>
   <mws:qvar xmlns:mws="http://www.mathweb.org/mws/ns">A</mws:qvar>
  </apply>
 </mws:expr>
</mws:query>
```

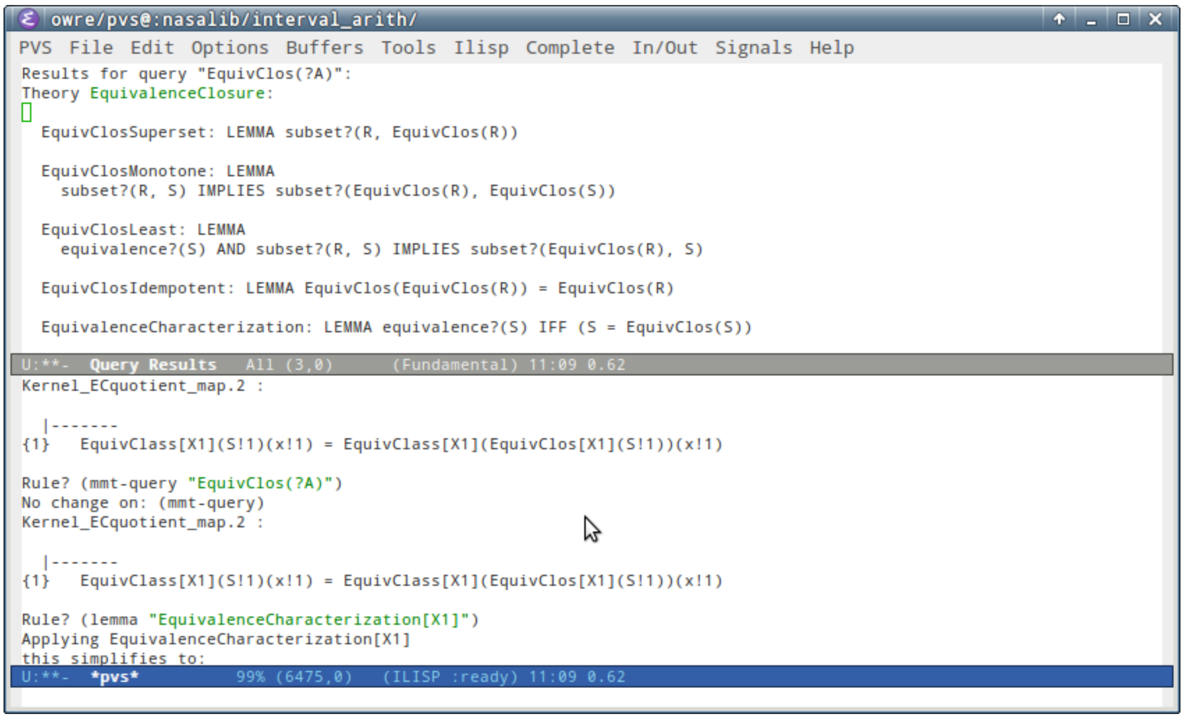
Figure 13.3: A Query for Applications of EquivClos

MMT exposes the search frontend both in its GUI for humans and as an HTTP service for other systems. Here we use the latter: We have added a feature to the PVS emacs interface that allows users to enter a search query in PVS syntax. PVS parses the query, type-checks it, and converts it to XML. The XML is sent to MMT, which acts as the mediator between the proof assistant — here PVS — and library management periphery — here MathWebSearch— and returns the search results to PVS.

```
[ {"lib_name" : "",
   "theory_name" : "EquivalenceClosure",
   "name" : "EquivClosMonotone",
   "Position" : "3_2_5_5_5_5"},
  {"lib_name" : "",
   "theory_name" : "EquivalenceClosure",
   "name" : "EquivalenceCharacterization",
   "Position" : "2_2_5_5_5_5"},
  ...
]
```

Figure 13.4: A Query Result for Figure 13.3

The PVS user enters the PVS query `EquivClos(?A)`, where we have extended the PVS syntax with query variables like `?A`. After OMDoc/MMT translation, this becomes the MathWebSearch query in Figure 13.3 — note the additional symbols from LF introduced by the representation in the logical framework. The representation also introduces unknown meta-variables for the domain and range of the `EquivClos` function, which become the additional query variables I1 and I1. MathWebSearch returns a JSON record with all results, and we show the first two in Figure 13.4: two occurrences of (instances of) `EquivClos(?A)` in two declarations in the theory `EquivalenceClosure` Figure 10.1. The attribute lib_name is the name of the library; by PVS convention, it is empty for the Prelude. The attributes theory_name and name give the declaration that contains the match, and Position gives the path to its subterm that matched the query.


owre/pvs@:nasalib/interval_arith/
PVS File Edit Options Buffers Tools Ilisp Complete In/Out Signals Help

```
Results for query "EquivClos(?A)":
Theory EquivalenceClosure:

  EquivClosSuperset: LEMMA subset?(R, EquivClos(R))

  EquivClosMonotone: LEMMA
    subset?(R, S) IMPLIES subset?(EquivClos(R), EquivClos(S))

  EquivClosLeast: LEMMA
    equivalence?(S) AND subset?(R, S) IMPLIES subset?(EquivClos(R), S)

  EquivClosIdempotent: LEMMA EquivClos(EquivClos(R)) = EquivClos(R)

  EquivalenceCharacterization: LEMMA equivalence?(S) IFF (S = EquivClos(S))

U:**-  Query Results   All (3,0)      (Fundamental) 11:09 0.62
Kernel_ECquotient_map.2 :

  |-------
{1}   EquivClass[X1](S!1)(x!1) = EquivClass[X1](EquivClos[X1](S!1))(x!1)

Rule? (mmt-query "EquivClos(?A)")
No change on: (mmt-query)
Kernel_ECquotient_map.2 :

  |-------
{1}   EquivClass[X1](S!1)(x!1) = EquivClass[X1](EquivClos[X1](S!1))(x!1)

Rule? (lemma "EquivalenceCharacterization[X1]")
Applying EquivalenceCharacterization[X1]
this simplifies to:
U:**-  *pvs*          99% (6475,0)   (ILISP :ready) 11:09 0.62
```

Figure 13.5: Example for Displaying the Query Result in PVS

Figure 13.5 shows what the query will look like while doing a PVS proof. The current implementation is just a proof-of-concept — for the mature version the part of PVS that sends the query to the MMT server and displays the results still has to be implemented thoroughly. But the remaining steps are straightforward.

Future work could potentially exploit this functionality to search specifically for existing theorems that may be helpful in a specific part of an ongoing PVS proof.

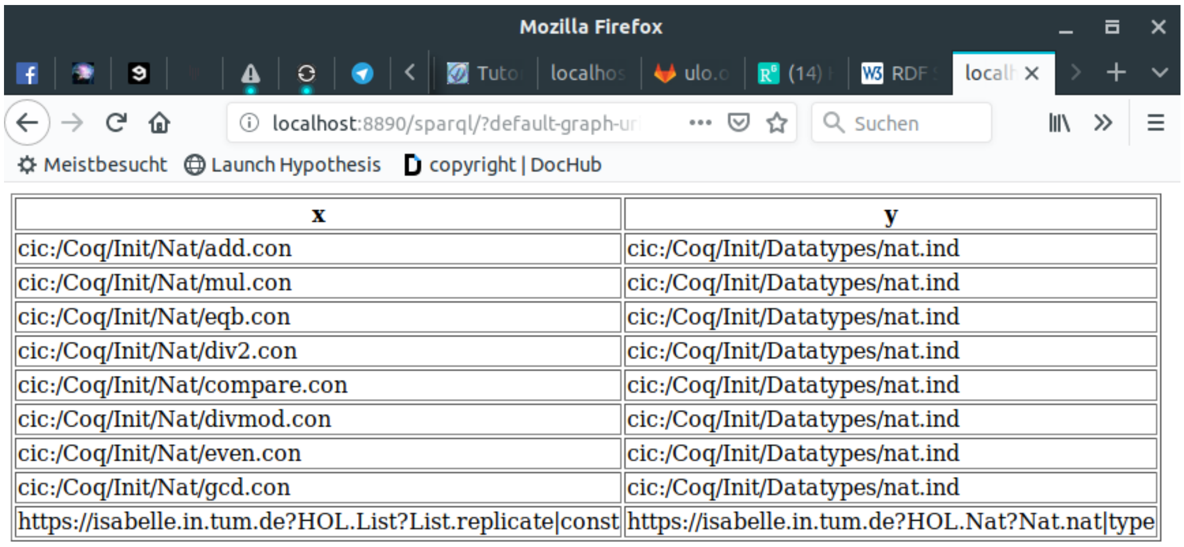
13.1.4 Querying Across Libraries

Going beyond a single system, we exported relational data for several theorem prover libraries as RDF triples for more advanced queries without relying on the MMT system (as published in [Con+]). This relational data is already contained in the underlying OMDoc representation of MMT archives and can thus be easily generated from imported libraries. In conjunction with alignments (which we will discuss in Chapter 14) this allows for conveniently querying for mathematical concepts across different theorem prover libraries using an arbitrary triple store as backend. For example, for the cited paper I set up an instance of Virtuoso Open-Source[2], providing a SPARQL endpoint. Using the query

```
SELECT ?x ?y WHERE {
  ?x ulo:inductive-on ?y .
  http://mathhub.info/MitM/Foundation?NatLiterals?nat_lit ulo:aligned-with ?y . }
```

we receive as results (which are shown in Figure 13.6) all functions and predicates defined by induction on the natural numbers regardless of implementation - by only considering those

[2] https://github.com/openlink/virtuoso-opensource



x	y
cic:/Coq/Init/Nat/add.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/mul.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/eqb.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/div2.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/compare.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/divmod.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/even.con	cic:/Coq/Init/Datatypes/nat.ind
cic:/Coq/Init/Nat/gcd.con	cic:/Coq/Init/Datatypes/nat.ind
https://isabelle.in.tum.de?HOL.List?List.replicate|const	https://isabelle.in.tum.de?HOL.Nat?Nat.nat|type

Figure 13.6: Virtuoso Output for the Example Query using Alignments

symbols that are aligned with the Math-in-the-Middle symbol for the type of natural numbers.

Part IV

Cross-Library Knowledge Management

Now that we have a unifying framework in which many different libraries are represented, we can approach the problem of integrating these libraries with each other, e.g. to translate knowledge between them.

Naturally, these libraries use different foundations; which is reflected in MMT as the contained theories having different meta theories. Consequently, as a first step one might want to integrate the foundations themselves, i.e. by implementing systematic translations between them.

An obvious challenge arises here, namely that the foundations used by formal systems are usually mutually incompatible. This is obvious for systems using fundamentally different logics, such as set theories vs. type theories, but even if the *platonic* foundation used by two systems is the same (e.g. higher-order logic), there are subtle differences in the specific implementations – and hence ontologies – of the foundations, that make implementing such translations less straight-forward than one might at first suspect. For example, the systems might offer different additional features (e.g. records, subtyping) where there is no universally valid elimination procedure. Additionally, systems usually offer distinct convenience operations, automations, module systems and conventions, that are suggestive of different approaches and best practices for formalizing content, leading to very different library developments for the same mathematical concept even if the foundation used by the systems is the same.

As a result, even when a translation between foundations is straight-forward (which it usually is not), the result of translating content between libraries *on the basis of* a translation between the foundations can yield awkward reformalizations of known content completely disconnected from the content already available in the target library.

Consequently, any generic methodology for integrating libraries needs to go beyond – or even entirely bypass – mere foundations and take distinct library developments into account. This part of this thesis presents our approach to library integration on that premise.

First, we introduce the notion of *alignments* in Chapter 14. This is a binary relation between symbols expressing that the two symbols (possibly from different libraries) denote the same abstract mathematical concept. In the simplest case, we can then translate symbol occurences in expressions across libraries by merely substituting along alignments. We then classify various more complex types of alignments (e.g. "up to argument order" or "up to additional arguments") that occur in practice and develop a convenient way to specify them.

Chapter 15 generalizes this approach to translating expressions; using alignments wherever possible, but additionally using theory morphisms (within or across libraries) and – as is crucial for translating across different foundations – programmatic translations, which can be supplied for more complex situations.

To avoid having to provide ad-hoc traslation mechanisms between all pairs of libraries, we instead try to use translations between any library and a fixed set of *interface theories*, in line with the Math-in-the-Middle approach (see Section 3.4). In a library of interface theories, we can implement all possible developments and definitions of the same mathematical concept and connect them via theory morphisms, lifting the more complicated translation problems to a realm governed by a single, flexible foundation using the logical frameworks developed in Part II. Not surprisingly, this methodology is used to implement the functionalities aimed at by the OPENDREAMKIT project (see Section 3.3) – Section 15.3 shows a real-world use case from the OPENDREAMKIT community that is covered by our approach.

In Chapter 16, we describe how to automatically find theory morphisms; both within and across different libraries. These can be used to identify overlap between libraries and consequently suggest new alignments and translations between the theories; ideally resulting in a positive feedback loop of ever increasing connections between libraries.

Lastly, Chapter 17 develops a method for using theory morphisms to refactor library content to increase modularity within a given library.

Chapter 14

Alignments

Disclaimer:

The contents of this chapter have largely been previously published as [Mül+17b] (Extended in [Mül+17c]), [Kal+16] and [Mül+17a] with coauthors Cezary Kaliszyk, Thibault Gauthier, Michael Kohlhase, Florian Rabe, Colin Rothgang and Yufei Liu. Both the writing as well as the theoretical results were developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors. The writing has been reworked for this thesis.

My contribution in this chapter consists of the actual implementations presented in Section 14.4 and the practical evaluation of the concept.

The sciences are increasingly collecting and curating their knowledge systematically in machine-processable corpora. For example, in biology many important corpora take the form of ontologies, e.g., as collected on BioPortal. These corpora typically overlap substantially, and much recent work has focused on integrating them. A central problem here is to find *alignments*: pairs (a_1, a_2) of identifiers from different corpora that describe the same concept, giving rise to *ontology matching* [ESC07].

In the certification of programs and proofs, the ontology matching problem is most apparent when trying to use multiple reasoning systems together. For example, Wiedijk [Wie06] explored a single theorem (and its proof) across 17 proof assistants implicitly generating alignments between the concepts present in the theorem's statement and proof. The Why3 system [Bob+11] maintains a set of translations into different reasoning systems for discharging proof obligations. Each translation must manually code individual alignments of elementary concepts such as integers or lists in order to fully utilize the respective system's automation potential. But automating the generation and use of alignments, which would be necessary to scale up such efforts, is challenging because the knowledge involves rigorous notations, definitions, and properties, which leads to very diverse corpora with complex alignment options. This makes it very difficult to determine whether an alignment is *perfect* (we will attempt to define this notion in the next section), or to predict whether an *imperfect* alignment will work just as well or not at all.

Alignment Use Cases Many practical services are enabled by alignments:

- Simultaneous browsing of multiple corpora. This is already enabled (so far for a limited number of corpora) by our system presented in Section 14.4.

- Imperfect alignments can be used to search for a single query expression in multiple corpora at once. This has been demonstrated in the Whelp search engine [Asp+06b] where Coq and Matita shared the URI syntax with the basic Calculus of Constructions constants aligned, as well as by the MathWebSearch engine [KR14] and a recent alignment-aware standardization of relational information (see [Con+] and Section 13.1.4).

- Statistical analogies extracted from large formal libraries combined with imperfect alignments can be used to create new conjectures and thus to automatically explore a logical corpus [GKU16]. This complements the more classical conjecturing and theory exploration mechanisms.

- Automated reasoning services can make use of alignments to provide more precise proof recommendations. The quality of the HOL(y) Hammer proof advice for HOL Light can be improved from 30% to 40% by using the imperfect alignments to HOL4 [GK15].

- Translations between systems. [KK13b] uses more than 70 manually discovered alignments between HOL Light and Isabelle/HOL to obtain translated theorems that talk about target system constants and types. Note, that it is not necessary for the translation for the definitions of the concepts to be the same. It is enough if the same properties are provable or if they yield the same computational behavior. Consider the real numbers: In some HOL proof assistants they are defined using Cauchy sequences, while others use Dedekind cuts. The two structures share all the relevant real number properties. However, they disagree with respect to irrelevant properties, e.g., in the construction of the former usually a canonical Cauchy sequence for each real number is introduced. Despite this minor difference, we can use such an alignment in a logical translation [KK13b].

  Translations along alignments will be covered in Chapter 15.

- Refactoring of proof assistant corpora. Aligning concepts across versions of the same proof corpus combined with statement normalization and consistent name hashing allowed discovering 39 symbols with equivalent definitions [KU15] in the Flyspeck development [Hal+15b].

  One such refactoring technique will be covered in Chapter 17.

Finding Alignments Even though we know that numerous alignments exist between libraries, not many alignments are known concretely or have been represented explicitly. Therefore, a major initial investment is necessary to obtain a large library of interface theories and alignments. There are three groups of alignment-finding approaches.

Human-based approaches examine libraries and manually identify alignments. This approach has been pursued ad hoc in various contexts. For example, the library translation of [OS06b] included some alignments for HOL Light and Isabelle/HOL, which were later expanded by Kaliszyk. The Why3 and FoCaLiZe systems include alignments to various theorem provers that they use as backends. Results from this approach will be discussed in Section 14.5.

The remaining two classes use *machine learning* methods.

Logical approaches align two concepts if they satisfy the same theorems. This cannot directly appeal to logical equivalence because that would require a translation between the corpora. Instead, they compare the theorems that are explicitly stated in the corpora. The theorems should be normalized first to eliminate differences that do not affect logical equivalence. The quality of the results depends on how completely those theorems characterize the respective concepts. In well-curated libraries, this can be very high [MK15]. The Viewfinder presented in Chapter 16 can be interpreted as implementing this approach.

Machine learning–based approaches are inherently based on statistical patterns and hence naturally inexact. The main research in this direction is carried out by Kaliszyk and others [GK14b].

Automatic Search for Alignments Finding alignments, preferably automatically, has proved extremely difficult in general. There are three reasons for this: the conceptual differences between logical corpora found in proof assistants, computational corpora containing algorithms from computer algebra systems, narrative corpora that consist of semi-formal descriptions from wiki-related tools; the diversity of the underlying formal languages and tools; and the differences between the organization of the knowledge in the corpora.

While finding alignments automatically is promising for *perfect alignments*, where two symbols only differ in name but are otherwise used identically, it is a different matter for *imperfect* alignments. For example, consider binary division which yields undefined or 0 when the divisor is zero, versus a strict division that requires an additional proof-argument that the divisor is nonzero. Here automation becomes much more difficult, because the imperfections often violate the conditions that an automatic approach uses to spot alignments.

I conjecture that the quality of artificial intelligence approaches will be massively improved if they are applied on top of a large set of guaranteed-perfect alignments, besides their obvious use as training data. This is based on two observations:

- The more alignments we know, the easier it is to find new ones. Consider a typical formalization in system S_1 that introduces a new concept c relative to some known ones. If perfect alignments to system S_2 for the known concepts have already been established, it becomes relatively easy to find the formalization of c in S_2, and add an alignment for it.
- It is very difficult to get alignment-finding off-the-ground. Because the foundations of S_1 and S_2 are often very different, almost nothing looks particularly similar in the absence of any alignments. Deeper alignments will become more apparent only when alignments for fundamental concepts such as booleans, sets and numbers are established.

Recently, heuristic methods for automatically finding alignments were developed [GK14b], targeted at integrating logical corpora, which were integrated into our developments in Section 14.1. Independently, Deyan Ginev built a library [GC14] of about 50,000 alignments between narrative corpora including Wikipedia, Wolfram Mathworld, PlanetMath and the SMGloM semantic multilingual glossary for mathematics. For this, the NNexus system indexes the corpora and applies clustering algorithms to discover concepts.

Related Work Alignments between computational corpora occur in bridges between the run time systems of programming languages. Alignments between logical and computational corpora are used in proof assistants with code generation such as Isabelle [WPN08] and Coq [Coq15]. Here functions defined in the logic are aligned with their implementations in the programming language in order to generate fast executable code from formalizations.

The dominant methods for integrating logical corpora so far have focused on truth-preserving translations between the underlying knowledge representation languages. For example, [KS10] translates from Isabelle/HOL to Isabelle/ZF. [KW10] translates from HOL Light to Coq, [OS06a] to Isabelle/HOL, and [NSM01] to Nuprl. Older versions of Matita [Asp+06a] were able to read Coq compiled theory files. [Cod+11] build a library of translations between different logics.

However, as mentioned in the introduction to this part, most translations are not alignment-aware, i.e., it is not guaranteed that a_1 will be translated to a_2 even if the alignment is known. This is because a_1 and a_2 may be subtly incompatible so that a direct translation may even lead to inconsistency or ill-typed results. [OS06a] was — to my knowledge — the first that could be parametrized by a set of alignments. The OpenTheory framework [Hur09] provides a number of higher-order logic concept alignments. In [KR16a], the corpus integration problem is discussed and concluded that alignments are of utmost practical importance. Indeed, corpus integration can succeed with only alignment data even if no logic translation is possible. Conversely, logic translations contribute little to corpus integration without alignment data.

14.1 Types of Alignments

Let us assume two corpora C_1, C_2 with underlying foundational logics F_1, F_2. We examine examples for how two concepts a_i from C_i can be aligned. Importantly, we allow for the case where a_1 and a_2 represent the same abstract mathematical concept without there being a direct, rigorous translation between them.

The types of alignments in this section are purely phenomenological in nature: they exemplify the difficulty of the problem and provide benchmarks for rigorous definitions. While some types are relatively straightforward, others are so difficult that giving a rigorous definitions remains an open problem. This is because alignments ideally legitimize translations from F_1 to F_2 that replace a_1 with a_2. But in many situations these translations, while possible in principle, are much more difficult than simply replacing one symbol with another. The alignment types below are roughly ordered by increasing difficulty of this translation.

Perfect Alignment If a_1 and a_2 are logically equivalent (modulo a translation φ between F_1 and F_2 that is fixed in the context), we speak of a perfect alignment. More precisely, all formal properties (type, definition, axioms) of a_1 carry over to a_2 and vice versa. Typical examples are primitive types and their associated operations. Consider:

$$\texttt{Nat}_1 : \texttt{Type} \qquad \texttt{Nat}_2 : \texttt{Type}$$

then translations between C_1 and C_2 can simply interchange a_1 and a_2.

The above example is deceptively non-trivial for two reasons. Firstly, it hides the problem that F_1 and F_2 do not necessarily share the symbol `Type`. Therefore, we need to assume that there are symbols `Type`$_1$ and `Type`$_2$, which have been already aligned (perfectly). Such alignments (on the foundations of C_1 and C_2) are crucial for all fundamental constructors that occur in the types and characteristic theorems of the symbols we want to align such as Type, $\to$, bool, $\wedge$, etc. These alignments can be handled with the same methodology as discussed here. Therefore, here and below, we assume we have such alignments and simply use the same fundamental constructors for F_1 and F_2.

Secondly, it ignores that we usually want (and can reasonably expect) only certain formal properties to carry over, namely those in the *interface theory* in the sense of [KR16a] — i.e. those properties that are still meaningful after abstracting away from the specific foundational logics F_i. We will look at some examples for perfect alignments between symbols that use different but equivalent definitions in Section 14.2.

Alignment up to Argument Order Two function symbols can be perfectly aligned except that their arguments must be reordered when translating.

The most common example is function composition, whose arguments may be given in application order $(g \circ f)$ or in diagram order $(f; g)$. Another example is given

$$\begin{array}{l}\texttt{contains}_1 : (\texttt{T} : \texttt{Type}) \to \texttt{SubSet}\,\texttt{T} \to \texttt{T} \to \texttt{bool} \\ \texttt{in}_2 : (\texttt{T} : \texttt{Type}) \to \texttt{T} \to \texttt{SubSet}\,\texttt{T} \to \texttt{bool}\end{array}$$

Here the expressions $\texttt{contains}_1(\texttt{T}, \texttt{A}, \texttt{x})$ and $\texttt{in}_2(\texttt{T}, \texttt{x}, \texttt{A})$ can be translated to each other.

Alignment up to Determined Arguments The perfect alignment of two function symbols may be broken because they have different types even though they agree in most of their properties. This often occurs when F_1 uses a more fine-granular type system than F_2, which requires additional arguments.

Examples are untyped and typed (polymorphic, homogeneous) equality: The former is binary, while the latter is ternary

$$\begin{array}{l}\texttt{eq}_1 : \texttt{Set} \to \texttt{Set} \to \texttt{bool} \\ \texttt{eq}_2 : (\texttt{T} : \texttt{Type}) \to \texttt{T} \to \texttt{T} \to \texttt{bool}\end{array}.$$

The types can be aligned, if we apply eq_2 to $\varphi(\mathsf{Set})$. Similar examples arise between simply- and dependently-typed foundations, where symbols in the latter take additional arguments.

These additional arguments are uniquely determined by the values of the other arguments, and a translation from C_1 to C_2 can drop them, whereas the reverse translations must infer them – but F_1 usually has functionality for that (e.g. the type parameter of polymorphic equality is usually uniquely determined).

The additional arguments can also be proofs, used for example to represent partial functions as total functions, such as a binary and a ternary division operator

$$\begin{array}{l}\texttt{div}_1 : \texttt{Real} \to \texttt{Real} \to \texttt{Real} \\ \texttt{div}_2 : \texttt{Real} \to (\texttt{d} : \texttt{Real}) \to \text{DED}\ \texttt{d} \neq 0 \to \texttt{Real}\end{array}$$

Here inferring the third argument is undecidable in general, and it is unique only in the presence of proof irrelevance.

Alignment up to Totality of Functions The functions a_1 and a_2 can be aligned everywhere where both are defined. This happens frequently since it is often convenient to represent partial functions as total ones by assigning values to all arguments. The most common example is division. Two implementations div_1 and div_2 might both have the type $\mathsf{Real} \to \mathsf{Real} \to \mathsf{Real}$ with $x\,\mathsf{div}_1\,0$ undefined and $x\,\mathsf{div}_2\,0 = 0$.

Here a translation from C_1 to C_2 can always replace div_1 with div_2. The reverse translation can usually replace div_2 with div_1 but not always. In translation-worthy data-expressions, it is typically sound; in formulas, it can easily be unsound because theorems about div_2 might not require the restriction to non-zero denominators.

Alignment for Certain Arguments Two function symbols may be aligned only for certain arguments. This occurs if a_1 has a smaller domain than a_2.

The most fundamental case is the function type constructor $\to$ itself. For example, $\to_1$ may be first-order in F_1 and $\to_2$ higher-order in F_2. Thus, a translation from C_1 to C_2 can replace $\to_1$ with $\to_2$, whereas the reverse translation must be partial.

Another important class of examples is given by subtyping (or the lack thereof). For example, we could have

$$\begin{array}{l}\texttt{plus}_1 : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \\ \texttt{plus}_2 : \texttt{Real} \to \texttt{Real} \to \texttt{Real}\end{array} .$$

Alignment up to Associativity An associative binary function (either logically associative or notationally right- or left-associative) can be defined as a flexary function, i.e., a function taking an arbitrarily long sequence of arguments. In this case, translations must fold or unfold the argument sequence. For example

$$\texttt{plus}_1 : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \qquad \texttt{plus}_2 : \texttt{List}\,\texttt{Nat} \to \texttt{Nat} .$$

All of the above types of alignments allow us to translate expressions between our corpora by modifying the lists of arguments the respective symbols are applied to, even if not always in a straight-forward way. The following types of alignments are more abstract, and any translation along them might be more dependent on the specifics of the symbols under consideration.

Contextual Alignments Two symbols may be aligned only in certain contexts. For example, the complex numbers are represented as pairs of real numbers in some proof assistant libraries and as an inductive data type in others. Then only selected occurrences of pairs of real numbers can be aligned with the complex numbers.

Alignment with a Set of Declarations Here a single declaration in C_1 is aligned with a set of declarations in C_2. An example is a conjunction a_1 in C_1 of axioms aligned with a set of single axioms in C_2. More generally, the conjunction of a set of C_1-statements may be equivalent to the conjunction of a set of C_2-statements.

Here translations are much more involved and may require aggregation or projection operators.

Alignment between the Internal and External Perspective on Theories When reasoning about complex objects in a proof assistant (such as algebraic structures, or types with comparison) it is convenient to express them as theories that combine the actual type with operations on it or even properties of such operations. The different proof assistants often have incompatible mechanisms of expressing such theories including type classes, records and functors, with the additional distinction whether they are first-class objects or not. This roughly corresponds to the distinction between *stratified* and *integrated* groupings in Chapter 8.

We define the crucial difference for alignments here only by example. We speak of the internal perspective ($\approx$ stratified grouping) if we use a theory like

$$\mathsf{theory}\ \mathsf{Magma}_1 = \{\mathsf{u}_1 : \mathsf{Type},\ \circ_1 : \mathsf{u}_1 \to \mathsf{u}_1 \to \mathsf{u}_1\}$$

and of the external perspective (≈ integrated grouping) if we use operations like

$$\mathsf{Magma}_2 : \mathsf{Type},\ \mathtt{u}_2 : \mathsf{Magma}_2 \to \mathsf{Type},$$

$$\circ_2 : (\mathtt{G} : \mathsf{Magma}) \to \mathtt{u}_2\,\mathtt{G} \to \mathtt{u}_2\,\mathtt{G} \to \mathtt{u}_2\,\mathtt{G}$$

Here we have a non-trivial, systematic translation from C_1 to C_2. A reverse may also be possible, depending on the details of F_1.[1]

Corpus-Foundation Alignment Orthogonal to all of the above, we have to consider alignments, where a symbol is primitive in one system but defined in another. More concretely, a_1 can be built-into F_1 whereas a_2 is defined in F_2. This is common for corpora based on significantly different foundations, as each foundation is likely to select different primitives. Therefore, it mostly occurs for the most basic concepts. For example, the boolean connectives, integers and strings are defined in some systems but primitive in others, as in some foundations they may not be easy to define.

The corpus-foundation alignments can be reduced to previously considered cases if we follow the approach generally followed in this thesis, where the foundations themselves are represented in an appropriate logical framework. Then a_1 is simply an identifier in the corpus of foundations of the framework F_1.

Opaque Alignments The above alignments focused on logical corpora, partially because logical corpora allow for precise and mechanizable treatment of logical equivalence. Indeed, alignments from a logical into a computational or narrative corpus tend to be opaque: Whether and in what way the aligned symbols correspond to each other is not (or not easily) machine-understandable. For example, if a_2 refers to a function in a programming language library, that function's specification may be implicit or given only informally. Even worse, if a_2 is a wiki article, it may be subject to constant revision.

Nonetheless, such alignments are immensely useful in practice and should not be discarded. Therefore, we speak of opaque alignments if a_2 refers to a symbol whose semantics is unclear to machines.

Probabilistic Alignments Orthogonal to all of the above, the correctness of an alignment may be known only to some degree of certainty. In that case, we speak of probabilistic alignments. These occur in particular when machine-learning techniques are used to find large sets of alignments automatically. This is critical in practice to handle the existing large corpora.

The problem of probabilistically estimating the similarity of concepts in different corpora was studied before in [GK14b]. I briefly restate the relevant aspects in our setting, as described by Thibault Gauthier in [Mül+17b].

Let T_i be the set of top level expressions occurring in C_i, e.g., the types of all constants and the formulas of all theorems. We assume a fixed set F of alignments, covering in particular the foundational concepts in F_1 and F_2.

[1]In MMT, we can bridge these two representations using the Mod-types described in Chapter 8

Definition 14.1:

The pattern $P(f)$ of an expression f is obtained by normalizing f to $N(f)$ and abstracting over all occurrences of concepts that are not in F, resulting in $P(f) = \lambda c_1 \dots c_n.\ N(f)$. If two formulas $f \in T_1$ and $g \in T_2$ have α-equivalent patterns $\lambda d_1 \dots d_m.\ N(g)$ and $\lambda e_1 \dots e_m.\ N(h)$, we define their induced alignments by $I(f,g) = \{(d_1, e_1), \dots, (d_m, e_m)\}$. We write $J(p)$ for the union of all $I(f,g)$ with $P(f) =_\alpha P(g) =_\alpha p$.

Example 14.1:
For the formula $\forall x.\ x = 2 \cdot \pi \Rightarrow cos(x) = 0$ with F not covering the concepts 2, π, 0, and cos, and using a normal form N that exploits the symmetry of equality, we get the pattern $\lambda c_1\ c_2\ c_3\ c_4.\ \forall x.\ x = c_1 \cdot c_2 \Rightarrow c_3 = c_4(x)$.

Let $a_1, \dots, a_n$ be the set of all alignments in any $J(p)$. We first calculate an initial vector containing the similarities sim_i for each a_i by

$$sim_i = \sum_{\{p \mid a_i \in J(p)\}} \frac{1}{ln(2 + card\ \{\ f \mid P(f) = p\ \})}$$

Intuitively, an alignment has a high similarity value if it was produced by a large number of rare patterns.

Secondly, we iteratively transform this vector until its values stabilize. The idea behind this dynamical system is that the similarity score of an alignment should depend on the quality of its co-induced alignments. Each iteration step consists of two parts: we multiply the vector with the matrix

$$cor_{kl} = card\ \{\ (f,g) \mid\ a_k \in I(f,g) \wedge a_l \in I(f,g)\ \}$$

which measures the correlation between a_k and a_l, and then (in order to ensure convergence and squash all values into the interval $[0;1]$) apply the function $x \mapsto \frac{x}{x+1}$ to each component.

14.2 Examples of Alignments

An essential requirement for relating logical corpora is standardizing the identifiers so that each identifier in the corpus can be uniquely referenced. It is desirable to use a uniform naming schema so that the syntax and semantics of identifiers can be understood and implemented as generically as possible. MMT URIs have been specifically designed for that purpose, and Part III shows by example how to systematically generate URIs during importing a library. Additionally, in [Mül+17c] we present URI schemas for a list of selected theorem provers.

Using these URIs with abbreviated proof assistant names, the following presents alignments across proof assistants for two representative concepts. Also included are some alignments to programming languages, which are relevant for code generation. Thousands of other alignments, both perfect and imperfect, can be explored on mathhub [PRA].

Cartesian Product In constructive type theory, there are two common ways of expressing the non-dependent Cartesian product. First, if the foundation has inductive types such as the Calculus of Inductive Constructions, it can be an inductive type with one binary constructor. This is the case for:

- `Coq ? Init/Datatypes ? prod.ind`
- `Matita ? datatypes/constructors ? Prod.ind`

Second, if the foundation defines a dependent sum type, it is possible to express the Cartesian product as its non-dependent special case:

- `Isabelle ? CTT/CTT ? times`

In higher-order logic, the only way to introduce types is by using the `typedef` construction, which constructs a new type that is isomorphic to a certain subtype of an existing type. In particular, most HOL-based systems introduce the Cartesian product $A \times B$ by using a unary predicate on $A \to B \to \mathsf{bool}$:

- `HOLLight ? pair/type ? prod`
- `HOL4 ? pair/type ? prod`
- `Isabelle ? HOL/Product ? prod`

In set theory, it is also possible to restrict dependent sum types to obtain the Cartesian product. This approach is used in Isabelle/ZF:

- `Isabelle ? ZF/ZF ? cart_prod`

In Mizar the Cartesian product is defined as functor in first-order logic. The definition invloves discharging the well-definedness condition. We defined functor is:

- `Mizar ? ZFMISC_1 ? K2`

In PVS, the product type constructor is part of the system foundation:

- `PVS ? foundation.PVS ? tuple_tp`

Cartesian products appear also in most programming languages and the code generators of various proof assistants do use a number of these:

- `OCaml ? core ? *`
- `Haskell ? core ? ,`
- `Scala ? core ? ,`
- `CPP ? std ? pair`

Informal sources that can be aligned are e.g.:

- https://en.wikipedia.org/wiki/Cartesian_product
- https://en.wikipedia.org/wiki/Product_type
- http://mathworld.wolfram.com/CartesianProduct.html

Concatenation of Lists In constructive type theory (e.g. for Matita, Coq), the append operation on lists can be defined as a fixed point. In higher-order logic, append for polymorphic lists can be defined by primitive recursion, as done by HOL Light and HOL4. Isabelle/HOL slightly differs from these two because it uses lists that were built with the co-datatype package [Bla+14]. PVS and Isabelle/ZF also use primitive recursion for monomorphic lists. In Mizar, lists are represented by finite sequences, which are functions from a finite subset of natural numbers (one-based `FINSEQ` and zero-based `XFINSEQ`) with append provided. Concatenation of lists is also common in programming languages.

- `Coq ? Init/Datatypes ? app`
- `HOLLight ? lists/const ? APPEND`
- `HOL4 ? list/const ? APPEND`
- `Isabelle ? HOL/List ? append`
- `PVS?Prelude.list_props?append`
- `Isabelle ? ZF/List_ZF ? app`
- `Mizar ? ORDINAL4 / K1`
- `OCaml ? core ? @`

- `Haskell ?  core ?  ++`
- `Scala ?  core ?  ++`

14.3 A Standard Syntax for Alignments

Based on the observations of the previous sections, we now define a standard for alignments. Because many of the alignment types described in Section 14.1 are very difficult to handle rigorously and additional alignment types may be discovered in the future, we opt for a very simple and flexible definition.

Concretely, we use the following formal grammar for collections of alignments:

Collection	::=	(Comment | NSDef | Alignment)*
Comment	::=	// String
NSDef	::=	`namespace` String URI
Alignment	::=	URI URI (String = "String")*

Our definition aims at practicality, especially considering the typical case where researchers exchange and manipulate large collections of alignments. Therefore, our grammar allows for comments and for the introduction of short namespace definitions that abbreviate long namespaces. Our grammar represents each individual alignment as a pair of two URIs with arbitrary additional data stored as a list of key-value pairs.

The additional data in alignments makes our standard extensible: any user can standardize individual keys in order to define specific types of alignments. For example, for alignments up to argument order, we can add a key for giving the argument order. Moreover, this can be used to annotate metadata such as provenance or system versions.

Below we standardize some individual keys and use them to implement the most important alignment types from Section 14.1. In all definitions below, we assume that a_1 and a_2 are the aligned symbols.

Definition 14.2:

The key direction has the possible values forward, backward, or both. Its presence legitimizes, respectively, the translation that replaces every occurrence of a_1 with a_2, its inverse, or both.

Alignments with direction key subsume the alignment types of perfect alignments (where the direction is both) and the unidirectional types of alignment up to totality of functions or up to associativity, and alignment for certain arguments. The absence of this key indicates those alignment types where no symbol-to-symbol translation is possible, in particular opaque alignments.

Definition 14.3:

The key arguments has values of the form $(r_1, s_1) \dots (r_k, s_k)$ where the r_i and s_i are natural numbers. Its presence legitimizes the translation of $a_1(x_1, \dots, x_m)$ to $a_2(y_1, \dots, y_n)$ where each y_k is defined by

- if $k = s_i$ for some i: the recursive translation of x_{r_i}
- otherwise: inferred from the context.

Alignments with arguments key subsume the alignment types of alignments up to argument order and of alignment up to determined arguments.

Example 14.2:
We obtain the following argument alignments for some of the examples from Section 14.1:

$$\begin{gathered}\mathsf{Nat}_1\ \mathsf{Nat}_2\ \mathsf{direction} = \text{"both"}\\ \mathsf{eq}_1\ \mathsf{eq}_2\ \mathsf{arguments} = \text{"}(1,2)(2,3)\text{"}\\ \mathsf{contains}_1\ \mathsf{in}_2\ \mathsf{arguments} = \text{"}(1,1)(2,3)(3,2)\text{"}\end{gathered}$$

Finally, we standardize a key for probabilistic alignments:

Definition 14.4:

The key similarity has values that are real numbers in $[0;1]$. If used together with other keys like direction and arguments, it represents a certainty score for the correctness of the corresponding translation. If absent, its value is 1 indicating perfect certainty.

14.4 Implementation

I have implemented alignments in the MMT system. Moreover, I have created a public repository [PRA] and seeded it with a number of alignments (currently ≈ 12000) including the ones mentioned above and below in Section 14.5, the README of this repository furthermore describes the syntax for alignments above as well as the URI schemata for several proof assistants. The MMT system can be used to parse and serve all these alignments, implement the transitive closure, and (if possible) translate expressions according to alignments (see Chapter 15 for the details). Available alignments are shown in the MMT browser.

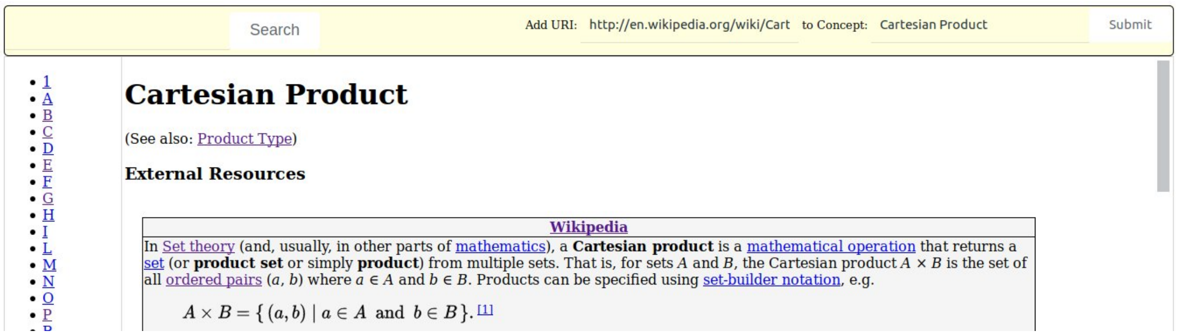
As an example service, I built a prototypical alignment-based math dictionary collecting formal and informal resources.[2] For this we extend the above grammar by the following:

$$\text{Alignment} \quad ::= \quad \text{String URI } (\text{String} = \text{"String"})^*$$

This assigns a mathematical concept (identified by the string) to a formal or informal resource (identified by the URI). The dictionary uses the above public repository, so additions to the latter will be added to the former. We have imported the ≈ 50,000 conceptual alignments from [GC14], although we chose not to add them to the dictionary yet, since the majority of them are (due to the different intention behind the conceptual mappings in Nnexus) dubious, highly contextual or otherwise undesirable.

Each entry in the dictionary shows snippets from select online resources if available (Figure 14.1), lists the associated formal statements (Figure 14.2) and available alignments between them (Figure 14.3), and allows for conveniently adding new individual URIs to concept entries as well as new formal alignments (Figures 14.1 and 14.4 respectively).

[2]Unfortunately, due to recent changes in the MMT API, the original dictionary is no longer compatible with the current code base and needs to be reimplemented at some point in the future.



Add URI: http://en.wikipedia.org/wiki/Cart to Concept: Cartesian Product

Cartesian Product

(See also: Product Type)

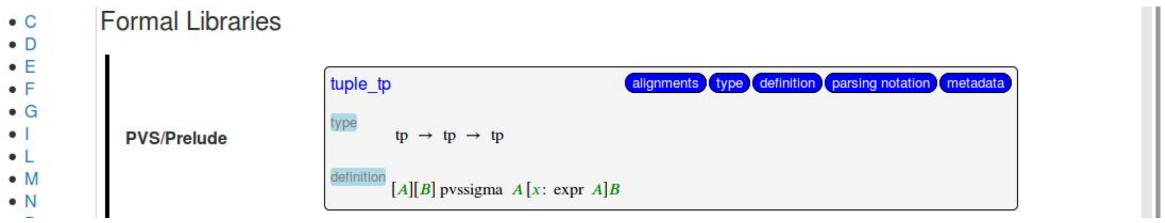
External Resources

Wikipedia

In Set theory (and, usually, in other parts of mathematics), a **Cartesian product** is a mathematical operation that returns a set (or **product set** or simply **product**) from multiple sets. That is, for sets A and B, the Cartesian product $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. Products can be specified using set-builder notation, e.g.

$$A \times B = \{(a,b) \mid a \in A \text{ and } b \in B\}.[1]$$

Figure 14.1: The Alignment-based Dictionary — External Resources



Formal Libraries

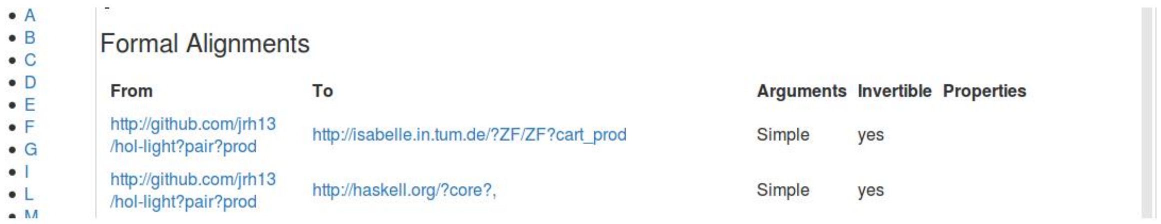
PVS/Prelude

tuple_tp

type $\text{tp} \to \text{tp} \to \text{tp}$

definition $[A][B]$ pvssigma $A[x\colon \text{expr } A]B$

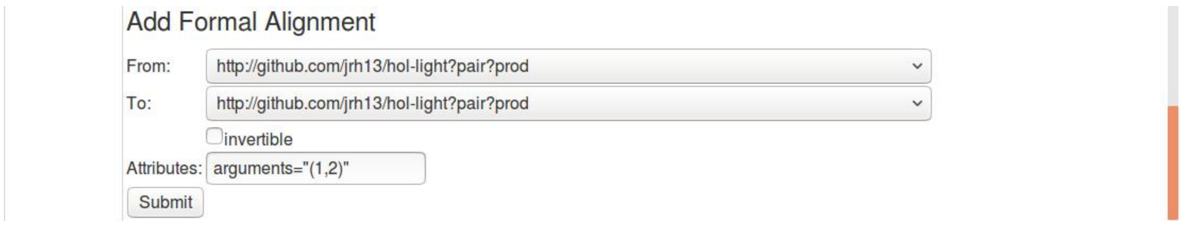
Figure 14.2: The Alignment-based Dictionary — Formal Resources



Formal Alignments

From	To	Arguments	Invertible	Properties
http://github.com/jrh13/hol-light?pair?prod	http://isabelle.in.tum.de/?ZF/ZF?cart_prod	Simple	yes	
http://github.com/jrh13/hol-light?pair?prod	http://haskell.org/?core?,	Simple	yes	

Figure 14.3: The Alignment-based Dictionary — Available Alignments



Add Formal Alignment

From: http://github.com/jrh13/hol-light?pair?prod

To: http://github.com/jrh13/hol-light?pair?prod

☐ invertible

Attributes: arguments="(1,2)"

Submit

Figure 14.4: The Alignment based Dictionary - Field for Adding Alignments

14.5 Manually Curated Alignments

As an experiment,we hired two mathematics students at Jacobs University in Bremen – Colin Rothgang and Yufei Liu – to manually comb through libraries of HOL Light, PVS, Mizar and Coq to find alignments. Specifically, they picked the mathematical areas of numbers, sets (as well as lists), abstract algebra, calculus, combinatorics, logic, topology, and graphs as a sample. This produced around 900 declarations overall, from which they constructed interface theories, as presented in Section 15.1. Notably neither of the two students had prior in-depth knowledge about theorem prover systems or their libraries.

Concept	PVS (Standard)	HOL Light (Standard)	Mizar (Standard)	Coq (Standard)
$\mathbb{N}$	naturalnumbers?naturalnumber	nums?nums	ORDINAL1?modenot.6	Coq.Init.Datatypes?nat
successor function	naturalnumbers?succ	nums?SUC	ORDINAL1?func.1	Coq.Init.Nat?succ
addition	number_fields?+	arith?ADD	ORDINAL2?func.10	Coq.Init.Nat?add
multiplication	number_fields?*	arith?MULT	ORDINAL2?func.11	Coq.Init.Nat?mul
less than	number_fields?<=	arith?<=	XXREAL_0?pred.1	Coq.Init.Nat?leb

Table 14.1: Alignments for NaturalNumbers (libraries in brackets)

Concept	PVS (NASA[3])	HOL Light (Standard)	Mizar (Standard)	Coq (coq-topology[4])
topology	topology_prelim?topology	topology?topology	PRE_TOPC?modenot.1	TopologicalSpaces?TopologicalSpace
open	topology?open?	topology?open_in	PRE_TOPC?attr.3	TopologicalSpaces?open
closed	topology?closed?	topology?closed_in	PRE_TOPC?attr.4	TopologicalSpaces?closed
interior	topology?interior	topology?interior	TOPS_1?func.1	InteriorsClosures?interior
closure	topology?Cl	topology?closure	PRE_TOPC?func.2	InteriorsClosures?closure

Table 14.2: Alignments for Topology (libraries in brackets)

For example, Tables 14.1 and 14.2 show some of these alignments for natural numbers and topology.

Alignments in topology pose some additional difficulties. Firstly, HOL Light defines a topology on some *subset* of the universal set of a type, whereas PVS defines it on a type directly. Thus, the alignment from HOL Light to the interface theory is unidirectional. Secondly, Mizar does not define the notion of a topology, but instead the notion of a topological space. Therefore, the students aligned all these symbols to two different symbols in an interface theory (`topology` and `topological_space`) and defined `topological_space` based on `topology`.

The set of all alignments they found can be inspected at `https://gl.mathhub.info/alignments/Public/tree/master/manual`. Many of these alignments are by now outdated, due to recent developments on MMT, including an import for Coq (see [MRS]) which had us rethink their MMT URI schema, and changes in the surface syntax that require the archive of interface theories (discussed in Section 15.1) to be cleaned up and partially reimplemented. However, as an experiment, it shows that finding and implementing alignments manually is surprisingly easy even with very little prior knowledge of the formal systems.

During the course of collecting alignments, Rothgang and Liu have identified the following two aspects to be the most common causes for imperfect alignments:

- *Subtyping* In the PVS library, the arithmetic operations on all the number fields are defined on a common supertype `numfields`. Therefore, a translation from PVS to the other two languages may not be viable.
- *Partiality* The result of division by zero in the libraries of HOL Light and Mizar is defined as zero; in PVS, however, the divisor must be nonzero. Therefore, certain theorems in HOL Light and Mizar involving division no longer hold in PVS, and the translation is unidirectional.

Topic	HOL Light	PVS	Mizar	Coq
Algebra	0/0	18/1	17/0	14/0
Calculus	15/0	14/0	16/0	5/15
Categories	0/0	0/0	9/1	5/0
Combinatorics	24/0	15/0	1/0	1/0
Complex Numbers	9/2	4/6	7/2	11/2
Graphs	5/5	17/0	20/0	7/2
Integers	10/0	0/0	5/2	47/3
Lists	16/0	9/0	8/0	36/2
Logic	7/0	7/5	7/0	24/1
Natural Numbers	19/0	8/10	9/0	34/1
Polynomials	4/0	1/0	7/0	0/0
Rational Numbers	0/14	2/11	0/10	14/3
Real Numbers	13/2	3/10	7/4	12/2
Relations	4/0	16/5	18/3	1/12
Sets	23/0	28/0	18/0	19/0
Topology	15/0	10/0	9/0	17/1
Vectors	13/0	7/0	15/0	0/0
Sum	**177/23**	**159/48**	**173/22**	**240/42**

Table 14.3: Number of Bidirectional/Unidirectional Alignments per Library

Chapter 15

Alignment-based Translations Using Interface Theories

Disclaimer:

A significant part of the contents of this chapter (except for Section 15.3) has been previously published as [Mül+17a] with coauthors Florian Rabe, Colin Rothgang and Yufei Liu. Both the writing as well as the theoretical results were developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors. In particular, Section 15.1 is primarily the result of a student experiment executed by Rothgang and Liu, under joint supervision by Rabe and me. The writing has been reworked for this thesis.

My contribution in this chapter consists of the actual implementation presented in Section 15.2, which I have rewritten from scratch for this thesis. Additionally, I have implemented the alignments, interface theories and translators for the use case in Section 15.3.

Using alignments, we can equivocate symbols from different libraries with each other. Ideally, in the case of a perfect alignment we can then reduce translation to a one-to-one symbol substitution. However, given a multitude of libraries in our framework, naively approaching the task of aligning them in the first place still means sifting through two libraries in parallel, implying n^2 alignment tasks for n libraries.

In recent work, our research group has come to understand this problem more clearly and suggested a systematic solution. Firstly, in [KRSC11] and [KR16b], Michael Kohlhase, Florian Rabe and Claudio Sacerdoti Coen developed the idea of **interface theories**. Using an analogy to software engineering, we can think of interface theories as specifications and of theorem prover libraries as implementations of formal knowledge.

Secondly, in the OpenDreamKit project [Deh+16; ODK] (see Section 3.3) we pursue the same approach (Math-in-the-Middle, see Section 3.4) in the context of computer algebra systems. In this context, we have already developed some interface theories for basic logical operations such as equality, with approximately 300 alignments to theorem prover libraries.

Not surprisingly, the methodology described in this chapter, while in theory originally developed for fully formal libraries originating from theorem prover systems, has consequently been used to realize the objectives of OpenDreamKit as well.

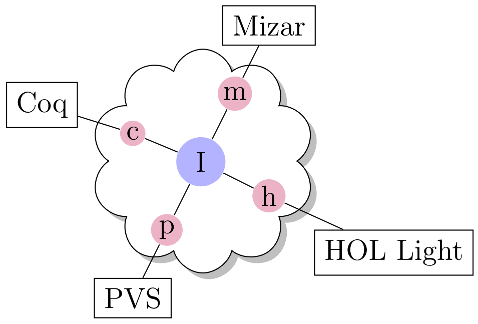


Figure 15.1: Aligning Libraries via Interface Theories

While alignments have the big advantage that they are cheap to find and implement, they have the disadvantage of not being very expressive; in fact, the definition of alignment itself is (somewhat intentionally) vague. In particular when a translation needs to consider foundational aspects beyond the individual system dialects, alignments alone are insufficient. This is where interface theories come into play: given different implementations of the same mathematical concept, their interface theory contains only those symbols that are a) common to all implementations and b) necessary to *use* the concept (as opposed to formalizing it in detail) – which in practice turn out to be the same thing.

Libraries of interface theories hence must critically differ from typical theorem prover libraries: they must follow the axiomatic method (as opposed to the method of definitional extensions), be written with minimal foundational commitment, and largely not rely on definitions or proofs in order to stay compatible with development approaches based on alternative definitions.

Using interface theories and the Math-in-the-Middle approach, we can see two symbols being aligned via a sequence of consecutive steps of abstraction:

1. We import the libraries of formal systems to a generic, foundationally neutral formal language (namely OMDoc/Mmt) with the help of a formalization of the system's primitives (see Chapter 10).

2. We align the system specific symbols (which we collectively refer to as the *system dialect*) with generic representations of the same concept. E.g. we align a symbol for HOL Light's function-application with a generic symbol for function application.

3. Ultimately, we (possibly after additional steps) end up with the same purely mathematical concept expressed in a system- and foundationally neutral language.

Consider for a simple example the PVS symbol `member` and the HOL Light symbol `IN`, we capture this with

- a symbol for elementhood in a (new) interface theory for sets,
- two alignments of this new symbol with the symbols `member` and `IN`, respectively.

This yields a star-shaped network as in the diagram in Figure 15.1 with various formal libraries on the outside, which are connected by alignments via representations of their system dialects (lower-case letters) to the interface theory in the center. Note the very deliberate similarity to Figure 3.4 – the primary difference is that we substituted OpenDreamKit systems with OAF systems (see Section 3.5).

15.1 Interface Theories

As mentioned in Section 14.5, we had two students (Colin Rothgang and Yufei Liu) collect alignments between various theorem prover libraries. In the course of this project, they also developed interface theories for the symbols they aligned.

As a basis for this experiment, they used the foundational theory for the Math-in-the-Middle archive, a simplified excerpt of which is shown in Listing 15.1, as an interface for basic logical constants.

Listing 15.1: An Interface Theory for Logic in MMT [1]

```
theory Logic : lfx :/TypedHierarchy?LFHierarchy =
        // the type of booleans, i.e., all formulas are represented as terms of LF−type $prop$ ❙
        prop  : type ❘ @ bool ❙
        rule rules ?BooleanLiterals ❙

        ded   : prop → type                ❘ # ⊢1 prec −500 ❘ role Judgment ❙

        // Equality on terms. The type A is left implicit and can be inferred by MMT ❙
        eq    : {A:𝒰 100} A → A → bool     ❘ # 2 ≐ 3 prec −5 ❘ role Eq ❙

        not   : bool → bool                ❘ # ¬1 prec −100 ❙
        neq   : {A: 𝒰 100} A → A → prop    ❘ # 2 ≠ 3 prec −5 ❙
        and   : bool → bool → bool         ❘ # 1 ∧2 prec −110 ❙
        or    : bool → bool → bool         ❘ # 1 ∨2 prec −120 ❙
        impl  : bool → bool → bool         ❘ # 1 ⇒2 prec −130 ❙
        iff   : bool → bool → bool         ❘ # 1 ⇔2 prec −140 ❙

        forall : {A : 𝒰 100} (A → bool) → bool ❘ # ∀2 prec −100❙
        exists : {A : 𝒰 100} (A → bool) → bool ❘ # ∃2 prec −100 ❙
```

This theory actually includes most of the features presented in Part II, in order to be as flexible as possible in formalizing content – although most of them were not yet available when the experiment described in this section was originally done.

15.1.1 Example: Natural Numbers

Listing 15.2 shows an example of a simple theory of natural numbers, using the theory `Logic` and the symbols declared therein. Note, that this theory is basically an implementation of the Peano axioms, which can be seen as the interface theory for all possible definitions of a concrete set (or type) of "the" natural numbers.

Listing 15.2: An (excerpt of an) interface theory for natural numbers with `Logic` as meta-theory[2]

```
theory Naturals : fnd:?Logic =
       zero         :  ℕ❘ = 0 ❙
       successor    :  ℕ→ ℕ            ❘ # S 1 ❙
       plus         :  ℕ→ ℕ→ ℕ         ❘ # 1 + 2 ❙
       times        :  ℕ→ ℕ→ ℕ         ❘ # 1 · 2 ❙
       leq          :  ℕ→ ℕ→ bool      ❘ # 1 ≤2 ❙

       Induction    : {P} ⊢P zero → ({n} ⊢P n → ⊢P (S n)) → ⊢∀ P ❙
       Succ_inj     : {n:ℕ,m:ℕ} ⊢(S n) ≐ (S m) ⊢n ≐ m ❙

       plus_zero    : {n} ⊢ (n + zero) ≐ n ❘ role Simplify ❙
       plus_succ    : {n,m} ⊢(n + (S m)) ≐S (n + m) ❘ role Simplify ❙
```

[1] https://gl.mathhub.info/MitM/Foundation/blob/master/source/math.mmt

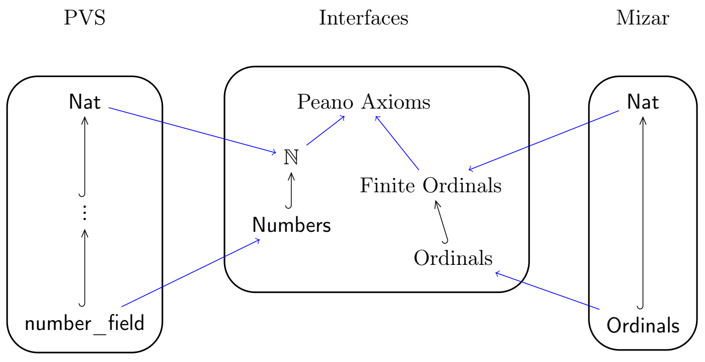


Figure 15.2: A Graph Showing Different Theories for Natural Numbers

```
        times_zero        :  {n} ⊢ (n ·  zero) ≐ zero | role  Simplify  |
        times_succ        :  {n,m} ⊢(n ·  (S m)) ≐ ((n ·  m) + n) |
```

In Mizar [Miz], which is based on Tarski-Grothendieck set theory, the set of natural numbers `NAT` is defined as omega, the set of finite ordinals. Arithmetic operations are defined directly on those.

In contrast, in PVS (see Chapter 10) natural numbers are defined as a specific subtype of the integers, which in turn are a subtype of the rationals etc. up to an abstract type `number` which serves as a maximal supertype to all number types. The arithmetic operations are inherited from a subtype `number_field` of `number`.

These are two fundamentally different approaches to describe and implement an abstract mathematical concept, but for all practical purposes the concept they describe is the same; namely the natural numbers. The interface theory for both variants would thus only contain the symbols that are relevant to the abstract concept itself, independent of their specific implementation – hence, things like the type of naturals, the arithmetic operations and the Peano axioms. The interface theory thus provides everything we need to work with natural numbers, and at the same time everything we know about them independently of the logical foundation or their specific implementation within any given formal system.

However, there is an additional layer of abstraction here, namely that in stating that the natural numbers in Mizar are the finite ordinals we have already ignored the system dialect (in the sense of item 2 in the introduction to this chapter). This step of abstraction (from the concrete definition using only Mizar-specific symbols) yields another interface theory for finite ordinals, which in turn can be aligned not just with Mizar natural numbers, but also e.g. with MetaMath [MeMa], which is built on ZFC set theory.

Figure 15.2 illustrates this situation. Blue arrows point from more detailed theories to their interfaces. The arrows from PVS or Mizar to interfaces merely strip away the system dialects; the arrows within Interfaces abstract away more fundamental differences in definition or implementation.

[2]https://gl.mathhub.info/MitM/interfaces/blob/master/source/Math/Numbers.mmt

Consider again Listing 15.2, a possible interface theory for natural numbers. Note, that symbols such as `leq` *could* be defined, but don't actually need to be. Since they are only interfaces, all we need is for the symbols to exist.

In fact, the more abstract the interface, the less we *want* to define the symbols – given that there's usually more than one way to define symbols, definitions are just one more thing we might want to abstract away from completely. There are exceptions to this – note that the symbol `zero` is actually defined as the literal 0.[3] This is because literals themselves are mere MMT terms, but not symbols with a stable MMT-URI, which we need for alignments. On the other hand, not having literals excludes those systems where literals are actually used. In this case, providing a definition to the symbols enables aligning both systems with and without literals.

The symbols in this interface theory can then be aligned either with symbols in other formal systems directly, or with additional interfaces in between, such as a theory for Peano arithmetic, or the intersection of all inductive subsets of the real numbers, or finite ordinals or any other possible formalization of the natural numbers.

15.1.2 Additional Interface Theories

The foundation independent nature of MMT allows us to implement interface theories with almost arbitrary levels of detail and for vastly different foundational settings.

We have started a repository of interface theories specifically for translation purposes [Mitb] and also aligned to already existing interfaces (as in the case of arithmetics, see below) in a second and third MathHub repository [Mitc] and [Mita] extending them when necessary.

Crucially, this interface repository contains interface theories for basic type-related symbols like the function type constructors (see Listing 15.3[4]), that are aligned with the respective symbols in HOL Light and PVS. These symbols are so basic as to be primitive in systems based on type theory, and consequently they occur in the vast majority of expressions. To have these symbols aligned is strictly necessary to get any further use of alignments off the ground.

Listing 15.3: Interface Theories for Type-Theoretical Foundations[5]

```
theory TypeSystem : ur:?LF =
         tp :  type |
         tm : tp → type |
|

theory SimpleFunctionTypes : ?TypeSystem =
         Arrowtp : tp → tp → tp | # 1 ⟹ 2 |
         Applytp : {A,B} tm (A ⟹ B) → tm A → tm B |# 3 @ 4 |## 3 4 |
         Lambdatp : {A,B} (tm A → tm B) → tm (A ⟹ B) |# λ3 |
|

theory DependentFunctionTypes : ?TypeSystem =
         Pitp : {A} (tm A → tp) → tp |# Π2 |
         Applytp : {A,B} tm (Pitp A B) → {x: tm A} tm (B x) |# 3 @ 4 | ## 3 4 |
```

[3]See Section 6.2.1 for more on Literals in MMT.

[4]This interface theory, like most formalizations of foundations, uses types as terms (via the symbols `tp` and `tm`), whereas the interface theories above (like `NaturalNumbers`) use the universe of types provided by the logical framework directly. During translation, special *higher-order abstract syntax rules* take care of eliminating or inserting the corresponding symbols appropriately to make aligning between the two formalization levels possible (see Section 4.3.2).

```
        Lambdatp : {A,B} ({x: tm A} tm (B x)) → tm (Π B) |# λ3 |
        structure  simple  :  ?SimpleFunctionTypes =
                Arrowtp = [A,B] Pitp A ([x] B) |
                Applytp = [A,B][f][ a]  Applytp A ([x] B) f  a |
                Lambdatp = [A,B][f] Lambdatp A ([x]B) f |
        |
|
```

Here, a *structure* is used to include the theory for simple function types in the theory for dependent function types, while providing definitions for the symbols in terms of the latter. This automatically yields a translation from the simple to the dependent variant and demonstrates the advantage of using interface theories for translations: The general problem of converting between (systems using) simple and dependent function types is lifted to the level of the interface theories, where we can use theory morphisms (such as structures) to specify the appropriate translation once and for all; independent of any specific external system.

Table 14.3 shows the total number of alignments the students found for PVS, HOL Light and Mizar. The following are some additional examples of mathematical areas covered by the resulting interface theories. The numbers here reflect the result of the student experiment and refer to the number of symbols actually used for alignments therein; the theories themselves have been partially adapted and significantly extended by now:

Calculus contains the following 3 subinterfaces:

Limits contains 17 symbols related to sequences and limits, including `metric_space` and `complete` (metric spaces and their completeness).

Differentiation contains 4 symbols, namely differentiability in a point and on a set and the derivative in a point and as a function

Integration contains 6 symbols, namely integrability and the integral over a set for Riemann, Lebesgue and Gauge-integration.

For **Arithmetics** we use the already existing theories from [Mitc]. It contains interfaces for below number arithmetics (each split into two interfaces for the basic number type definitions and the arithmetics on them).

Complex Numbers contains 11 symbols for complex numbers aligned to their counterparts in HOL Light, PVS and Mizar. Besides the usual arithmetic operations similar to `NaturalNumbers`, it contains `i` (the imaginary unit), `abs` (the modulus of a complex number) and `Re`, `Im` (the real and imaginary parts of a complex number).

Integers contains 9 symbols for the usual arithmetic operations on integers and for comparison between two integers.

Natural Numbers contains 21 symbols and is already described above.

[5]https://gl.mathhub.info/MitM/interfaces/blob/master/source/TypeSystem.mmt and https://gl.mathhub.info/MitM/interfaces/blob/master/source/TypeConstructors.mmt

Real Numbers contains 15 symbols, again very similar in nature to the other number spaces.

Lists contains the 13 most important symbols for lists, including `head`, `tail`, `concat`, `length` and `filter` (filter a list using another list) as well as some auxiliary definitions. There are no lists in Mizar, instead finite sequences are used. These however deserve their own interface.

For **Logic** we use the already existing theories in the [Mita] repository. It contains 9 symbols for boolean algebra that are all perfectly aligned to HOL Light, PVS and Mizar, and sometimes also to Coq.

Sets are also already existent in [Mitc], split into many subtheories. 28 of the contained symbols have been aligned. sets contains symbols for typed sets as in a type theoretical setting, including axioms and theorems. Here we have the most alignments so far. It also contains the following two interfaces:

Relations contains 23 symbols for alignments to relations and their properties, including orders.

Functions contains 7 symbols for alignments to functions and their relations, that are not already contained in relations.

Topology contains 25 symbols for both general topological spaces as well as the standard topology on $\mathbb{R}^n$ specifically.

In order to translate an expression from one library to another, the concepts in the expression must at least exist in both libraries. This creates the need to inspect the intersection of the concepts in these libraries. Table 15.1 gives an overview of the library intersection for various interface theories.

Figure 15.3 shows a small part of the theory graph of the MitM libraries. The full MitM theory graph can be explored on MathHub[6].

15.2 Implementation

I have implemented a prototypical expression translator in the MMT system that uses alignments, theory morphisms and programmatic tactics to translate expressions in the form of MMT terms. The translation mechanism is already used in the OPENDREAMKIT project and is exposed via the MMT query language [Rab12].

Crucially, the algorithm returns partial translations when no full translation to a target library can be found. These can be used for finding new alignments automatically by the techniques described in Chapter 16.

[6]https://mathhub.info/mh/mmt/graphs/tgview.html?graphdata=MitM

Topic	1 System	2 Systems	3 Systems	4 Systems
Algebra	17	9	5	0
Calculus	35	7	8	0
Categories	4	5	0	0
Combinatorics	25	6	0	1
Complex Numbers	10	5	3	3
Graphs	72	6	3	0
Integers	52	2	7	0
Lists	28	8	9	0
Logic	18	0	2	5
Natural Numbers	53	2	10	2
Polynomials	12	0	0	0
Rational Numbers	11	4	2	7
Real Numbers	9	3	5	5
Relations	21	15	4	0
Sets	56	10	9	10
Topology	62	2	8	0
Vectors	25	5	0	0
Sum	**510**	**91**	**79**	**33**

Table 15.1: Number of Concepts Found in Exactly One, Two, Three or Four Systems

The Algorithm is parametric in

1. An MMT term t,
2. A termination predicate Fin on MMT terms indicating whether an MMT term is finished translating. In the case of across-library translation, this predicate can e.g. check that all symbols occuring in a term are from a targeted MMT archive or in a specific namespace.
3. an ordered list of individual translators $\mathcal{T} = \{T_1, \ldots T_n\}$,which are partial functions from MMT terms to MMT terms.

We can obtain translators in several ways:

- For an alignment `S1 S2 direction = "both"`, we obtain the translator T with $T(S_1) = S_2$ and $T(S_2) = S_1$. Analogously, the translator we obtain from alignments with direction values forward or backward only maps S_1 to S_2 or, respectively, the other way around.
- For an alignment `S1 S2 arguments = "(i1,j1)...(in,jn)"`, we obtain the translator T that maps any term $S_1(x_1, \ldots, x_m)$ to $S_2(y_1, \ldots, y_n)$ according to the reordering from Definition 14.3. As in the case above, T also maps the reverse case depending on the direction-key.
- A theory morphism is already a partial function on MMT terms, its defined domain being the well-formed terms over the domain theory of the morphism.
- A Scala object of class AcrossLibraryTranslation, that implements two functions:
  - `applicable(tm: Term): Boolean` that determines its domain (should be fast regarding runtime) and
  - `apply(tm: Term):Term` that computes the translation.

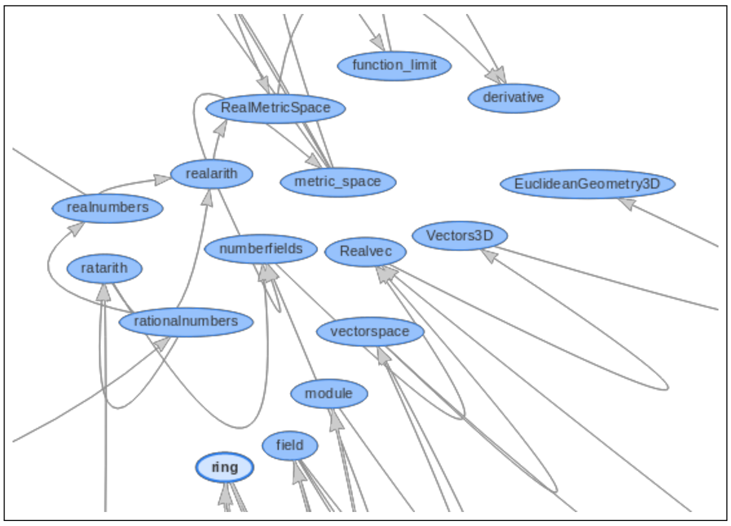


Figure 15.3: A Small Part of the MitM Theory Graph

- As a last option, we choose definition expansion, which maps a symbol reference S for a defined constant to its definiens.

The Algorithm Translate$(t,\mathsf{Fin},\mathcal{T})$:(Term,Boolean) then reduces to a simple tree search, iteratively applying applicable translators in $\mathcal{T}$ to subterms of t until Fin is satisfied, or return a partial translation if no full translation of t can be found.

This implementation is naive and potentially inefficient and should be considered prototypical, but suitably demonstrates the capabilities of the general approach.

Before we cover one example in detail, Table 15.2 lists three rather simple example translations between PVS and HOL Light. We use the Pi-types of LF to bind variables on the outside of the intended expressions.

Note that even something like the application of a function entails a translation from function application in HOL Light to function application in PVS, since they belong to their respective system dialects. Furthermore, the translation from simple function types in HOL Light to the dependent Π-type in PVS is actually done on the level of interface theories, where simple function types are defined as a special case of dependent function types (see Section 15.1.2). Unlike the latter two, the first example uses only primitive symbols in both systems that are not part of any external library. Meta-variables have been left out in the third example for brevity.

Furthermore, an argument alignment was used in the third example, that switches the two (non-implicit) arguments – namely the predicate and the list. Whereas this usually trips up automated translation processes or needs to be manually implemented, in our case this is as easy as adding (in this case) the key-value-pair `arguments = “(2,3)(3,2)”` to the alignment.

HOL Light	PVS
{A:holtype, P:term A⟹bool, a:term A}⊢P(a)	{A:tp,P:expr $\Pi_{_:A}$boolean,a:expr A}⊢P(a)
{T:holtype,a:term T,A:term T⟹bool}a IN A	{T:tp, a:expr T, A:expr $\Pi_{_:T}$boolean}member(a,A)
FILTER (Abs x:bool. x) c :: b :: a :: NIL	filter (c :: b :: a :: null) (λx : boolean. x)

Table 15.2: Three Expressions Translated

15.3 Example

To exemplify the usefulness of such translations across libraries, we will examine one example with computer algebra systems – which the OpenDreamKit research community has come to call *Jane's Use Case* [Koh+17a] – in detail. This example is interesting in that it was (chiefly) developed by John Cremona as a real-world situation a computational group theorist might find themself in, and where they would want to use a combination of computer algebra systems.

A user, Jane, wants to experiment with invariant theory of finite groups. She does so, by considering some group G as acting on the variables $X_1, \ldots, X_n$ of the polynomial ring $\mathbb{Z}[X_1, \ldots, X_n]$ and studying the ideals generated by the orbit of some polynomial, since these happen to be fixed by the group action. Naturally, in a computational setting one would want to have the Gröbner bases of these ideas as well. GAP happens to be efficient in computing orbits, whereas Singular is known to be very efficient in computing Gröbner bases. For the purposes of this section (and in light of Chapter 12) we will substitute Singular by Sage.

Starting from the Math-in-the-Middle ontology (see Section 3.4), Jane wants to study the dihedral group D_4. So she declares a polynomial p – exemplary $3X_1 + 2X_2$ – over $R = \mathbb{Z}[X_1, X_2, X_3, X_4]$, and wants to compute the orbit $O(\mathbb{Z}, D_4, p)$ in GAP, translate the resulting ideal back to MitM as I, declare its Gröbner base $G = \texttt{Groebner}(\mathbb{Z}, I)$, translate this to Sage in order to compute it and have the result presented to her back in MitM.

There are two things that make this example particularly non-trivial:

1. The terminologies employed by GAP and Sage differ: The dihedral group of order 8 is called D_4 in Sage, and D_8 in GAP; the conversion between the two is not easily expressed in a single alignment.

2. The ways polynomials are constructed in GAP and Sage differ considerably.

   The polynomial $3X_1 + 2X_2$ is formally constructed in GAP using the (internal) syntax

   $$\texttt{Poly}_{\text{GAP}}(\texttt{RationalFunctions}(\texttt{FamilyObject}(1)), [[1,1], 3, [2,1], 2])\,,$$

   where $[\ldots]$ represents lists, and $[[k_1, n_1, \ldots, k_m, n_m], a]$ represents the monomial $aX_{k_1}^{n_1} \cdot \ldots \cdot X_{k_m}^{n_m}$. The precise meaning of the `FamilyObject` and `RationalFunctions` symbols is technical and largely irrelevant.

   In Sage, the same polynomial is instead constructed as

   $$\texttt{Poly}_{\text{SAGE}}(\texttt{PolyRing}(\mathbb{Z}, [\texttt{X}_1, \texttt{X}_2, \texttt{X}_3, \texttt{X}_4]), [[[1,0,0,0], 3], [[0,1,0,0], 2]])\,,$$

   where $[[i_1 \ldots i_n], a]$ represents the monomial $aX_1^{i_1} \cdot \ldots \cdot X_n^{i_n}$. Consequently, monomials refer to the specific named variables of the polynomial ring $\mathbb{Z}[X_1, X_2, X_3, X_4]$ provided as an argument.

   In general and not surprisingly, there seems to be no universally agreed upon method to formally represent polynomials.

The symbols for lists, integers, Gröbner bases, ideals (which in a computational setting can be represented as a list of generators) and orbits can be easily expressed as typed constants in an interface theory and connected via alignments, so we can focus on those aspects of the translation that are non-trivial.

The discrepancy regarding dihedral groups is reflective of different notational conventions used in the mathematical community, hence it makes sense to have both of them represented in an interface. Correspondingly, for the interface theory in the Math-in-the-Middle library we use both and simply use definitions to convert between them:

Listing 15.4: An Interface Theory for Dihedral Groups

```
theory Dihedral : base:?Logic =
    dihedralD     : ℕ+ → group ❙
    dihedral      : ℕ+ → group ❘= [z] dihedralD (2 · z) ❙
```

Which allows us to use simple alignments to translate from GAP/Sage to MitM and definition expansion to translate from the second to the first variant. For the other direction, we need a theory morphism, e.g. a view:

Listing 15.5: An View for Translating Dihedral Groups

```
view DihedralTranslation : ?Dihedral -> ?Dihedral =
    dihedralD = [n] dihedral (n / 2) ❙
```

Regarding converting polynomials, we are stuck with using a programmatic translator instead. For a MitM representation of polynomials, we use a more generic method that carries the names of the variables:

Listing 15.6: An Interface Theory for Integer Polynomials[7]

```
theory Polynomials : base:?Logic =
    polynomialRing : ring → type ❙
    monomial : ring → type ❘ # monomial 1❙
    multi_poly_con : {r : ring} List (monomial r) → multi_polynomial r ❘ # mpoly 2 ❙
    monomial_con : {r: ring} (List (string × ℕ)) × r.universe → monomial r ❘ # monomial_con 1 2❙
❚

theory IntegerPolynomials : base:?Logic =
    include ?Polynomials ❙
    include algebra?IntegerRing ❙
    int_polynomial : List (monomial ℤ) → multi_polynomial ℤ❙
    int_monomial : (List (string × ℕ)) × ℤ → monomial ℤ❙
❚
```

The polynomial $3X_1 + 2X_2$ would then be represented as

```
int_polynomial([int_monomial(⟨[⟨"X1",1⟩],3⟩),int_monomial(⟨[⟨"X2",1⟩],2⟩)]) .
```

The only thing missing then are programmatic translators from GAP and Sage to MitM and their inverses. As an example, Listing 15.7 shows the translator from MitM polynomials to GAP polynomials, using helper objects (analogous to those described in Section 5.2.1).

Listing 15.7: A Translator from MitM Polynomials to GAP[8]

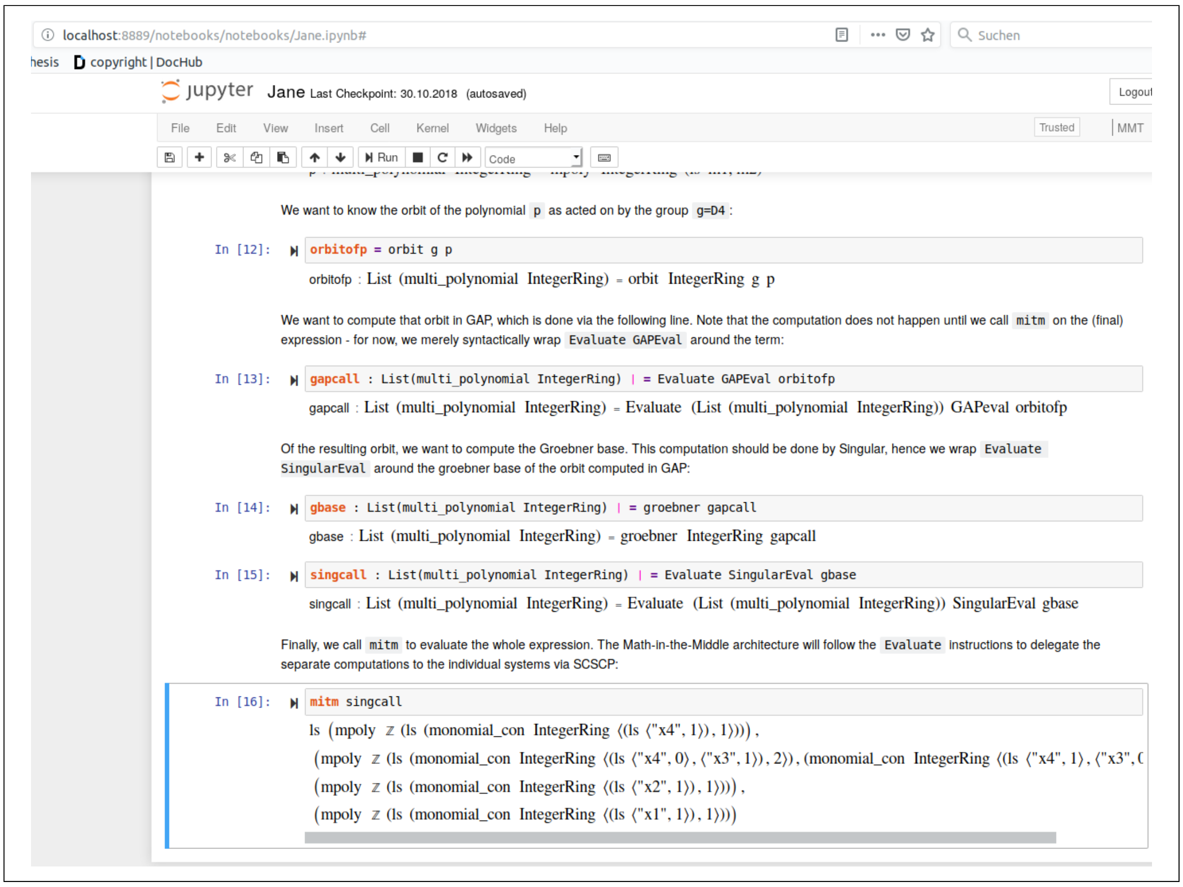
```
val toPolynomials = new AcrossLibraryTranslation {
    def applicable(tm: Term): Boolean = tm match {
        case MitM.MultiPolynomial(_,_) => true
        case _ => false
    }
    def apply(tm: Term): Term = tm match {
        case MitM.MultiPolynomial(_,ls) =>
```

[7] Adapted and simplified from https://gl.mathhub.info/MitM/smglom/blob/master/source/algebra/polynomials.mmt

```
        Poly(ls.flatMap{
          case (vars,coeff,_) =>
            LFList(vars.flatMap(p => List(Integers(variables.get(p._1)),Integers(p._2)))) :: Integers(coeff) :: Nil
        })
    }
  }
```

Given these translators, Jane's Use Case can be realized as a sequence of applications of the translation algorithm described in Section 15.2.

Figure 15.4 shows the realization of this use case in a Jupyter [Jup] notebook running an MMT kernel. Here, the expression `Evaluate(S,t)` is used to instruct the MMT system to translate the expression t to the ontology of system S and have the target system evaluate the expression.


We want to know the orbit of the polynomial `p` as acted on by the group `g=D4` :

```
In [12]:  orbitofp = orbit g p
```

orbitofp : List (multi_polynomial IntegerRing) = orbit IntegerRing g p

We want to compute that orbit in GAP, which is done via the following line. Note that the computation does not happen until we call `mitm` on the (final) expression - for now, we merely syntactically wrap `Evaluate GAPEval` around the term:

```
In [13]:  gapcall : List(multi_polynomial IntegerRing) | = Evaluate GAPEval orbitofp
```

gapcall : List (multi_polynomial IntegerRing) = Evaluate (List (multi_polynomial IntegerRing)) GAPeval orbitofp

Of the resulting orbit, we want to compute the Groebner base. This computation should be done by Singular, hence we wrap `Evaluate SingularEval` around the groebner base of the orbit computed in GAP:

```
In [14]:  gbase : List(multi_polynomial IntegerRing) | = groebner gapcall
```

gbase : List (multi_polynomial IntegerRing) = groebner IntegerRing gapcall

```
In [15]:  singcall : List(multi_polynomial IntegerRing) | = Evaluate SingularEval gbase
```

singcall : List (multi_polynomial IntegerRing) = Evaluate (List (multi_polynomial IntegerRing)) SingularEval gbase

Finally, we call `mitm` to evaluate the whole expression. The Math-in-the-Middle architecture will follow the `Evaluate` instructions to delegate the separate computations to the individual systems via SCSCP:

```
In [16]:  mitm singcall
```

```
ls (mpoly ℤ (ls (monomial_con IntegerRing ⟨(ls ⟨"x4", 1⟩⟩, 1⟩))) ,
 (mpoly ℤ (ls (monomial_con IntegerRing ⟨(ls ⟨"x4", 0⟩, ⟨"x3", 1⟩⟩, 2⟩), (monomial_con IntegerRing ⟨(ls ⟨"x4", 1⟩, ⟨"x3", (
 (mpoly ℤ (ls (monomial_con IntegerRing ⟨(ls ⟨"x2", 1⟩⟩, 1⟩))) ,
 (mpoly ℤ (ls (monomial_con IntegerRing ⟨(ls ⟨"x1", 1⟩⟩, 1⟩)))
```

Figure 15.4: Jane's Use Case in an MMT Jupyter Notebook

[8]https://github.com/UniFormal/MMT/blob/master/src/mmt-odk/src/info/kwarc/mmt/odk/GAP/GAPSystem.scala

Chapter 16

Viewfinding

Disclaimer:

A significant part of the contents of this chapter has been previously published as [MKR18] with coauthors Michael Kohlhase and Florian Rabe. The writing was developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors.

My contribution in this chapter consists of developing the algorithm presented in this chapter, its implementation and running and evaluating the case study in Section 16.3.

Theory morphisms underly the notion of theory graphs. They are one of the primary structuring tools employed in MMT and enable the kind of modularity we aimed for in this thesis, most notably Part II. As they induce new theorems in the target theory for any of the source theory, *views* (as opposed to includes and other trivial morphisms, see Section 4.1.1) in particular are high-value elements of a modular formal library. Usually, these are manually encoded, but this practice requires authors who are familiar with source and target theories at the same time, which limits the scalability of the manual approach.

To remedy this problem, I have developed a view finder algorithm that automates theory morphism discovery, and implemented it in MMT. While this algorithm conceptually makes use of the definitional property of theory morphisms that they are to preserve typing judgements, it ultimately operates on the level of individual declarations after preprocessing which may include translations as in Chapter 15 using alignments – as such, it can also be thought of as an alignment finder across libraries, simultaneously utilizing and extending the set of possible translations between them.

Here, we focus on one specific application of *theory classification*, where a user can check whether a (part of a) formal theory already exists in some library, potentially avoiding duplication of work or suggesting an opportunity for refactoring.

The basic use case is the following: Mary, a mathematician, becomes interested in a class of mathematical objects, say – as a didactic example – something she initially calls “beautiful subsets” of a base set $\mathcal{B}$ (or just “beautiful over $\mathcal{B}$”). These have the following properties Q:

1. the empty set is beautiful over $\mathcal{B}$
2. every subset of a beautiful set is beautiful over $\mathcal{B}$
3. If A and B are beautiful over $\mathcal{B}$ and A has more elements than B, then there is an

$x \in A \backslash B$, such that $B \cup \{x\}$ is beautiful over $\mathcal{B}$.

To see what is known about beautiful subsets, she types these three conditions into a theory classifier, which computes any theories in a library $\mathcal{L}$ that match these (after a suitable renaming). In our case, Mary learns that her "beautiful sets" correspond to the well-known structure of matroids [MWP], so she can directly apply matroid theory to her problems.

In extended use cases, a theory classifier may find theories that share significant structure with Q, so that Mary can formalize Q modularly with minimal effort. Say Mary was interested in "dazzling subsets", i.e. beautiful subsets that obey a fourth condition, then she could just contribute a theory that extends the theory `matroid` by a formalization of the fourth condition – and maybe rethink the name.

Existing systems have so far only worked with explicitly given views, e.g., in IMPS [FGT93] or Isabelle [Isa]. Automatically and systematically searching for new views was first undertaken in [NK07] in 2006. However, at that time no large corpora of formalized mathematics were available in standardized formats that would have allowed easily testing the ideas in practice.

This situation has changed since then as multiple such exports have become available. In particular, we now have OMDoc/Mmt as a uniform representation language for such corpora. Building on these developments, we are now able, for the first time, to apply generic methods — i.e., methods that work at the Mmt level — to search for views in formal libraries.

While inspired by the ideas of [NK07], the design and implementation presented here are completely novel. In particular, the theory makes use of the rigorous language-independent definitions of *theory* and *view* provided by Mmt.

[GK14a] applies techniques related to ours to a related problem. Instead of views inside a single corpus, they use machine learning to find similar constants in two different corpora. Their results can roughly be seen as a single partial view from one corpus to the other.

16.1 The Viewfinder Algorithm

Let C be a corpus of theories with (for now) the same fixed meta-theory M. We call the problem of finding views between theories of C the **view finding problem** and an algorithm that solves it a **view finder**. Note that a view finder is sufficient to solve the theory classification use case above: Mary provides a M-theory Q of beautiful sets, the view finder computes all (total) views from Q into C.

Efficiency Considerations The cost of this problem quickly explodes. First of all, it is advisable to restrict attention to simple views (see Section 4.1.1). Eventually we want to search for arbitrary views as well. But that problem is massively harder because it subsumes theorem proving: a view from Σ to Σ' maps Σ-axioms to Σ'-proofs, i.e., searching for a view requires searching for proofs.

Secondly, if C has n theories, we have n^2 pairs of theories between which to search. (It is exactly n^2 because the direction matters, and even views from a theory to itself are interesting.) Moreover, for two theories with m and n constants, there are n^m possible *simple* views (It is exactly n^m because views may map different constants to the same one.) Thus, we can in principle enumerate and check all possible simple views in C. But for large C, it quickly becomes important to do so in an efficient way that eliminates ill-typed or uninteresting views early on.

Thirdly, it is desirable to search for *partial* views as well. In fact, identifying refactoring potential in libraries is only possible if we find partial views: then we can refactor the involved theories in a way that yields a total view (see Chapter 17). Moreover, many proof assistant libraries do not follow the little theories paradigm or do not employ any theory-like structuring mechanism at all. These can only be represented as a single huge theory, in which case we have to search for partial views from this theory to itself – facilitating the aforementioned refactoring technique. While partial views can be reduced to and then checked like total ones, searching for partial views makes the number of possible views that must be checked much larger.

Finally, even for a simple view, checking reduces to a set of equality constraints, namely the constraints $\vdash_{\Sigma'} \overline{\sigma}(E) = E'$ for the type-preservation condition (see Section 4.1.1). Depending on M, this equality judgment may be undecidable and require theorem proving.

A central motivation for my algorithm is that equality in M can be soundly approximated very efficiently by using a normalization function on M-expressions. This has the additional benefit that relatively little meta-theory-specific knowledge is needed, and all such knowledge is encapsulated in a single well-understood function. This way we can implement view–search generically for arbitrary M.

The algorithm consists of two steps. First, we preprocess all constant declarations in C with the goal of moving as much intelligence as possible into a step whose cost is linear in the size of C. Then, we perform the view search on the optimized data structures produced by the first step.

16.1.1 Preprocessing

The preprocessing phase computes for every constant declaration $c : E$ a normal form E' and then efficiently stores the abstract syntax tree (defined below) of E'. Both steps are described below.

Normalization involves two steps: **MMT-level normalization** performs generic transformations that do not depend on the meta-theory M. These include flattening and definition expansion. Importantly, we do not fully eliminate defined constant declarations $c : E = e$ from a theory Σ: instead, we replace them with primitive constants $c : E$ and replace every occurrence of c in other declarations with e. If Σ is the domain theory, we can simply ignore $c : E$ (because views do not have to provide an assignment to defined constants). But if the Σ is the codomain theory, retaining $c : E$ increases the number of views we can find; in particular in situations where E is a type of proofs, and hence c a theorem.

Meta-theory-level normalization applies an M-specific normalization function. In general, we assume this normalization to be given as a black box. However, because many practically important normalization steps are widely reusable, we provide a few building blocks, from which specific normalization functions can be composed. Skipping the details, these include:

1. Top-level universal quantifiers and implications are rewritten into the function space of the logical framework using the Curry-Howard correspondence.
2. The order of curried domains of function types is normalized as follows: first all dependent argument types are ordered by the first occurrence of the bound variables; then all non-dependent argument types A are ordered by the abstract syntax tree of A.

3. Implicit arguments, whose value is determined by the values of the others are dropped, e.g. the type argument of an equality. This has the additional benefit of shrinking the abstract syntax trees and speeding up the search.
4. Equalities are normalized such that the left hand side has a smaller abstract syntax tree.

Above multiple normalization steps make use of a total order on abstract syntax trees. We omit the details and only remark that we try to avoid using the names of constants in the definition of the order — otherwise, declarations that could be matched by a view would be normalized differently. Even when breaking ties between requires comparing two constants, we can first try to recursively compare the syntax trees of their types.

Abstract Syntax Trees We define **abstract syntax trees** as pairs (t, s) where t is subject to the grammar

$$t ::= C_{Nat} \mid V_{Nat} \mid t\,[t^+]\,(t^+)$$

(where Nat is a non-terminal for natural numbers) and s is a list of constant names.

We obtain an abstract syntax tree from an MMT expression E by (i) switching to de-Bruijn representation of bound variables and (ii) replacing all occurrences of constants with C_i in such a way that every C_i refers to the i-th element of s.

Abstract syntax trees have the nice property that they commute with the application of simple views σ: If (t, s) represents E, then $\sigma(E)$ is represented by (t, s') where s' arises from s by replacing every constant with its σ-assignment.

The above does not completely specify i and s yet, and there are several possible canonical choices among the abstract syntax trees representing the same expression. The trade-off is subtle because we want to make it easy to both identify and check views later on. We call (t, s) the **long** abstract syntax tree for E if C_i replaces the i-th occurrence of a constant in E when E is read in left-to-right order. In particular, the long tree does not merge duplicate occurrences of the same constant into the same number. The **short** abstract syntax tree for E arises from the long one by removing all duplicates from s and replacing the C_i accordingly.

Example 16.1:
Consider the axiom $\forall x : \mathsf{set}, y : \mathsf{set}.\ \mathsf{beautiful}(x) \wedge y \subseteq x \Rightarrow \mathsf{beautiful}(y)$ with internal representation

$$\forall\,[\mathsf{x : set, y : set}]\,(\Rightarrow (\wedge\,(\mathsf{beautiful}\,(\mathsf{x})\,, \subseteq (\mathsf{y}, \mathsf{x}))\,, \mathsf{beautiful}\,(\mathsf{y})))\,.$$

The *short* syntax tree and list of constants associated with this term would be:

$$t = C_1\,[C_2, C_2]\,(C_3\,(C_4\,(C_5\,(V_2)\,, C_6\,(V_1, V_2))\,, C_5\,(V_1)))$$

$$s = (\forall, \mathsf{set}, \Rightarrow, \wedge, \mathsf{beautiful}, \subseteq)$$

The corresponding long syntax tree is :

$$t = C_1\,[C_2, C_3]\,(C_4\,(C_5\,(C_6\,(V_2)\,, C_7\,(V_1, V_2))\,, C_8\,(V_1)))$$

$$s = (\forall, \mathsf{set}, \mathsf{set}, \Rightarrow, \wedge, \mathsf{beautiful}, \subseteq, \mathsf{beautiful})$$

For our algorithm, we pick the *long* abstract syntax tree, which may appear surprising. The reason is that shortness is not preserved when applying a simple view: whenever a view maps two different constants to the same constant, the resulting tree is not short anymore. Length, on the other hand, is preserved. The disadvantage that long trees take more time to traverse is outweighed by the advantage that we never have to renormalize the trees.

16.1.2 Search

Consider two constants $c : E$ and $c' : E'$, where E and E' are preprocessed into long abstract syntax trees (t, s) and (t', s'). It is now straightforward to show the following Lemma:

Lemma 16.1:

The assignment $c \mapsto c'$ is well-typed in a view σ if $t = t'$ (in which case s and s' must have the same length l) and σ also contains $s_i \mapsto s'_i$ for $i = 1, \ldots, l$.

Proof. The claim is simply the typing-preservation condition of theory morphisms for the special case of simple views; expressed using abstract syntax trees. □

Of course, the condition about $s_i \mapsto s'_i$ may be redundant if s contains duplicates; but because s has to be traversed anyway, it is cheap to skip all duplicates. We call the set of assignments $s_i \mapsto s'_i$ the **prerequisites** of $c \mapsto c'$.

This lemma describes our search algorithm:

Lemma 16.2: Core Algorithm

Consider two constant declarations c and c' with $t = t'$ in theories Σ and Σ'. We define a view by starting with $\sigma = \{c \mapsto c'\}$ and recursively adding all prerequisites to σ until

- *either the recursion terminates*
- *or σ contains two different assignments for the same constant, in which case we fail,*
- *or the recursion requires $d \mapsto d'$, where d and d' have incompatible abstract syntax trees, in which case we fail.*

If the above algorithm succeeds, then σ is a well-typed partial simple view from Σ to Σ'.

Proof. Note that the algorithm constructs a partial view by recursively ensuring that the premises of Theorem 16.1 holds. □

Example 16.2:

Consider two constants c and c' with types $\forall x : \mathsf{set}, y : \mathsf{set}.\ \mathsf{beautiful}(x) \wedge y \subseteq x \Rightarrow \mathsf{beautiful}(y)$ and $\forall x : \mathsf{powerset}, y : \mathsf{powerset}.\ \mathsf{finite}(x) \wedge y \subseteq x \Rightarrow \mathsf{finite}(y)$. Their syntax trees are

$$t = t' = C_1 \left[C_2, C_3\right] \left(C_4 \left(C_5 \left(C_6 \left(V_2\right), C_7 \left(V_1, V_2\right)\right), C_8 \left(V_1\right)\right)\right)$$

$$s = (\forall, \mathsf{set}, \mathsf{set}, \Rightarrow, \wedge, \mathsf{beautiful}, \subseteq, \mathsf{beautiful})$$

$$s' = (\forall, \mathsf{powerset}, \mathsf{powerset}, \Rightarrow, \wedge, \mathsf{finite}, \subseteq, \mathsf{finite})$$

Since $t = t'$, we set $c \mapsto c'$ and compare s with s', meaning we check (ignoring duplicates) that $\forall \mapsto \forall$, set $\mapsto$ powerset, $\Rightarrow \mapsto \Rightarrow$, $\wedge \mapsto \wedge$, beautiful $\mapsto$ finite and $\subseteq \mapsto \subseteq$ are all valid.

To find all views from Σ to Σ', we first run the core algorithm on every pair of Σ-constants and Σ'-constants. This usually does not yield big views yet. For example, consider the typical case where theories contain some symbol declarations and some axioms, in which the symbols occur. Then the core algorithm will only find views that map at most one axiom.

Depending on what we intend to do with the results, we might prefer to consider them individually (e.g. to yield *alignments*). But we can also use these small views as building blocks to construct larger, possibly total ones:

Lemma 16.3: Amalgamating Views

We call two partial views ***compatible*** *if they agree on all constants for which both provide an assignment.*

The union of compatible well-typed views is again well-typed.

Proof. Note that it is sufficient to show that for any constant $c : T$ in its domain, the amalgamated view σ satisfies $\sigma(c) \Leftarrow \sigma(T)$. Since the partial views amalgamated to σ are assumed to be compatible and well-typed, this immediately follows from the assignment $c \mapsto \sigma(c)$ being in (at least) one of the constituent partial views. □

Example 16.3:
Consider the partial view from Example 16.2 and imagine a second partial view for the axioms beautiful($\varnothing$) and finite($\varnothing$). The former has the requirements

$$\forall \mapsto \forall, \quad \mathsf{set} \mapsto \mathsf{powerset} \quad \Rightarrow \mapsto \Rightarrow \quad \wedge \mapsto \wedge \quad \mathsf{beautiful} \mapsto \mathsf{finite} \quad \subseteq \mapsto \subseteq$$

The latter requires only set $\mapsto$ powerset and $\varnothing \mapsto \varnothing$. Since both views agree on all assignments, we can merge all of them into a single view, mapping both axioms and all requirements of both.

16.1.3 Optimizations

The above presentation is intentionally simple to convey the general idea. We now consider a few advanced features of the implementation to enhance scalability.

Caching Preprocessing Results Because the preprocessing performs normalization, it can be time-consuming. Therefore, we allow for storing the preprocessing results to disk and reloading them in a later run.

Fixing the Meta-Theory We improve the preprocessing in a way that exploits the common meta-theory, which is meant to be fixed by every view. All we have to do is, when building the abstract syntax trees (t, s), to retain all references to constants of the meta-theory in t instead of replacing them with numbers. With this change, s will never contain meta-theory constants, and the core algorithm will only find views that fix all meta-theory constants. Because s is much shorter now, the view search is much faster.

It is worth pointing out that the meta-theory is not always as fixed as one might think. Often we want to consider certain constants, that are defined early on in the library and then used widely, to be part of the meta-theory. In PVS (see Chapter 10), this makes sense, e.g., for all operations defined in the Prelude library. For across-library view finding, we might also apply a partial translation (in the sense of Chapter 15) to the target library first and consider all succesfully translated symbols as part of the meta-theory, ensuring that all morphisms found are compatible with and predicated on already known alignments and morphisms.

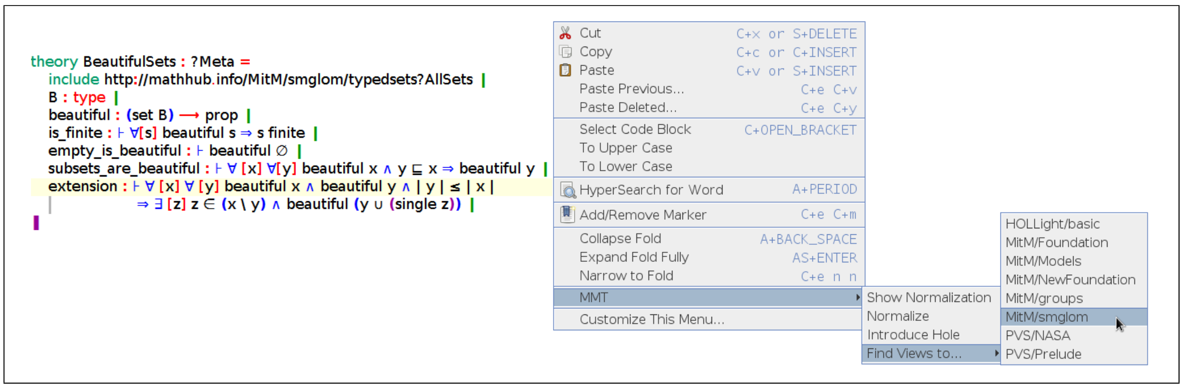
Note that we still only have to cache one set of preprocessing results for each library: changes to the meta-theory only require minor adjustments to the abstract syntax trees without redoing the entire normalization.

Biasing the Core Algorithm The core algorithm starts with an assignment $c \mapsto c'$ and then recurses into constant that occur in the declarations of c and c'. This occurs-in relation typically splits the constants into layers. A typical theory declares types, which then occur in the declarations of function symbols, which then occur in axioms. Because views that only map type and function symbols are rarely interesting (because they do not allow transporting non-trivial theorems), we always start with assignments where c is an axiom, but other conditions for starting declarations are possible.

Exploiting Theory Structure Libraries are usually highly structured using imports between theories. If Σ is imported into Σ', then the set of partial views out of Σ' is a superset of the set of partial views out of Σ. If implemented naively, that would yield a quadratic blow-up in the number of views to consider.

Instead, when running our algorithm on an entire library, we only consider views between theories that are not imported into other theories. In an additional postprocessing phase, the domain and codomain of each found partial view σ are adjusted to the minimal theories that make σ well-typed.

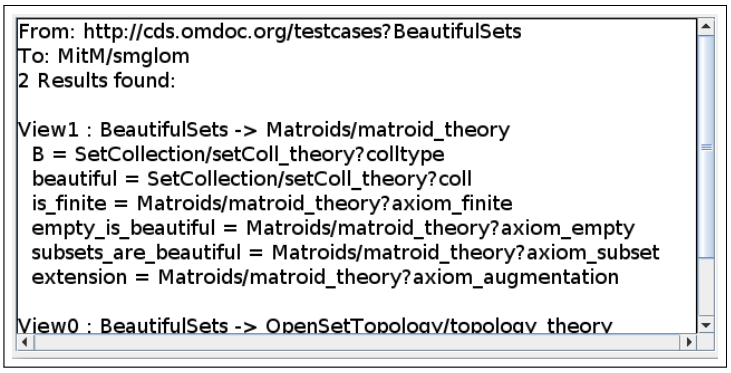
16.2 Implementation



```
theory BeautifulSets : ?Meta =
   include http://mathhub.info/MitM/smglom/typedsets?AllSets |
   B : type |
   beautiful : (set B) ⟶ prop |
   is_finite : ⊦ ∀[s] beautiful s ⇒ s finite |
   empty_is_beautiful : ⊦ beautiful ∅ |
   subsets_are_beautiful : ⊦ ∀ [x] ∀[y] beautiful x ∧ y ⊑ x ⇒ beautiful y |
   extension : ⊦ ∀ [x] ∀ [y] beautiful x ∧ beautiful y ∧ | y | ≤ | x |
   |             ⇒ ∃ [z] z ∈ (x \ y) ∧ beautiful (y ∪ (single z)) |
```

Figure 16.1: “Beautiful Sets” in MMT Surface Syntax

I have implemented our view finder algorithm in the MMT system and exposed it in MMT's jEdit IDE. A screenshot of Jane's theory of beautiful sets is given in Figure 16.1. Right-clicking anywhere within the theory allows Jane to select MMT → Find Views to... → MitM/smglom. The latter menu offers a choice of known libraries in which the view finder should look for codomain theories; MitM/smglom is the Math-in-the-Middle library (see Section 3.4).
After choosing MitM/smglom, the view finder finds two views (within less than one second) and shows them (Figure 16.2).



```
From: http://cds.omdoc.org/testcases?BeautifulSets
To: MitM/smglom
2 Results found:

View1 : BeautifulSets -> Matroids/matroid_theory
  B = SetCollection/setColl_theory?colltype
  beautiful = SetCollection/setColl_theory?coll
  is_finite = Matroids/matroid_theory?axiom_finite
  empty_is_beautiful = Matroids/matroid_theory?axiom_empty
  subsets_are_beautiful = Matroids/matroid_theory?axiom_subset
  extension = Matroids/matroid_theory?axiom_augmentation

View0 : BeautifulSets -> OpenSetTopology/topology_theory
```

Figure 16.2: Views Found for "Beautiful Sets"

The first of these (View1) has a theory for matroids as its codomain, which is given in Listing 16.1. Inspecting that theory and the assignments in the view, we see that it indeed represents the well-known correspondence between beautiful sets and matroids.

Listing 16.1: The Theory of Matroids in the MitM Library[1]

```
theory Matroids : base:?Logic =
          include ?Properties |
          include typedsets?FiniteCardinality |
          include arithmetics ?NaturalArithmetics |

          theory matroid_theory : base:?Logic > X : type | =
                    include typedsets?SetCollection /setColl_theory (X) |

                    axiom_finite : {s : set X} ⊢s ∈ coll → ⊢s finite |
                    axiom_empty : ⊢prop_emptyInColl coll |
                    axiom_subset : ⊢ prop_subsetClosed coll |
                    axiom_augmentation : ⊢∀[A] ∀[B] A ∈ coll ∧ B ∈ coll ∧ | B | ≤ | A |
                                        ⇒∃ [x] x ∈ (A \ B) ∧ (B ∪ (single x)) ∈ coll |
          ❚
❚
```

The latter uses predefined propositions in its axioms and uses a type coll for the collection of sets, while the former has the statements of the axioms directly in the theory and uses a predicate beautiful – since (typed) sets are defined as predicates, definition expansion is required for matching. Additionally, the implication that beautiful sets (or sets in a matroid) are finite is stated as a logical formula in the former, while the latter uses the Curry/Howard correspondence.

[1]https://gl.mathhub.info/MitM/smglom/blob/master/source/collections/matroids.mmt

16.3 Cross-Library Viewfinding

We now generalize view finding to the situation where domain and codomain live in different libraries written in different logics. Intuitively, the key idea is that we now have two fixed meta-theories M and M' and a fixed meta-view $m : M \to M'$. However, due to the various idiosyncrasies of logics, tools' library structuring features and individual library conventions, this problem is significantly more difficult than *intra*-library view finding. For example, unless the logics are closely related, meta-views usually do not even exist and must be approximated. Therefore, a lot of tweaking is typically necessary, and it is possible that multiple runs with different trade-offs give different interesting results.

As an example, we present a large case study where we find views from the MitM library used in the running example so far into the PVS/NASA library (see Chapter 10).

Theory Structure Normalization PVS's complex and prevalently used parametric theories critically affect view finding because they affect the structure of theories. For example, the theory of groups `group_def` in the NASA library has three theory parameters $(\mathsf{T}, *, \mathsf{one})$ for the signature of groups, includes the theory `monoid_def` with the same parameters, and then declares the axioms for a group in terms of these parameters. Without special treatment, we could only find views from/into libraries that use the same theory structure.

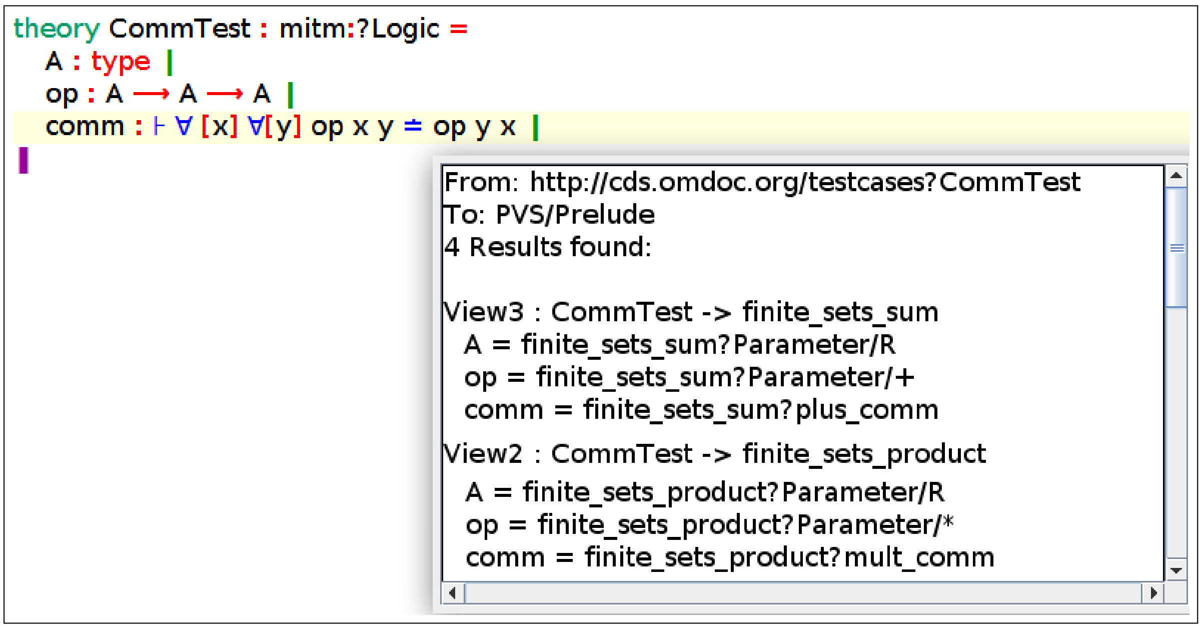
We have investigated three approaches of handling parametric theories:

1. *Simple treatment:* We drop theory parameters and interpret references to them as free variables that match anything. This is of course not sound so that all found views must be double-checked. However, because practical search problems often do not require exact results, even returning all potential views can be useful.
2. *Covariant elimination:* We treat theory parameters as if they were constants declared in the body. In the above mentioned theory `group_def`, we would hence add three new constants T, $*$ and one with their corresponding types. This works well in the common case where a parametric theory is not used with two different instantiations in the same context.
3. *Contravariant elimination:* The theory parameters are treated as if they were bound separately for every constant in the body of the theory. In the above mentioned theory `group_def`, we would change e.g. the unary predicate `inverse_exists`? with type $T \to \mathsf{bool}$ to a function with type $(T : \mathsf{pvstype}) \to (* : T \to T \to T) \to (\mathsf{one} : T) \to (T \to \mathsf{bool})$. This is closest to the actual semantics of the PVS module system. But it makes finding interesting views the least likely because it is the most sensitive to the modular structure of individual theories.

I have implemented the first two approaches. The first is the most straightforward but it leads to many false positives and false negatives. I have found the second approach to be the most useful for inter-library search since it most closely corresponds to simple formalizations of abstract theories in other libraries. The third approach will be our method of choice when investigating *intra*-library views of PVS/NASA in future work.

Implementation Example

As a first use case, we can write down a theory for a commutative binary operator using the MitM foundation, while targeting the PVS Prelude library – allowing us to find all commutative operators, as in Figure 16.3 (using the simple approach to theory parameters).



```
theory CommTest : mitm:?Logic =
  A : type |
  op : A ⟶ A ⟶ A |
  comm : ⊦ ∀ [x] ∀[y] op x y ≐ op y x |
```

```
From: http://cds.omdoc.org/testcases?CommTest
To: PVS/Prelude
4 Results found:

View3 : CommTest -> finite_sets_sum
  A = finite_sets_sum?Parameter/R
  op = finite_sets_sum?Parameter/+
  comm = finite_sets_sum?plus_comm
View2 : CommTest -> finite_sets_product
  A = finite_sets_product?Parameter/R
  op = finite_sets_product?Parameter/*
  comm = finite_sets_product?mult_comm
```

Figure 16.3: Searching for Commutative Operators in PVS

This example also hints at a way to iteratively improve the results of the view finder: since we can find properties like commutativity and associativity, we can use the results to in turn inform a better normalization of the theory by exploiting these properties. This in turn would potentially allow for finding more views.

To evaluate the approaches to theory parameters we used a simple theory of monoids in the MitM foundation and the theory of monoids in the NASA library as domains for viewfinding with the whole NASA library as target using simple and covariant approaches. The results are summarized in Figure 16.4.

Domain	Normalization	Simple Views	Aggregated
NASA/monoid	simple	388	154
MitM/monoid	simple	32	17
NASA/monoid	covariant	1026	566
MitM/monoid	covariant	22	6

Figure 16.4: Results of Inter- and Intra-Library View Finding in the PVS NASA Library

Most of the results in the simple MitM→NASA case are artifacts of the theory parameter treatment and view amalgamation – in fact only two of the 17 results are meaningful (to operations on sets and the theory of number fields). In the covariant case, the additional requirements lead to fuller (one total) and less spurious views. With a theory from the NASA library as domain, the results are already too many to be properly evaluated by hand. With the simple approach to theory parameters, most results can be considered artifacts; in the covariant case, the most promising results yield (partial) views into the theories of semigroups, rings (both the multiplicative and additive parts) and most extensions thereof (due to the duplication of theory parameters as constants).

16.4 Applications for Viewfinding

We have seen how a view finder can be used for theory *classification* and finding constants with specific desired properties, but many other potential use cases are imaginable. The main problems to solve with respect to these is less about the algorithm or software design challenges, but user interfaces.

The theory classification use case described in Section 16.2 is mostly desirable in a setting where a user is actively writing or editing a theory, so the integration in jEdit is sensible. However, the across-library use case in Section 16.3 already would be a lot more useful in a theory exploration setting, such as when browsing available archives on MathHub [Ian+14] or in the graph viewer integrated in MMT [RKM17]. Additional specialized user interfaces would enable or improve the following use cases:

- **Model-/Countermodel Finding:** If the codomain of a view is a theory representing a specific model, it would tell a user that those are *examples* of her abstract theory.

  Furthermore, partial views – especially those that are total on some included theory – could be insightful *counterexamples*.

- **Library Refactoring:** Given that the view finder looks for *partial* views, we can use it to find natural extensions of a starting theory. Imagine Jane removing the last of her axioms for “beautiful sets” – the other axioms (disregarding finitude of her sets) would allow her to find e.g. both Matroids and *Ideals*, which would suggest to her to refactor her library accordingly.

  Additionally, *surjective* partial views would inform her, that her theory would probably better be refactored as an extension of the codomain, which would allow her to use all theorems and definitions therein.

- **Theory Generalization:** If we additionally consider views into and out of the theories found, this can make theory classificationeven more attractive. For example, a view from a theory of vector spaces intro matroids could inform Jane additionally, that her beautiful sets, being matroids, form a generalization of the notion of linear independence in linear algebra.

- **Folklore-based Conjecture:** If we were to keep book on our transfomations during preprocessing and normalization, we could use the found views for translating both into the codomain as well as back from there into our starting theory.

  This would allow for e.g. discovering and importing theorems and useful definitions from some other library – which on the basis of our encodings can be done directly by the view finder.

  A useful interface might specifically prioritize views into theories on top of which there are many theorems and definitions that have been discovered.

For some of these use cases it would be advantageous to look for views *into* our working theory instead.

Note that even though the algorithm is in principle symmetric, some aspects often depend on the direction – e.g. how we preprocess the theories, which constants we use as starting points or how we aggregate and evaluate the resulting (partial) views (see Sections 16.1.3 and 16.3).

Chapter 17

Refactoring and Theory Discovery via Theory Intersections

Disclaimer:

A significant part of the contents of this chapter has been previously published as Work-in-Progress as [MK15] with coauthor Michael Kohlhase. Both the writing as well as the theoretical results were developed in close collaboration between the authors, hence it is impossible to precisely assign authorship to individual authors.

My contribution in this chapter consists of all implementations described herein.

An important driver of mathematical development is the discovery of new mathematical objects, concepts and theories. Even though there are many different situations that give rise to a new theory, it seems that a common instance is the discovery of commonalities between apparently different mathematical structures (if such exist). In fact, many of the algebraic theories in Bourbaki naturally arise as the "common part" of two (or more) different mathematical structures – for instance, one could interpret the theory of groups as the common theory of $(\mathbb{Z}, +, 0)$ and the set of symmetrical operations on e.g. a square.

In [Koh14], Kohlhase proposes a notion of *theory intersection* – elaborating an earlier formalization from [Nor08] to MMT [RK13b; HKR12b] that captures this phenomenon in a formal setting: Let two theories S and T, a partial view $\sigma : S \to T$ with domain D and codomain C, and its partial inverse δ (as in Figure 17.1 on the left) be given. Then we can pass to a more modular theory graph, where $S' := S \backslash D$ and $T' := T \backslash C$ (as in Figure 17.1 on the right). In this case we think of the isomorphic theories D and C as the *intersection* of theories S and T along σ and δ.

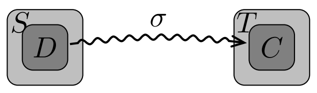
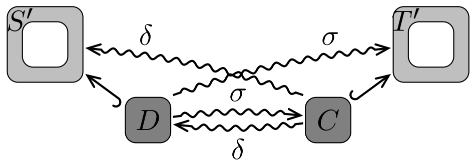
Situation	Theory Intersection
	

Figure 17.1: Theory Intersection

17.1 Theory Intersection by Example

To fortify our intuitions, we examine a concrete mathematical example in detail.

We start out with a theory `PosPlus` of positive natural numbers with addition and intersect it with `StrConc` of strings with concatenation (as in Figure 17.2). Note that we do not start with modular developments; rather the modular structure is (going to be) the result of intersecting with various examples. Also, note that the views k and l are both partial.

Positive	Strings

theory PosPlus =
    ℕ : type ❙
    + : ℕ → ℕ→ ℕ
    assoc : ...
    comm : ...
	theory StrConc =
	    A∗ : type ❙
	    :: : A∗ → A∗ → A∗ ❙
	    assoc : ...
	    ε : A∗ ❙
	    unit : ... ❙

view k : ?PosPlus −> ?StrConc =
    ℕ = A∗ ❙
    + = :: ❙
    assoc = assoc ❙
	view l : ?StrConc −> ?PosPlus =
	    A∗ = ℕ❙
	    :: = + ❙
	    assoc = assoc ❙

Figure 17.2: Positive Integers and Strings

Now the intersection as proposed above yields Figure 17.3, which directly corresponds to the schema in Figure 17.1. Note, that the new pair of (equivalent) theories is completely determined by the partial morphism k; the only thing we have to invent are the names – and we have not been very inventive here.

Positive	Strings

theory A =
    ℕ : type ❙
    + : ℕ → ℕ→ ℕ❙
    assoc : ...
❚
theory PosPlus =
    include ?A ❙
    comm : ...
❚
	theory B =
	    A∗ : type ❙
	    :: : A∗ → A∗ → A∗ ❙
	    assoc : ... ❙
	❚
	theory StrConc =
	    include ?B ❙
	    ε : A∗ ❙
	    unit : ... ❙
	❚

Figure 17.3: Positive Integers and Strings, Refactored

Intuitively, the intersection theory is the theory of semigroups which is traditionally written down in MMT as in Figure 17.4. And indeed, `sg` is a renaming of both `A` and `B`.

```
theory sg =
    G : type ❙
    ∘: G → G → G ❙
```

Figure 17.4: Semigroups

For this situation, we should have a variant theory intersection operator that takes a partial morphism and returns *one* intersection theory.

In this situation, we want a theory intersection operator that follows the schema in Figure 17.5.

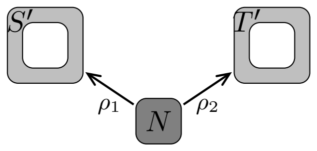


Figure 17.5: Unary TI

Let us call this operation *unary TI* and the previous one *binary TI*. To compute it, we need to specify a name N for the new theory and two renamings $\rho_1 : N \rightarrow \mathrm{Dom}(\sigma)$ and $\rho_2 : N \rightarrow \mathsf{Img}(\sigma)$. Note that in this TI operator, the intersection is connected to the "rest theories" via MMT structures – rather than mere inclusions – which carry the assignments induced by the partial morphisms suitably composed with the renamings (see next section).

In our example we can obtain the theory sg from Figure 17.4 via the renamings $\rho_1 := \{\mathsf{G} \mapsto \mathbb{N}, \circ \mapsto +, \mathsf{assoc} \mapsto \mathsf{assoc}\}$ and $\rho_2 := \{\mathsf{G} \mapsto A^*, \circ \mapsto ::, \mathsf{assoc} \mapsto \mathsf{assoc}\}$.

Unary TI is often the more useful operation on theory graphs, but needs direct user supervision, whereas binary TI is fully automatic if we accept generated names for the intersection theories.

17.2 Reducing Partial Morphisms to Renamings

In the previous example it is noteworthy, that the morphisms k and l are fully inverse to each other. In fact, they are *renamings*. By a renaming, we mean a partial morphism $\sigma : S \rightarrow T$ such, that for every constant c declared in S, $\sigma(c)$ is again a constant (as opposed to a complex term).

Naturally, theory intersections are a lot simpler for renamings. In fact, we only need a single renaming $\sigma : S \rightarrow T$ to intersect along (since the inverse of a renaming is again a renaming and uniquely determined). It turns out, that we can always reduce the situation in Figure 17.1 to the much easier case, where we have a single renaming $\sigma : S' \rightarrow T'$, where S' and T' are conservative extensions of our original theories:

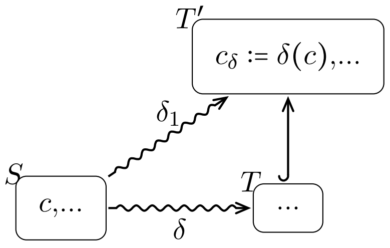


Figure 17.6: Obtaining a Renaming From a Morphism

Let $\delta : S \rightarrow T$ be a partial theory morphism and $\delta(c) = t$ for a complex term t. We can then conservatively extend T to a theory T_1 that contains a new defined constant $c_\delta = t$ and adapt δ to a new morphism δ_1 with $\delta_1(c) = c_\delta$ (and $\delta_1 \restriction S \setminus \{c\} = \delta \restriction S \setminus \{c\}$), as in Figure 17.6. If we repeat this process for every mapping $\delta(c) = t$ that is not already a simple renaming, we yield a new conservative extension T' such, that the corresponding morphism δ' (accordingly adapted) is a renaming.

Doing the same with a partial inverse $\rho : T \to S$, we ultimately get a single renaming $\sigma : S' \to T'$ that is equivalent to the original pair of partial morphisms. We can therefore w.l.o.g. reduce ourselves in every instance to this case.

17.3 Implementation

I have implemented a method for unary TI in an earlier version of the MMT API.

The intersection method takes as parameters two theories, a name for the intersection and a list of pairs of declarations, which can be obtained from a pair of morphisms, a single morphism or use the view finder presented in Chapter 16. It returns the intersection theory and the refactored versions of (conservative extensions of) the original theories, depending on the original morphisms provided.

Both methods are integrated into a refactoring panel, which is part of the MMT plugin for jEdit. To intersect two theories, the user can either provide one or two morphisms between the theories or allow the Viewfinder to pick for them. The declarations in the intersection theory can optionally be named.

An annotated video demonstration of the refactoring panel and its components can be found on Youtube [IntV15].

If we accept automatically generated names for the theory intersections, we can eliminate user interaction and use the view finder to automate the intersection operation on a set of theories without having to provide the morphisms to intersect along beforehand. This could be used to refactor whole libraries in a more modular way (in concordance with the *little theories* approach [FGT92b]). Additionally, by supplying the view finder with alignments and other translators between libraries, our approach could be used to refactor a whole library along the theory graph of an already modular library.

We do not yet have a heuristic to evaluate (the resulting) theories automatically with respect to their interest; however, given a morphism between two interesting theories, we have observed that the corresponding intersection tends to be interesting as well, as long as the morphism is not meaningless itself.

Part V

Conclusion and Future Work

We have seen how we can take on the integration problem for formal libraries by representing them in a unifying meta-framework (Part III), using a flexible logical framework to specify their foundations (Part II), aligning their contents (Chapter 14) and implementing foundation-independent alignment-aware algorithms for identifying overlap (Chapter 16) suggestive of new alignments, translating expressions across libraries (Chapter 15) and modularize library content by refactoring along theory intersections (Chapter 17).

The knowledge management techniques described in Part IV are certainly not exhaustive; many more services are imaginable, but the ones presented in this thesis demonstrate the effectiveness of the general approach to library integration described herein. Furthermore, these facilitate the aims of the OAF (see Section 3.5) and OpenDreamKit (see Section 3.3) projects, as demonstrated in Section 15.3. Additionally they represent steps towards a system integrating all aspects of the tetrapod (see Chapter 1); demonstrably *inference*, *computation*, *tabulation* and *organization*.

Naturally, this opens up many avenues for future research and improvements, which fall broadly into three categories:

1. Fundamentals Many implementations described in this thesis are prototypical and can be optimized or otherwise improved. In particular:

1. The implementation of alignments (see Chapter 14) is not deeply integrated into the Mmt system and offers no change management facilities. As mentioned in Section 14.5, many of the alignments we once had are by now outdated for various reasons, suggesting that additional services are needed that e.g. keep track of (and adequately adapt) changes in URIs, check for the existence (or maybe even adequacy) of aligned symbols, allow for globally replacing URIs in multiple alignments, etc.

2. The current syntax for alignments is, while simple and convenient, not exhaustive for many types of alignments, such as those that need simple arithmetic operations on arguments (as the dihedral groups described in Section 15.3), or the cases where a function $f(x)$ is aligned with a function $g(t, x)$ for a fixed expression t. Currently, this introduces a need for programmatic translators, but a more expressive syntax could obviate that need in many situations.

3. The implementation of alignment-based translation presented in Chapter 15 is naive and potentially inefficient. While this does not show for the use cases that have occured in practie so far, I strongly suspect that this might become inconvenient once the number of available alignments and other translators grows, or the input expressions reach a certain degree of complexity.

4. I mentioned in the introduction to Part II that it is desirable to not have inference rules in a logical framework active that are not needed for some specific content – one of the reasons why we prefer a modular approach to our logical frameworks. This is also the reason we shied away from a naive implementation of declared subtyping in Chapter 7. To expand on that claim, the reason is that naively applying inference rules can be computationally expensive and slow down the solver. This is not just the case for unnecessary inference rules, but also for more basic conflicting strategies for checking a

judgement (*critical pairs*). A simple example is the basic approach of a bi-directional type checking algorithm:

To check a judgement $t \Leftarrow T$ we have two strategies: We can either simplify T until we find an applicable foundation-dependent typing rule specifically for (the head symbol of) T, or we can try to infer the principal type T' of t and check $T' <: T$. In most cases, the existence of a typing rule implies that the former strategy will quickly yield a result, whereas inferring the type can be expensive, and hence the former strategy is attempted first. However, imagine T being a single reference to a constant with a complex definiens, and t being a function application $f(a)$, where the return type of f is T. In this case, inferring the principal type of t is fast and the subsequent check $T <: T$ trivial, whereas simplifying T leads to definition expansion and subsequent further simplifications, which will ultimately need to be repeated for the inferred type of t as well, since a function application can only ever check against any type via the inference strategy. This kind of situation seems to occur regularly in high-level formalizations, and is correspondingly often encountered in the development of the Math-in-the-Middle library.

To remedy this problem, I am currently actively working on a reimplementation of the MMT solver algorithm using multithreading; not just to exploit multicore processors, but to be able to compute conflicting checking strategies on separate threads in parallel, such that if *either* strategy terminates quickly, the checking algorithm can return a result just as quickly. This would obviate the current advantage of being cautious with adding potentially slow inference rules.

2. Scaling Up Many of the implementations and approaches presented in this thesis scale in usefulness and coverage depending on a certain amount of available data, features and implementations. All of these can and should be scaled up in the future, such as:

1. extending our logical framework by more desirable features (Part II),
2. Importing more formal libraries and system ontologies into OMDoc/MMT (Part III),
3. expanding the Math-in-the-Middle library by more content and interface theories (Chapter 15),
4. collecting more alignments between libraries (Chapter 14),
5. covering more library contents with dedicated translators where needed (Chapter 15),
6. investigating both the available and additional libraries for more useful preprocessing strategies to increase the coverage and usefulness of the view finder algorithm (Chapter 16),

3. Workflows One of the major obstacles in using all of the techniques in this thesis is a certain lack of tool support with respect to specific workflows for various use cases. Especially the algorithms presented in Part IV are primarily usable by programmatically calling methods in the MMT API, or via commands in the MMT shell. Exemplary, in Chapter 16 my coauthors and I (on the original publication) already conjecture on possible applications for the view finder algorithm, most of which are predicated on the existence of a bespoke user interface for that particular use case. Designing such interfaces entails thinking deeply about realistic and convenient workflows for different situations and users.

This also applies to MMT's surface syntax, which requires a very reasonable, but (for example, for working mathematicians not familiar with formal systems) potentially prohibitive amount of time and effort to get accustomed to.

I have written an MMT plugin for IntelliJ IDEA[1], a java-based IDE which offers a rich plugin API, to experiment with integrating various generic services in a more flexible IDE than the only previously supported one, jEdit. In the future, I would like to investigate the possibility of using e.g. sTEX [Koh08] or possibly even plain LATEX as an alternative input language for formal MMT content, as well as designing a unifying online IDE on MathHub for working with MMT and its connected systems and libraries, obviating the need to set up and use MMT (and the multitude of systems with which MMT can interact) locally – a process which (even using docker images, jupyter notebooks and similar solutions) can be prohibitively complicated for non-experts, especially when lacking a unifying user interface.

[1] https://www.jetbrains.com/idea/, the plugin itself is available at https://github.com/UniFormal/IntelliJ-MMT

Chapter 18

Bibliography

Online Source References

[LFX] *MathHub MMT/LFX Git Repository*. URL: http://gl.mathhub.info/MMT/LFX (visited on 05/15/2015).

[Mita] *MitM/Foundation*. URL: https://gl.mathhub.info/MitM/Foundation (visited on 09/01/2017).

[Mitb] *MitM/Interfaces*. URL: https://mathhub.info/MitM/interfaces (visited on 06/21/2017).

[Mitc] *MitM/smglom*. URL: https://gl.mathhub.info/MitM/smglom (visited on 02/01/2018).

[MMT] *UniFormal/MMT – The MMT Language and System*. URL: https://github.com/UniFormal/MMT (visited on 10/24/2017).

[OMU] *OMDoc/MMT Urtheories*. URL: https://gl.mathhub.info/MMT/urtheories (visited on 06/02/2017).

[PRA] *Public Repository for Alignments*. https://gl.mathhub.info/alignments/Public. (Visited on 05/21/2017).

Publications by the Author

[Con+] A. Condoluci, M. Kohlhase, D. Müller, F. Rabe, C. S. Coen, and M. Wenzel. “Relational Data Across Mathematical Libraries”. Accepted for CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-ulo.pdf.

[Deh+16] P.-O. Dehaye et al. “Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach”. In: *Intelligent Computer Mathematics 2016*. Conferences on Intelligent Computer Mathematics. (Bialystok, Poland, July 25–29, 2016). Ed. by M. Kohlhase, M. Johansson, B. Miller, L. de Moura, and F. Tompa. LNAI 9791. Springer, 2016. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/CICM2016/published.pdf.

[IntV15] D. Müller. *Theory Intersections in MMT*. URL: https://www.youtube.com/watch?v=qXKaGuV7kLY (visited on 06/29/2015).

[Kal+16] C. Kaliszyk, M. Kohlhase, D. Müller, and F. Rabe. “A Standard for Aligning Mathematical Concepts”. In: *Work in Progress at CICM 2016*. Ed. by A. Kohlhase, M. Kohlhase, P. Libbrecht, B. Miller, F. Tompa, A. Naummowicz, W. Neuper, P. Quaresma, and M. Suda. CEUR-WS.org, 2016, pp. 229–244.

[Koh+17a] M. Kohlhase, D. Müller, M. Pfeiffer, F. Rabe, N. Thiéry, V. Vasilyev, and T. Wiesing. “Knowledge-Based Interoperability for Mathematical Software Systems”. In: *Mathematical Aspects of Computer and Information Sciences*. Ed. by J. Blömer, I. Kotsireas, T. Kutsia, and D. Simos. Springer, 2017, pp. 195–210.

[Koh+17b] M. Kohlhase, D. Müller, S. Owre, and F. Rabe. “Making PVS Accessible to Generic Services by Interpretation in a Universal Format”. In: *Interactive Theorem Proving*. Ed. by M. Ayala-Rincón and C. A. Muñoz. Vol. 10499. LNCS. Springer, 2017. URL: http://kwarc.info/kohlhase/submit/itp17-pvs.pdf.

[Koh+17c] M. Kohlhase, T. Koprucki, D. Müller, and K. Tabelow. “Mathematical models as research data via flexiformal theory graphs”. In: *Intelligent Computer Mathematics (CICM) 2017*. Conferences on Intelligent Computer Mathematics. Ed. by H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke. LNAI 10383. Springer, 2017. DOI: 10.1007/978-3-319-62075-6. URL: http://kwarc.info/kohlhase/papers/cicm17-models.pdf.

[Kop+18] T. Koprucki, M. Kohlhase, K. Tabelow, D. Müller, and F. Rabe. “Model pathway diagrams for the representation of mathematical models”. In: *Journal of Optical and Quantum Electronics* 50.2 (2018), p. 70. DOI: 10.1007/s11082-018-1321-7.

[MK15] D. Müller and M. Kohlhase. “Understanding Mathematical Theory Formation via Theory Intersections in MMT”. 2015. URL: http://cicm-conference.org/2015/fm4m/FMM_2015_paper_2.pdf.

[MKR18] D. Müller, M. Kohlhase, and F. Rabe. “Automatically Finding Theory Morphisms for Knowledge Management”. In: *Intelligent Computer Mathematics (CICM) 2018*. Conferences on Intelligent Computer Mathematics. Ed. by F. Rabe, W. Farmer, A. Youssef, and … LNAI. in press. Springer, 2018. URL: http://kwarc.info/kohlhase/papers/cicm18-viewfinder.pdf.

[MR19] D. Müller and F. Rabe. “Rapid Prototyping Formal Systems in MMT: 5 Case Studies”. Accepted for LFMTP 2019. 2019. URL: https://kwarc.info/people/dmueller/pubs/RapidPrototyping.pdf.

[MRK] D. Müller, F. Rabe, and M. Kohlhase. *Theories as Types*. URL: http://kwarc.info/kohlhase/submit/tatreport.pdf.

[MRK18] D. Müller, F. Rabe, and M. Kohlhase. “Theories as Types”. In: ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Springer Verlag, 2018. URL: http://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

[MRS] D. Müller, F. Rabe, and C. Sacerdoti Coen. “The Coq Library as a Theory Graph”. Submitted to CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-coq.pdf.

[Mül+17a] D. Müller, C. Rothgang, Y. Liu, and F. Rabe. “Alignment-based Translations Across Formal Systems Using Interface Theories”. In: *Proof eXchange for Theorem Proving*. Ed. by C. Dubois and B. Woltzenlogel Paleo. Open Publishing Association, 2017, pp. 77–93.

[Mül+17b] D. Müller, T. Gauthier, C. Kaliszyk, M. Kohlhase, and F. Rabe. “Classification of Alignments between Concepts of Formal Mathematical Systems”. In: *Intelligent Computer Mathematics (CICM) 2017*. Conferences on Intelligent Computer Mathematics. Ed. by H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke. LNAI 10383. Springer, 2017. DOI: 10.1007/978-3-319-62075-6. URL: http://kwarc.info/kohlhase/papers/cicm17-alignments.pdf.

[Mül+17c] D. Müller, T. Gauthier, C. Kaliszyk, M. Kohlhase, and F. Rabe. *Classification of Alignments between Concepts of Formal Mathematical Systems*. Tech. rep. 2017. URL: https://gl.mathhub.info/alignments/Public/tree/master/doc/report.pdf (visited on 05/21/2017).

[RKM16] D. Rochau, M. Kohlhase, and D. Müller. "FrameIT Reloaded: Serious Math Games from Modular Math Ontologies". In: *Intelligent Computer Mathematics – Work in Progress Papers*. Ed. by M. Kohlhase, A. Kohlhase, P. Libbrecht, B. Miller, A. Naumowicz, W. Neuper, P. Quaresma, F. Tompa, and M. Suda. 2016. URL: `http://ceur-ws.org/Vol-1785/W50.pdf`.

[RKM17] M. Rupprecht, M. Kohlhase, and D. Müller. "A Flexible, Interactive Theory-Graph Viewer". In: *MathUI 2017: The 12th Workshop on Mathematical User Interfaces*. Ed. by A. Kohlhase and M. Pollanen. 2017. URL: `http://kwarc.info/kohlhase/papers/mathui17-tgview.pdf`.

[RM18] F. Rabe and D. Müller. "Structuring Theories with Implicit Morphisms". In: *24th International Workshop on Algebraic Development Techniques 2018*. 2018. URL: `https://kwarc.info/people/frabe/Research/RM_implicit_18.pdf`.

References for Systems, Languages and Libraries

[AFP] AFP. *Archive of Formal Proofs*. URL: `http://afp.sf.net` (visited on 12/20/2011).

[Art] R. Arthan. *ProofPower*. `http://www.lemma-one.com/ProofPower/`.

[Asp+06a] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. "Crafting a Proof Assistant". In: *TYPES*. Ed. by T. Altenkirch and C. McBride. Springer, 2006, pp. 18–32.

[Aus+03] R. Ausbrooks et al. "Mathematical Markup Language (MathML) v. 2.0." In: *World Wide Web Consortium recommendation* (2003).

[BCH12] M. Boespflug, Q. Carbonneaux, and O. Hermant. "The λΠ-calculus modulo as a universal proof language". In: *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*. Ed. by D. Pichardie and T. Weber. 2012, pp. 28–43.

[Bra13] E. Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593.

[Bus+04] S. Buswell, O. Caprotti, D. P. Carlisle, M. C. Dewar, M. Gaetano, and M. Kohlhase. *The Open Math standard*. Tech. rep. Version 2.0. The Open Math Society, 2004.

[CC] *CoCalc: Collaborative Calculation in the Cloud*. URL: `https://cocalc.com` (visited on 01/28/2019).

[CFO10] J. Carette, W. Farmer, and R. O'Connor. *The MathScheme Project*. `http://www.cas.mcmaster.ca/research/mathscheme`. 2010.

[Con+86] R. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[Dev16] T. S. Developers. *SageMath, the Sage Mathematics Software System (Version 7.0)*. 2016. URL: `http://www.sagemath.org`.

[DT] G. Dowek and F. Thiré. "Logipedia: a multi-system encyclopedia of formal proofs". URL: `http://www.lsv.fr/~dowek/Publi/logipedia.pdf`.

[Dun+15] C. Dunchev, F. Guidi, C. S. Coen, and E. Tassi. "ELPI: Fast, Embeddable, lambda-Prolog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by M. Davis, A. Fehnker, A. McIver, and A. Voronkov. 2015, pp. 460–468.

[FGT] W. M. Farmer, J. Guttman, and J. Thayer. *The IMPS online theory library*. URL: `http://imps.mcmaster.ca/theories/theory-library.html` (visited on 12/17/2014).

[FGT93] W. Farmer, J. Guttman, and F. Thayer. "IMPS: An Interactive Mathematical Proof System". In: *Journal of Automated Reasoning* 11.2 (1993), pp. 213–248.

[Gap] *GAP – Groups, Algorithms, and Programming, Version 4.8.2*. The GAP Group. 2016. URL: http://www.gap-system.org.

[GC14] D. Ginev and J. Corneli. "NNexus Reloaded". In: *Intelligent Computer Mathematics 2014*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. LNCS 8543. Springer, 2014, pp. 423–426. URL: http://arxiv.org/abs/1404.6548.

[Gog+93] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. "Introducing OBJ". In: *Applications of Algebraic Specification using OBJ*. Ed. by J. Goguen, D. Coleman, and R. Gallimore. Cambridge, 1993.

[Gro16] T. G. Group. *GAP – Groups, Algorithms, and Programming*. http://www.gap-system.org. [Online; accessed 30 August 2016]. 2016. URL: http://www.gap-system.org.

[Har+12] T. Hardin et al. *The FoCaLiZe Essential*. http://focalize.inria.fr/. 2012.

[Har96] J. Harrison. "HOL Light: A Tutorial Introduction". In: *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*. Springer, 1996, pp. 265–269.

[HHP93a] R. Harper, F. Honsell, and G. Plotkin. "A framework for defining logics". In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HOL4] The HOL4 development team. *HOL4*. URL: http://hol.sourceforge.net/ (visited on 12/17/2014).

[Hur09] J. Hurd. "OpenTheory: Package Management for Higher Order Logic Theories". In: *Programming Languages for Mechanized Mathematics Systems*. Ed. by G. D. Reis and L. Théry. ACM, 2009, pp. 31–37.

[Isa] *Isabelle*. Mar. 9, 2013. URL: http://isabelle.in.tum.de (visited on 03/27/2013).

[Jup] *Project Jupyter*. URL: http://www.jupyter.org (visited on 08/22/2017).

[KMM00] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[Lmfa] *LMFDB Knowledge Database*. URL: http://lmfdb.org/knowledge/.

[Lmfb] *LMFDB Knowledge Database entry for* Minimal Weierstrass equation over the rationals. URL: http://lmfdb.org/knowledge/show/ec.q.minimal_weierstrass_equation.

[LN12] F. Lübeck and M. Neunhöffer. *GAPDoc, A Meta Package for GAP Documentation, Version 1.5.1*. 2012. URL: http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc.

[Mat] *Mathematical Components*. URL: http://www.msr-inria.fr/projects/mathematical-components-2/.

[MC] *Mathematical Components*. URL: http://www.msr-inria.fr/projects/mathematical-components-2/ (visited on 12/17/2014).

[MeMa] *Metamath Home page*. URL: http://us.metamath.org.

[Mil+97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, Revised edition*. MIT Press, 1997.

[Miz] *Mizar*. http://www.mizar.org. 1973–2006. URL: http://www.mizar.org.

[MizLib] *Mizar Mathematical Library*. URL: http://www.mizar.org/library (visited on 09/27/2012).

[MN86] D. Miller and G. Nadathur. "Higher-order logic programming". In: *Proceedings of the Third International Conference on Logic Programming*. Ed. by E. Shapiro. Springer, 1986, pp. 448–462.

[Nor05] U. Norell. *The Agda WiKi.* http://wiki.portal.chalmers.se/agda. 2005.

[NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Springer, 2002.

[OAF] *The OAF Project & System.* URL: http://oaf.mathhub.info (visited on 04/23/2015).

[ODK] *OpenDreamKit Open Digital Research Environment Toolkit for the Advancement of Mathematics.* URL: http://opendreamkit.org (visited on 05/21/2015).

[ORS92] S. Owre, J. Rushby, and N. Shankar. “PVS: A Prototype Verification System”. In: *11th International Conference on Automated Deduction (CADE).* Ed. by D. Kapur. Springer, 1992, pp. 748–752.

[PC93] L. Paulson and M. Coen. *Zermelo-Fraenkel Set Theory.* Isabelle distribution, ZF/ZF.thy. 1993.

[PS99] F. Pfenning and C. Schürmann. “System Description: Twelf - A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction.* Ed. by H. Ganzinger. 1999, pp. 202–206.

[PVS] *NASA PVS Library.* URL: http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/ (visited on 12/17/2014).

[Qed] *The QED Project.* http://www-unix.mcs.anl.gov/qed/. 1996. URL: http://www-unix.mcs.anl.gov/qed/.

[Ran04] A. Ranta. “Grammatical Framework — A Type-Theoretical Grammar Formalism”. In: *Journal of Functional Programming* 14.2 (2004), pp. 145–189.

[RK13a] F. Rabe and M. Kohlhase. “A Scalable Module System”. In: *Information and Computation* 230.1 (2013), pp. 1–54.

[SC] N. M. T. et al. *Elements, parents, and categories in Sage: a primer.* URL: http://combinat.sagemath.org/doc/reference/categories/sage/categories/primer.html.

[SNG] *Singular.* URL: https://www.singular.uni-kl.de/ (visited on 08/22/2017).

[Sut09] G. Sutcliffe. “The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 337–362.

[SW83] D. Sannella and M. Wirsing. “A Kernel Language for Algebraic Specification and Implementation”. In: *Fundamentals of Computation Theory.* Ed. by M. Karpinski. Springer, 1983, pp. 413–427.

[Tea03] T. C. D. Team. *The Coq proof assistant reference manual (version 7.4).* Tech. rep. INRIA, Rocquencourt, France, 2003.

[Wen09] M. Wenzel. *The Isabelle/Isar Reference Manual.* http://isabelle.in.tum.de/documentation.html, Dec 3, 2009. 2009.

[WPN08] M. Wenzel, L. C. Paulson, and T. Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics (TPHOLs 2008).* Ed. by A. Mohamed, Munoz, and Tahar. LNCS 5170. Springer, 2008, pp. 33–38.

[Coq15] Coq Development Team. *The Coq Proof Assistant: Reference Manual.* Tech. rep. INRIA, 2015.

[LMF] T. LMFDB Collaboration. *The L-functions and Modular Forms Database.* http://www.lmfdb.org. [Online; accessed 27 August 2016].

Other References

[AH15] S. Autexier and D. Hutter. “Structure Formation in Large Theories”. In: *Intelligent Computer Mathematics 2015*. Conferences on Intelligent Computer Mathematics. (Washington DC, USA, July 13–17, 2015). Ed. by M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge. LNCS 9150. Springer, 2015, pp. 155–170.

[Arx] `arXiv.org` *e-print archive*. http://www.arxiv.org. 1994.

[Asp+06b] A. Asperti, F. Guidi, C. S. Coen, E. Tassi, and S. Zacchiroli. “A Content Based Mathematical Search Engine: Whelp”. In: *Types for Proofs and Programs, International Workshop, TYPES 2004, revised selected papers*. Ed. by J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner. LNCS 3839. Springer Verlag, 2006, pp. 17–32.

[Bet18] J. Betzendahl. “Translating the IMPS Theory Library to MMT / OMDoc”. Master’s Thesis. Informatik, Universität Bielefeld, Apr. 2018. URL: https://gl.kwarc.info/supervision/MSc-archive/blob/master/2018/jbetzendahl/thesis_imps2omdoc.pdf.

[BKR] K. Berčič, M. Kohlhase, and F. Rabe. “Towards a Unified Mathematical Data Infrastructure: Database and Interface Generation”. submitted to CICM 2019. URL: https://kwarc.info/kohlhase/submit/cicm19-MDH.pdf.

[BL] T. Breuer and S. Linton. “The GAP 4 Type System: Organising Algebraic Algorithms”. In: *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*. ISSAC ’98. ACM, pp. 38–45.

[Bla+14] J. C. Blanchette et al. “Truly Modular (Co)datatypes for Isabelle/HOL”. In: *ITP*. Ed. by G. Klein and R. Gamboa. Vol. 8558. LNCS. Springer, 2014, pp. 93–110. DOI: 10.1007/978-3-319-08970-6.

[Bob+11] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64.

[Bou64] N. Bourbaki. “Univers”. In: *Séminaire de Géométrie Algébrique du Bois Marie - Théorie des topos et cohomologie étale des schémas*. Springer, 1964, pp. 185–217.

[Car+19] J. Carette, W. M. Farmer, M. Kohlhase, and F. Rabe. “Big Math and the One-Brain Barrier – A Position Paper and Architecture Proposal”. submitted to Mathematical Intelligencer. 2019. URL: https://arxiv.org/abs/1904.10405.

[CF58] H. Curry and R. Feys. *Combinatory Logic*. Amsterdam: North-Holland, 1958.

[CH88] T. Coquand and G. Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2/3 (1988), pp. 95–120.

[Chu40] A. Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5.1 (1940), pp. 56–68.

[Cod+11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. “Project Abstract: Logic Atlas and Integrator (LATIN)”. In: *Intelligent Computer Mathematics*. Ed. by J. Davenport, W. Farmer, F. Rabe, and J. Urban. Springer, 2011, pp. 289–291.

[Dyb97] P. Dybjer. “Representing Inductively Defined Sets by Wellorderings in Martin-LÖF’s Type Theory”. In: *Theor. Comput. Sci.* 176.1-2 (Apr. 1997), pp. 329–335. DOI: 10.1016/S0304-3975(96)00145-4. URL: http://dx.doi.org/10.1016/S0304-3975(96)00145-4.

[EI] *EINFRA-9: e-Infrastructure for Virtual Research Environment*. URL: http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/2144-einfra-9-2015.html.

[ESC07] J. Euzenat, P. Shvaiko, and E. Corporation. *Ontology matching*. Springer, 2007.

[FGT92a] W. Farmer, J. Guttman, and F. Thayer. “Little Theories”. In: *Conference on Automated Deduction*. Ed. by D. Kapur. 1992, pp. 467–581.

[FGT92b] W. M. Farmer, J. Guttman, and J. Thayer. “Little Theories”. In: *Proceedings of the 11th Conference on Automated Deduction*. Ed. by D. Kapur. LNCS 607. Saratoga Springs, NY, USA: Springer Verlag, 1992, pp. 467–581.

[Geu+17] H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, eds. *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics. LNAI 10383. Springer, 2017. DOI: 10.1007/978-3-319-62075-6.

[Gin+16] D. Ginev et al. “The SMGloM Project and System. Towards a Terminology and Ontology for Mathematics”. In: *Mathematical Software - ICMS 2016 - 5th International Congress*. Ed. by G.-M. Greuel, T. Koch, P. Paule, and A. Sommese. Vol. 9725. LNCS. Springer, 2016. DOI: 10.1007/978-3-319-42432-3. URL: http://kwarc.info/kohlhase/papers/icms16-smglom.pdf.

[GK14a] T. Gauthier and C. Kaliszyk. “Matching concepts across HOL libraries”. In: *Intelligent Computer Mathematics*. Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. Springer, 2014, pp. 267–281.

[GK14b] T. Gauthier and C. Kaliszyk. “Matching concepts across HOL libraries”. In: *CICM*. Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. Vol. 8543. LNCS. Springer Verlag, 2014, pp. 267–281. DOI: 10.1007/978-3-319-08434-3_20.

[GK15] T. Gauthier and C. Kaliszyk. “Sharing HOL4 and HOL Light Proof Knowledge”. In: *LPAR*. Ed. by M. Davis, A. Fehnker, A. McIver, and A. Voronkov. Vol. 9450. LNCS. Springer, 2015, pp. 372–386. DOI: 10.1007/978-3-662-48899-7.

[GKU16] T. Gauthier, C. Kaliszyk, and J. Urban. “Initial Experiments with Statistical Conjecturing over Large Formal Corpora”. In: *Work in Progress at CICM 2016*. Ed. by A. Kohlhase et al. Vol. 1785. CEUR. CEUR-WS.org, 2016, pp. 219–228.

[Gon+13] G. Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving*. Ed. by S. Blazy, C. Paulin-Mohring, and D. Pichardie. 2013, pp. 163–179.

[Hal+15a] T. Hales et al. *A formal proof of the Kepler conjecture*. 2015. URL: http://arxiv.org/abs/1501.02155.

[Hal+15b] T. C. Hales et al. “A formal proof of the Kepler conjecture”. In: *CoRR* abs/1501.02155 (2015). URL: http://arxiv.org/abs/1501.02155.

[HHP93b] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[HKR12a] F. Horozal, M. Kohlhase, and F. Rabe. “Extending MKM Formats at the Statement Level”. In: *Intelligent Computer Mathematics*. Ed. by J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel. Springer, 2012, pp. 64–79.

[HKR12b] F. Horozal, M. Kohlhase, and F. Rabe. “Extending MKM Formats at the Statement Level”. In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by J. Jeuring, J. A. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 65–80. URL: http://kwarc.info/kohlhase/papers/mkm12-p2s.pdf.

[HKR14] F. Horozal, M. Kohlhase, and F. Rabe. “Flexary Operators for Formalized Mathematics”. In: *Intelligent Computer Mathematics 2014*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. LNCS 8543. Springer, 2014, pp. 312–327. URL: http://kwarc.info/kohlhase/papers/cicm14-lfs.pdf.

[Hor14] F. Horozal. “A Framework for Defining Declarative Languages”. PhD thesis. Jacobs University Bremen, Nov. 2014. URL: https://svn.kwarc.info/repos/fhorozal/pubs/phd-thesis.pdf.

[How80] W. Howard. “The formulas-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, 1980, pp. 479–490.

[HS02] M. Hofmann and T. Streicher. “The Groupoid Interpretation of Type Theory”. In: (Apr. 2002).

[Ian+13] M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. “The Mizar Mathematical Library in OMDoc: Translation and Applications”. In: *Journal of Automated Reasoning* 50.2 (2013), pp. 191–202.

[Ian+14] M. Iancu, C. Jucovschi, M. Kohlhase, and T. Wiesing. “System Description: MathHub.info”. In: *Intelligent Computer Mathematics.* Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. Springer, 2014, pp. 431–434.

[Ian17] M. Iancu. “Towards Flexiformal Mathematics”. PhD thesis. Bremen, Germany: Jacobs University, 2017. URL: https://opus.jacobs-university.de/frontdoor/index/index/docId/721.

[Jeu+12] J. Jeuring, J. A. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, eds. *Intelligent Computer Mathematics.* Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012.

[KK13a] C. Kaliszyk and A. Krauss. “Scalable LCF-style proof translation”. In: *Interactive Theorem Proving.* Ed. by S. Blazy, C. Paulin-Mohring, and D. Pichardie. Springer, 2013, pp. 51–66.

[KK13b] C. Kaliszyk and A. Krauss. “Scalable LCF-style proof translation”. In: *ITP.* Ed. by S. Blazy, C. Paulin-Mohring, and D. Pichardie. Vol. 7998. LNCS. Springer Verlag, 2013, pp. 51–66.

[Koh08] M. Kohlhase. “Using LaTeX as a Semantic Markup Format”. In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304.

[Koh13a] M. Kohlhase. “The Flexiformalist Manifesto”. In: *14th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012).* Ed. by A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie. Timisoara, Romania: IEEE Press, 2013, pp. 30–36. URL: http://kwarc.info/kohlhase/papers/synasc13.pdf.

[Koh13b] M. Kohlhase. “The OMDoc2 Language Design”. KWARC Blue Note. 2013. URL: http://gl.kwarc.info/omdoc/blue/raw/master/design/note.pdf.

[Koh14] M. Kohlhase. “Mathematical Knowledge Management: Transcending the One-Brain-Barrier with Theory Graphs”. In: *EMS Newsletter* (June 2014), pp. 22–27. URL: https://kwarc.info/people/mkohlhase/papers/ems13.pdf.

[KP93] M. Kohlhase and F. Pfenning. “Unification in a λ-Calculus with Intersection Types”. In: *Proceedings of the International Logic Programming Symposion ILPS’93.* Ed. by D. Miller. MIT Press, 1993, pp. 488–505. URL: Http://kwarc.info/kohlhase/papers/ilps93.pdf.

[KR14] C. Kaliszyk and F. Rabe. “Towards Knowledge Management for HOL Light”. In: *Intelligent Computer Mathematics.* Ed. by S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban. Springer, 2014, pp. 357–372.

[KR16a] M. Kohlhase and F. Rabe. “QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge”. In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 201–234.

[KR16b] M. Kohlhase and F. Rabe. “QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge”. In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 201–234. URL: http://jfr.unibo.it/article/download/4570/5733.

[KRSC11] M. Kohlhase, F. Rabe, and C. Sacerdoti Coen. “A Foundational View on Integration Problems”. In: *Intelligent Computer Mathematics*. Ed. by J. Davenport, W. Farmer, F. Rabe, and J. Urban. LNAI 6824. Springer Verlag, 2011, pp. 107–122. URL: http://kwarc.info/kohlhase/papers/cicm11-integration.pdf.

[KS10] A. Krauss and A. Schropp. “A Mechanized Translation from Higher-Order Logic to Set Theory”. In: *Interactive Theorem Proving*. Ed. by M. Kaufmann and L. Paulson. Springer, 2010, pp. 323–338.

[KU15] C. Kaliszyk and J. Urban. “HOL(y)Hammer: Online ATP Service for HOL Light”. In: *Mathematics in Computer Science* 9.1 (2015), pp. 5–22.

[KW03] H. Kanayama and H. Watanabe. “Multilingual translation via annotated hub language”. In: *MT-Summit IX*. 2003, pp. 202–207.

[KW10] C. Keller and B. Werner. “Importing HOL Light into Coq”. In: *Interactive Theorem Proving*. Ed. by M. Kaufmann and L. Paulson. Springer, 2010, pp. 307–322.

[KŞ06] M. Kohlhase and I. Şucan. “A Search Engine for Mathematical Formulae”. In: *Artificial Intelligence and Symbolic Computation*. Ed. by T. Ida, J. Calmet, and D. Wang. Springer, 2006, pp. 241–253.

[Luo09] Z. Luo. “Manifest Fields and Module Mechanisms in Intensional Type Theory”. In: *Types for Proofs and Programs*. Ed. by S. Berardi, F. Damiani, and U. de’Liguoro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–255.

[MH] *MathHub.info: Active Mathematics*. URL: http://mathhub.info (visited on 01/28/2014).

[ML74] P. Martin-Löf. “An Intuitionistic Theory of Types: Predicative Part”. In: *Proceedings of the ’73 Logic Colloquium*. North-Holland, 1974, pp. 73–118.

[ML94] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1994.

[MWP] *Matroid — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=Matroid (visited on 04/04/2018).

[NK07] I. Normann and M. Kohlhase. “Extended Formula Normalization for ε-Retrieval and Sharing of Mathematical Knowledge”. In: *Towards Mechanized Mathematical Assistants. MKM/Calculemus*. Ed. by M. Kauers, M. Kerber, R. Miner, and W. Windsteiger. LNAI 4573. Springer Verlag, 2007, pp. 266–279.

[Nor08] I. Normann. “Automated Theory Interpretation”. PhD thesis. Bremen, Germany: Jacobs University, 2008. URL: https://kwarc.info/people/archive/pubs/phd-2008/normann.pdf.

[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*. Ed. by R. Boulton and P. Jackson. Springer, 2001.

[OS06a] S. Obua and S. Skalberg. “Importing HOL into Isabelle/HOL”. In: *Automated Reasoning*. Ed. by N. Shankar and U. Furbach. Vol. 4130. Springer, 2006.

[OS06b] S. Obua and S. Skalberg. “Importing HOL into Isabelle/HOL”. In: *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*. Ed. by U. Furbach and N. Shankar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 298–302. DOI: 10.1007/11814771_27.

[Pfe+03] F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. *The Logosphere Project*. http://www.logosphere.org/. 2003.

[Pfe01] F. Pfenning. “Logical frameworks”. In: *Handbook of automated reasoning*. Ed. by J. Robinson and A. Voronkov. Elsevier, 2001, pp. 1063–1147.

[Pfe93] F. Pfenning. “Refinement Types for Logical Frameworks”. In: *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*. Ed. by H. Geuvers. Nijmegen, The Netherlands: University of Nijmegen, May 1993, pp. 285–301.

[Rab12] F. Rabe. “A Query Language for Formal Mathematical Libraries”. In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (CICM). (Bremen, Germany, July 9–14, 2012). Ed. by J. Jeuring, J. A. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge. LNAI 7362. Berlin and Heidelberg: Springer Verlag, 2012, pp. 142–157. arXiv: 1204.4685 [cs.LO].

[Rab17a] F. Rabe. “A Modular Type Reconstruction Algorithm”. In: *ACM Transactions on Computational Logic* (2017). accepted pending minor revision; see https://kwarc.info/people/frabe/Research/rabe_recon_17.pdf.

[Rab17b] F. Rabe. “How to Identify, Translate, and Combine Logics?” In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[RK13b] F. Rabe and M. Kohlhase. “A Scalable Module System”. In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[RKS11] F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. “A Foundational View on Integration Problems”. In: *Intelligent Computer Mathematics*. Ed. by J. Davenport, W. Farmer, F. Rabe, and J. Urban. Springer, 2011, pp. 107–122.

[Sol95] R. Solomon. “On Finite Simple Groups and Their Classification”. In: *Notices of the AMS* (Feb. 1995), pp. 231–239.

[SS89] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. LNAI 395. Springer Verlag, 1989.

[Sto+17] C. Stolze, L. Liquori, F. Honsell, and I. Scagnetto. “Towards a Logical Framework with Intersection and Union Types”. In: *11th International Workshop on Logical Frameworks and Meta-languages, LFMTP*. Oxford, United Kingdom, Sept. 2017, pp. 1 –9. URL: https://hal.inria.fr/hal-01534035.

[SW11] B. Spitters and E. van der Weegen. “Type Classes for Mathematics in Type Theory”. In: *CoRR* abs/1102.1323 (2011). arXiv: 1102.1323. URL: http://arxiv.org/abs/1102.1323.

[VJS] *vis.js - A dynamic, browser based visualization library*. URL: http://visjs.org (visited on 06/04/2017).

[Wat+14] S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, eds. *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics. (Coimbra, Portugal, July 7–11, 2014). LNCS 8543. Springer, 2014.

[Wie06] F. Wiedijk. *The Seventeen Provers of the World*. Springer, 2006.

[Wie92] G. Wiederhold. “Mediators in the architecture of future information systems”. In: *Computer* 25.3 (1992), pp. 38–49.

[WKR17] T. Wiesing, M. Kohlhase, and F. Rabe. “Virtual Theories – A Uniform Interface to Mathematical Knowledge Bases”. In: *MACIS 2017: Seventh International Conference on Mathematical Aspects of Computer and Information Sciences*. Ed. by J. Blömer, T. Kutsia, and D. Simos. LNCS 10693. Springer Verlag, 2017, pp. 243–257. URL: https://github.com/OpenDreamKit/OpenDreamKit/blob/master/WP6/MACIS17-vt/crc.pdf.

[WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1913.

[Uni13] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `http://homotopytypetheory.org/book`, 2013.