A Modular Formalization of Set Theory

Annika Schmidt
Friedrich-Alexander-Universität Erlangen-Nürnberg

20. September 2021

Abstract

The development of a reliable and flexible program to support the automation of reasoning gets more important considering the increasing amount of data. However, the formalizations in such a program require design choices that are unnecessary in informal mathematics, like the commitment to a specifc type system. Therefore formalizations are hardly comparable, although from a user perspective they are equivalent. To investigate improvements, some common features are formalized in multiple ways and variations. Especially interesting are morphisms that realize a feature in a different type system.

The inspiration for this work is the LATIN project which had similar goals using the LF framework in Twelf. However, Mmt is used for the formalizations, because it has some advantages compared to Twelf. Concretely, Mmt has the two additional morphisms realize and structure, new typing tricks, notations and roles (e.g. "role Simplify"). Furthermore, Mmt is build to use systematically different base languages to formalize features.

The purpose of this paper is to introduce the method to formalize set theoretical features in Mmt. Additionally, the idea of transformations between different base languages is partially realized. Furthermore the problems of such a high scaling program and solution approaches are discussed. The results of this work led to a new release of Mmt's library *LATIN2* and a graphical representation of it.

Declaration

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 20.09.2021

Annika Schmidt

1 Introduction

Motivation Normally mathematicians do not give an exact definition of their theoretical foundation (e.g. axiomatic set theory). Instead they leave the foundation implicit and just assume the foundation is available, in which all the advanced concepts they want to use have been defined in an indefinite way.

This is a problem for theorem provers which have one specific foundational language, that cannot be changed. For example a major design decision in the development of proof assistants is, whether the foundational language should be based on set theory or type theory. Even then there are numerous type theories and variants of set theory which can be chosen. Because every theorem prover uses a different foundation, the transmission of formalizations between two theorem provers is difficult. This does not only cause redundant work, but it also slows down the progress of formalizing mathematics in general.

Logical frameworks have been introduced to address this problem, e.g. the Twelf implementation of LF. In a logical framework we can formalize all foundations in parallel and then develop formal translations between them. In particular we can capture the semantics of type theory as the translation of type theory to set theory.

The Twelf module system by Rabe and Schürmann [RS09] was constructed to support the development of large collections of formalizations of different foundations and translations between them. This system was used in the *LATIN project* [Cod+11] to build an atlas of logics, translations and associated topics.

The *LATIN2* library is a modern reimplementation of *LATIN*'s library using the latest support features in Mmt [Rab20]. The present thesis is a central part of the *LATIN2 project* by systematically developing formalization of set theory and the semantics of the various type theories in *LATIN2* translated in set theory.

Contribution More concretely, the present work is a formalization of set theory as a theory graph containing more than 70 different theories and additionally multiple views. These formalizations are modular which allows reusing them and build different variants of set theory. For example, *Zermelo-Fraenkel set theory* can be obtained by combining the modules for basic definitions of set theory, the necessary axioms and the features representing their realizations. Optionally, multiple desired features can be added as long as a valid realization for them exists.
For example, a necessary axiom would be the pairing axiom, its realization are unordered pairs and an additional feature based on unordered pairs could be singletons.

More precisely, the formalization includes views that show how various set theoretical features can be realized in terms of each other. For example, unordered pairs realize the pairing axiom, which is expressed as a view. Another example is that the feature for singletons can

be realized by the feature for unordered pairs, if the unordered pair containing twice the element a is interpreted as the singleton a.

A small sketch of this idea is shown in figure 1, where blue rectangles represent collections of theories and purple ones with rounded corners display the (example) theories. Normal black lines represent that at least one theory of the collection is included in the collection to which the arrow points. If the arrows starts at a theory and ends in a collection, that theory is part of the collection. The bigger white arrows represent views, where the theory with the starting point is used to realize the theory at the arrow end.

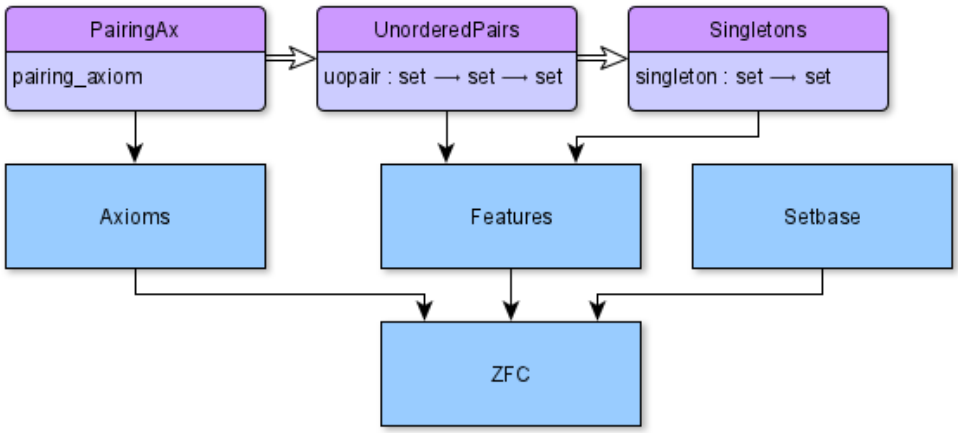

Figure 1: Sketch for ZFC

When systematically refactoring individual features of set theory into their own theories, in a way that abstracts from the set theoretical foundation, we notice a gradual transition from formalizations of set theory to formalizations of type theory. For example, if the feature of Cartesian products is formalized in its own abstract theory, it is isomorphic to the feature of product types in type theory.
Contrary to the usual dichotomy of type and set theory, this approach shows them to be just different ways of interpreting the same feature. In particular, the views defining the semantics of type theory in set theory add up just being structurally similar to the views relating different language features. For example, if types are interpreted as sets, product types are equivalent to the Cartesian product.

This work is one of the first major case studies evaluating recent features of MMT. And thus provides valuable feedback to the design of MMT as a whole. In particular, we had to improve MMT regarding issues that become most apparent at large scale, such as the organization of content into packages and files as well as build processes and the management of names for large libraries.

Overview Section 2 discusses related work. In section 3 the syntax and morphisms in MMT/LF are explained using some examples from set theory. The following sections 4 and 5 describe our formalization of set theory in MMT and transformations of various features into set theory. Lastly, section 7 recaps our work and outlines future work.

2 Related Work

This work is mostly about the formalization of set theory and transformations of type theory into set theory. Therefore it is helpful to take a look at the five systems *Coq*, *Lean*, *Isabelle/ZF*, *Mizar* and *Nuprl*. Information about modularity and module systems has already been given by Rabe and Kohlhase in [RK13].

Coq The formal language of Coq is *Calculus of Inductive Constructions (CIC)* which is based on type theory. The library *Sets* contains set theoretical features in Coq. Within this library, often a variable U is used to denote the universe that contains all elements. Since U is just formalized as a type, each element of the universe has type U.
The type for sets is *Ensemble* which is a function that receives an element of type U and returns a *Prop*. Basically, Ensemble has already an inbuilt membership function, but still an explicit membership function *In* is defined as $A\ x$ with $A :$ Ensemble and $x : U$. Features of type Ensemble are formalized with natural deduction rules [Cst; Cty; Cmo; RK13].

Furthermore Coq has formalizations for the properties of features in form of lemmas or definitions. Often these are separated from the formalization of the feature itself. For some features Coq introduces a new type, like relations get the type Relation which is defined as $U \to U \to Prop$. Another example would be *nat* for the Peano natural numbers, which is already defined in *Init* using the abstract keyword *Set*. *Set* is also used for Coq's approach to migrate sets to type by interpreting *Set* as *Type*.
Moreover Coq provides useful definitions for set theoretic features like *Disjoint* and *Same_set*. The latter is based on the subset relation *Included* and is used to define the extensionality axiom for Ensembles. However, this seems to be the only axiom for set theory in Coq's standard library. Further implementations of set theory in Coq have been performed in e.g. [Kai12] or [SY20]. A formalization for ZFC in Coq is provided at [Czf] [Cst].

Lean Like *Coq* the Lean theorem prover uses the *Calculus of Inductive Constructions* and is based on dependent type theory. Lean has different levels for the design of set theory. The basic level is in its core library, where the type for set and some set theoretic features are formalized. First of all, two universes u and v are introduced that contain all elements. Then set is defined as $\alpha \to Prop$ with $\alpha :$ Type u and therefore α is an element of the universe u. The membership function and subsets are defined just like in Coq.
In the core library features like *sep* for separation are defined using the set builder notation, e.g. $\{a | a \in s \wedge p\ a\}$. Therefore it seems like Lean might have an even deeper level which supports the common handwritten notations [Lpr; Lwh; AEU18; Lma; Lin; Lse].

The next level is located in the mathematical library, where the datatype set is defined in more detail. This includes lemmas and properties for the features that were defined in core as well as the introduction of new features. For example, the Cartesian product is defined in this level together with its properties.
The final level is a model of ZFC, for which pre-sets are defined. Later in the formalization, ZFC sets are defined with pre-sets and then they are used to define classes. While pre-sets are defined, the axioms for ZFC get formalized as well [Lzf; Lax; Lma; Lse].

Isabelle/ZF In general, Isabelle is based on simple type theory. However, Isabelle/ZF imitates untyped set theory by using type i for all sets which is defined as a term. Isabelle/ZF supports axiomatic formalizations, e.g. for the membership relation or ZF axioms. Still most of the features have a definition that either depends on first-order logic or previously formalized features.
Natural deduction rules and some properties of the features are formalized as lemmas. For the proofs of the lemmas abstract definitions of the features and schemes for an automated theorem prover are used. This approach can be useful for more modular systems and could be interesting for Mmt as well [RK13; Wen21b; Wen21a; Wen21c; Ido; Izf; Ipu].

There exists a formalization of ZF(C) in Isabelle/HOL as well which formalizes ZF(C) as a HOL type. Interestingly there is already a theory Set which formalizes sets as predicates among the imported theories of HOLZF, but in HOLZF the new type *ZF* is declared and used for sets. Additionally, the feature *Elem* is axiomatically defined with the type $ZF \to ZF \to bool$. Therefore Elem receives two sets x and A and returns whether x is an element of A which is equivalent to a membership function.
Features in HOLZF get either defined axiomatically or with previously formalized features. In general the whole structure of HOLZF is similar to the one in e.g. ZF_Base of Isabelle/ZF. Still there appears to be no connection between these two theories, except that they formalize the same set theory in different variants [Wen21b; Wen21a; Wen21c; Ido; Izf; Iho].

Mizar Although Mizar is based on untyped set theory, it has a type system that can be interpreted as soft typing. The *Mizar Mathematical Library (MML)* is based on Tarski-Grothendieck set theory (TG) and first-order logic. The axioms of TG are formalized twice: once completely axiomatically and then as features that get proven using the axioms. Additionally, properties of the features are defined and proven.
Mizar has numerous formalizations of features with their properties, lemmas and models. However, all formalizations except the TG axioms are not axiomatically. Instead there are proofs that refer to previously defined features [Wie07; Mmm; Mgr; Min].

At the start of a Mizar article, the environment is defined which is basically a series of includes of vocabulary, notations, theorems and more. Then variables might be reserved to represent an element of a special type, e.g. *set*. Often in a formalization a statement about

the type of a variable is made. For example, "for x being set" or "let x be set".
In order to have a meaningful soft-type system, it is necessary to define the types. For example, for ordered pairs a new type *pair* is introduced and defined as exists $x1, x2$ such that $x = [x1, x2]$. The interpretation of $[x1, x2]$ as Kuratowski pair has already been defined together with the TG features that realize the axioms [Wie07; Mmm; Mgr; Min].

Nuprl Nuprl is based on an extended version of Martin-Löf's intuitionistic type theory, but its logic is designed in such a way that its primitives can build a set theory with a similar expressiveness to *ZFC* on top of the type theory. Interestingly, the membership relation $x \in T$ is defined for types instead of sets to denote that an object x belongs to type T. In general it seems like types and sets are basically the same in Nuprl.
This idea is already supported in Nuprl's core library, where e.g. *True* is defined as $0 \in \mathbb{Z}$. Furthermore, *subtype* is defined as $\forall x : S.x \in T$ which is equivalent to the definition of subsets in set theory [Kre02; Nov; Nma; Nbr; Ncr; Neq; Nty].

Moreover, other set theoretic features are formalized in Nuprl's type theory as well. Some examples are singletons, Cartesian products and disjoint union that all work on types instead of sets.
Properties of features are stated as theorems that get proven using subgoals. Since Nuprl tends to have a documentation for its features, the properties are often just part of the documentation. Additionally, the documentation refers to sections that are required by a feature and mostly these sections belong to the Nuprl standard library [Kre02; Nov; Nma; Nbr; Ncr; Neq; Nty].

3 Preliminaries

The "Meta Meta Tool" (or MMT) gets developed at FAU Erlangen-Nürnberg. It is a framework that is generic enough to design logical frameworks. Additionally MMT has a standard library that already contains some logical frameworks. The most popular one is the Edinburgh Logical Framework which is abbreviated as *LF*. In case the reader is not familiar with *LF* it is recommended to take a look at [HHP93].
Thus the term MMT/LF refers to the logical framework *LF* which is used in the MMT environment. Therefore typical *LF* features can be used with MMT/LF's syntax. A
λ-abstraction $\lambda_{x:A}t$ is written as $[x : A]t$, where A is a type, x is of type A and t is a term. Correspondingly, the dependent function type $\Pi_{x:A}B$ is noted as $\{x : A\}B$ which can be abbreviated as $A \to B$. Of course types, inferable brackets and arguments can be omitted as well. Figure 2 shows the grammar of MMT/LF. The MMT/LF specific features are explained in more detail below [IR11; Rab20; RR21; RK13].

A MMT document consists of multiple declarations. The two most important declaration types are *theory declarations* and *constant declarations*. The former has the keyword *theory*

[1] m has to be a previously defined morphism or an implicitly generated identity morphism

Δ	::=	$\cdot$	diagrams
	$\mid$	$\Delta, \text{theory } T := \{\Theta\}$	theory declaration
	$\mid$	$\Delta, \text{view } S \to T := \{\vartheta\}$	view declaration
Θ	::=	$\cdot$	declarations in a theory
	$\mid$	$\Theta, c[: A][= t]$	optionally typed or defined constant
	$\mid$	$\Theta, \text{include } S \mid \Theta, \text{realize } S$	include/realization of a theory
	$\mid$	$\Theta, [\text{total}]\ \text{structure } s : S := \{\vartheta\}$	optionally total structure
ϑ	::=	$\cdot \mid \vartheta, c = t \mid \vartheta, \text{include } m^1$	declarations in a morphism
	$\mid$	$\vartheta, [\text{total}]\ \text{structure } s : S := \{\vartheta\}$	optionally total structure
Γ	::=	$\cdot \mid \Gamma, x : A$	contexts
t, A, f	::=	$c \mid x \mid type \mid kind \mid \lambda_{x:A} t \mid \Pi_{x:A} B \mid ft$	MMT/LF expressions

Figure 2: MMT/LF grammar [RR21; RK13]

and is followed by a local name T. Optionally it can have a type M which is a *meta-theory* (see [Rab20]). Finally, a theory declaration has a body Θ which consists of constant declarations and morphisms. The syntax of a theory declaration is theory $T[: M] = \Theta$.
A constant declaration has no keyword and appears normally only within the body of a theory declaration. Similarly to a theory declaration, the constant declaration consists of a local name c and an optional type A. Also, it has an optional definiens t and a notation N which is often used to introduce a mathematical symbol. The syntax of a constant definition is $c[: A][= t][\#N]$. Although only the local name c is required, it is common to give at least the type A or a definiens t as well [IR11; Rab20; RR21].

Additionally, MMT supports the theory morphisms *include*, *realize*, *view* and *structure* which are indicated by their corresponding keywords. If a theory T includes a previously defined theory S all declarations and includes made in S are available in T as well. An include can be used any time definitions of another theory S are needed.
A similar morphism is realize. Similar to include, all declarations of S can be used in T, if a theory T realizes a previously defined theory S. However, every declaration in S that has no definition has to be defined in T. Even includes in S have to be defined in T, but mostly the identity morphisms of the theories just get included in T as well. The realize morphism is useful, if a declaration can have multiple different definitions. In the example 3 `KuratowskiPairs` realizes `OrderedPairs`, since it is wanted to keep all properties of `OrderedPairs`. However, *pair* should not be defined directly in `OrderedPairs`, since there are many definitions for pairs and some users might prefer Wiener's definition of a pair [IR11; RR21; SM08].

The first named morphism is view which has a domain theory S and a codomain theory T. A view v with domain S and codomain T is basically the same as if T realizes S. Still a view is different from a realize, since it is not written within the theory T it does not get included whenever T is included in a theory U. Additionally, a view might allow multiple

codomains in future versions of Mmt. For now, multiple codomains can still be imitated by creating a new theory C that includes all codomains. Then C is used as the codomain of the view. Therefore a view should be used, if there are either multiple codomains or the definition should not be bound to a theory T [IR11; RR21].

In example 3 there is the view `SelfPair` with domain `Singletons` and codomain `UnorderedPairs`. This could be done as a realize in `Singletons`, but then *singleton* would always be defined with `UnorderedPairs` and therefore a dependency would be created. When using a view there is no direct dependency between both theories. If `Singletons` should be included with the definition in `SelfPair`, this is possible by writing include `Singletons` = `SelfPair` [IR11; RR21].

theory `Singletons` =
  singleton : $set \to set$

theory `UnorderedPairs` =
  uopair : $set \to set \to set$

theory `OrderedPairs` =
  pair : $set \to set \to set$

theory `KuratowskiPairs` =
  include `Singletons`
  include `UnorderedPairs`
  realize `OrderedPairs`
  pair = $[a, b]$ uopair (singleton a) (uopair a b)

view `SelfPair` : `Singletons` → `UnorderedPairs` =
  singleton = $[a]$ uopair a a

Figure 3: Example for morphisms in Mmt/LF

The last morphism is structure which is also named. Theory T can have a structure referring to a previously defined theory S. A *total structure* is basically the same as a realize, but every declaration has to be called using the structure name. In the example 4 the *refl* in the structure *eq* would be called using eq/refl. In a realize, it would be called just using refl. If *urefl* would have a notation in `RelationRefl`, it is recommended to repeat the notation definition in the structure. Otherwise there might occur an error when using the build script. Another difference is that a structure which is not total does not have to define everything of S. Therefore it can be useful, when not all undefined declarations of S shall be defined in T or when the defined declarations should not be used by default [IR11; RR21].

4 Formalization of Set Theory

For mathematicians set theory is one of the most important concepts. Therefore a theorem prover should provide at least some set-theoretic features, if it wants to gain popularity. Mmt's predecessor *Twelf* already implemented some features of *ZFC*. However, these formalizations are not modular and therefore it was not possible to translate them exactly into Mmt/LF. Still they provided assistance regarding proofs and an overview of some important features. Of course, there also might have been some inspiration of the theorem provers that have been discussed in Related Work [Sal19; Twe; IR11].

theory `RelationRefl` =
  carrier : *type* # c
  rel : $c \to c \to c$ # $
  refl : $\{x\}\ x\$x$

theory `UntypedLogic` =
  prop : *type*
  ded : *prop* → *type* # ⊢
  term : *type*

theory `UntypedEquality` =
  include `UntypedLogic`
  uequal : *term* → *term* → *term* # $=^u$
  urefl : $\{x\} \vdash x =^u x$
  total structure eq : `RelationRefl` =
    carrier = term
    rel = $[x, y] \vdash x =^u y$
    refl = [x] urefl

Figure 4: Example for structures in MMT/LF

The graph in figure 5 displays an overview of the contents in this chapter, where each rectangle represents a collection of theories. *FOL* stands for first-order logic on which set theory is based. Although all remaining collections are part of set theory, *Type Base* will not be discussed until section Transformations into set theory. If an arrow points toward a collection, at least one theory of this collection is dependent on a theory in the collection with the starting point. For example, *Finite Sets* is dependent on *Set Base*. Additionally, the dotted arrow indicates that *Cartesian product* is realized in *Pair Definitions*.

4.1 Basics of set theory

There exist different variants of set theory. Beside the distinction of *naïve* and *axiomatic* set theory in general, the latter can be divided into multiple variants. Some of the most famous axiomatic set theories are *Zermelo-Fraenkel set theory* with or without the choice axiom (ZFC or ZF) and *Tarski-Grothendieck set theory* (TG). However, MMT shall be as independent of a specific set theory as possible. Therefore multiple different set theories can be formalized and features are defined axiomatically whenever this is practicable. Thus every user can work with his own desired set theory and does not need to redefine each feature [IR11; Pin14; Wie07].

Set Base Although there are different set theories, they have some characteristics in common. First of all they need a definition of a set. Since the formalizations of set theory are based on first-order logic, in MMT a set is defined as a term. Therefore all operations of first-order logic can be applied to sets as well.
Also, membership needs to be formalized which determines, whether an element belongs to a set or not. In MMT this is just an abstract formalization [IR11; Pin14; Sal19].

Other important features are subsets and the equality of sets which is normally defined using the extensionality axiom. These can either be considered individually for each further feature or they get added to the core definitions of set theory. The former approach is

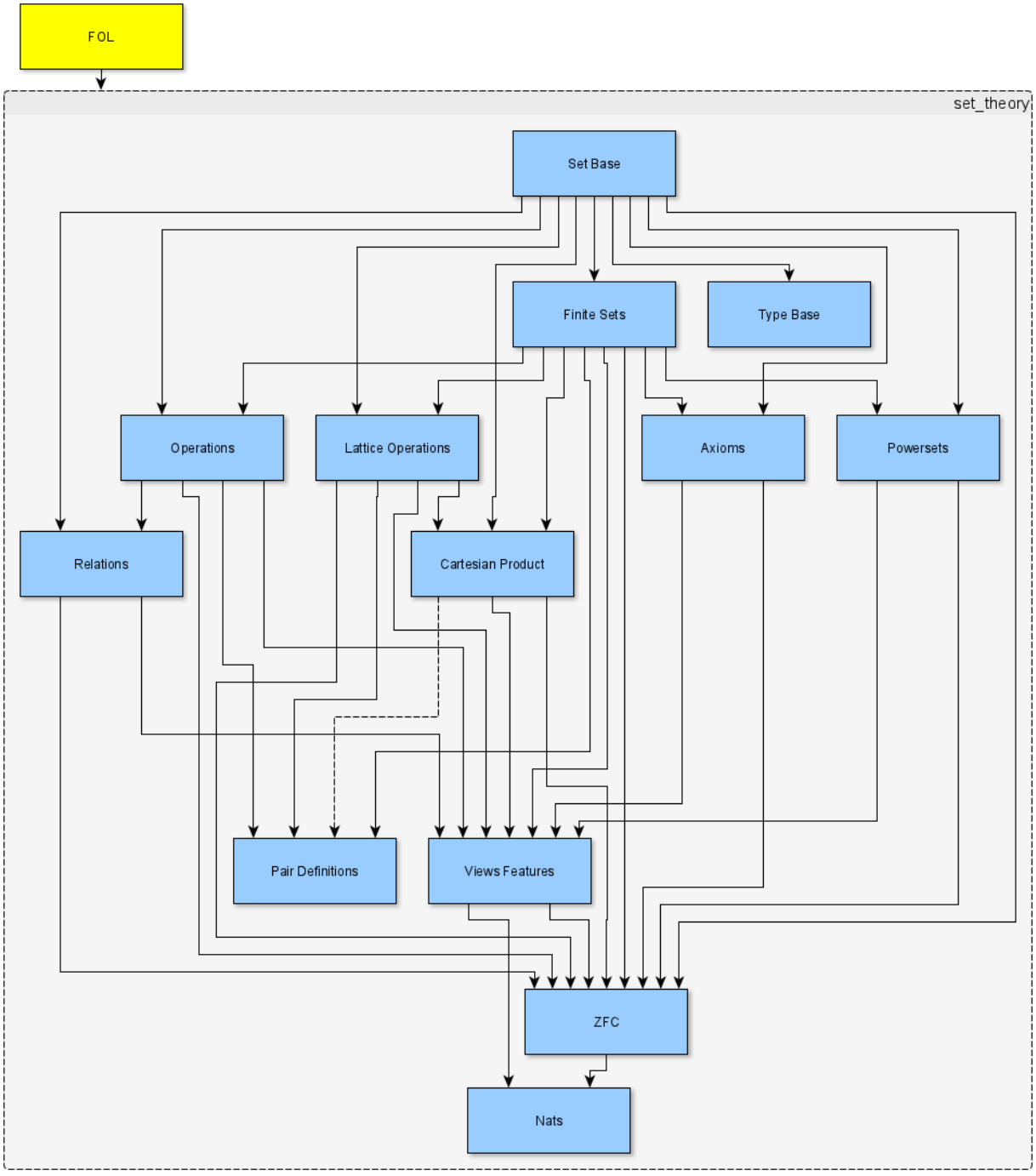

Figure 5: Overview of set theory

called "cross-cutting" and has the benefit that the specific set theories are not required to implement subsets and extensionality. However, this might be needless, because subsets and extensionality are key features of set theory and it is very unlikely that a specific set theory does not use them.
The downside of cross-cutting is an impractical amount of additional theories. Every time a later formalized feature has a property regarding subsets or extensionality, a new theory for this property has to be defined. Therefore the number of theories would grow faster than necessary which could soon be confusing. Thus the MMT formalization includes subsets and extensionality in the basic definitions for set theories which are done in *setbase.mmt* [IR11; Pin14; Sal19].

theory `SetDefinitions` =
  set = term
  in : $set \to set \to prop \ \# \in$

theory `ExtensionalityAx` =
  include `SetDefinitions`
  isextensional = $[A, B](\forall^u[x](x \in A \Leftrightarrow x \in B)) \Leftrightarrow A =^u B$
  extensionality_axiom : $\{A, B\} \vdash$ isextensional $A\ B$

theory `SubsetDefinitions` =
  include `SetDefinitions`
  subset : $set \to set \to prop \ \# \subseteq$
    $= [A, B]\forall^u[x]x \in A \Rightarrow x \in B$

theory `SetBase` =
  include `SetDefinitions`
  include `ExtensionalityAx`
  include `SubsetDefinitions`
  include `UniverseNonEmpty`

Figure 6: Excerpt of *setbase.mmt*

An excerpt of *setbase.mmt* is given in figure 6. It contains the most important concepts of set theory that result in the theory `SetBase`. The missing theory `UniverseNonEmpty` had already been defined in MMT's type theory. It allows an MMT user to introduce fresh variables within a proof. The remaining included theories in `SetBase` are only sketched and might consist of more declarations. However, natural deduction rules and further definitions for proving are not relevant for examples and therefore will be usually skipped.
Also, it should be mentioned that *setbase.mmt* provides some additional theories that define properties of sets like being empty. These definitions are not included in `SetBase`, but

they are useful for the formalization of some features which include their required definitions.

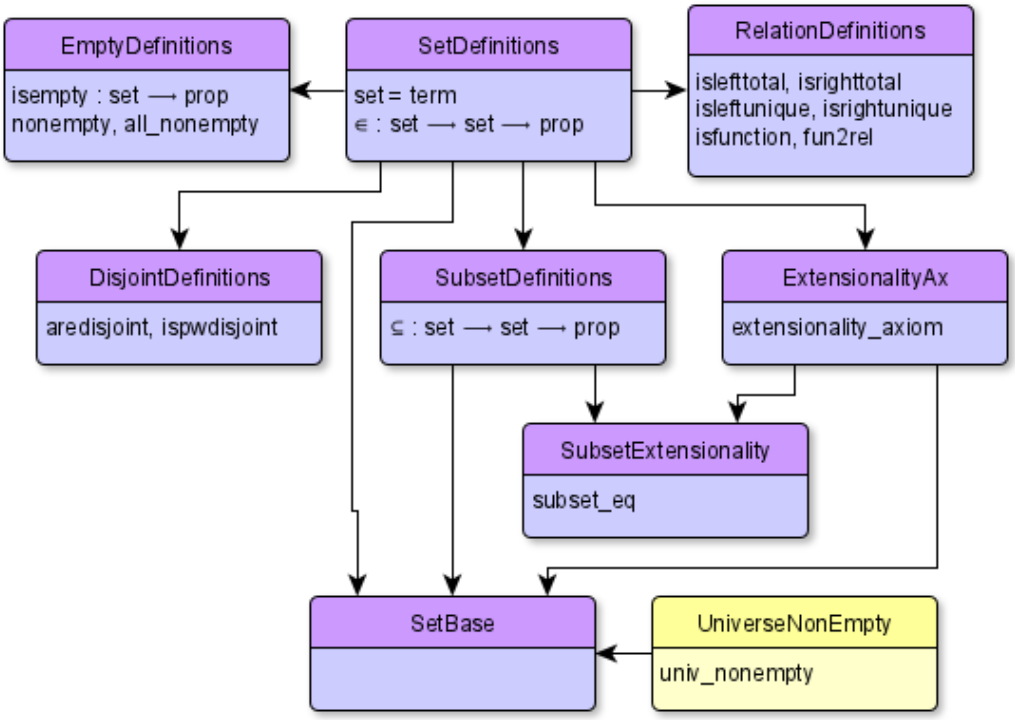

Figure 7: Overview of *setbase.mmt*

The graph 7 displays the theories that just have been defined. Purple rectangles with rounded edges represent theories which are formalized in *setbase.mmt*. Yellow rectangles illustrate theories that are not part of this MMT file. However, some violet theories include the yellow ones, therefore they are necessary in the graph. The includes are represented as arrows, where the included file is the starting point. For further graphs analogous interpretations can be used.

ZFC axioms Although MMT shall be independent of a specific set theory, it is necessary to use a set theory for consistency. Therefore a version of ZFC that is similar to the one in *Twelf* is formalized in MMT. The used axioms of ZFC are [Twe; IR11; Hal60; Wel20]:

- Extensionality: $\forall A \forall B (\forall x. x \in A \Leftrightarrow x \in B) \Leftrightarrow A = B$
- Set Existence: $\exists x$ true (included to reduce the dependency on the infinity axiom)
- Unordered Pairing: $\forall A \forall B \exists Z (\forall x. x \in Z \Leftrightarrow x = A \vee x = B)$
- (General) Union: $\forall A \exists Z (\forall x. x \in Z \Leftrightarrow \exists X. X \in A \wedge x \in X)$

- Powerset: $\forall A \exists Z(\forall B.B \in Z \Leftrightarrow (\forall x.x \in B \Rightarrow x \in A))$
- Comprehension: $\forall A \exists Z(\forall x.x \in Z \Leftrightarrow x \in A \wedge P(x))$, where P is a unary predicate that may contain free variables
- Replacement: $\forall A(\forall x.x \in A \Rightarrow \exists^! y.f(x,y)) \Rightarrow \exists Z(\forall y.y \in Z \Leftrightarrow \exists x.x \in A \wedge f(x,y))$, where $\exists^!$ abbreviates the quantifier for unique existence and f is a binary predicate that may contain free variables
- Regularity: $\forall A(\exists x.x \in A) \Rightarrow \exists B.B \in A \wedge \neg(\exists x.x \in A \wedge x \in B)$
- Infinity: $\exists A(\varnothing \in A \wedge \forall x.x \in A \Rightarrow \text{succ}(x) \in A)$, where $\varnothing$ is the empty set and succ is a successor function (normally $\text{succ}(x) = x \cup \{x\}$; this means succ is the union of x and the set containing x)
- Choice: Let A be a family of pairwise disjoint non-empty sets, then $\forall A \exists C \forall X.X \in A \Rightarrow \exists^! c.c \in C \wedge c \in X$, where $\exists^!$ abbreviates the quantifier for unique existence

In general the formalizations of axioms are based on `SetBase` and are located in *axioms.mmt*. However, extensionality is part of `SetBase` and set existence is equivalent to `UniverseNonEmpty` and therefore not formalized again. Whenever it is practical natural deduction rules for an axiom are defined together with the axiom itself. To avoid inconsistencies, the rules need to be proven.
Even though all axioms belong to ZFC by now, there exists a separate file for ZFC. Since other set theories shall be formalized as well, it can be helpful to reuse some of the axioms. Especially, if an individual set theory only differs slightly from an already existing one.

4.2 Features without element construction

Now that the foundation is done, features for set theory can be formalized. However, there are some different types of features. Also, sometimes it makes sense to locate features in different MMT files in order to make the MMT files clearer.

Finite Sets The first group of features are finite sets. Admittedly this name might be irritating, since most features result in finite sets. Therefore a name change to "countable sets" is worth considering, but finite sets seems to be more common.
Finite sets contain the empty set, singletons and unordered pairs which are also called doubletons. Accordingly each of these features creates a set containing exactly no, one or two distinguishable elements. Apart from the empty set which has no arguments, every finite set gets its elements as arguments. Of course there could be formalized an infinite amount of features of this sort, but in most cases these are sufficient [Pin14; Sal19; IR11].

In 3 the theories for singletons and unordered pairs have been just sketched. Figure 8 shows how the theory for unordered pairs actually looks like in MMT. After the include of `SetBase`

theory `UnorderedPairs` =
  include `SetBase`
  uopair : $set \to set \to set$
  uopairIl : $\{A, B\} \vdash A \in \text{uopair } A\ B$
  uopairIr : $\{A, B\} \vdash B \in \text{uopair } A\ B$
  uopairE : $\{A, B, x, C\} \vdash x \in \text{uopair } A\ B \to$
    $(\vdash x =^u A \to \vdash C) \to (\vdash x =^u B \to \vdash C) \to \vdash C$
  uopair_comm : $\{A, B\} \vdash \text{uopair } A\ B =^u \text{uopair } B\ A =$ ... (proof omitted)

Figure 8: Theory `UnorderedPairs`

the type of an unordered pair gets defined. Then there are two introduction rules for natural deduction which state that both arguments of an unordered pair are actually elements of the unordered pair. For the elimination rule it would be sufficient to infer that if x is an element of the unordered pair $A\ B$, then x has to be either A or B. However, in actual proofs it would be necessary to eliminate the disjunction as well. Therefore the disjunction elimination is already build into the elimination for an unordered pair. Lastly, an unordered pair is commutative which gets proven using extensionality. It shall be noted that neither the unordered pair itself nor its natural deduction rules have a proof. Since these features shall be defined axiomatically to support modularity, it is impossible to proof this now [Pin14; Sal19; IR11].

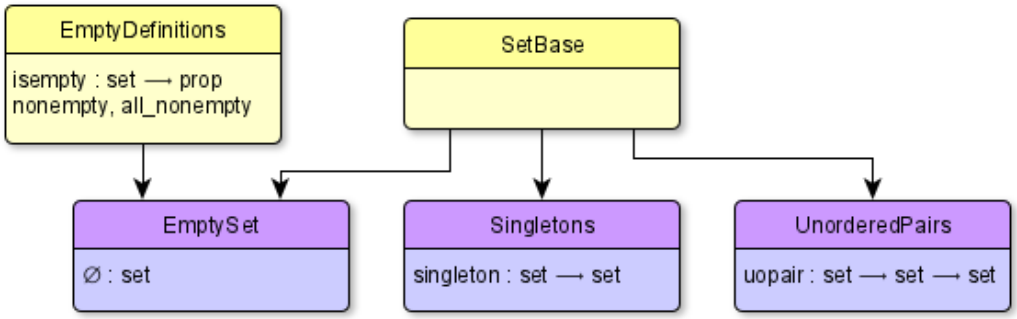

Figure 9: Overview of *finite_sets.mmt*

Powerset The power set of a set A is the set containing all subsets of A. This offers two options for the formalization: either referring to subsets or to indirectly formalize subsets again. The latter would be possible in a set theory without subsets as well. However, it would probably not be rational to have power sets in a set theory without the concept of subsets. Therefore this consideration is meaningless, but the aspect of usability is not. If the variant with subsets is chosen, then an extra proof step to show the subset relation might be necessary. The variant without subsets does not need this extra step. At least, if its

argument is not already a subset relation.
Since both variants can be beneficial, both are formalized. This does not cause much more work, because only the natural deduction rules have to be formalized in both variants. The second variant can even be proven using the first variant, which is useful for realizations [Pin14; IR11].

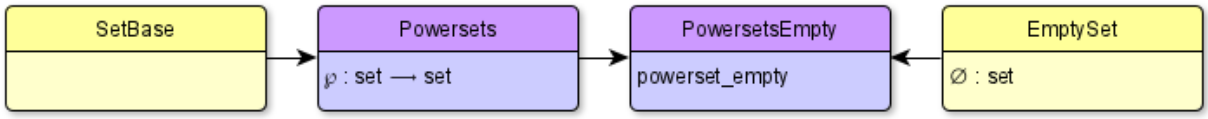

Figure 10: Overview of *powersets.mmt*

Lattice Operations Here lattice operations mean union and intersection, general union and intersection (labeled as "BigUnion" and "BigIntersection"), difference and complement. These operations are some of the most typical operations of set theory that should never be left out. BigUnion is the union operation for a family of sets and correspondingly BigIntersection is the intersection operation for a family of sets. However, the formalization of these features is still axiomatically. Therefore there is no dependency between the general operations and union or intersection. Due to the fact that there cannot be a universal set containing all other sets, difference and complement are nearly identical in their formalization. They only differ noticeably in their properties [Pin14; Sal19; IR11].

theory `EmptySet` =
  include `SetBase`
  emptyset : $set \# \varnothing$

theory `Union` =
  include `SetBase`
  union : $set \to set \to set \# 1 \cup 2$

theory `UnionEmpty` =
  include `Union`
  include `EmptySet`
  union_empty : $\{A\} \vdash A \cup \varnothing =^u A = \ldots$ (proof omitted)

Figure 11: Sketch of the theory `UnionEmpty` and its included theories

Some features have properties that rely another feature. An example for this is *union_empty* which states that a union of a set A with the empty set results in A. Basically there a three options to formalize this property. First would be to formalize this in the theory `EmptySet`, then `Union` would have to be included there. Second is the formalization in the theory

`Union`, where `EmptySet` would have to be included. The third option is to do cross-cutting and create a new theory `UnionEmpty` which includes both `EmptySet` and `Union`. The only purpose of `UnionEmpty` would be to define properties that solely rely on these two theories. Since a modular formalization is anticipated, the last option is preferably, because it does not create a preventable dependency between the two features. However, it creates an additional theory which has to be included every time this property is needed. Fortunately, these properties are used seldom and since it can be defined easily without creating further dependencies, it does not produce much extra work. The formalization of `UnionEmpty` is sketched in figure 11 [Pin14; Sal19].

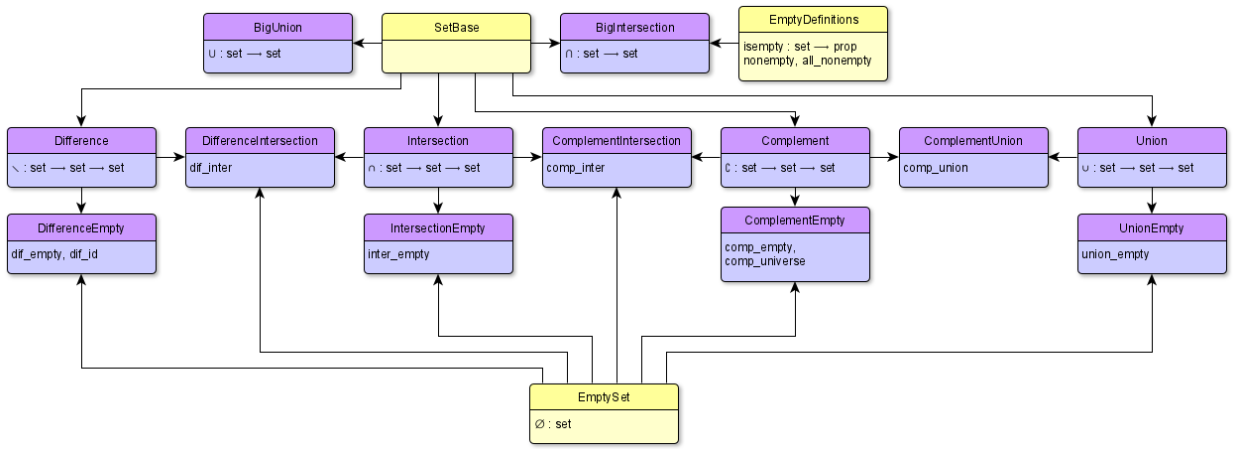

Figure 12: Overview of *lattice_ operations.mmt*

Operations The operations filter, replace, image, symmetric difference, adjoin and remove are not formalized in a different way than lattice operations. Still there had to be different files to increase the performance and maintainability of the corresponding MMT files. Filter and replace are the corresponding features to the comprehension and replacement axiom. Image does not rely on set theory and therefore could be realized from a typed theory in a later version. Nearest to the lattice operations are symmetric difference, adjoin and remove, since they are similar set modifications. Therefore they could be added to lattice operations, but if more similar features get formalized, the perfomance of lattice operations might decrease a lot. Thus for the moment, all these features belong to *operations.mmt* [Pin14; IR11].

Scheme for features without element construction Each feature mentioned in this chapter does not create new elements. This means that every element in the resulting set is either an input set or some kind of element of an input set. The feature just "picks" the desired elements from its arguments. If such a feature shall be formalized in MMT, the following scheme can be useful:

1. State the type of the feature without defining it.

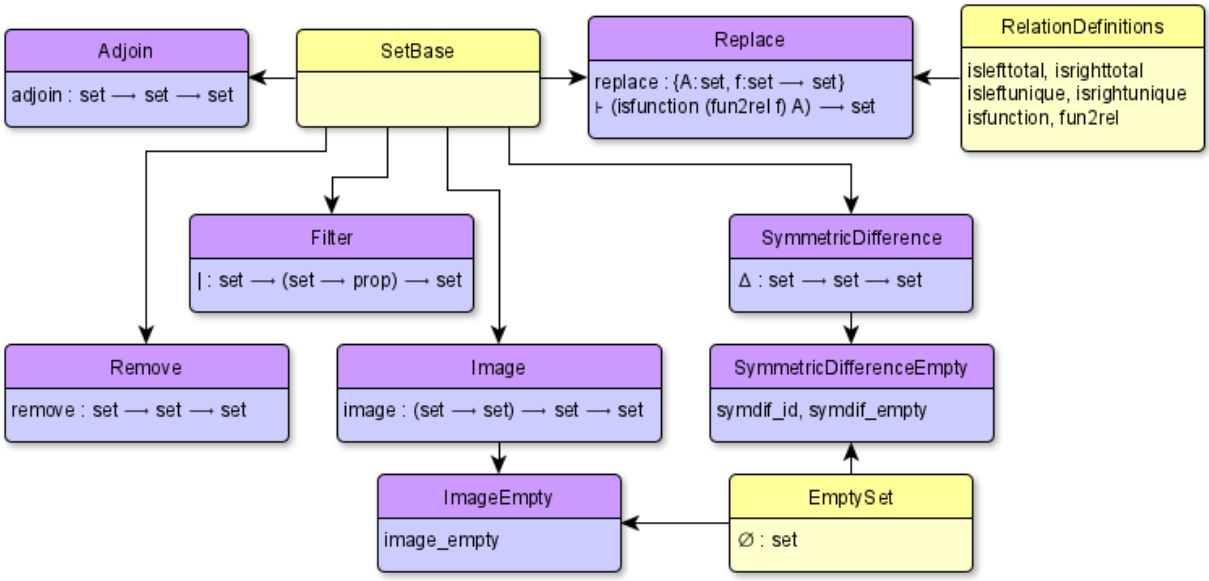

Figure 13: Overview of *operations.mmt*

2. Provide rules for natural deduction without proving them:
   (a) At least one rule to introduce the type.
   (b) At least one rule to eliminate the type.
3. Define properties of this feature that are independent of other features (except the features of `SetBase`).
4. Define properties that rely on this and other features in additional theories.

If the feature shall not be formalized axiomatically, a definition has to be added in 1 and 2. Steps 3 and 4 should be proven in either case. Figure 14 demonstrates how this scheme applies for `Union`. The definition of `UnionEmpty` in figure 11 is equivalent to step 4 [Pin14; Sal19; IR11].

4.3 Features with element construction

Sometimes a "normal" set is not expressive enough. Mostly this is because sets are unordered and do not allow duplicates. Then it is necessary to construct a new element which is based on sets but still imitates the desired property [Pin14; Sal19].

Cartesian Product The Cartesian product of two sets A and B is the set consisting of all pairs a b, where $a \in A$ and $b \in B$. Assuming it does not need a new element, the pairs could be build using unordered pairs. However, when it comes to the elimination rule it would be impossible to determine, whether a was originally an element of set A or set B. Therefore

theory `Union` =
include `SetBase`
// Step 1: stating the type (and a notation)
union : $set \to set \to set \ \# \ 1 \cup 2$
// Step 2(a): introduction rules
unionIl : $\{A, B, x\} \vdash x \in A \to\vdash x \in A \cup B$
unionIr : $\{A, B, x\} \vdash x \in B \to\vdash x \in A \cup B$
// Step 2(b): elimination rule
unionE : $\{A, B, x, C\} \vdash x \in A \cup B \to (\vdash x \in A \to\vdash C) \to (\vdash x \in B \to\vdash C) \to\vdash C$
// Step 3: definition of properties
union_subl : $\{A, B\} \vdash A \subseteq A \cup B = \ldots$ (proof omitted)
union_idem : $\{A\} \vdash A \cup A =^u A = \ldots$ (proof omitted)
...

Figure 14: Application of the scheme for `Union`

an ordered pair has to be used, since it allows to distinguish the elements based on their order [Pin14].

There are different definitions for an ordered pair (a, b). Wiener defined it 1914 as $(a, b) = \{\{\{a\}, \varnothing\}, \{\{b\}\}\}$, whereas later Kuratowski chose $(a, b) = \{\{a\}, \{a, b\}\}$. While Kuratowski's definition is better readable and independent of the empty set, Wiener's definition does not need a case distinction to check whether $a = b$. Overall there are various definitions for ordered pairs, often it is just a slight change of Wiener's or Kuratowski's variant. Since so many varations exist, MMT should not be fixed on a single definition but instead support multiple variants. As hinted in figure 3 MMT has a realization of Kuratowski pairs and Wiener pairs. The corresponding files are located in the subfolder *pair_definitions* [Pin14; SM08].

Since MMT strives for modularity there exists an abstract theory `OrderedPairs` (Figure 15), which just gets realized as a Kuratowski pair or a Wiener pair. The formalization of `OrderedPairs` starts with declarations to construct and deconstruct an ordered pair. The deconstructions $pi1$ and $pi2$ select the first and second element of the ordered pair. However, these declarations cannot be defined without an exact definition of pair. Therefore rules have to be formalized that specify the interactions of the pair construction and deconstruction. The computation rules determine that $pi1$ applied to an ordered pair (a, b) results in a and correspondingly $pi2$ leads to b. However, this rule has to be proven in the realizations, when the definition of pair is known. To assure the next two rules are meaningful, they shall only be used for pairs. That is why a auxiliary function *ispair* gets defined. The representation rule states that a pair p is equal to the ordered pair $(pi1\ p, pi2\ p)$. Finally, the extensionality rule defines that two pairs are equal, if they have the same first and second component [Pin14; SM08; IR11].

theory `OrderedPairs` =
  include `SetBase`
  pair : $set \to set \to set \ \#\ 1\ ,^u\ 2$
  pi1 : $set \to set \ \#\ 1_{1^u}$
  pi2 : $set \to set \ \#\ 1_{2^u}$

  ispair : $set \to prop = [p]\exists^u[a]\exists^u[b]p =^u (a,^u b)$

  compL : $\{a, b\} \vdash (a,^u b)_{1^u} =^u a$
  compR : $\{a, b\} \vdash (a,^u b)_{2^u} =^u b$
  repr : $\{p\} \vdash$ ispair $p \to\vdash p =^u (p_{1^u},^u p_{2^u}) =$ ... (proof omitted)
  ext : $\{p, q\} \vdash$ ispair $p \to\vdash$ ispair $q \to\vdash p_{1^u} =^u q_{1^u} \to\vdash p_{2^u} =^u q_{2^u} \to p =^u q$
    = ... (proof omitted)

  pair_inj : $\{A, B, C, D\} \vdash (A,^u B) =^u (C,^u D) \to\vdash A =^u C \wedge B =^u D$
    = ... (proof omitted)

Figure 15: MMT formalization of ordered pairs

theory `CartesianProduct` =
  include `OrderedPairs`
  prod : $set \to set \to set \ \#\ 1 \times^u 2$
  prodI : $\{A, B, a, b\} \vdash a \in A \to\vdash b \in B \to\vdash (a,^u b) \in A \times^u B$
  prodEl : $\{A, B, p\} \vdash p \in A \times^u B \to\vdash p_{1^u} \in A \ \#\ 4$ prodEl
  prodEr : $\{A, B, p\} \vdash p \in A \times^u B \to\vdash p_{2^u} \in B \ \#\ 4$ prodEr
  prodE : $\{A, B, p\} \vdash p \in A \times^u B \to\vdash$ ispair $p \ \#\ 4$ prodE

Figure 16: MMT formalization of the Cartesian product

With the abstract formalization of `OrderedPairs`, the Cartesian product can be implemented. However, the formalization follows the scheme for features without element construction. The only notable difference is that `OrderedPairs` are included and are used as the element. Figure 16 displays the formalization of `CartesianProduct` [Pin14; SM08; IR11].

Also, using `OrderedPairs` the dependent product (labeled as "Sigma") can be formalized. The dependent product is nearly the same as the Cartesian product, but the second component of the ordered pair is dependent on the first component. Additionally, there exists a modification of sigma which requires a proof that the first component is an element of its first argument. This modification is useful for transformations in section 5.
Another feature that is dependent on `OrderedPairs` is called img. Img is a different variant of image that works on sets R where the elements are ordered pairs. Then img gets an argument a that represents the first component of an ordered pair. It generates a set containing

all components b that satisfy $(a,^u b) \in R$ [Pin14; SM08; IR11].

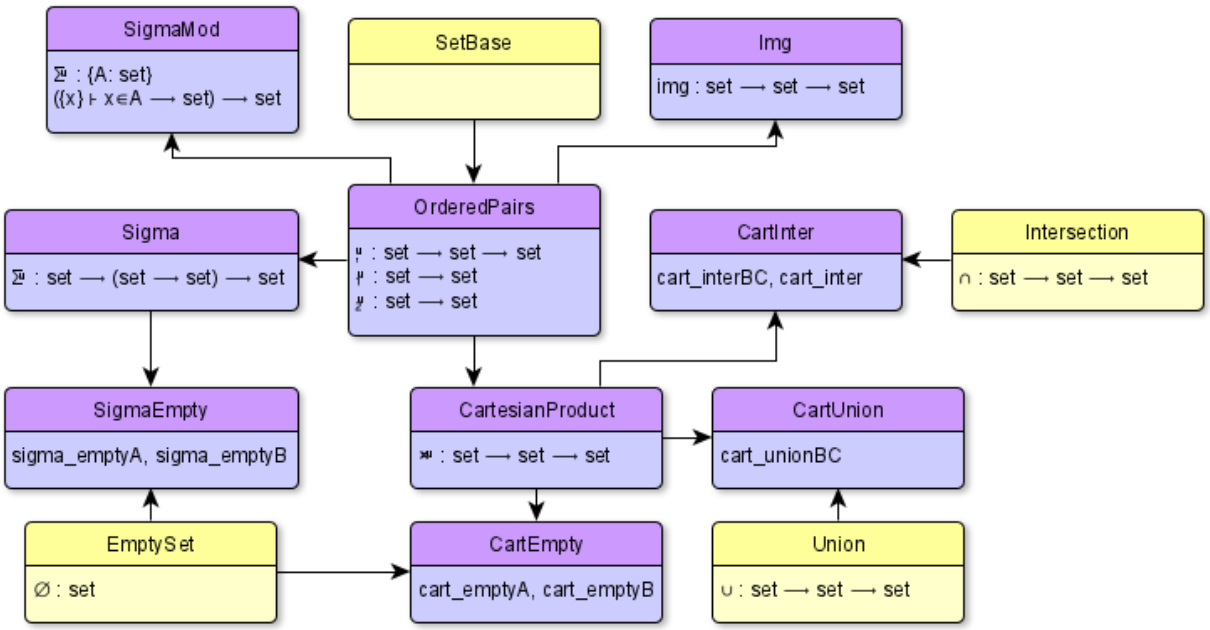

Figure 17: Overview of *cartesian_product.mmt*

Scheme for element construction The generation of a new element in set theory follows always the same scheme. This can be either done in an extra theory, like in `OrderedPairs`, or in the theory requiring this element, like `CartesianProduct`. For the modularity of MMT it is advisable to create a new theory. However, both schemes could be combined to formalize features with element construction in a single theory. In any case the following steps should be considered for the definition of a new element:

1. Provide element constructor(s) and deconstructor(s).
   (a) State the type of the constructor without defining it.
   (b) State the type of the deconstructor without defining it.
2. Define an auxiliary function that checks, whether an element is of this newly defined "type".
3. State the rules defining the behaviour of the constructor and destructor.
   (a) Computation: Apply the destructor to a newly constructed element and compute the result.
   (b) Representation: Reconstruct an element using its components.

(c) Extensionality: Define that two elements are equal, if they have the same components.

4. Define properties of the element that are independent of other features (except the features of `SetBase`).

5. (Define properties that rely on this element and other features in additional theories.)

The proof of step 3(a) is impossible without a definition of constructor and destructor. The remaining rules 3(b) and 3(c) can be proven, if step 2 can be proven in an axiomatic formalization. Otherwise each proof has to be done in a realization of this element [IR11].

An example for this scheme is `OrderedPairs` which formalization is shown in figure 15. However, there does not exist an example of step 5 by now. Still it might be possible that an element might have a special interaction with e.g. the empty set. Considering the definition of Kuratowski pairs there could be a property that if a and b are equal, then $(a, b) = \{\{a\}\}$. This property uses singletons, but Kuratowski's definition uses singletons anyway and therefore no new theory would be needed. Still the last step of the scheme is worth considering. Optionally, steps 1(a) and (b) (and therefore step 3(a)) can be defined, if no axiomatic formalization is wanted. The remaining steps can also be proven in an axiomatic formalization of `OrderedPairs` [Pin14; SM08; IR11].

Relations Another feature that needs a new element is the set of all relations of the sets A and B. Apparently every relation of A and B has to be an element of this set. Therefore the element constructor needs to create a specific relation with a binary predicate P and the deconstructor checks, whether two elements a and b belong to this relation. In contrast to `OrderedPairs`, the deconstructor of `TheRelation` does not return the components of a relation, but rather the membership in that relation [Pin14; IR11].

Then the auxiliary function *isrelAB* is formalized which preferably would be done similar to *ispair* in `OrderedPairs`. However, this would result in an expression with an existential quantification of a predicate. Since set theory is based on first-order logic, such an expression is not allowed. Therefore *isrelAB* has to be defined later, when `TheRelation` gets realized. The rules describing the behaviour of the constructor and destructor are formalized according to the scheme step 3. Because *isrelAB* is not defined, the rules cannot be proven in `TheRelation`.
Apart from that, `TheRelation` specifies the relation definitions that have been formalized in *setbase.mmt*. In *setbase.mmt* the definitions had to be done rather abstract, because it cannot refer to `TheRelation` to avoid cyclic dependencies. Therefore `TheRelation` has declarations that adjust the relation definitions by providing the destructor *inRel* as an argument [Pin14; IR11].

With the element `TheRelation` the feature for relations can be formalized. Relations is the set containing all relations of two sets A and B. It can be formalized according to the

scheme for features without element construction. The only difference is, that it includes `TheRelation` as its elements. Additionally, the two features for the sets of all partial functions and all functions on the sets A and B can be formalized exactly the same. Then the set of all partial functions on A and B contains all right-unique relations. The set of all functions on A and B consists of all relations that are right-unique and left-total. The three features for relations, partial functions and functions on A and B are all formalized axiomatically [Pin14; IR11].

Since basically every function is just a relation, `TheRelation` should be enough. However, it can be quite complex to express functions with `TheRelation`, especially when a function shall be applied to an element. Therefore another element called lambda function is introduced that can be used similarly to a function in practice.
The function that is build with its constructor creates a set that equals a relation generated by `TheRelation`. In contrast to `TheRelation`, the deconstructor returns the element that is the result of the function application. Since lambda function shall only be used to represent functions, the rules 3(b) and 3(c) require functions to be provable. Also, the function properties could be helpful to proof a resemblance of the deconstructors *apply* and *inRel* [IR11].

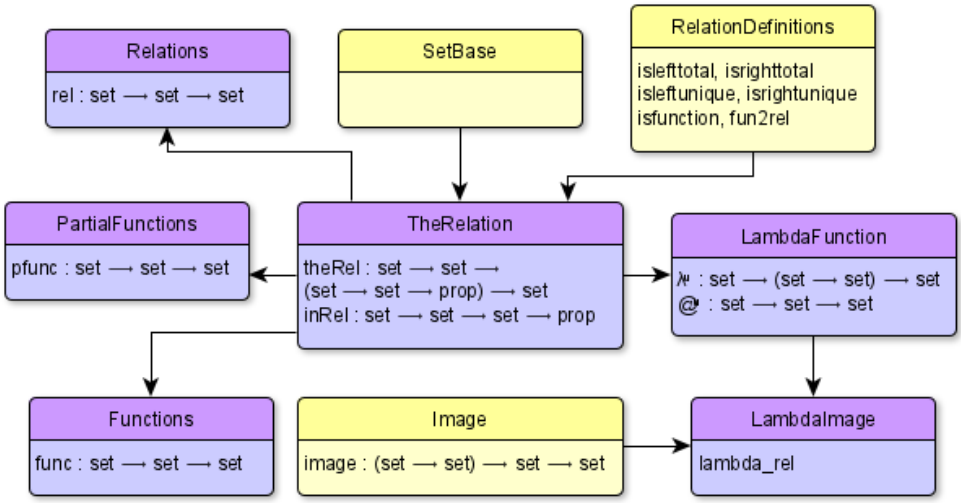

Figure 18: Overview of *relations.mmt*

4.4 Building ZFC

The axioms of ZFC have already been stated in section 4.1. However, it also has been mentioned that the axioms and the formalization of ZFC are located in different MMT files. Therefore the structure and issues of *zfc.mmt* shall be explained in more detail [IR11].

First a theory `ZFBase` is created that solely includes `SetBase` and the axioms that ZF is based on. The only exception is the infinity axiom that relies on empty set and cannot be included before empty set has been realized. Otherwise there would be a morphism error in MMT. Then the theory `ZF` is build that contains the features that are strongly connected to the axioms in `ZFBase`. Empty set gets realized in `ZFBase` such that the infinity axiom can be included. An example for another feature in `ZFBase` is `UnorderedPairs`, which is essentially the feature defined by the axiom *Unordered Pairing*. Based on `ZF` the theory `ZFFeatures` is generated which consists of all features that can be realized using `ZF`. The same idea applies for ZFC.
The separation of the theories is not necessary, but it might be helpful to keep track of the dependencies. While features of `ZFFeatures` can be removed relatively loosely, a removed feature in `ZF` indicates that probably an axiom is not needed. Also, there could be multiple theories similar to `ZFFeatures` that contain different features. Then the modularity of `ZF` could be useful, since it could be reused as the foundation of these theories.

Until now an important aspect has been disregarded. Because features are formalized axiomatically, they cannot be included in `ZF` and `ZFFeatures` without a realization. Otherwise just new axioms would be added to ZF, which is pointless.
For the purpose of modularity views are created to realize features independently of ZF(C). There has to be at least one view for each feature in order to include it in ZF(C). Sometimes it is possible that a feature could be realized by multiple different views and one of these views has to be chosen [IR11].

Figure 19 is an overview of the views between set theoretic features. The axioms in `SetBase`, inclusions of elements like `OrderedPairs` and extra theories for multiple codomains are ignored in the graph. Each arrow represents a view, where the arrow points towards the domain. If the view has multiple codomains, the arrow has multiple starting points that are combined in one line. For example, `Singletons` and `Adjoin` are the codomains for a view with the domain `UnorderedPairs`.
A huge problem in this graph is that especially the theory `Filter` can be used in numerous views as (part of) a codomain to realize another feature. Therefore there are many lines that have a connection to `Filter` which makes the graph confusing. Another problem is that some features rely on multiple codomains which is hard to illustrate and leads to many crossing lines. However, the main issue is that the flexible formalization of views leads to a high connectivity between different theories. Therefore, the more theories and accordingly views are formalized, the more complicated the graph gets.

Fortunately, a complicated graph does neither affect the formalization nor the usability. Therefore the formalization of ZF(C) can be continued using one view for each feature. For example, for the realization of `Singletons` either the view `SelfPair` with codomain `UnorderedPairs` or a view with the codomains `Adjoin` and `EmptySet` can be chosen. However, these might not be the only possibilities to realize `Singletons`.

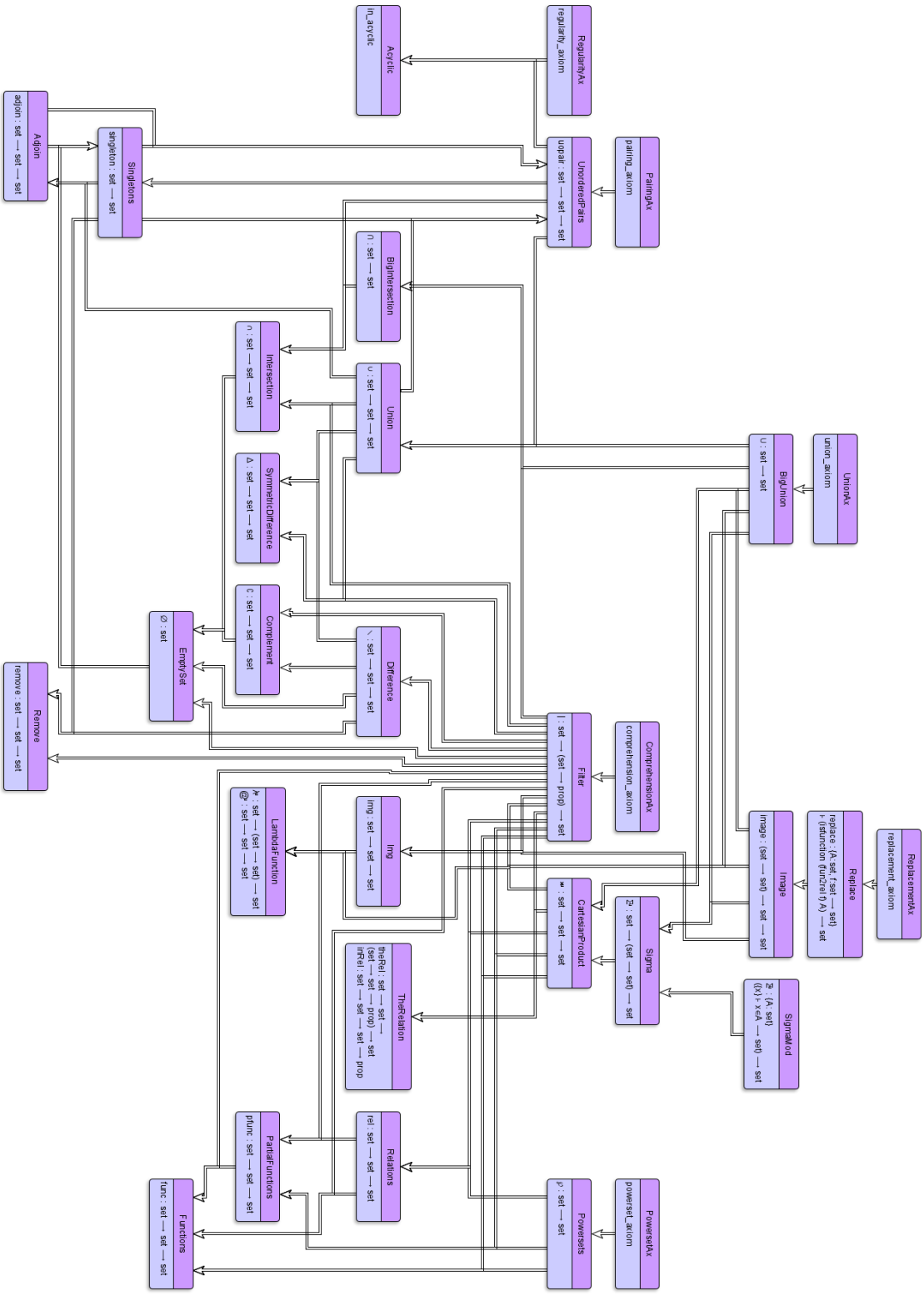

Figure 19: Overview of views in MMT

The formalization of the view `SelfPair` has already been sketched in figure 3. When `Singletons` are included in `ZFFeatures`, the view `SelfPair` has to be stated as the realization of `Singletons`. Of course, this creates dependencies within `ZFFeatures` and therefore the order of its features is partially fixed. However, this also assures that features are not included before they can be proven in ZF(C) [IR11].

```
theory ZFBase =
  include SetBase
  include PairingAx
  include UnionAx

theory ZF =
  include ZFBase
  include UnorderedPairs = UopairFromAx
  include BigUnion = BigUnionFromAx

theory ZFFeatures =
  include ZF
  include Singletons = SelfPair
  include Union = UnionAsBigUnion
  include Adjoin = AdjoinUnion
```

Figure 20: Excerpt of the formalization of *zfc.mmt*

Figure 20 displays an example of the idea of *zfc.mmt* which is illustrated in the graph in figure 21. For this example `ZFBase` consists of `SetBase` and the axioms *Unordered Pairing* (`PairingAx`) and *(General) Union* (`UnionAx`). In `ZF` the features `UnorderedPairs` and the general union `BigUnion` are included with the realization from their corresponding views. `UopairFromAx` is the view that defines `UnorderedPairs` using `PairingAx` and accordingly `BigUnionFromAx` realizes `BigUnion` with `UnionAx`. To keep this example simple, the formalizations of features and their views have been omitted.
In `ZFFeatures` further features are defined on the basis of `ZF`. The view `SelfPair` defines `Singletons` as `UnorderedPairs` and `UnionAsBigUnion` realizes `Union` as the `BigUnion` of `UnorderedPairs`. Within the view `AdjoinUnion Adjoin` is defined using `Union` and `Singletons`. Since these two features are not part of `ZF`, it is necessary to include them before `Adjoin` can be included [IR11].

Though this concept is great for modularity, there are some downsides. First of all, since views cannot have multiple codomains by now, there needs to be an extra theory C for the codomains of a view. This additional theory has to be included in `ZFFeatures` before the view can be used. This is a manageable but nonetheless annoying problem. In the example 20 there would be additional theories `BigUnionUnorderedPairs` and `UnionSingleton` that

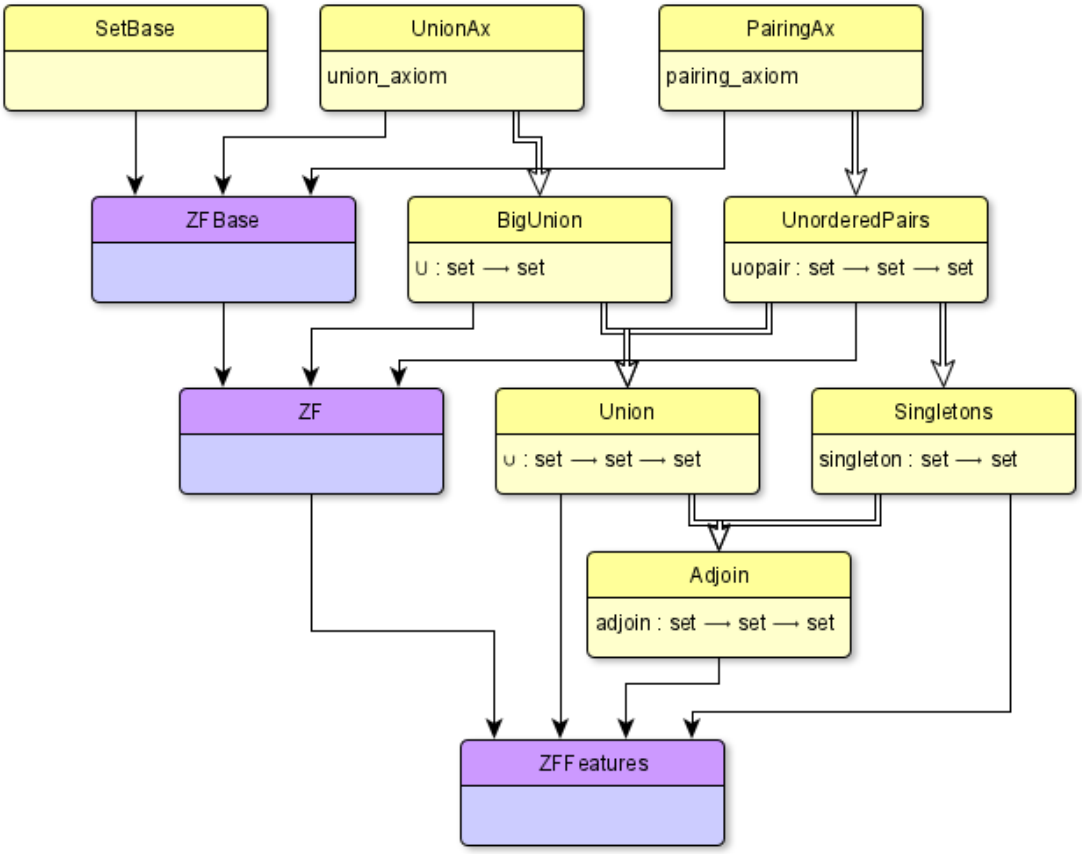

Figure 21: Graphical visualization of the formalization in figure 20

include all codomains of the corresponding view.

Another issue occurs because of the axiomatic formalization of elements. For example, when Kuratowski's definition should be used for ordered pairs, this leads to morphism errors in `ZFFeatures` as soon as the Cartesian product or dependent product is realized. Just the inclusion does not lead to an error, therefore the problem is related to the view to realize the features. A view needs to realize all undefined declarations of its domain. Consequently, if the domain is the Cartesian product, `OrderedPairs` have to be realized as well. Since no specific definition of `OrderedPairs` shall be given, the identity morphism is used. However, as soon as `OrderedPairs` are realized as e.g. Kuratowski pairs, MMT has two morphisms for `OrderedPairs` that are not equal. This creates the morphism error. The temporarily solution for this problem is to use the abstract `OrderedPairs` instead of a realization [Pin14; SM08; IR11].

The last is a performance issue. If every include has a realization, MMT checks all of them, whenever `ZFFeatures` is referred. Even if no changes have been made to `ZFFeatures` and

it is just included in another theory, MMT gets really slow. Therefore all realizations of the includes are commented out, until a better solution for this problem is found. Mostly, the realizations are not necessary except to show that ZF(C) is built correctly. Also, if a realization is needed in a `ZFFeatures` based theory, it can be explicitly added in the corresponding theory.
For example, the theory `NatNums` which formalizes the natural numbers in set theory includes `ZFFeatures`. For the definition of natural numbers, the successor function $s = [n]n \cup \{n\}$ is used that defines the successor of the natural number n as the union of n and the singleton of n. This successor function would be equivalent to the realization of the feature adjoin in `ZFFeatures`. Since realizations in `ZFFeatures` are not possible because of the performance issues, this realization of adjoin is added directly in `NatNums` [IR11; Pin14].

5 Transformations into set theory

Set theory is a very powerful concept, if it is slightly modified. Twelf already has some transformations, where types are realized as sets and even data types like *Booleans* are expressed with set theory. These formalizations are used as a model for the MMT transformations into set theory [Twe; IR11].

5.1 Preliminaries

Since set theory is untyped, a few declarations are necessary in order to realize types. Therefore the notions of *Class* and *Elem* are formalized. A Class P is the class of all sets that satisfy an unary predicate P. An element of a Class P (*celem*) is like a pair (x, p), where x is a set and p is a proof that x satisfies P. The operation *cwhich* applied to this element returns the set x and the application of *cwhy* leads to the proof p. Therefore these operations are similar to $pi1$ and $pi2$ of ordered pairs.
Elem of the set A is basically the Class of all sets that are elements of A. Thus, Elem A represents all elements of A. For Elem the operations *elem*, *which* and *why* are defined with the corresponding Class operations. The MMT formalization of this is shown in figure 22 [IR11].

Additionally, the property *ceq_ which* is formalized for a Class. This property states that a term with a proof is equal to the original term. Correspondingly, *eq_ which* is defined for Elem. However, *eq_ which* has the MMT feature "role Simplify". According to the name, role Simplify uses *eq_ which* as a simplification rule. Therefore, every time a term consists of "which (elem x p)", MMT simplifies this term automatically to x. Often this makes proofs a lot easier, but sometimes this can be a downside as well. Especially, if something of the form "which (elem x p)" shall be the result of a proof. The advantages and disadvantages of this decision will be continued in more detail in section 5.3 [IR11].

Another feature for Elem is equality, which states that two elements of Elem A are equal, if

theory TypeBase =
include SetBase

Class : $(set \to prop) \to type$
celem : $\{P, x\} \vdash P\ x \to \text{Class}\ P$
cwhich : $\{P\}\ \text{Class}\ P \to set$
cwhy : $\{P, c : \text{Class}\ P\} \vdash (P(\text{cwhich}\ P\ c))\ \#\ \text{cwhy}\ 1\ 2$

ceq_which : $\{P, x, p\} \vdash (\text{cwhich}\ P(\text{celem}\ P\ x\ p)) =^u x$

Elem : $set \to type = [A]\ \text{Class}\ [x]x \in A$
elem : $\{A, x\} \vdash x \in A \to \text{Elem}\ A\ \#\ \text{elem}\ 2\ 3$
$= [A, x, p]\ \text{celem}\ ([u]u \in A)\ x\ p$
which : $\{A\}\ \text{Elem}\ A \to set\ \#\ \text{which}\ 2$
$= [A, e]\ \text{cwhich}\ ([x]x \in A)\ e$
why : $\{A, e : \text{Elem}\ A\} \vdash \text{which}\ e \in A\ \#\ \text{why}\ 2$
$= [A, e]\ \text{cwhy}\ \ ([x]x \in A)\ e$

eq_which : $\{A, x, p :\vdash x \in A\} \vdash \text{which}\ (\text{elem}\ x\ p) =^u x$
$= [A, x, p]\ \text{ceq\_which}$

Eq : $\{A\}\ \text{Elem}\ A \to \text{Elem}\ A \to prop\ \#\ 2\ \text{Eq}\ 3$
$= [A, e, d]\ \text{which}\ e =^u \text{which}\ d$

Figure 22: Formalization of Class and Elem in MMT

their set is equal. The proofs of the elements can be left out, because there can be multiple proofs for a set being an element of A. Therefore untyped equality and the *which* operation are enough to define the equality. However, for the transformations some additional declarations to use this equality have to be formalized. Mostly they state that if two elements of Elem A are equal and one of them satisfies a predicate, the other element fulfills that predicate as well [IR11].

5.2 Transformations

The transformations of type theory (and logic) into set theory shall have a similar structure as type theory. On the one hand this helps to keep an overview of the files and on the other hand the modularity of type theory is reused.

Fundamentals First of all type related theories that are located in fundamentals have to be transformed into set theory. Figure 23 displays a short version of this. In MMT this is done in multiple theories that build on each other, but for simplification the example is

reduced to two theories. `TypedFund` represents the theories for "normal" type theory, while `SoftTypedFund` combines the theories for soft-typed theory. It shall be noted that the example skips some includes as well, especially if the corresponding theories are part of the first-order logic on which set theory is based.
`TypedFund` and `SoftTypedFund` include `TypeBase` and in both theories *tp*, which represents the type, is defined as a set. In `TypedFund` a term (*tm*) of type A is equal to Elem A and the typed equality *tequal* corresponds to *Eq*. Furthermore, *trefl* and *tcongB* are represented by the matching operations in `TypeBase`. `SoftTypedFund`'s only further addition is to define *of* as the membership operation *in*. In soft-typed theory "x of A" states that a term x is of the specific type A.

theory `TypedFund` =
  include `TypeBase`

  realize `TypedEqualityND`
  tp = set
  tm = $[A]$ Elem A
  tequal = Eq
  trefl = $[A, e : \text{Elem } A]$ Refl
  tcongB = EqcongB

theory `SoftTypedFund` =
  include `TypeBase`

  realize `SoftTypedLogic`
  tp = set
  of = in

Figure 23: Short version of *fundamentals_ transformed.mmt*

Product Types In type theory there are multiple variants of product types which are similar to the Cartesian and dependent product. The transformations of `SimpleProducts` into the `CartesianProduct` will be explained in more detail.
At first the type *simpprod* is defined as the `CartesianProduct` of two sets A and B. Since *tp* has been defined as set, this step is trivial [IR11].

Now `OrderedPairs` have to be defined with the `CartesianProduct` as their type. According to `TypedFund` a term of type A is equivalent to Elem A. Therefore the transformation of *simppair* has the type $\{A, B\}$ Elem $A \to$ Elem $B \to$ Elem $A \times^u B$ which MMT can infer. The arguments a : Elem A and b : Elem B are used to generate a new Elem $A \times^u B$. The pair (which a,^{u} which b) matches the set of the Elem and the proof is prodI (why a)(why b)).
The transformation of the destructors *simppi1* and *simppi2* follows a similar procedure. They get an Elem $A \times^u B$ as an argument and generate an Elem A (respectively Elem B). The associated theory `SimpleProductsTransformation` in figure 24 displays the MMT code. However, in the actual MMT file these definitions are distributed in multiple theories [IR11].

Of course, `SimpleProducts` has also formalizations for the computation, representation and extensionality rules of `OrderedPairs` which are called *compute1*, *compute2*, *expand* and *exten* correspondingly. The transformations of these are straightforward, because they can be directly transformed into their set theoretic equivalents [IR11].

theory `SimpleProductsTransformation` =
  include `TypedFund`
  include `CartesianProduct`

  simpprod = $[A, B]\ A \times^u B$
  simppair = $[A, B, a : \text{Elem } A, b : \text{Elem } B]$
    elem (which a,^{u} which b) (prodI (why a) (why b))
  simppi1 = $[A, B, e : \text{Elem } (A \times^u B)]$ elem ((which e)$_{1^u}$) ((why e) prodEl)
  simppi2 = $[A, B, e : \text{Elem } (A \times^u B)]$ elem ((which e)$_{2^u}$) ((why e) prodEr)

  compute1 = $[A, B, a : \text{Elem } A, b : \text{Elem } B]$ compL
  compute2 = $[A, B, a : \text{Elem } A, b : \text{Elem } B]$ compR
  expand = $[A, B, e : \text{Elem } (A \times^u B)]$ eq/sym (repr ((why e) prodE))
  exten = $[A, B, e : \text{Elem } (A \times^u B), d : \text{Elem } (A \times^u B), p, q]$
    ext ((why e) prodE) ((why d) prodE) p q

Figure 24: Transformation of `SimpleProducts` into set theory

Dependent Product Types Another type is for dependent products with `Sigma` as the corresponding feature in set theory. However, `Sigma` had to be changed in order to transform `DependentProducts` into set theory. Since in type theory $B(a)$ is only defined for an a of type A, but in set theory $B(a)$ for $\neg a \in A$ is possible as well. Although the result would still not be part of `Sigma`.

To get a valid transformation of `DependentProducts`, either a case distinction for $a \in A$ and $\neg a \in A$ has to be performed or `Sigma` has to be rewritten such that B requires that a is an element of A. The former option was discarded, because it would make the transformations more complex and the case $\neg a \in A$ would still be hard to define. Instead `SigmaMod` got introduced which is a modification of `Sigma`. `SigmaMod` is formalized such that B gets an additional argument $\vdash a \in A$ to solve this issue. It should be noted, that `Sigma` can be realized using `SigmaMod`. However, it is impossible to realize `SigmaMod` with `Sigma`.
Figure 25 displays short versions of the MMT formalizations for `SigmaMod`, `DependentProducts` and the transformation `DependentProductsTransformation`. The transformations of the natural deduction rules are similar to the ones for `SimpleProducts`.

Typed Existential Quantification Types do not only occur in type theory, but also in logic. An interesting examples of typed first-order logic is existential quantification. The transformation depends highly on the first-order logic on which set theory is based, because it contains dependent propositional logic.

In general, `TypedExistentialQuantification` shall be represented as the untyped exis-

theory SigmaMod =
  sigma : $\{A : set\}$
    $(\{x\} \vdash x \in A \to set) \to set$

theory DependentProducts =
  depprod : $\{A\}(\text{tm } A \to tp) \to tp$

theory DependentProductsTransformation =
  include TypedFund
  include SigmaMod

  realize DependentProducts
  depprod = $[A, B : (\text{Elem } A \to set)]$
    sigma $A([x, p]\ B(\text{elem } x\ p))$

Figure 25: MMT formalizations to transform DependentProducts into SigmaMod

tential quantifier. However, tm A has to be transformed into Elem A and therefore set theory is necessary. Since *texists* does neither get an Elem A nor a proof that some x is an element of A, there is no possibility to generate an Elem A without using the dependent conjunction. Figure 26 shows how the formalization looks like in MMT. The dependent conjunction is written as "$\wedge^d$" in infix notation [IR11].

theory TypedExistentialQuantification =
  texists : $\{A\}(tm\ A \to prop) \to prop\ \#\ \exists^t\ 2$
  texistsI : $\{A, P\}\{x : tmA\} \vdash P\ x \to\vdash \exists^t P$
  texistsE : $\{A, P, C\} \vdash \exists^t P \to (\{x : tm\ A\} \vdash P\ x \to\vdash C) \to\vdash C$

theory TypedExistentialTransformation =
  include TypedFund

  realize TypedExistentialQuantification
  texists = $[A, P : (\text{Elem } A \to prop)]$ uexists $[x](x \in A) \wedge^d ([p]P\ (\text{elem } x\ p))$
  texistsI = $[A, P : (\text{Elem } A \to prop), e : \text{Elem } A, p]$
    uexistsI (which e) (dandI (why e) (simpP P equivEl p))
  texistsE = $[A, P, C, p, Q]\ p$ uexistsE $([x, px]\ Q\ (\text{elem } x\ (px\ \text{dandEl}))\ (px\ \text{dandEr}))$

Figure 26: Transformation of the TypedExistentialQuantification into set theory

Scheme for transformations into set theory Like the formalizations of features and elements in section 4, transformations into set theory are also formalized according to a scheme. Although this scheme might vary depending on the feature that shall be formalized, it can still be used as an orientation. Especially, if these transformations shall be automated in the future. In this scheme, F stands for the feature that shall be transformed.

1. Check whether a feature S exists that resembles the type of F. If there exists no suitable feature, S has to be formalized first (see schemes in section 4).

2. Create a new theory T for the transformation. T includes S and realizes F. Alternatively, the transformation could be formalized as a view.

Step	`SimpleProducts`	`DependentProducts`	`TypedExistentialQuantification`
1	`CartesianProduct`	`SigmaMod`	`ExistentialQuantification`
2	...-Transformation	...-Transformation	...-Transformation
3	simpprod ≋ prod	depprod ≋ sigma (modified version)	texists ≋ uexists
4(a)	ordered pairs	ordered pairs	*results in prop and therefore creates no elem*
4(b)	simppair ≋ prodI simppi1 ≋ prodEl simppi2 ≋ prodEr	deppair ≋ sigmaI deppi1 ≋ sigmaEl deppi2 ≋ sigmaEr	texistsI ≋ uexistsI texistsE ≋ uexistsE
5	compute1 ≋ compL ...	compute1 ≋ compL ...	*no properties*

Table 1: Transformations of `SimpleProducts`, `DependentProducts` and `TypedExistentialQuantification` with the corresponding steps of the scheme

3. Define the type of F as the feature S. If F is a dependent type, perhaps S has to be rewritten to match it. Alternatively, a new feature $FMod$ for the modified version of F can be introduced. If F has a predicate operating on a term of type A, it might be necessary to use dependent propositional logic.

4. Define the natural deduction rules of F:

   (a) The set of *elem* corresponds to the element of S. Use *which* whenever the set of an argument of type *Elem* is needed. This step does not apply, if no *elem* is created.

   (b) The proof of *elem* makes use of the corresponding natural deduction rule of S. In any case the proof has to introduce/eliminate an element of the transformed type. Use *why* whenever the proof of an argument of type *Elem* is needed. If no *elem* is created, only this proof is used as definition.

5. Transform undefined properties of F considering the previous definitions.

As an addition to step 4, it should be noted that there is no case distinction whether or not S is feature with constructed elements. In either case the natural deduction rules of F resemble the natural deduction rules of S.
Because this scheme is really dependent on F, table 1 is used to point out the steps for the transformations of `SimpleProducts`, `DependentProducts` and `TypedExistentialQuantification`.

5.3 Advantages and disadvantages of using "role Simplify"

As already mentioned before, the decision to give *eq_which* the role Simplify needs to be looked at in more detail. The transformations before are helpful as they can serve as exam-

ples to get a better understanding for the benefits and issues.

Gain of role Simplify Transforming *compute1* of Product Types without simplification is more complicated. *compute1* states for a : tm A and b : tm B:

$$\vdash \text{simppi1 (simppair } a\ b) =^t a$$

To get a better understanding, already defined parts of this expression get transformed into set theory. This result that has to be proven for a : Elem A and b : Elem B is:

$$\vdash \text{elem } ((\text{which (elem (which } a,^u \text{ which } b)(\text{prodI (why } a)(\text{why } b))))_{1^u})$$

$$((\text{why (elem (which } a,^u \text{ which } b)(\text{prodI (why } a)(\text{why } b)))) \text{ prodEl}) \text{ Eq } a$$

Since "e Eq d" is equivalent to "which e $=^u$ which d", *eq_ which* can be used to reduce the expression every time "which (elem x p)" appears. However, this has to be done manually and is not performed by MMT automatically. Therefore *eq_ which* needs the proof of the Elem as an argument. Then *eq_ which* has to be combined with e.g. *ucongP* and the exact predicate P for reduction. This has to be done twice until it is possible to apply *compL* which is accessed using transitivity. An accepted transformation for *compute1* without Simplify is shown in figure 27.

theory `SimpleProductsTransformationWithoutSimplify` =
compute1 = $[A, B, a : \text{Elem } A, b : \text{Elem } B]$ ((eq_which (prodI (why a) (why b)))
ucongP ([x] which (elem ((which (elem (which a,^{u} which b)
(prodI (why a) (why b))))$_{1^u}$)
((why elem (which a,^{u} which b) (prodI (why a) (why b))) prodEl)) $=^u$ (x_{1^u}))
(eq_which (why (elem (which a,^{u} which b) (prodI (why a) (why b))) prodEl)))
eq/trans compL

Figure 27: Transformation of *compute1* without automatical simplification

Admittedly, this proof would not look that complicated, if *eq_ which* would not need an argument and the predicate for *ucongP* would not have to be specified. However, the idea of the proof is basically the same as what the automatical simplification does. As a comparison, the steps of the simplification shall be pointed out, starting with the equation that has to be proven:

$$\vdash \text{which (elem ((which (elem (which } a,^u \text{ which } b)(\text{prodI (why } a)(\text{why } b))))_{1^u})$$

$$((\text{why (elem (which } a,^u \text{ which } b)(\text{prodI (why } a)(\text{why } b)))) \text{ prodEl}) =^u \text{which } a$$

The first simplification reduces the expression to:

$$\vdash (\text{which (elem (which } a,^u \text{ which } b)(\text{prodI (why } a)(\text{why } b))))_{1^u} =^u \text{which } a$$

The result of the next simplification is of the same form as *compL* which completes the proof:

$$\vdash (\text{which } a,^u \text{ which } b)_{1^u} =^u \text{which } a$$

Disadvantages of role Simplify The transformation of the natural deduction rules of `TypedExistentialQuantification` (figure 26) have not been discussed. While *texistsE* is basically the transformation that could be expected, *texistsI* uses *simpP* which has not been defined by now. Essentially *simpP* states that a predicate P applied to some e : Elem A is equivalent to P applied to "elem (which e) (why e)".
This statement cannot be defined as long as *eq_ which* is used as a simplification rule. For the proof it would be necessary to use the equality defined by *eq_ which*. But whenever *eq_ which* is written, it gets immediately simplified to "which $e =^u$ which e". Since it is not possible to turn simplify off, *eq_ which* is basically just another name for the reflexivity of untyped equality.

As a consequence, declarations like *simpP* and *simpF* (similar to *simpP*, but for functions and not predicates) have been added to `TypeBase`. These declarations can be used to define further transformations that could not be defined otherwise.
Since the declarations cannot be defined, they are basically axioms. In actual fact the addition of new axioms to a set theory is risky, because it could lead to paradoxes. In this specific case it should not lead to a problem, because the statements would be proveable, if *eq_ which* had not been defined as role Simplify.

Another issue with role Simplify is related to transformations as well. Whenever a feature has a declaration with role Simplify, the declaration and the simplification rule need to be defined. While the declaration itself is relatively easy to transform, it is unkown how to realize its simplification rule. Therefore some transformations are not complete which might lead to errors in later formalizations.

Opinion on the use of role Simplify In general, it can be concluded that defining *eq_ which* with role Simplify is useful. It saves a lot of time formalizing the transformations and the results are much better readable. Since additional declarations like *simpP* and *simpF* solve the greatest issue. The downside of defining them axiomatically is negligible, because the statements would be provable with *eq_ which*.
The issue that the simplification rule has to be realized is not a concern for *eq_ which*. On the one hand `TypeBase` and therefore *eq_ which* should not have to be realized, especially because *eq_ which* is already defined as *ceq_ which*. On the other hand *eq_ which* has no further influence on other features that have been defined as role Simplify. Therefore this problem is independent of the formalization of *eq_ which*.

6 Library Management

While adding MMT files to LATIN2 in sections 4 and 5, some problems occurred that were related to the high scalability of MMT. The first issue arised because of the amount of files and their dependencies. Therefore it was necessary to (re-)organize the files with regard to their dependencies and their scientific area. The next problem was also related to the number of files. Since this number is growing, it gets harder to build all files manually and a build script was required. The last issue emerged because different concepts have formalizations of similar features that share their notation or name. This led to disambiguity problems, when views between two concepts were created.

6.1 Organization of MMT files

Due to the many files and theories in MMT, the dependencies are sometimes unclear. Therefore the build order of the MMT library *LATIN2* was hard to figure out. To reduce the problem, a bit of reorganization was necessary. However, this section might leave out less interesting files.
A simplified version of the resulting dependency graph is shown in figure 28. In this graph, blue rectangles visualize collections of theories. The arrows point towards the collection that is dependent on the collection with the starting point. For simplification, collections that are equivalent to subfolders are named similar to <folder name/subfolder name>. Additionally, implicit dependencies have not been illustrated, like e.g. *Logic/Fol Like* being dependent on *Fundamentals*.

fundamentals As the folder name suggests, fundamentals is the root of *LATIN2*. In the file *concepts.mmt* concepts like *Propositions*, *Terms* and *Types* are formalized that are used by nearly every other theory. However, *relations.mmt*, which defines the type and properties of relations, is independent of *concepts.mmt*. These two files are necessary to formalize the remaining files of fundamentals.
One of the more important files is *base_languages.mmt* which formalizes differently typed logics. It is required to define *equality.mmt* that is necessary to formalize equality for types in *type_equality.mmt* and *subtyping.mmt*.

logic/propositional Another very significant folder is logic. It is separated into subfolders to get a better overview of the different types of logic and dependencies within the folder. Propositional contains every file belonging to propositional logic which is often the basis for further logics. Its most important file is *pl.mmt* that defines connectives in propositional logic. Every other file in this subfolder is dependent on *pl.mmt* and formalizes variations and additions to propositional logic.

logic/fol_like First-order logic depends on propositional logic. In fol_like there are three main versions to formalize differently typed first-order logic. In *fol.mmt* untyped quantifiers and description operators are formalized. The typed variants of these features are formalized

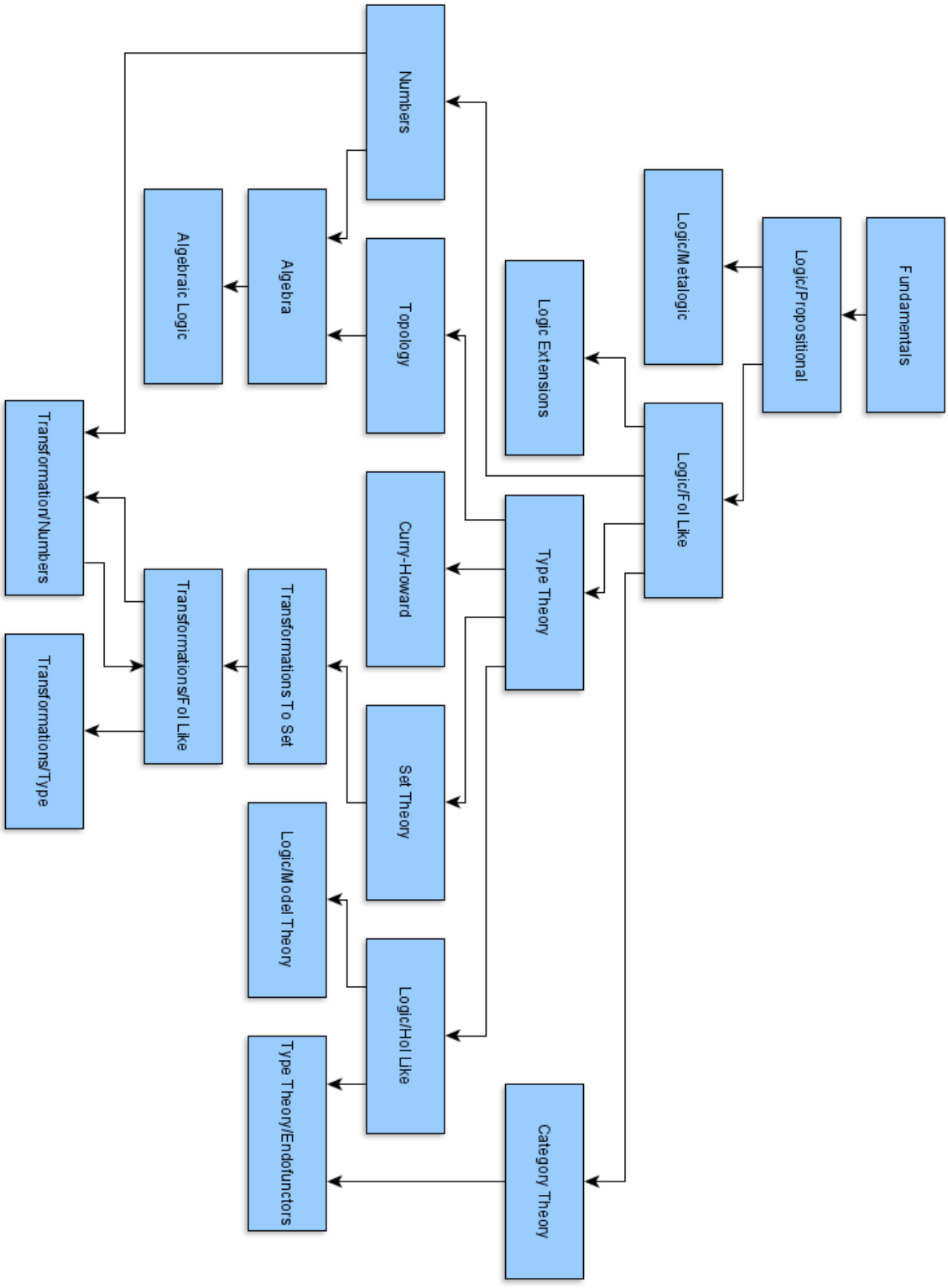

Figure 28: Simplified overview of dependencies in *LATIN2*

in *sfol.mmt*. While *fol.mmt* and *sfol.mmt* are independent of each other, *stfol.mmt* relies on *fol.mmt*. The formalizations of *stfol.mmt* define soft-typed first-order logic.
A part of the remaining files in fol_like formalizes variations and additions to *fol.mmt* which are similar to the ones in propositional logic. The other remaining files are based on *sfol.mmt*. For example, *booleans.mmt* is defined using *sfol.mmt*. Then *pl-sfol.mmt* sketches how propositional logic could be transformed into typed first-order logic.

type_theory Some subfolders in logic have not been considered, because they rely on (parts of) type theory. Since *power_types.mmt* relies on parts of logic, logic and type theory have a cyclic dependency that could be solved by splitting the different logics.
Most files in type theory are independent of each other and their names are self explanatory. For example, *nonempty.mmt* formalizes that neither the universe nor types are empty and therefore fresh elements can be picked. Another example is *function_types.mmt* which formalizes typed and soft-typed types for "simple" and dependent functions. This file is necessary for *type_erasure.mmt* that contains views to transform typed features into soft-typed features.

logic/hol_like Using type theory, higher-order logic is formalized in *hol.mmt*. The variant *hol_andrews.mmt* realizes a theory of *hol.mmt* and is therefore dependent on it.
logic/model_theory Based on higher-order logic is model_theory. For now only Kripke models are formalized in *kripke.mmt*. The variants *kripke_dynamic.mmt* and *kripke_multimodal.mmt* rely on *kripke.mmt* as well as fol_like.

category_theory Category_theory contains exactly one file *category.mmt* which formalizes a category. However, this file is necessary for the remaining files of type_theory.
type_theory/endofunctors Before endofunctors have been ignored, but now all of their requirements are formalized. *endofunctors.mmt* is the root of this subfolder and based on it is *endomagmas.mmt*. The file *collection_types.mmt* that defines different collections as well as lists and multisets is dependent on *endomagmas.mmt*. The last file *monads.mmt* relies on *endofunctors.mmt* and higher-order logic.

set_theory Set_theory is based on fol_like. Within set_theory, *setbase.mmt* is the root that formalizes basic definitions like *Sets*. The formalizations of the features (see sections 4.2 and 4.3) solely rely on *setbase.mmt*. The subfolder pair_definitions contains the realizations of ordered pairs and is therefore dependent on multiple features. *axioms.mmt* depends on *finite_sets.mmt*, because the formalization of the infinity axiom relies on the empty set. The subfolder views_features is dependent on features and axioms, since it contains multiple views to realize features with axioms and other features.
The file *typebase.mmt* contains the declarations of *Class* and *Elem* which are necessary to interpret type theory in set theory. With *typebase.mmt* datatypes like *nat.mmt* and *bool.mmt* can be formalized. Finally, *zfc.mmt* defines ZFC with axioms and all features are realized with views. Additionally, a typed version of ZFC is formalized.

logic_extensions As the name suggests, this folder is dependent on logic. Within logic_extensions *Contexts* and other logical extensions are formalized with the goal of creating *prolog.mmt*. Overall the dependencies in smaller folders are relatively straightforward, which is why they are omitted here.
algebra Algebra is dependent on typed first-order logic and has its own declarations of *Sets*, *Relations* and *Magmas*, although most of these are already formalized similarly in another folder. Based on *magmas.mmt* different algebraic concepts like *groups.mmt* and *ringoids.mmt* are defined.

transformations_to_set Finally the folder transformations_to_set is considered. Since many concepts can be transformed into set theory, it is helpful to state this folder at the end. The names of the subfolders are self explanatory, e.g. type_theory contains the transformations of type theory into set theory. The structure of the files in a subfolder is the same as in the folder that gets transformed.
Obviously, this folder depends on *set_theory* and every other feature that shall be transformed. Some fundamental transformations are done in *fundamentals_transformed.mmt*. The subfolders are sometimes dependent on each other, when e.g. typed universal quantification is required in the transformation of natural numbers. For now there exists only a workaround, but these dependencies should be acyclic as well.

6.2 Generation of a build script

Although there already existed a build script for LATIN2 when the formalizations started, it soon turned out to be deprecated. Not only because new files were added to formalize set theory and transformations, but also because the build script was more of a quick solution that barely considered the dependencies of the files. Therefore complete folders were built multiple times in order to get working MMT files. Since building all files manually would be time consuming and requires users to know the dependencies or lose more time trying to get it somehow correct, a new version of a build script was necessary.

MMT has its own shell and in its documentation information regarding scripts is given [Mmtc]. This includes information about logging and archives that was already used in the deprecated build file. However, the deprecated file was only able to build the complete LATIN2 library (or at least all folders that were mentioned in the script). Since this can take some time and is therefore not always wanted, the new build script should be more flexible to build only a specific part. Therefore independent commands like logging and the mathpath archives are now in an extra configuration file which is used by all build scripts.
To achieve the flexibility each folder has its own script to build the MMT files within it. However, small folders with only one or two files did not get a script since they can be build manually or their order does not matter. Additionally, they get build within a larger script that builds every file in LATIN2 except those that should be skipped by now. The larger script named *build-omdoc.msl* uses all of the other build scripts. Every folder without a

script gets build here according to the documentation [Mmta]. Also, it creates a html file to have a better overview of the shell output.

Now it shall be explained how the build script(s) can be started. The main build script *build-omdoc.msl* is located directly in LATIN2, while the smaller build scripts are included in their corresponding folder. One possibility to run a build script is to use *run-file.bat* located in MMT/deploy. *run-file.bat* takes a build script as an argument and runs it. However, this is specific for Windows.
A more general approach to run a build script is to start the MMT shell. Then a build script can be run with the *file* command, though it is very important that *build-omdoc.msl* has to be started with the option *--noqueue*. Otherwise the queue that is activated by default could ruin the special build order [Mmtc].

```
C:/User> java -jar MMT/deploy/mmt.jar
mmt>file C:/User/MMT/LATIN2/build-omdoc.msl --noqueue
```

This can also be done in the MMT editor itself. For example, jEdit has at its bottom next to the error list a console. By default the console is set to *system*, but, by clicking the arrow next to it, it can be changed to *mmt*. Then the build script can be run with the *file* command just like before.
If the build script is started outside of an MMT editor, it might be necessary to *clear* the MMT memory of this editor. In jEdit there exists a button for this case. Also, it is advisable to delete the files of the LATIN2 *content* folder sometimes, since it might contain deprecated files. These files could lead to errors or prevent error messages.

6.3 Name resolution

Another problem that occurred is related to disambiguity. For example, the untyped and typed universal quantifier could both have the local name *forall* and use the notation $\forall$. This works as long as both theories are not used at the same time, otherwise MMT is not able to refer to the right theory. However, to formalize transformations it was necessary to have both theories in one view. Therefore changes to the local name and notation had to be made [Rab20].

The solution for local names is straightforward. Each theory that also occurs in another type system or variant needs a unique name. To achieve this, each type system gets associated with a letter: **u**ntyped, **t**yped and **r**elativized (soft typed). Also the dependent variants of e.g. propositional conjunction get associated with the letter **d**.
Then each local name that needs to be changed gets the corresponding letter as a prefix. For example, untyped universal quantification gets named *uforall* and its typed counterpart is *tforall*. The same applies for their natural deduction rules.

Although these changes would be enough to solve the disambiguity issue, they are not enough since shorter notations like $\forall$ still cannot be used. There are different approaches for this

problem.
First, different symbols could be used. For example, *uforall* could be denoted with $\forall$ while *tforall* uses $\bigvee$. Obviously this solution has some disadvantages, because many symbols are necessary. On the one hand, a formula that does not use the standard mathematical symbols is less intuitive to read. Everyone using MMT would have to learn the MMT specific notations to a certain degree. This could make it harder for MMT to be established. On the other hand, mathematicians already use many symbols. Therefore it gets hard to find suitable notations that are not already used. Overall this approach does not seem to be the right one.

Another idea consisted of modifying the notation symbols which are unicode characters. Unicode provides already some packages to change the appearance of its characters like *Nonspacing Mark* or *Modifier Letter*. These categories contain more than 2000 characters. However, many characters cannot be illustrated in all programs. This problem is not only MMT related, because even websites may fail to display these characters. For example, the combining latin small letter *s* is encoded as U+1DE4, but most likely searching online for it will result in a disappointing square.
Unicode is aware of this issue and suggests to change the font. Still these characters should be avoided in order to increase the user friendliness.
Additionally, the used modification should be easy to understand and distinguish. Therefore small dots, lines and different oriented apostrophes might not be suited, whereas small letters seem to be predestined for this purpose. Especially because the associated letters from before can be reused [Ucc; Udi].

Still there is a decision to be made since both *Modifier Letter* and *Nonspacing Mark* contain small shifted letters. *Modifier letters* contain small letters that are shifted up or down. Latter are marked as "subscript" whereas the first mentioned ones are sometimes called "superscript", but most often they are denoted as "modifier letter". Small letters in *Nonspacing Marks* are combining characters like the combining latin small letter *s* from before. In general, a combining character "merges" with its previous character to create a new character that can be merged as well. Therefore they can even appear before or below their previous character. Combining latin small letters are coded to appear above the previous letter, but in the MMT editor jEdit they get shifted to the right. Therefore there is nearly no visible difference between both categories in jEdit [Ucc].

Like mentioned before, the combining latin small letter *s* cannot be displayed in all fonts and should therefore be avoided. However, this applies only for about half the combining latin small letters. The other half that contains the desired letters *u*, *t*, *r* and *d* works perfectly fine [Ucc].
Modifier letters can all be displayed correctly, but they have already been used in MMT formalizations and count as "normal" letters in Java class *Character*. This is a problem, because the parser had to be changed in order to handle the modified notations properly.

These changes consisted of adding the (combining) letter to the previously scanned token and ending the token after all (combining) letters have been read.
It would have been more ponderous to do these changes for modifier letters without ruining formalizations. The only unicode character of *Modifier letters* that got explicitly included in these changes is the modifier letter raised exclamation mark (U+A71D) which is used to denote the unique existential quantifier. Otherwise combining latin small letters are chosen to modify the notation symbols. Also, this has the nice effect that modifier letters can still be used like normal letters for e.g. local names without getting unwanted behaviour [Jch; Ucc].

Additionally to the parser changes new abbreviations had to be added in *mmt-api*. First of all, the working combining characters needed an MMT abbreviation in order to use them. Then shortcuts for frequently needed modified symbols like $\forall^u$ were added. However, some useful abbreviations might still be missing.
If wanted, they can be added to the file *unicode-latex-map* of the *mmt-api*. Then *mmt.jar* has to be build according to [Mmtb]. To update the MMT plugin in the MMT editor, e.g. jEdit, the shell command

```
C:/User/MMT/deploy> java -jar ./mmt.jar :jeditsetup install
```

has to be made [Mmtb].

7 Conclusion and Future Work

Overview We have created a modular formalization of set theory in MMT's library *LATIN2* (`https://gl.mathhub.info/MMT/LATIN2/-/tree/devel/source/set_theory`). Then we have visualized our formalizations in a graph (`https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/source/set_theory/graph/set_theory.svg`). We have presented the features with examples to explain the process of the formalization. Additionally, schemes have been provided to support the definitions of features in the future.

Another aspect has been transformations into set theory, especially the transformation of type theory. We have transformed some type theoretical features and the universal and existential quantifier of logic into set theory. Also we have defined a scheme for the transformation into set theory. These transformations have been formalized in `https://gl.mathhub.info/MMT/LATIN2/-/tree/devel/source/transformations_to_set`. An incomplete graph showing the interactions of the formalizations in *LATIN2* is located at `https://gl.mathhub.info/MMT/LATIN2/-/blob/devel/LATIN2-graph.svg`.

Because of the high scaling, we have reorganized *LATIN2* and provided a build script for it. Additionally, we have introduced a modification for notations and names to support the disambiguation. Therefore not only suitable modifiers had to be chosen, but the MMT API had to be adapted as well.

Thus the result of our work led to a new *LATIN2* release which can easily be compiled using the build script. Also, the graphs help especially new users to understand the parts of *LATIN2* and how they are connected with each other.

Future Work There exist multiple variants of set theories, like different versions of ZF(C) and Tarski-Grothendieck, that still have to be formalized. Additionally, many more features can be formalized in set theory. While some are necessary to perform further transformations of type theory in set theory, other features could be removed from set theory (e.g. Image), because they are basically equivalent to the soft-typed version.

Certainly, further transformations into set theory have to be formalized as well. Some features of type theory could not be transformed, because the set theoretical feature is not defined by now. However, sometimes (soft-)type theory is not complete or incorrect which prevented a transformation. Therefore (soft-)type theory has to be improved as well.
Furthermore the dependency issue for transformations has to be solved. Other parts of *LATIN2* can be transformed into set theory as well, like different logics. In addition, it is desired to automate these transformations in the future for more efficiency.

Finally, some suggestions for the improvement of MMT shall be provided:

- Views should be able to support multiple codomains.
- An approach for a realization of role Simplify (and Solve) has to be discussed.
- The realization of structures and especially nested structures need to be improved.
- Error messages regarding morphisms have to be fixed. For example, MMT does not give a warning when `TypeEqualityND` is realized in set theory. However, when this transformation is used, MMT throws an error because the realization is not total. Either this error message is incorrect or there should be a warning that the realization is not complete.
- Issues have to be solved that the identity morphism has to be included every time, although it should be inherited. For example, in *fundamentals_transformed.mmt* it should not be necessary to include *Propositions* every time in order to have a total realization.
- The identity morphisms in views and realizes lead to problems, since MMT is unable to substitute them if they have been realized before. For example, if `OrderedPairs` have been realized as `KuratowskiPairs` and later the `CartesianProduct` shall be realized with a view, an error occurs. Because MMT cannot substitute the identity morphism with `KuratowskiPairs`, realizations as in *zfc.mmt* are not possible.

- In general, the performance must be improved. Especially in the case of *zfc.mmt* it would be useful, if each realization would only be checked when it got changed. Otherwise they could be ignored to improve the performance.
- Role Simplify should work smarter such that it does not get applied, if the result is of the "expanded" form. At least there should be an option to explicitly forbid the automatical use of simplification in a proof step.

References

[AEU18] J. Avigad, G. Ebner, and S. Ullrich. *The Lean Reference Manual*. 2018.

[Cmo] *The Module System - Coq 8.13.2 documentation*. `https://coq.inria.fr/distrib/current/refman/language/core/modules.html`. Accessed: 2021-08-05.

[Cod+11] M. Codescu et al. "Project Abstract: Logic Atlas and Integrator (LATIN)". In: *Intelligent Computer Mathematics*. Springer, 2011, pp. 289–291.

[Cst] *Standard Library | The Coq Proof Assistant*. `https://coq.inria.fr/library/`. Accessed: 2021-08-05.

[Cty] *Typing rules - Coq 8.13.2 documentation*. `https://coq.inria.fr/distrib/current/refman/language/cic.html`. Accessed: 2021-08-04.

[Czf] *An Encoding of Zermelo-Fraenkel Set Theory in Coq*. `https://github.com/coq-contribs/zfc`. Accessed: 2021-09-16.

[Hal60] P.R. Halmos. *Naive Set Theory*. Van Nostrand, 1960.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. "A Framework for Defining Logics". In: (Jan. 1993). DOI: 10.1145/138027.138060. URL: https://doi.org/10.1145/138027.138060.

[Ido] *Documentation*. `https://isabelle.in.tum.de/documentation.html`. Accessed: 2021-08-08.

[Iho] *Session HOL-ZF*. `https://isabelle.in.tum.de/library/HOL/HOL-ZF/index.html`. Accessed: 2021-08-09.

[Ipu] *Theory Pure*. `https://isabelle.in.tum.de/dist/library/Pure/Pure/Pure.html`. Accessed: 2021-09-19.

[IR11] M. Iancu and F. Rabe. "Formalizing Foundations of Mathematics". In: *Mathematical Structures in Computer Science* 21.4 (2011). DOI: 10.1017/S0960129511000144. URL: `https://www.cambridge.org/core/services/aop-cambridge-core/content/view/2E3F3CC119D3AAAEFD3D21CE5BAD7E33/S0960129511000144a.pdf/formalising_foundations_of_mathematics.pdf`.

[Izf] *Session ZF*. `https://isabelle.in.tum.de/dist/library/ZF/ZF/index.html`. Accessed: 2021-08-08.

[Jch] *Character (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html. Accessed: 2021-07-28.

[Kai12] J. Kaiser. *Formal Construction of a Set Theory in Coq*. 2012.

[Kre02] C. Kreitz. *The Nuprl Proof Development System, Version 5*. 2002.

[Lax] *23. Axiomatic Foundations - Logic and Proof 3.18.4 documentation*. https://leanprover.github.io/logic_and_proof/axiomatic_foundations.html. Accessed: 2021-09-16.

[Lin] *index - mathlib docs*. https://leanprover-community.github.io/mathlib_docs/. Accessed: 2021-08-08.

[Lma] *Mathematics in mathlib*. https://leanprover-community.github.io/mathlib-overview.html. Accessed: 2021-08-08.

[Lpr] *Programming in Lean*. https://leanprover.github.io/programming_in_lean/. Accessed: 2021-08-08.

[Lse] *12. Sets in Lean - Logic and Proof 3.18.4 documentation*. https://leanprover.github.io/logic_and_proof/sets_in_lean.html. Accessed: 2021-08-08.

[Lwh] *What is Lean - Lean Manual*. https://leanprover.github.io/lean4/doc/whatIsLean.html. Accessed: 2021-08-08.

[Lzf] *mathlib/zfc.lean*. https://github.com/leanprover-community/mathlib/blob/b7593841620449def9435f0b9f3a1002afecff53/src/set_theory/zfc.lean. Accessed: 2021-09-16.

[Mgr] *Grammar of Mizar*. http://www.mizar.org/language/mizar-grammar.xml. Accessed: 2021-08-06.

[Min] *Index of /version/current/mml*. http://mizar.uwb.edu.pl/version/current/mml/. Accessed: 2021-08-06.

[Mmm] *Mizar Home Page: Mizar Mathematical Library*. http://www.mizar.org/library/. Accessed: 2021-08-06.

[Mmta] *MMT - Building Documents*. https://uniformal.github.io//doc/archives/building.html. Accessed: 2021-07-14.

[Mmtb] *MMT - Details on Building using SBT*. https://uniformal.github.io/doc/setup/sbt. Accessed: 2021-07-28.

[Mmtc] *MMT - The MMT Shell*. https://uniformal.github.io/doc/applications/shell.html. Accessed: 2021-07-14.

[Nbr] *Nuprl Browsing - how to*. http://www.nuprl.org/sfa/Nuprl/Shared/Xweb_projecting_sections_doc.html. Accessed: 2021-08-05.

[Ncr] *PRL Cross-links to Type Theory, Set Theory and Domain Theory*. http://www.nuprl.org/Intro/TypeSetDomain/typesetd.html. Accessed: 2021-08-05.

[Neq] *Nuprl Basics - Equality and Membership*. http://www.nuprl.org/sfa/Nuprl/NuprlPrimitives/Xequality_doc.html. Accessed: 2021-09-20.

[Nma] *PRL Project Library of Formal Definitions and Proofs*. http://nuprl.org/MathLibrary/. Accessed: 2021-08-05.

[Nov] *Overview*. http://www.nuprl.org/book/Overview.html. Accessed: 2021-08-05.

[Nty] *Nuprl Basics - Types: Ontic, Semantic, and Intensional*. http://www.nuprl.org/sfa/Nuprl/NuprlPrimitives/Xtype_doc.html. Accessed: 2021-09-20.

[Pin14] C. Pinter. *A Book of Set Theory*. 2014.

[Rab20] F. Rabe. "MMT: The Meta Meta Tool (system description)". In: 2020.

[RK13] F. Rabe and M. Kohlhase. "A scalable module system". In: *Information and Computation* (2013). DOI: https://doi.org/10.1016/j.ic.2013.06.001.

[RR21] F. Rabe and N. Roux. "Modular Formalization of Formal Systems". In: 2021.

[RS09] F. Rabe and C. Schürmann. "A Practical Module System for LF". In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. Association for Computing Machinery, 2009, 40–48. URL: https://doi.org/10.1145/1577824.1577831.

[Sal19] M. Saltzman. "A Little Set Theory (Never Hurt Anybody)". In: (2019). URL: https://cecas.clemson.edu/~mjs/courses/misc/settheory.pdf.

[SM08] D. Scott and D. McCarty. "Reconsidering Ordered Pairs". In: *The Bulletin of Symbolic Logic* 14.3 (2008), pp. 379–397. URL: http://www.jstor.org/stable/20059989.

[SY20] T. Sun and W. Yu. "A Formal System of Axiomatic Set Theory in Coq". In: *IEEE Access* 8 (2020), pp. 21510–21523. DOI: 10.1109/ACCESS.2020.2969486.

[Twe] *source/foundations/zfc*. https://gl.mathhub.info/MMT/LATIN/-/tree/master/source/foundations/zfc. Accessed: 2021-08-11.

[Ucc] *Unicode Character Categories*. https://www.compart.com/en/unicode/category. Accessed: 2021-07-27.

[Udi] *Unicode Display Problems*. http://www.unicode.org/help/display_problems.html. Accessed: 2021-07-27.

[Wel20] P.D. Welch. *Axiomatic Set Theory*. 2020.

[Wen21a] M. Wenzel. *The Isabelle System Manual*. 2021.

[Wen21b] M. Wenzel. *The Isabelle/Isar Implementation*. 2021.

[Wen21c] M. Wenzel. *The Isabelle/Isar Reference Manual*. 2021.

[Wie07] F. Wiedijk. "Mizar's Soft Type System". In: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 2007, pp. 383–399.