Friedrich-Alexander-Universität
Erlangen-Nürnberg

Bachelor Thesis in Computer Science

Refactoring of Theory Graphs in Knowledge Representation Systems

---

Navid Roux

Advisors: Prof. Dr. Michael Kohlhase, PD Dr. Florian Rabe

Erlangen, 1st July 2019

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, 1. Juli 2019 _____

Contents

List of Figures

1. Introduction

The Need for Formalization In recent years there has been increasing interest in computer-assisted and computer-verified theorem proving in both the mathematical and computer science community. Notable theorems with long and dependency-rich proofs have been verified thereby [Hal+17; Gon+13; Gon08]. Furthermore, it has become a niche trend to submit papers teogether with machine-checked proofs. For such tasks large mathematical corpora needed and still need to be formalized. Unlike natural language proofs, where importing key results of a whole field of mathematics can be as short as few sentences in a proof, in the setting of (fully) computer-verified proofs all transitive dependencies require formalization. Another interest in such corpora particularly stems from the Mathematical Knowledge Management (MKM) community [CF09] which desires to work with mathematical knowledge as accessible data. Similar to databases, structuredness of data in a digitized form with rich semantics and interconnections is a key enabler for many useful operations. Those operations include "cataloguing, retrieval, refactoring, change propagation [...] and in some cases even application of mathematical knowledge." ([Koh14]) Indeed, possessing formalized knowledge may help to overcome the "One-Brain-Barrier" [Koh14; Car+19]: While humans are particularly good at tasks on small amounts of data requiring intuition and insights, computers are far better in handling large datasets – albeit with more shallow tasks.

Computer Assistance in Formalization In the previous paragraph we motivated the need for formalization from different perspectives. In all cases, this gives rise to the tasks of creation and maintenance of such formalized knowledge. To foster their understanding and possibly support humans performing them, we can borrow and evaluate field-tested concepts from the field of software engineering, where instead of formalized knowledge we create and maintain code. To name a few of such concepts in software engineering, we have for example full-blown IDEs with syntax highlighting, autocomplete, (boilerplate) code generation, linting and refactoring tools. In addition, the ecosystem – "DevOps" – is well developed, for instance code changes in pull requests on GitHub can be made to be automatically evaluated for their code quality and possibly rejected if they do not fulfill specified requirements. Overall, the success of such tools in software engineering is a good indicator for the usefulness of transferring them to formal systems and MKM. In fact, we posit that the converse could also be true, namely that some tools with inherent strict formalism could be fruitful to software engineering.[1]

Furthermore, we suppose assistance in rapid prototyping might also have applicability

[1] We hint at our refactoring principle presented in Chapter 5 together with its running example, which is borderline to formal specifications.

in the realm of developing and formalizing new mathematics. As definitions, theorems and proofs often undergo various revisions during development at the research frontier, so must formalizations and computer-verified proofs. Hence, appropriate refactoring tools accounting for such moving targets would allow a tighter feedback loop between conceptual development and computer-assisted formalization.

Contribution of the Thesis We focus ourselves on refactoring, precisely on refactorings to generalize existing knowledge. Basing our discussions on the MMT knowledge representation framework [RK13], we first formally define what a generalization refactoring is and then present two such examples, namely Theory Splitting and App-Gen, which is short for "application of existing generalization". We consider the main contribution to be two-fold: First, our definition of a generalization refactoring allows us to express *and* partially verify behavior preservation within the system. Second, the refactoring principle App-Gen, makes the ubiquitous practice of abstraction in mathematics and computer science explicit in form of an algorithm and an IDE graphical plugin implementation. Concretely, we have the following use cases in mind, which we shortly comment on below.

a) Help making existing knowledge more abstract

b) Help playing around with formalizations of new things

c) Help formalizing abstract concepts / proofs by *only* giving them in a concrete setting and letting them generalize automatically

d) Help students understand abstractions and limitations thereof

Point a) is common in authoring big libraries in programming, which often necessitate a flexible and thus abstract API for consumers. Indeed as history has shown, mathematics is strikingly similar. While b) is very frequent in programming as well, we suppose it has not yet gained likewise attraction in mathematics, partially due to usability differences between a theorem prover and pen & paper. Hopefully, the advent of refactoring tools in the future is able to marginally bridge this gap. We imagine use case c) being applicable in two scenarios. First, formalization in a concrete setting might be cognitively easier. Second, in narration or "text book" settings [Car+19] for diactic reasons, things are presented differently, potentially more concrete than in library settings. Hence, automatic generalization might allow the reader to firm their abstract knowledge after having understood the concrete setting. As a special case alluded by d), knowing when concepts do *not* generalize is useful in people's mathematical education.

Independent of human-oriented use cases above, saving more abstract knowledge together with concrete incarnations widens the induced knowledge space and hence is beneficial for MKM.

Structure of the Thesis In Chapter 2 we convey preliminaries regarding MMT and review related work. Chapter 3 motivates and presents our framework for generalization

refactorings. In particular it defines theory- and diagram-level generalizations as well as corresponding requirements on behavior preservation. Based on this framework, Chapters 4 and 5 present our two refactoring principles Theory Splitting and App-Gen, respectively. In Chapter 6 we discuss relevant implementation details and classify our principles into the MMT ecosystem from an end user's perspective. Finally, Chapter 7 concludes the thesis and shares ideas about future work.

2. Preliminaries

2.1. MMT

In this work we leverage MMT [RK13] on the one hand as a theoretical framework to formulate our discussions in and on the other hand as an implemented knowledge representation system [MMTb; Rab18] to build our algorithms on. We strive to distinguish them by denoting the theoretical framework as "MMT" and the practical implementation as the "MMT system". In this section, we shortly introduce and recapitulate a simplified version of MMT's theoretical side necessary to understand this work and follow the exposition in [RM18a].

There are many point of views from which MMT and the MMT language can be introduced. Central to this work is its intention to allow a scalable module system for mathematical knowledge management while being foundation and logic independent [Rab15]. We see in a minute what this means in practice. The essential way in which formal knowledge[1] is organized in MMT is by means of theories and theory morphisms.

Theories Theories are ordered lists of typed declarations with unique names and, for structuring purposes, also inclusions:

Definition 2.1.1 (Theory, adapted from [RM18a])**.** The grammar for **theories and expressions** is

$TDec$	$::=$	$T = \{Dec, \dots, Dec\}$	theory declaration
Dec	$::=$	$c : E[= E] \mid$ include T	constant or include declaration
E	$::=$	$c \mid \dots$	expressions built from constants

We write $\texttt{dom}(T)$ for all declarations of a theory, and $\texttt{Obj}(T)$ for all induced closed and well-typed expressions over $\texttt{dom}(T)$. If S is included in T, we write $S \hookrightarrow T$.

Assumption of Well-typedness

Except where otherwise noted, we assume well-typed theories and below well-typed (total) morphisms throughout this thesis. Furthermore, we omit the actual typing rules as they are mostly irrelevant for this thesis and the reader may safely read

[1] Although we focus on formal knowledge in this thesis, do note that MMT and its related ecosystem can also be used for stating flexiformal knowledge, e.g. see [Ian+16] for a concrete implementation and [Ian17] for more theoretical background.

over the parts where they are nevertheless used.

For the definition of morphisms later we need one more utility definition:

Definition 2.1.2 (Flat Theory)**.** A **flat theory** is a theory without inclusions. Every theory T admits a flattening $T^\flat$, which replaces every inclusion by the flattening of the included theory.[a]

[a] One important aspect, which might go unnoticed with our informal definition, is that diamond inheritance via inclusions does not lead to duplicated symbols, but rather unifies them, i.e. only copies those symbols once over to the target theory. We refer to [RM18a] for details.

Declarations are able to subsume many established notions such as "type / function / predicate symbols, axioms [and] theorems" ([RM18a]) as well as proofs by employing the propositions-as-types idiom. Let us elaborate on that by showing how the concept of a monoid could be formalized:

```
1  theory Monoid =
2    include ?LF ❚
3    include ?NatDed ❚
4
5    U: type ❚
6    e: U ❚
7    op: U → U → U | # #1 ∘ #2 ❚
8    associative: ⊢ ∀ [a: U] ∀ [b: U] ∀ [c: U] (a ∘ b) ∘ c ≐ a ∘ (b ∘ c) ❚
9
10   neutral: U → prop | = [e'] ∀ [a: U] (a ∘ e' ≐ a) ∧ (e' ∘ a ≐ a) ❚
11   e_neutral: ⊢ neutral e ❚
12   e_unique: ⊢ ∀ [e': U] (neutral e') ⇒ e' ≐ e
13     | = … ❚  /T Proof omitted ❚
14
15   power: U → ℕ → U | # #1 ^ #2 | = … ❚
16   thm_power: ⊢ ∀ [a: U] ∀ [n: ℕ] ∀ [m: ℕ]  a ^ (n + m) ≐ (a ^ n) ∘ (a ^ m)
17     | = … ❚  /T Proof omitted ❚
18 ❚
```

Remark 2.1.1 (MMT Surface Syntax). For convenience, instead of adhering to the formal grammar above, we use the actual MMT surface syntax [MMTa; Rab14] employed by the MMT system to present listings. Apart from two aspects we mention below, we expect the reader to be able to easily match this syntax to the formal grammar.

The three symbols | ❚ ❚ are terminators at the levels of being within a declaration, outside a declaration within a theory, and outside a theory, respectively.

They can be thought of as refined versions of semicoli in C-inspired programming languages and can be safely ignored when reading the present thesis. Another difference to the formal grammar is the ability to create custom notations for declarations. For example, in the declaration for `op` we write `# #1 ∘ #2` to introduce a new infix notation ∘ receiving two arguments corresponding exactly to the first and second argument of `op` itself.

Given the hint that `[…]` denotes lambda abstraction in the listing above, we assume the reader is familiar enough with basic (higher-order) logic to understand the listing at surface level, possibly disregarding synctactic unfamiliarities. Let us spot the different meanings MMT declarations subsume. Declarations without a definiens (`U`, `e`, `op`, …) constitute the signature and language for formulating the knowledge in. They can be thought of as (formal) constructors. As a special case, we have axioms (`associative`, `e_neutral`) via propositions-as-types. In contrast, declarations with a definiens serve the following two purposes. First, they can act as definitions as such (`neutral`, `power`) to shorten other declarations. Second, arguably as a special case, they can represent theorems (`e_unique`, `thm_power`), where the definiens is an inhabitant of the type – or dually a valid proof.

Let us now revisit the declarations we skipped over. Precisely, we neither explained what the two inclusions do nor where all the symbols (`type`, `⟶`, `⊦`, …) come from given that they were not part of the formal grammar definition above. Since MMT strives to be foundation and logic independent, many if not almost all symbols of the listing above are actually not baked into MMT's core language. That is where the inclusions come into play. On the one hand, we include the Logical Framework LF [HHP93] to serve as our foundation, which provides us with the symbols and notations `type⟶[…]` together with their corresponding typing rules. Indeed, the MMT framework only has very few typing rules on its own and outsources this work to foundations. On the other hand, we include `NatDed` which we let stand for a generic theory formalizing basic logic, natural deduction calculi and proof types. This allows us to use the symbols `⊦∀∧≐⟹` and the proof constructors we left out in the omitted proofs.[2] For brevity, we will readily assume both inclusions to be always existent in our listings and will thus omit them.

Remark 2.1.2 (Omittance of Proofs). At time of writing, the MMT system does not offer a practical way to formalize proofs since proofs can only be given as "raw" proof terms with constructors of the chosen natural deduction calculus. As a consequence, we omit proofs in definienses throughout the listings of the thesis.

[2] In the MMT system, LF is provided by an archive called `MMT/urtheories` of which parts are implemented as Scala extensions to the pluggable MMT typechecker. The natural deduction calculus can be found in `MitM/Foundation`. For both see https://gl.mathhub.info/MMT/urtheories/blob/master/source/lf.mmt and https://gl.mathhub.info/MitM/Foundation/blob/master/source/math.mmt, respectively.

Morphisms The other building block of knowledge organization in MMT are morphisms between theories. A (total) morphism $\varphi\colon S \rightsquigarrow T$ is a set of assignments $c := e$ for every $c \in \texttt{dom}(S^\flat)$ with $e \in \texttt{Obj}(T^\flat)$. For practical purposes, we can instead consider its "more structured" definition on non-flat theories as well:

Definition 2.1.3 (Morphism, adapted from [RM18a])**.** The grammar for **morphisms** is

$MDec$	$::=$	$m : T \to T =^{[M]} \{Ass, \dots, Ass\}$	flat morphism declaration
Ass	$::=$	$c := E \mid$ include M	assignment to declaration or inclusion of morphism

A morphism can "inherit" the assignment of another morphism by including it. For example, this is useful when we have theories $R \hookrightarrow S$, an existing morphism $\varphi\colon R \rightsquigarrow T$ and wish to extend that morphism to $\psi : S \rightsquigarrow T$. That extension is possible with just `include φ` without mimicking all the assignment of φ.

Naturally, every morphism $S \rightsquigarrow T$ can be lifted in its domain $\texttt{dom}(S)$ to $\texttt{Obj}(S)$, which we call the **homomorphic extension** $\overline{\varphi} : \texttt{Obj}(S) \to \texttt{Obj}(T)$.[3] As an immediate consequence, we realize that morphisms only need to defined on constants without definiens as their homomorphic extension accounts for them anyway. Overall, morphisms provide a means to *translate* expressions of one theory to another and thus to connect existing knowledge. The importance and meaning of morphisms is elaborated on in Chapter 3. Nonetheless, we would like to highlight one particularly important corollary here [Rab17]:

Corollary 2.1.0.1 (Truth Preservation)**.** *If $c :\vdash E = e$ is a theorem in a theory T and $\varphi\colon T \rightsquigarrow S$ a morphism leaving the logic foundation unchanged, then $\vdash \overline{\varphi}(E)$ is a theorem in S as well with proof $\overline{\varphi}(e)$*

Proof. This readily follows from the fact that all entities were well-typed, which implies that images under the homomorphic extension of the morphism are well-typed as well. □

In fact, inclusions $R \hookrightarrow S$ are morphisms as well, namely, they just assign every domain declaration in $R^\flat$ its corresponding copy in $S^\flat$.

In MMT surface syntax, morphisms are also known as "views" and are written as follows:

```
1 /T This corresponds to a morphism φ: S ⇝ T
2 view φ : S -> T =
```

[3] In this view, we consider the homomorphic extension as a function on sets and therefore write $\to$ instead of $\rightsquigarrow$. In Section 5.4 we will briefly note that homomorphic extensions can be thought of pushouts, which would justify them being similar to morphisms again.

```
3   a = b
4   c = d
5
```

Category $\mathbb{T}\mathrm{hy}$ In Section 5.4, we make use of categorical constructions, namely pushouts and pullbacks, to motivate a categorical interpretation of the refactoring principle presented there. Hence, we define the category we are working in by

Definition 2.1.4 (Category)**.** By $\mathbb{T}\mathrm{hy}$ we denote the category of (well-typed) theories and theory morphisms with the apparent identity morphisms and the composition of $\varphi\colon R \nrightarrow S$, $\psi\colon S \nrightarrow T$ given by

$$(\psi \circ \varphi)(c) := \overline{\psi}(\varphi(c))$$

For convenience, we will use the categorical term "diagram" to refer to a finite subset of theories and morphisms. When stressing the contained knowledge such a diagram represents, we occasionally refer to it as *knowledge graph*.

Paths We omitted the fact that in the more general framework [RK13] many entities are assigned a unique path, a so-called MMT URI. In particular, theories, morphisms as well as declarations inside theories and morphisms all have corresponding paths. We will only occasionally refer to them.

2.2. MMT Dependency Order

The refactoring principle Theory Splitting we state in Chapter 4 is able to reorganize a theory's declarations onto multiple new theories. In particular, this requires knowing dependencies between declarations, which this section will define. For example, in the monoid formalization we saw above, it is evident that we need the declaration of the universe type U – among others – to state the axiom of associativity. At the end of this section we will be able to formally state $U \leq_T$ *associative* to precisely mean this dependency.

To allow for a more general framework we might exploit in the future, we desire to first define a finer dependency relation, namely on declaration components:

Definition 2.2.1. Declaration components of

- an include T declaration is the singleton set $\{\text{include } T\}$
- a constant declaration $c : E$ without definiens is the singleton set $\{E\}$
- a defined constant declaration $c : E = e$ is the (multi[a])set $\{E, e\}$.

For a theory T let $\texttt{comps}(T)$ denote all (tagged[b]) declaration components.

[a] With the right foundation loaded into MMT we could in theory have a declaration whose type is its definiens at the same time.
[b] If two declarations have the same type and/or definiens, we still consider their components to be distinct in $\texttt{comps}(T)$.

Definition 2.2.2 (Dependency Relations)**.** Let T be a theory. The **component dependency relation** $\leq_C$ on $\texttt{comps}(T)$ is given by the smallest relation fulfilling

$$\frac{}{x \leq_C x} \qquad \frac{x \text{ is looked up (transitively) while typechecking } y}{x \leq_C y}$$

Likewise the **declaration dependency relation** $\leq_T$ on $\texttt{dom}(T)$ is given by the smallest relation fulfilling

$$\frac{x \in \delta \qquad y \in \delta' \qquad x \leq_C y}{\delta \leq_T \delta'}$$

where x and y are components of declarations δ and δ', respectively.
Unless stated otherwise, $<_C$ and $<_T$ denote the induced strict orders.

Remark 2.2.1. For declarations $c : E = e$ we especially have $E \leq_C e$.

Lemma 2.2.1. *Let T be a theory.*

(i) $\leq_C$ is a partial order.

(ii) $\leq_T$ is a partial order.

(iii) The theory with the same declarations as T stemming from a reordering given by any linear extension of $\leq_T$ is well-typed.

Proof. ad (i), (ii): Reflexivity and transitivity can be easily seen. Antisymmetry is due to the wellformedness of the theory.
ad (iii): Suppose δ were ill-typed in the reordering given by $\sqsubseteq_T$. This means that while typechecking, another declaration $\delta' \neq \delta$ was attempted to be looked up, but not found because it appeared after δ, i.e. $\delta' \leq_T \delta$ and $\delta \sqsubseteq_T \delta'$. This implies $\delta' = \delta$ leading to a contradiction. □

Example 2.2.1. The sequential notation of declarations of theories in this thesis and in MMT surface syntax is just one possible linear extension of $\leq_T$.

Remark 2.2.2. Narrative Order If the particular total order is irrelevant, we will speak of "the" narrative order.

2.3. Related Work

We describe related work for refactoring in general and specific to both generalization principles we present in this work, respectively.

2.3.1. Refactoring

The concept and act of refactoring is well-known in the discipline of software engineering in both academia and industry. A recent literature review can for example be found in [SK18]. Pioneering works [Opd92; Fow+99] defined the central idea of refactoring as "[changing the] internal structure of software to make it easier to understand [...] without changing its observable behavior." ([Fow+99, ch. 2]) In the present thesis we have a similar goal in mind for formal knowledge graphs. However, we interpret it slightly more losely to capture additional supposedly useful refactorings: Apart from making the knowledge graph easier to understand for users, we also consider the act of abstraction and increase of knowledge-induced items to be a refactoring, namely from the MKM point of view. We remark that our standpoint shares similarities with database engineering where on the one hand database views ease accessibility by users (and possibly performance) and on the other hand normal forms remove redundancies and increase "buildable" knowledge.

Refactorings have been suggested and applied not only on programming languages, but also on more formal documents such as UML models [Sun+01], formal specifications in Z [SPT02; LZ08] and tactic-based proofs [Whi13; SH02]. In this work we present refactorings on MMT theories, which subsume formal specifications and (linearized[4]) proofs via the propositions-as-types idiom. We focus ourselves on refactorings to generalize existing knowledge. Such generalization refactorings have already been suggested in [Opd92; Fow+99], however mostly in the realm of classes such as Pull Up Field / Method, Extract Subclass / Superclass / Interface, Collapse Hierarchy. These principles are all instances of class hierarchy reorganization and hence are closely related to Theory Splitting. Therefore, we continue this discussion in the respective subsection below.

[4]Indeed, as noted in Remark 2.1.2 proofs are practically *not* tractable in the MMT system.

Behavior Preservation The crucial property of behavior preservation has been defined and dealt with varying degrees of formality throughout the refactoring discipline. However, neglecting the fluctuating formality in all works cited below, common to all is that they deal with behavior preservation at most in natural language on the meta level. We explain below how our framework allows us to express *and* partially verify that property *within* the MMT system itself.

The early works [Opd92; Fow+99] targetting Java as their exemplary language leave behavior preservation undefined as a fuzzy concept. Even though [Opd92] states preconditions for their refactorings as first-order logic formulae and argues larger refactorings be behavior preserving as compositions of smaller ones, eventually behavior preservation for those building blocks is argued informally. Indeed, to tackle the inherent semantic complexity of Java, there have been efforts to formally reason about behavior by using an executable formal semantics of Java [GM06] or by temporary compilation to a simpler intermediate format [Sch+12]. In the latter case, the refactoring is described for and applied in the simpler format, after which the code is again translated back to Java. Eventually, formal systems such as proof languages inherently admit a completely rigorous treatment [Whi13]. In fact Liu and Zhu argue that formal systems make it easier to validate behavior preservation [LZ08].

Whiteside argues that while defining behavior preservation by type-checking would be possible, it is infeasible performance-wise and possibly insufficient to ensure preservation of intended meaning for theorem statements as users would expect [Whi13, ch. 7.4]. We counter the first point insofar that we imagine our generalized theories be represented next to their original theories in the knowledge graph anyway, making typechecking necessary in any case. Precisely, we call the theory G a generalization of a theory T if there is a morphism $G \nrightarrow T$. We posit that storing all three entities is favorable MKM-wise.[5] Thus, being a generalization amounts to type-checking all three entities. Furthermore, we define behavior preservation as the property that every T-declaration has a suitable G-expression as its preimage, i.e. all T-declarations can be recovered by following the morphism. In case of renamings, this property can be easily verified by the system. Indeed, the refactorings we propose in the present thesis produce renamings witnessing their generalizations. We remark that we have not yet built such verification into the MMT system.

2.3.2. Theory Splitting

We describe Theory Splitting as a reorganization of a single theory's declarations onto multiple theories which are then suitably linked via inclusions to account for declaration interdependencies. Employing the analogy between theories and classes, such

[5] Again we can employ the analogy to databases: Let T correspond to the rows of a database view, let $G \nrightarrow T$ correspond to the SQL statement the database view is decribed by, and finally let G be standing for the underlying tables the statement draws from. Then storing T is primarly useful user-wise, whereas storing G is useful system-wise, perhaps because other data analytics could be drawn from it. Lastly, storing the morphism (view) helps keeping T up-to-date when doing changes at G.

reorganization methods can be traced back to pioneering works [Opd92; Fow+99] and probably beyond. Especially, the following principles from [Fow+99, chs. 7, 11] all describe related reorganization actions:

- Move Method/Field
- Inline Class
- Pull Up Field/Method
- Pull Down Field/Method
- Extract Class/Subclass/Superclass/Interface
- Collapse Hierarchy

Hence, we consider Theory Splitting primarly as an example fitting both the realm of formal knowledge representation systems and our generalization framework. Most of the advantages attributed to the above refactorings can also be applied to our principle. In particular the principles prefixed by Move and Extract can be used to increase the cohesion of individual classes (theories), which is a measure on the relatedness between the members. This in turn is known to have the benefit of reducing complexity and improving maintainability of code [Ing18].

However, an arguably fortunate difference to programming languages is that declarations within a theory cannot share "hidden" behavioral dependencies as is for example the case with complex synchronization properties in mainstream languages. This is at least true for commonly found logic foundations.[6]

Source of Refactoring Albeit in the present work we do not consider automatic suggestion of theory splits in knowledge graphs, we would like to highlight previous work in software engineering tackling effectively the same problem with class hierarchies. Namely, [ST00; SS04] applied methods from the discipline of concept formation on both Java class hierarchies as such and its usage patterns in client code. As a result, new behavior-preserving class hierarchies can be created which optimally reflect usage patterns. In knowledge representation systems, we posit that usage patterns of a theory correspond to incoming and outgoing morphisms. The outgoing morphisms especially subsume child theories via inclusions. This standpoint is consistent with additional desirable properties on morphisms we suggest in a side note in Section 6.1, namely that they be total, surjective and simple. Indeed, [ST00] similarly suggests usage patterns be simple and exhaustive of features, e.g. a class must neither outsource its business logic onto the consumer nor have unused fields.

[6] In the MMT system, a logic foundation contributes typing rules by means of Scala code. Thus, in theory arbitrarily complex and unforeseeable restrictions could be imposed.

2.3.3. App-Gen

We describe App-Gen as a principle which, given a generalization of one theory T encoded as a morphism $G \rightsquigarrow T$, tries to apply the same syntactic generalization to theories including T. Put differently, it tries generalizing theories along (user-defined) morphisms.

Relation to Subtype Lifting We can choose the morphisms along we generalize in such a way that they represent subtyping. Namely, if $s \in S$ is a type symbol within a theory and $t \in T$ a subtype symbol of s, then a morphism can assign $s \mapsto t$. By the operating principle of App-Gen, we would now generalize T-expressions by inverting (rewriting) t to s, or in other words, by lifting a subtype to a supertype. In many programming languages, classes and interfaces are indeed very similar to types and for a specific class we can naturally consider the classes/mixins/interfaces it extends/implements as supertypes of that class. With these analogies, generalizing in our sense is related to the action of choosing a more general type for variable declarations. Searching for these more general types within class hiearchies and applying the generalization has already been explored and automated in [BFS07; Tip07; SM07; SMM06], mainly to decrease unnecessary coupling between classes.

Let us shortly elaborate on the refactoring algorithm implemented in [BFS07]. It can either be run in a mode such that it proposes the most general type on the existing class hierarchy (subtype) lattice or be instructed to create new minimal maximally-fitting supertypes. In both cases, the protocol given by the variable's usage, i.e. how and which methods or fields on it are accessed, is statically inspected to infer the more general type. Neglecting actually finding the more general type, App-Gen is arguably more general insofar that it operates on the "morphism lattice", which subsumes the subtype lattice. Concretely, we are able to generalize not only to inherently compatible supertypes, i.e. included theories, but also to structurally different theories under the assumption that a translating morphism is given. Independently of this fact, creation of new better fitting types can be considered to be an application of Theory Splitting.

Based on those subtype lifting refactorings, graphical user interfaces and linting tools have also been created to help users with applying the principles: based on [Tip07] the Eclipse Java IDE offers the refactorings Generalize Declared Type and Use Supertype where Possible [Fou19]. In addition, the reference [BFS07] we shortly detailled in the last paragraph provides an Eclipse plugin as well. Similar features are offered by the IntelliJ IDEA Java IDE under the name of Use Interface where Possible [Jet19].

Relation to Automated Theorem Proving Since App-Gen works on MMT theories and expressions therein, it particularly subsume theorem statements and proofs via propositions-as-types. Hence our work is related to automated abstraction of proofs as well. Indeed, in the context of automated theorem proving and understanding proofs by analogies, abstractions of proofs have been studied as an automated means of proof reuse and proof discovery [BC87; TK92; WK00; JL04].

Let us elaborate on the differences and similarities of [BC87; TK92] and our work. In these works the authors desire to exploit proofs by analogies in the following way: Suppose a user has got a theorem `thmA` they proved via `proofA` and a theorem `thmB` in mind they want to prove by (synctactically) analogous arguments. The authors then suggest the user provide a "transformation rule" representing the analogy in mind. Concretely, the transformation rule acts on a "proof scheme" of `proofA` to output a proof scheme for `proofB`. Here, proof scheme means that the rule does not necessarily act on the actual terms in `proofA`, but may rather act on free variables such that the rule's LHS is first (HO-)pattern-matched with `proofA`. Then applications of the resulting unifiers on the rule's RHS gives rise to proof candidates for `thmB`.

We exhibit very similar transformation rules under the hood in App-Gen, which we call rewrite rules instead. However, contrary to [BC87; TK92], our framework sees analogies and specializations[7] in the reversed direction. Namely, consider we have a theory R and a theory S, then a morphism $R \nrightarrow S$ represents an analogy or specialization. Now let R contain `thmB` and S contain $\{thmA, proofA\}$. Then within the framework of [BC87; TK92], we could come from S to $(R \cup proofB)$ by specifying transformation rules exactly in that direction. However in our framework, we directly use the morphism $R \nrightarrow S$ to automatically generate applicable transformation rules going from S to R. They then hopefully capture the transformation rules the user had in mind as well.

We argue that a generalization algorithm should first exploit pre-existing specialization morphisms before resorting to ask the user to manually enter additional transformation rules. In fact, independently, there are several reasons to record the morphisms in the knowledge management system anyway, cf. Chapter 3 and citations therein. However, we also note that we canont hope for an algorithm which exploits every morphism to an extent a human would be able to. Hence, we propose in our future work section that our algorithm be extensible by user-defined rewrite rules as well.

Source of Refactoring Finally, as was the case with Theory Splitting, we again defer batch application of App-Gen to future work. As the rich existence of morphisms is crucial for the applicability of this principle, we would like to refer to approaches of automatic morphism discovery in knowledge representation systems [NK07; MKR18].

[7] Analogies form a strict subset of specializations, possibly involutions, e.g. consider the prime example of $Magma \nrightarrow Magma$ swapping the operation's arguments.

3. Framework for Generalization Refactorings

MAN MUSS IMMER generalisieren.

Jacobi in the 1840s [DH82, p. 134]

The principles we present in later chapters generalize theories to more abstract theories or even to a bunch of more abstract theories, i.e. a diagram. Therefore, in this chapter we define theory-level and subsequently diagram-level generalization refactorings[1]. Roughly speaking, we will define them in terms of the existence of morphisms from the more general to the original part, and in case of diagrams with some commutation properties as well.

Structure of the Chapter In Section 3.1 we first motivate by means of examples that morphisms are indeed a sensible choice to express specializations or – if read backwards – generalizations. Then, in Section 3.2 we formally define theory- and diagram-level generalizations, for which Section 3.3 then presents initial formal notations of behavior preservation. Finally, Section 3.4 overviews some of generalization principles the authors have in mind, of which two constitute the contents of the present thesis.

3.1. Intuition for Morphisms going from the General to the Concrete

Albeit there are several different angles from which we can get a feel for MMT morphisms, common to all is that they represent *specialization* and *refinement* from their poorer, more general domain to their richer, more concrete codomain. Dually, every morphism also represents a generalization in the inverse direction, which is central to the idea of our framework. Note that in the inverse direction we do not necessarily have a "generalization morphism", but only a (multi)map. Nonetheless, the inverse map can be successfully exploited to generalize *along morphisms*, as we do in Chapter 5.
To exemplify the mentioned notions, we list a few slogans and concrete examples below, which are partly taken and adapted from [OMT; Rab17; OMT].[2]

[1] Since we consider all generalizations in this thesis to be refactorings, we will stick to only saying "generalizations" from now on.

[2] We especially refer the reader to [Rab17] for more advanced (categorical) constructions with morphisms.

"Every model of the codomain gives rise to a model of the domain" Consider the formalization of natural numbers *Nat*, monoids *Monoid* and the fact that natural numbers form monoids in a canonical way, which is expressed by the morphism $\sigma\colon \mathit{Monoid} \rightsquigarrow \mathit{Nat}$:

```
1  theory Monoid =
2    U: type ▌
3    e: U ▌
4    op: U ⟶ U ⟶ U | # #1 ∘ #2 ▌
5    associative: ⊢ ∀ [a: U] ∀ [b: U] ∀ [c: U] (a ∘ b) ∘ c ≐ a ∘ (b ∘ c) ▌
6
7    neutral: U ⟶ prop | = [e'] ∀ [a: U] (a ∘ e' ≐ a) ∧ (e' ∘ a ≐ a) ▌
8    e_neutral: ⊢ neutral e ▌
9  ▌
10
11 theory Nat =
12   /T Natural numbers as usual with Peano axioms ▌
13
14   ℕ: type ▌
15   0: ℕ ▌
16   s: ℕ ⟶ ℕ ▌
17   plus: ℕ ⟶ ℕ ⟶ ℕ | = … | # #1 + #2 ▌
18
19   /T … ▌
20 ▌
21
22 view σ : Monoid -> Nat =
23   U = ℕ ▌
24   e = 0 ▌
25   op = plus ▌
26   associative = … ▌
27   e_neutral = … ▌
28 ▌
```

Indeed, the morphism refines the poorer, more general concept of monoids to the richer, more concrete concept of natural numbers. Precisely, the morphism tells us that every model of natural numbers also induces a model of monoids. Namely, let a model of natural numbers be given with interpretation function $[\![\cdot]\!]_{\mathbb{N}}$ over *Nat*-expressions. To induce a monoid model (along σ), we look at the assignments of the morphism. They

instruct us what the interpretation function $[\![\cdot]\!]_{Mon}$ of the monoid model shall do:

$$\begin{aligned}
[\![U]\!]_{Mon} &:= [\![\sigma(U)]\!]_{\mathbb{N}} &&= \mathbb{N} \\
[\![e]\!]_{Mon} &:= [\![\sigma(e)]\!]_{\mathbb{N}} &&= 0 \\
[\![op]\!]_{Mon} &:= [\![\sigma(op)]\!]_{\mathbb{N}} &&= + \\
[\![associative]\!]_{Mon} &:= [\![\sigma(associative)]\!]_{\mathbb{N}} &&= \dots \\
[\![e_neutral]\!]_{Mon} &:= [\![\sigma(e_neutral)]\!]_{\mathbb{N}} &&= \dots
\end{aligned}$$

This extends homomorphically to all *Monoid*-expressions and thus we have a total interpretation function. This can concisely be written as $[\![\cdot]\!]_{Mon} := [\![\cdot]\!]_{\mathbb{N}} \circ \bar{\sigma}$. Put in words, the monoid's interpretation function first uses the morphism $Monoid \rightsquigarrow Nat$ to rewrite to a *Nat*-expression and then employs the *Nat*-model we have at hand to find a fitting interpretation. Note that the result is in fact a valid monoid model since the interpretations of *associative* and *e_neutral* exactly represent proofs of the corresponding monoid axioms ensuring validity.

The same reasoning can be found in many hierarchies of (algebraic) structures. For example, the chain of statements "Hilbert spaces induce normed vector spaces, which induce metric spaces, which induce topological spaces" exactly translates to a refinement chain of morphisms in MMT:

$$TopologicalSpace \rightsquigarrow MetricSpace \rightsquigarrow NormedVectorSpace \rightsquigarrow HilbertSpace$$

On the left side, we see the poorest structure and conversely on the right side, we can find the richest structure. Walking the chain forwards means incarnating existing notions, e.g. continuity in topologies, in richer structures, e.g. $\varepsilon\delta$-definition in metric spaces. In contrast, walking backwards means generalizing statements. In fact, our principle AppGenpresented in Chapter 5 exactly does this by means of rewrite rules.

"Logic Translation" In the previous example, we focussed on the aspect of model induction by a morphism. Dually, we have the aspect of *translation*. Indeed, where we have translation on the formulae side of things, we have model induction on the model side of things. With the algebraic hierarchy, one usually stresses the latter viewpoint. Hence, we would like to present an example to stress the translational aspect. Namely, consider $\sigma : \mathcal{L} \rightsquigarrow \mathcal{L}'$ with $\mathcal{L} := PL$ (propositional logic) and $\mathcal{L}' := FOL$ (first-order logic), where the morphism embeds PL into FOL:

```
1  theory PL =
2    /T The type of propositions ▌
3    o: type ▌
4    ∧: o ⟶ o ⟶ o ▌
5    ¬: o ⟶ o ▌
6
7    /T To provide an easier example, we postulate the existence
8       of two variables ▌
```

```
  a: o ▮
  b: o ▮
▮

theory FOL =
  /T The type of individuals (and hence terms) ▮
  ι: type

  /T The type of propositions (and hence formulae) ▮
  o: type ▮

  ∧: o ⟶ o ⟶ o ▮
  ¬: o ⟶ o ▮
  ∀: (ι ⟶ o) ⟶ o ▮
  =: ι ⟶ ι ⟶ o ▮

  a: o ▮
  b: o ▮
▮

view σ : PL -> FOL =
  o = o ▮
  ∧ = ∧ ▮
  ¬ = ¬ ▮

  a = a ▮
  b = b ▮
▮
```

By means of the (homomorphic extension of the) morphism, we can now translate PL-sentences to FOL-sentences, e.g.

$$\overline{\sigma}\,(\neg_{PL}(\neg_{PL}a_{PL} \wedge_{PL} \neg_{PL}b_{PL})) = \neg_{FOL}(\neg_{FOL}a_{FOL} \wedge_{FOL} \neg_{FOL}b_{FOL})$$

where we indexed all constants with the theory they are coming from to ease reading. Thus we have translation (refinement) into a richer structure.

"Analogies as Non-Strict Refinements" Consider again the theory of monoids, but now with an endomorphism:

```
theory Monoid =
  U: type ▮
  e: U ▮
  op: U ⟶ U ⟶ U ▮ # #1 ∘ #2 ▮
```

```
5    associative: ⊢ ∀ [a: U] ∀ [b: U] ∀ [c: U] (a ∘ b) ∘ c ≐ a ∘ (b ∘ c) ▌
6
7    neutral: U ⟶ prop ▌ = [e'] ∀ [a: U] (a ∘ e' ≐ a) ∧ (e' ∘ a = a) ▌
8    e_neutral: ⊢ neutral e ▌
9  ▌
10
11 view σ : Monoid -> Monoid =
12   U = U ▌
13   e = e ▌
14   op = [a, b] b ∘ a ▌
15   associative = … ▌
16   e_neutral = … ▌
17 ▌
```

The morphism translates every monoid to its opposite monoid. We posit that many analogies can in fact be expressed by such a morphism in MMT. For example, a completely analogous construction with a formalization of categories allows precise definition and exploitation of dualities.

3.2. Theory- and Diagram-level Generalizations

Having seen that morphisms are specializations or dually generalization maps, we can directly state what it means to generalize a theory:

Definition 3.2.1. We call a theory G a **theory-level generalization** of a theory T if there is a specialization morphism $g\colon G \rightsquigarrow T$.

A **theory-level generalization principle** is a partial algorithm accepting T and outputting (G, g).

We have already seen representative examples in the previous section. For an example in a completely different spirit, we defer the reader to Chapter 5, where our running example generalizes a formal specification of a sorting algorithm.

Some generalization methods might have multiple theories and morphisms as input and/or have them as output. The principle of Theory Splitting, which we describe in Chapter 4, can disperse declarations of a single theory onto multiple theories (Figures 3.1a and 3.1b). In categorical tonus, these methods are acting on diagrams in the category $\mathbb{T}\mathrm{h}\mathrm{y}$ of theories and theory morphisms. To define a diagram-level generalization, we lift the specialization morphism from the previous definition to a specialization functor and require on top some commutation properties:

Definition 3.2.2. Let $\Delta_T : \mathcal{D}_T \to \mathbb{Thy}$ be a diagram. We call another diagram $\Delta_G : \mathcal{D}_G \to \mathbb{Thy}$ a **diagram-level generalization** if there is a specialization functor $F : \mathcal{D}_T \to \mathcal{D}_G$ and a natural transformation $\eta \colon D_T \to D_G \circ F$.

A **diagram-level generalization principle** is a partial algorithm accepting Δ_T and outputting $(\Delta_G, (F, \eta))$.

Let us describe the definition by viewing the just mentioned principle of Theory Splitting in light of it. Recall Figures 3.1a and 3.1b showing the initial single-theory and the refactored diagram. The functor is supposed to provide a scheme translation between the more general diagram Δ_G and the original diagram Δ_T. In our case of dispersed theories, the specialization functor identifies all theories again (Figure 3.1c).
However, this does not yet guarantee that the theories G_{1-4} are in fact a theory-level generalization of the theory T they are mapped to. This is why we require a natural transformation, which provides us two crucial aspects:

- On the one hand, it ensures what was said, i.e. it strengthens the mapping of the functor with providing morphisms, which are evidence for theory-level generalizations.

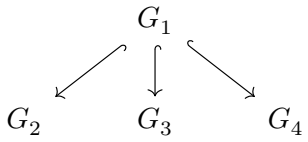
- On the other hand, being a natural transformation, it guarantees a strong commutation property, namely that going along those theory-level specializations and going along morphisms in the individual diagrams commutes.
  A specific instance of this property in our case can be seen in Figure 3.1d. Let us denote the morphisms from that figure by $g_{1,T} \colon G_1 \rightsquigarrow T$ and $g_{4,T} \colon G_4 \rightsquigarrow T$. Then we realize that the commutation is equivalent to $g_{4,T}|_{G_1} = g_{1,T}$. Put differently, the specializations *in our case* are limited insofar that they need to inherit specializations performed on "parental" theories.
  The general case of enforced commutation independent of Theory Splitting can be seen in Figure 3.2.

Unification of Concepts It might strike the reader that theory- and diagram-level generalizations share no syntactical resemblance in their definition. Therefore, we try to make it plausible that they indeed describe the same concept instantiated on different levels and start with a basic result:

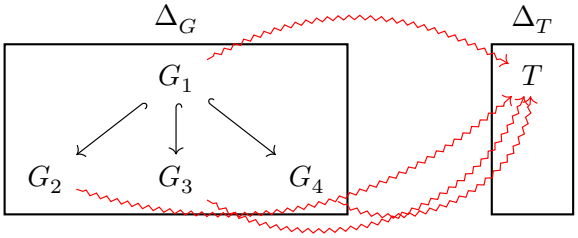
Lemma 3.2.1. *Let $g \colon G \rightsquigarrow T$ be a theory-level generalization. Then $\Delta_G := \{G, id_G\}$ is a diagram-level generalization of $\Delta_T := \{T, id_T\}$.*

Proof. Take the functor with $FG := T$, $Fid_G = id_T$ and the natural transformation with $\eta_G := g$. □

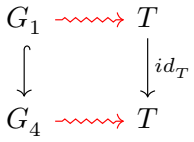
(a) Initial diagram Δ_T consisting of a single theory



(b) Refactored diagram Δ_G by dispersing T onto G_{1-4}



(c) Justification of the refactoring being a diagram-level generalization according to Definition 3.2.2. The red arrows both denote the mapping of the functor and the components of the natural transformation.



(d) One exemplary commutation property ensured by the natural transformation.

Figure 3.1.: Diagram-level generalizations for Theory Splitting

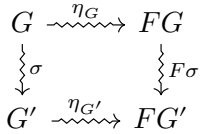


Figure 3.2.: Commutation property of the natural transformation for all $\sigma \in \Delta_G$

Since a formal relationship between both definitions is not needed in the rest of the thesis and for the sake of notational brevity, we switch to a very informal context to describe their "skeletal" equivalence. Overall, our strategy is to first inspect theory-level generalizations by unravelling the definition of a (well-typed) morphism. Then, we will extrapolate the gained skeleton to declarations and see that the definition of diagram-level generalizations we provided sensibly fits the family. Hence, let $g \colon G \nrightarrow T$ be a well-typed morphism, then g and as such the theory-level generalization along g consist of three ingredients [RK13]:

- a raw mapping: a mapping from G-symbols to T-expressions
- a well-typedness condition: especially for all constants $(c \colon E) \in G$ that $\vdash_T g(c) \colon \bar{g}(E)$
- a homomorphicity condition: the morphism must be a homomorphism wrt. definienses. Concretely, for every defined constants $(c \colon E = e) \in G$ we require $\vdash_S g(c) = \bar{g}(c) \colon \bar{g}(E)$

In a similar vein and just for the sake of this paragraph, we define *declaration-level generalizations* as being made up of

- a raw mapping: two declarations δ' and δ, where δ' is the supposed generalization,
- a well-typedness condition: $\vdash_T E' = E$, and
- an empty homomorphicity condition.

We can now identify the common skeleton, namely declaration- and theory-level generalizations both consist of a) a raw mapping of the next lower-level entities, b) the requirement of those mappings being generalizations in the next lower-level context, and c) a homomorphicity condition. Definition 3.2.2 of diagram-level generalizations indeed fits this pattern: The raw mapping on theories *and* morphisms is given by the functor. The generalization condition on morphisms is vacuous, while the one on theories is given by the components of the natural transformation. Finally, the homomorphicity condition can be taken as the very commutation requirements enforced by the natural transformation.

3.3. Behavior Preservation

Commonly, refactorings are defined as actions making code easier to understand while preserving behavior [Opd92; Fow+99]. So far we have defined the overall type of these actions in our setting, namely as generalizations, but have not given a definition of behavior preservation yet. Hence, in this section we strive to give such initial definitions for theory- and diagram-level generalizations. We note that they are likely to be subject to change in the future primarily because the authors devised them only very recently before submission deadline. Additionally, their applicability on further principles than the ones presented in this thesis needs to be evaluated.

Intuitively, we want to call a generalized theory G of T via some morphism $G \rightsquigarrow T$ behavior-preserving if no "information" is lost, but possibly only added. Furthermore, as we see our work particularly in the realm of mathematics, such information should especially include derivable theorems. Hence, a first attempt would require the *deduction closure* of T to be contained in G, i.e. G is behavior-preserving iff. all T-derivable-theorems are also derivable in G. However, in general we cannot even expect the statements of those T-theorems[3] to be well-formed G-expressions. For example, in Section 5.1 we will generalize the merge sort algorithm on lists over natural numbers (being T) to lists whose entries are sorted by an arbitrary total order (G). There we realize that the T-provable statement $\vdash$ `sort [1, 2, 3] ≐ [1, 2, 3]` cannot be expressed in G, simply due to the fact that G does neither include nor define natural numbers.

Fortunately, we can salvage our attempt while still retaining a notion similar to deduction closure. Recall that MMT morphisms are truth-preserving (cf. Corollary 2.1.0.1), i.e. the image of a theorem under a morphism is again a theorem. Especially, the proof is given as the image of the original proof. Now given a T-theorem, instead of requiring the potentially impossible rederivation in G, we require the existence of a preimage wrt. the considered specialization morphism[4]. Hence, by virtue of the morphism we can conclude the well-typedness and an approximate rederivability property of the T-theorem. Formally, we define this as follows:

Definition 3.3.1 ((Strong) Behavior Preservation)**.** Let T be a theory. A generalization (G, η) with $g\colon G \rightsquigarrow T$ is called behavior preserving iff.

$$T \subseteq \overline{g}(\texttt{Obj}(G))$$

where $\texttt{Obj}(G)$ denotes all closed and well-typed G-expressions and the subset relation is to be taken as follows: for every $(c : E[= e]) \in T^{\flat}$ either $c \in \overline{g}(\texttt{Obj}(G))$ or $e \in \overline{g}(\texttt{Obj}(G))$.[a]

[a] If the first condition does not apply, this especially means that c must have definiens e.

In a bold and sloppy way, the definiton can be stated as $T \subseteq g(G)$.

Example 3.3.1. Let $R \hookrightarrow S$. If S only contains defined constants, then R is a behavior-preserving generalization of R wrt. that inclusion. If on the other hand S contains at least one undefined constant, then we do not have behavior preservation. Indeed, this matches the intuition that defined constants do not provide any new information and thus can be safely dropped when behavior preservation one's only concern.

[3] Their type (propositions-as-types)

[4] Note that a generalization always encompasses three entities, namely G, T and some specialization morphism $G \rightsquigarrow T$.

We can conclude that the act of deleting constants from a theory is in general *not* behavior-preserving. Still, we imagine the usefulness of this action and list it in the next section under the name Theory Shrinking as a possible future generalization principle.

We believe that this definition is still too strong. Namely, there are cases where preimages wrt. g exist only up to some equational theory, i.e. instead of $T \subseteq \overline{g}(\texttt{Obj}(G))$ we should consider the requirement with some closure: $T \subseteq \mathrm{cl}(\overline{g}(\texttt{Obj}(G)))$. Further, we suppose that this closure shall encompass $\alpha\beta\eta$-equality, equality resulting from T-theorems, and proof irrelevance. We refer to Appendix A.1 for an example portraying groups axiomatized on the one hand via multiplication as usual and on the other hand via division. They are both behavior-preserving generalizations of each other precisely if one argues modulo the just mentioned equational theory. We remark that doing the naïve closure might allow for circular reasoning of behavior preservation. Therefore and for the reason that the principles in the present thesis are already behavior preserving in the strong sense, we defer further investigation to future work.

Finally, we extend the definition straightforwardly to diagram-level generalizations:

Definition 3.3.2 ((Strong) Behavior Preservation on Diagrams)**.** Let Δ_T be a diagram. A diagram-level generalization $(\Delta_G, (F, \eta))$ is called behavior-preserving iff. all theories in Δ_T can be covered by some theory-level behavior-preserving generalizations.

With the definition above, this formally means that for every $T \in \Delta_T$ we require the existence of theories $G_1, \ldots, G_n \in \Delta_G$[a] such that

$$T \subseteq \bigcup_i \overline{\eta_{G_i}}(\texttt{Obj}(G_i))$$

[a] where n may depend on T.

Concerning both definitions, for practical reasons it might be useful say that a theory (diagram) is only behavior-preserving generalization in subsets of another theory (diagram). For example, `Topology` is a behavior-preserving generalization of `MetricSpace` *not* overall, but in its continuity definition – again with some unspecified equational theory in mind.

3.4. Overview of Generalization Principles

Having defined what generalization principles are, we now shortly list some principles the authors have in mind in the table below. The two principles elaborated in the present thesis are highlighted with gray background color.

		specific to Foundation	Logic
Diagram-level	Theory Splitting		
Theory-level	App-Gen	×	
	Theory Shrinking		
	Symbol Splitting		
	Abstraction of Repeated Subterm Patterns	×	
Declaration-level	Identification of Bound Variables	×	×
	Increase/Decrease of a Function's Arity	×	
	Reordering of Existential Quantifiers	×	×

We structure the principles in two dimensions. First, we can group principles into the level they are working on, i.e. declarations, theories or diagrams. Second, we can inspect their dependence on specific foundations or logics. For example, App-Gen depends on lambda abstractions and is thus dependent on the LF foundation, however, it is not dependent on a specific logic or calculus of natural deduction. In contrast, to be able to identify bound variables, we must talk about $\forall$ quantifiers making us depend on specific logics.

4. Theory Splitting

As known from software engineering, cohesion within structuring elements in programming languages, such as classes, modules or namespaces, is a key quality measure of source code [Ing18]. We claim the same holds true for theories in knowledge representation systems and suggest a decomposition method for them, which happens to be a behavior-preserving diagram-level generalization. It takes a single theory, a partition on its declarations and then splits (reorganizes) the theory's declarations according to the partition to generate a bunch of new theories. Finally, the created diagram of new theories is output. Certainly, during the whole process we have to respect declaration interdependencies, especially theories in the output diagram need to be sensibly connected by inclusions. We will detail this discussion in this chapter, but let us first shortly outline why we consider Theory Splitting to be a useful refactoring principle.

We have already detailled in Chapter 2 on related work that actions to reorganize large classes are established refactorings in software engineering. This alone already motivates having a counterpart in formal knowledge representation systems. We can draw further motivation exemplarily from the fact that the popular HOL Light theorem prover [Har09], which is designed to be minimalistic wrt. MKM, has no structuring elements serving the role of theories.[1] Additionally, HOL Light has been exported to multiple other formal systems such as Coq, Hets and MMT [Wie02; MML07; KR14]. Thus, we could apply Theory Splitting on the exported MMT documents. In particular, this would be appealing if we had an automated way to intelligently perform our principle on large documents, which we defer to future work. Last but not least, this principle also serves as an exemplary behavior-preserving generalization principle to support our theoretical framework.

Structure of the Chapter Section 4.1 guides the reader through an example of splitting to the formal definition of Theory Splitting. Section 4.2 gives formal proofs on it outputting well-typed behavior-preserving generalized diagrams in terms of our overall framework.

[1] Having heard this claim in personal communications, we could not find a definite reference for it. However at time of writing, a strong indicator can be found in the files of the standard library, which bear no (computer-readable) structuring at all: https://github.com/jrh13/hol-light/tree/013324af7ff715346383fb963d323138cf011732

```
theory Monoid =
  U: type ▌
  e: U ▌
  op: U → U → U | # #1 ∘ #2 ▌
  associative: ⊢ ∀ [a: U] ∀ [b: U] ∀ [c: U] (a ∘ b) ∘ c ≐ a ∘ (b ∘ c) ▌

  neutral: U → prop | = [e'] ∀ [a: U] (a ∘ e' ≐ a) ∧ (e' ∘ a = a) ▌
  e_neutral: ⊢ neutral e ▌
  e_unique: ⊢ ∀ [e': U] (neutral e') ⇒ e' ≐ e
    | = sketch "omitted proof ▌

  power: U → ℕ → U | # #1 ^ #2 ▌
  thm_power: ⊢ ∀ [a: U] ∀ [n: ℕ] ∀ [m: ℕ]  a ^ (n + m) ≐ (a ^ n) ∘ (a ^ m)
    | = sketch "omitted proof" ▌

  /T … further (general) monoid declarations … ▌

  commutative: ⊢ ∀ [a: U] ∀ [b: U] a ∘ b ≐ b ∘ a ▌
  /T … further declarations (esp. theorems) based on commutative … ▌
▌
```

Figure 4.1.: A theory of commutative monoids to be split

4.1. Towards a Definition by Example

Consider a drafted formalization of commutative monoids *and* monoids in general in Figure 4.1. As already said, *valid* partitions on the declarations of a theory need to respect its declaration interdependencies. Hence, recall the dependency partial order $\leq_T$ on declarations we introduced in Definition 2.2.2. For example, we have $U \leq_T e$ and $op \leq_T associative$. It turns out that it is conceptionally easier to think in term of its equivalent graph representation instead. Indeed, we can express every partial order equivalently as a transitively reduced directed acyclic graph (DAG). The corresponding graph for our formalization is shown as part of Figure 4.2. Whenever we have an edge $\delta \to \delta'$, we have a dependency $\delta \leq_T \delta'$. We now want to give a partition on that graph which sensibly splits our theory of monoids.

A First Split To describe a first sensible (and valid) split, let us first note a few things about the graph. We drawed additional dummy dependencies in the graph which serve to hopefully illustrate the following property which we believe a truly representative graph for a theory of commutative and general monoids would have. Namely, it would have one subgraph for the theory of general monoids and one subgraph for commutative monoids only such that there are many edges from the former to the latter, but none

in the other way. Very similar to Extract Subclass from software engineering, this motivates moving the "child" of commutative monoids into its own theories. In other words, we should split the theory by exactly those two subgraphs into two new theories such that the theory for the commutative part would include the general theory. The code after the (yet informal) split is shown in Figure 4.3.

Note how we needed to account for an inclusion after the "raw splitting" as such. The underlying reason was that there was at least one dependency from a symbol within the commutative monoid part on a symbol within the general monoid part. Thus, the commutative part as a whole is dependent on the general part as a whole. We will make this precise below. Additionally, realize that it would be erroneous to have a dependency in the other way on top precisely for the reason that two theories in MMT cannot include each other.[2] This restriction will amount to a validity condition on partitions we require for spitting.

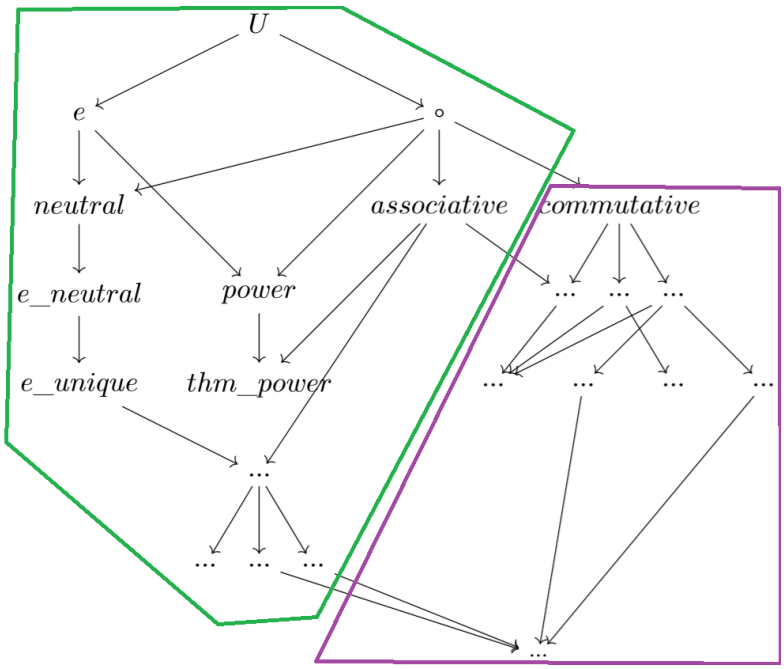


Figure 4.2.: Dependency DAG for the theory of monoids from Figure 4.1 with an overlaid suggested decomposition.

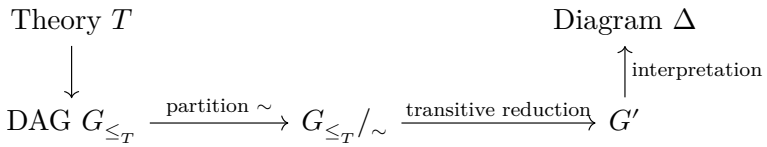
The Definitional Pipeline We now give a precise definition for Theory Splitting. Overall, it will encompass the following pipeline visualizing our approach, which we

[2]While this was a sensible design choice for MMT with inclusions, other encapsulation notions in other systems might call for a relaxation of the cyclefreeness requirement. For example, in C++ a namespace can be spread over multiple interdependent files, given that one uses forward declarations for such interdependencies.

```
22 theory GeneralMonoid =
23   U: type ▌
24   e: U ▌
25   op: U → U → U | # #1 ∘ #2 ▌
26   associative: ⊢ ∀ [a: U] ∀ [b: U] ∀ [c: U] (a ∘ b) ∘ c ≐ a ∘ (b ∘ c) ▌
27
28   neutral: U → prop | = [e'] ∀ [a: U] (a ∘ e' ≐ a) ∧ (e' ∘ a = a) ▌
29   e_neutral: ⊢ neutral e ▌
30   e_unique: ⊢ ∀ [e': U] (neutral e') ⇒ e' ≐ e
31     | = sketch "omitted proof ▌
32
33   power: U → ℕ → U | # #1 ^ #2 ▌
34   thm_power: ⊢ ∀ [a: U] ∀ [n: ℕ] ∀ [m: ℕ] a ^ (n + m) ≐ (a ^ n) ∘ (a ^ m)
35     | = sketch "omitted proof" ▌
36
37   /T … further (general) monoid declarations … ▌
38 ▌
39
40 theory CommutativeMonoid =
41   include ?GeneralMonoid ▌
42
43   commutative: ⊢ ∀ [a: U] ∀ [b: U] a ∘ b ≐ b ∘ a ▌
44   /T … further declarations (esp. theorems) based on commutative … ▌
45 ▌
```

Figure 4.3.: Split theories of general and commutative monoids

already informally traversed in the previous paragraphs. Concretely, the interpretation amounts to adding those inclusions we discussed above. Every step will be given a formal definition below.



We start by formally defining the

Definition 4.1.1 (Dependency DAG)**.** For a theory T, let $G_{\leq_T}$ be the reflexive-transitively reduced graph representation of $\leq_T$. By convention, an edge $\delta \to \delta'$ stands for the dependency $\delta \leq_T \delta'$ in that "forward" direction.

Having the dependency graph, we can now apply a partition:

Definition 4.1.2 (Quotient Graphs & Cyclefreeness)**.** Let $G = (V, E)$ be a graph and a partition on V be given, inducing the equivalence relation $\sim$. We define the **quotient graph** $G/_\sim$ as consisting of

- vertex set $V' := V/_\sim$, and
- edge set $E' := \{([u]_\sim, [v]_\sim) \mid (u, v) \in E\}$.

We call a partition for a graph **cyclefree** iff. the resulting quotient graph is cyclefree (except for trivial self-cycles).

We summarize the last two steps directly in the definition of the principle itself:

Definition 4.1.3 (Theory Splitting)**.**

- **Input:** A theory T and a cyclefree partition represented by its equivalence $\sim$
- **Action:** Build the reflexive-transitively reduced version of $G_{\leq_T}/_\sim$ and then interpret it as a diagram in $\mathbb{Thy}$ as follows:

  Every vertex is interpreted as an anonymous theory with the collected declarations as its contents together with inclusions as follows: for every direct predecessor vertex, include the interpretation thereof.
- **Output:**
  - The interpreted diagram

– A specialization functor detailled below in Lemma 4.2.2

The reflexive reduction of $G_{\leq_T}/_{\sim}$ is supposed to prevent self-inclusions in the interpretation afterwards. To see the desire for transitive reduction as well, again consider our monoid formalization with its DAG in Figure 4.2. Another sensible, finer split is depicted in Figure 4.4. It splits into a core theory of monoids, a theory of general monoid facts building on top, and a theory of commutative monoids. In that case, we would build theories `CoreMonoid`, `GeneralMonoid`, and `CommutativeMonoid`. Now without the transitive reduction of $G_{\leq_T}/_{\sim}$, we would have inclusions of both `CoreMonoid` *and* `GeneralMonoid` in `CommutativeMonoid`. While we have them both anyway as $\mathbb{T}\mathrm{hy}$ as a category respects composition, the former is redundant in concrete formalization code. Hence, transitive reduction is supposed to throw it away.

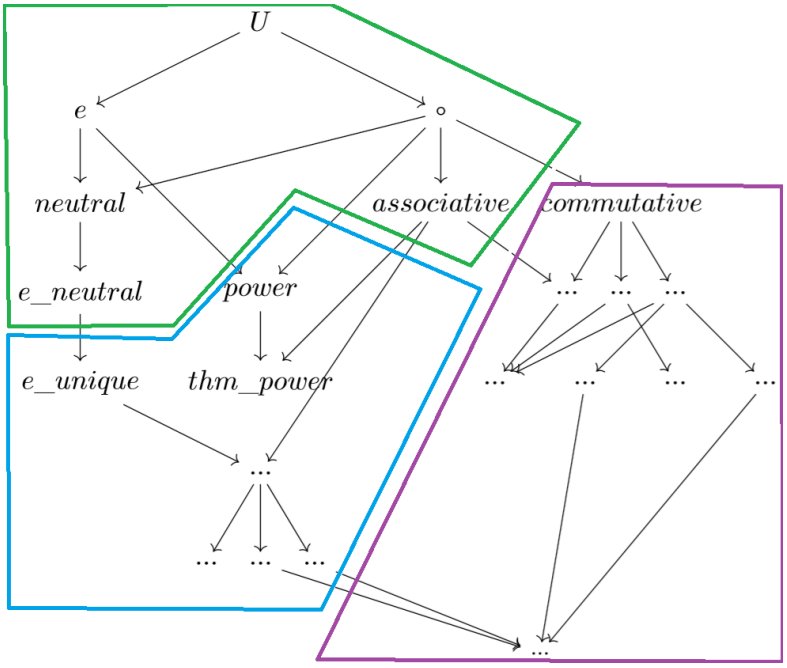


Figure 4.4.: An alternative, finer split for the theory of commutative monoids

Extreme Cases of Splitting There are two extreme cases of Theory Splitting:

(i) Empty split: By means of the "full" equivalence relation on the declarations of a theory T, i.e. $\delta \sim \delta'$ for all $\delta, \delta' \in T$, Theory Splitting only produces one output theory, namely T itself. Thus, the splitting was actually a *no-op* and thus useless.

(ii) Full split: By means of the smallest possible equivalence relation, i.e. the diagonal, Theory Splitting disperses the declarations to individual singleton theories[3]

[3]Apart from inclusions induced by interpretation, these theories only contain one declaration.

> Although there are systems advocating the use of such *tiny theories* [Car+11], this approach hinders usability since the theory graph is cluttered with theories, which are mostly uninteresting to a mathematician. For instance, imagine the axioms of a vectorspace being separated into, say, ≥ 8 tiny theories.

Hence, splits need either be selected manually by a human or good heuristics need to be developed accounting for careful splitting strategies. We leave the investigation of the latter, especially in the realm of MKM, to future work and refer to Section 2.3 having shown related work in the area of software engineering.

4.2. Validity and Behavior Preservation

In this section we prove that Theory Splitting is in fact a behavior-preserving generalization principle within our framework. First, let us convince ourselves that the output diagram is in fact well-typed:

Lemma 4.2.1 (Validity of Theory Splitting)**.** *Let T be a theory and a cyclefree partition be given. The corresponding theory split produces a well-typed diagram.*

Proof. Since we start with a well-typed theory by convention, it suffices to show

(i) The resulting diagram contains no cycles.

(ii) For every theory R in the diagram, every contained declaration $\delta \in R$ and every dependency $\delta' <_T \delta$:
δ' is "accessible" (in scope) at the location of the final placement of δ. That is, $\delta' \in R$ or it stems from a transitively included theory in R.

ad (i): Per assumption of cyclefreeness of the partition.
ad (ii): By the very notion of quotiening, there are vertices u, v of the final theory split, so that $\delta' \in u$ and $\delta \in v$, where R is the theory interpretation of v. In case of $u = v$ we are done, in case of $u \neq v$ we have the edge $u \to v$ retained from the initial dependence. Hence R includes the theory interpretation of u. □

In Chapter 3 we have already motivated diagram-level generalizations by the principle of Theory Splitting. Recall Figure 3.1c presenting a split diagram on the left and identifying its theories to the original theory T. Let us now formally state and prove this generalization proprety:

Lemma 4.2.2 (Theory Splitting being a Generalization)**.** *Let T be a theory and a cyclefree partition be given. The output theory-split diagram Δ_G is a diagram-level generalization of the trivial diagram Δ_T containing T.*

Proof. Use the constant functor $F : \Delta_G \to \Delta_T$ with $FG := T$ and the "embedding-like" natural transformation given by $(\eta_G : G \rightsquigarrow T)_{G \in \Delta_G}$, which sends every declaration $\delta \in G$ to its original counterpart $\text{"}\delta\text{"} \in T$. Note that those are formally not the same declaration, but rather copies of each other, which we denote on the present meta level with the same symbol. □

Having convined ourselves of Theory Splitting outputting well-typed generalized diagrams, we can also assert its behavior preservation, which is due to the fact that no declarations are thrown away. Hence, the morphism-induced closure is unaffected.

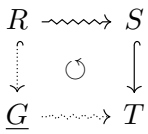
Lemma 4.2.3 (Behavior Preservation of Theory Splitting)**.** *Let T be a theory and a cyclefree partition be given. The output theory-split diagram Δ_G is behavior-preserving generalization of the initial trivial diagram Δ_T containing T wrt. to the functor and natural transformation given in Lemma 4.2.2.*

Proof. Every declaration of T is covered by (at least[4]) one theory in Δ_G, whose image under the specialization morphism from the natural transformation then covers the declaration itself in T as well. □

[4] In the theory it appears the first time and in all theories including that theory

5. App-Gen: Application of Existing Generalization

In this chapter we explore how existing (theory-level) generalizations can be exploited to come up with new generalizations. Suppose we have a theory S and a more general theory R witnessed by a morphism $R \nrightarrow S$. Let T be a theory using S by means of an inclusion $S \hookrightarrow T$.[1] The central question now is: Can we find a generalization G of T such that G now includes R instead of S? Pictorially, we can phrase this as follows:



Throughout the chapter such diagrams are assumed to commute. Additionally, by means of underlined objects and dotted morphisms we indicate that those entities are created from the remaining ones by the very action we consider in the specific context.

Put differently, we would like to investigate when and how an inclusion can be generalized and how declarations can be most sensibly carried over. Of course, the problem admits trivial solutions by choosing $G \in \{R, S, T\}$. We leave defining the precise problem statement ruling out such unwanted solutions to future work.[2] Instead, we derive a syntatical generalization algorithm App-Gen which tries to compute G by going over every declaration in T.

We consider the action of creating G to be a refactoring for two reasons. First, having a more general theory and its linkage to the old concrete one widens the induced knowledge space, which allows (proof) reuse and enhances data analytics operations from the Mathematical Knowledge Management discipline. Second, the generalization as such can be useful for the user. For example, G might contain more general mathematical identities, which followed by abstracting the dependence on S-expressions in T to corresponding R-expressions. Alternatively, in the realm of formal specifications with program extraction, T might be the formalization of a particular library class whose generalization G can be imagined as a "generified/templated" version of T.

Apart from the refactoring motive, there are two more reasons we can imagine. On the one hand, in some circumstances, theorems, which are known to be provable in the more general R-setting, can be more easily proven in the more concrete, and hence possibly

[1] Other possibilities of S being used are nested theories, where the inner theory T accesses the outer S, and implicit morphisms $S \nrightarrow T$.

[2] Possibly, the problem can be formulated with categorical semantics, which we will hint at in Section 5.4.

more comfortable S-setting. If the user now formalizes their proof in T, chances are that the proof can be generalized by App-Gen to a proof of the more general theorem.[3] On the other hand, playing with the notions of (syntactic) generalizability could be a learning resource for students. We detail our early ideas in that direction in Chapter 7.

Structure of the Remaining Chapter Section 5.1 introduces our running example and guides the reader from the conceptual notion of applying a generalization up to a first formal definition of such an algorithm. Section 5.2 motivates a revision thereof to allow for more complex scenarios. Some additional examples are presented in Section 5.3, after which Section 5.4 concludes the chapter with showing a possible interpretation of our principle from a categorical point of view and showcasing other related categorical constructions.

5.1. App-Gen-0: A First Version

To introduce our running example and to explain what we mean by "application of existing generalization", we start off with a made-up quote describing both things on their conceptual level:

Runninng Example (Conceptual)

"Conceptionally, merge sort on lists of natural numbers generalizes to merge sort on lists on arbitrary totally ordered sets (tosets)."

We expect the reader to comprehend the example from a generic mathematician's or computer scientist's point of view, and to maybe even have a few ideas on which properties of the merge sort algorithm on $\mathbb{N}$ are responsible for making the above statement true. With these assumptions, we devote this section to examining the quote and translating every conceptual idea to a rigid formal definition within the MMT framework. At the end of this section, the reader should have an idea how such implicit generalizability *and* generalization can be made explicit in terms of knowledge representation systems.

First we note that the quote asserts generalizability without making the act of generalization explicit. We will see that it suffices to understand the latter to acknowledge the former. Hence, as a first approximation, let us structure the action happening behind the scenes on an informal level, see the list below. In every step we distinguish the abstract action from its incarnation in the concrete, albeit still conceptual merge sort setting.

1. Given a concept T, look for ingredients of it,
   T being merge sort on lists of natural numbers
2. and choose one ingredient S.
   S being the entry type (natural numbers)

[3] We refer to the example of sequences and nets in Section 5.3.

3. Look at possible generalizations of S,
   Group, countable set, toset, ...
4. choose one generalization R,
   R being natural numbers forming a totally ordered set wrt. their standard ordering
5. and try generalizing T wrt. R to form the new concept G.
   merge sort on lists of totally ordered entries

In other words we might say that in step 5 we *apply the existing generalization* from step 4 to the concept chosen in step 1. Concretely, we apply the generalization that natural numbers form a toset in the obvious way, to the concept of merge sort on lists of natural numbers. This viewpoint gave App-Gen its name.

Before we try to formalize the steps in MMT tonus, let us emphasize one crucial aspect: The generalization in step 4 encompasses the more general concept R *and* the instantiation to get S again. Indeed, the natural numbers admit specializations of tosets in various ways, the standard ordering $\leq$ is just the most common one. Hence, it is important to state the more general concept *together* with the way it is incarnated in the concrete concept. Likewise, the newly created concept G from step 5 refers to itself and a specific instantiation as well.

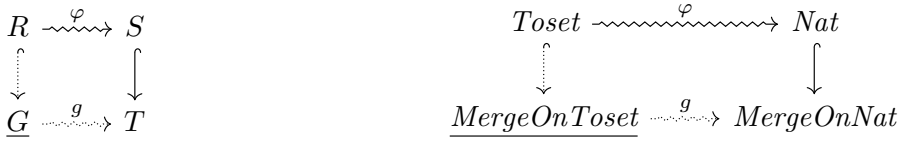
Let us now translate the above list to MMT: concepts become theories, ingredients become inclusions and generalizations are represented by morphisms (cf. Chapter 3), thus especially as a triple consisting of their domain, their codomain and their actual mapping:

1. Given a theory T, look for included theories,
   $T = \mathit{MergeOnNat}$
2. and choose one (transitively) included theory S.
   $S = \mathit{Nat}$
3. Look at incoming morphisms to S,
   $\mathit{Group} \rightsquigarrow \mathit{Nat}$, $\mathit{CountableSet} \rightsquigarrow \mathit{Nat}$, $\mathit{Toset} \rightsquigarrow \mathit{Nat}$, ...
4. choose one morphism $\varphi\colon R \rightsquigarrow S$
   $\varphi\colon \mathit{Toset} \rightsquigarrow \mathit{Nat}$ with $\varphi(X) = \mathbb{N}$, $\varphi(\leq_x) = \leq$
5. and try generalizing T wrt. φ to form the new theory G including T together with a morphism $g\colon G \rightsquigarrow T$.
   $G = \mathit{MergeOnToset}$

We will introduce the theories and morphisms mentioned in the list above on our way developing App-Gen. The avid reader might peek at corresponding formalizations in MMT surface syntax in Figures 5.2a and 5.2b.

Recall that we assume generalization principles be triggered manually by users and leave automatic batch processing to future work (cf. Chapter 7). Indeed, the steps, or rather inputs, 1 – 4 are exactly the inputs our principle App-Gen requires. However, as step 5 on its own can already be a highly complicated task for a mathematician on the conceptual level, we restrict ourselves to a manageable subset of purely syntactical generalizations. Fortunately, as we will see, the desired merge sort generalization is purely syntactical. We hope the reader gets an intuition about the manageable subset by reading this and the next section. By sticking to the running example, the remaining part of this section develops an initial algorithm for step 5.

Step 5 of Generalization In Detail Let us reiterate the current task we are trying to solve. Given theories R, S and T with $S \hookrightarrow T$ and a morphism $\varphi\colon R \nrightarrow S$, we seek a generalization $g\colon G \nrightarrow T$ such that $R \hookrightarrow G$ and the commutation depicted in Figure 5.1a holds. In terms of the running example, we are seeking *MergeOnToset* as in Figure 5.1b.



(a) The current abstract setting

(b) The current concrete setting

Figure 5.1.: The setting of the current task, both in abstract terms and in terms of the running example

Let us perform the act of generalization on concrete formalizations of the above entities and then extrapolate a general algorithm. For that consider the formalizations in Figures 5.2a and 5.2b. By close inspection, it is easy to verify that the following does the job of creating a well-typed generalized theory *MergeOnToset*

1. Copy the theory *MergeOnNat* to a new theory *MergeOnToset*,
2. replace the inclusion of Nat by an inclusion of *Toset*,
3. and finally change every occurrence of $\mathbb{N}$ and $\leq$ to X and $\leq_x$, respectively.

We realize the replacements in step 3 are given by the very morphism mapping φ read backwards (cf. Figure 5.2b). Indeed, the meaning of $X \mapsto \mathbb{N}$ via φ is that in the specific case of natural numbers, the toset is given by $\mathbb{N}$. Thus, to no longer rely on that specific case, we shall simply replace $\mathbb{N}$ by X. Analogous considerations hold for $\leq_x \mapsto \leq$. Indeed, such *link inversion* is typically the right thing to do for generalizing theories as examples in Section 5.3 will make plausible as well.

To bridge to the formal world, we modify two things of the above 3-step-procedure to enable an easier framework. First, we favor an iterative procedure over step 1, which produces the generalized theory declaration-by-declaration and component-by-component.

```
9  theory Nat =
10   ℕ: type ▌
11   zero: ℕ ▌
12   succ: ℕ ▌
13   add: ℕ ⟶ ℕ ⟶ ℕ ▌ = … ▌
14   ≤: ℕ ⟶ ℕ ⟶ prop ▌ = …▌
15 ▌
16
17 theory MergeOnNat =
18   include ?Nat ▌
19   include ?PolymorphicLists ▌ /T Some imaginary theory of finite lists ▌
20
21   /T Merge two pre-sorted lists ▌
22   merge: List ℕ ⟶ List ℕ ⟶ List ℕ ▌
23
24   /T Axioms on merge (≈ inductive definition) ▌
25   merge_base_case1: ⊦ ∀[l: List ℕ] merge nil l ≐ l ▌
26   merge_base_case2: ⊦ ∀[l: List ℕ] merge l nil ≐ l ▌
27   merge_ind_step: ⊦ ∀[x: ℕ] ∀[y: ℕ] ∀[xs: List ℕ] ∀[ys: List ℕ]
28     merge (cons x xs) (cons y ys) ≐
29       if (x ≤ y) cons x (merge xs (cons y ys))
30       else cons y (merge (cons x xs) ys) ▌
31
32   /T Sort a list ▌
33   mergesort: List ℕ ⟶ List ℕ ▌
34   mergesort_base_case: ⊦ mergesort nil ≐ nil ▌
35   mergesort_ind_step: ⊦ ∀[l: List ℕ] ∀[k: List ℕ]
36     mergesort (l ::: k) ≐ merge (mergesort l) (mergesort k) ▌
37 ▌
```

(a) Natural numbers and merge sort on lists over them (apparent by line 29)

Figure 5.2.: The running example: formalization of natural numbers, merge sort on lists over them, and tosets

```
39 theory Toset =
40   X: type ▌
41   ≤ₓ: X ⟶ X ⟶ prop ▌
42
43   ax_antisymmetry: ⊦ … ▌
44   ax_transivitivty: ⊦ … ▌
45   ax_connexity: ⊦ … ▌
46 ▌
47
48 view φ : Toset -> Nat =
49   X = ℕ ▌
50   ≤ₓ = ≤ ▌
51
52   /T Proofs of those properties being fulfilled in Nat ▌
53   ax_antisymmetry = … ▌
54   ax_transitivity = … ▌
55   ax_connexity = … ▌
56 ▌
```

(b) Tosets and their incarnation as natural numbrers expressing the fact that tosets are a generalization of natural numbers wrt. their standard ordering

Figure 5.2.: The running example: formalization of natural numbers, merge sort on lists over them, and tosets (cont.)

Second, in step 3 we instead consider rewrite rules on terms given by reading the morphism mapping backwards. This finally leads to our first version of our "application of existing generalization" (partial) algorithm, App-Gen-0, in Definition 5.1.1 below.

Definition 5.1.1 (App-Gen-0: A First Version)**.** Let the theories R, S, T, and the morphism $\varphi\colon R \nrightarrow S$ be given. Build G and $g\colon G \nrightarrow T$ as follows:

$$\begin{array}{ccc} R & \overset{\varphi}{\rightsquigarrow} & S \\ \vdots & & \downarrow \\ G & \overset{g}{\dashrightarrow} & T \end{array}$$

1. Set $G_0 := \{\text{include } R\}$, $g_0 := (\text{include } R \mapsto \varphi)$
2. In narrative order for every declaration $\delta \in T$ do the following:
   - In case of an inclusion `include U`: skip if $U = S$, else adopt unchanged.
   - In case of a constant declaration, try rewriting its type and possibly definiens component using the following rules until it is an G_{i-1}-expression:
     - For every $(c := c') \in \varphi$:
     $$\frac{c'}{c} \tag{5.1}$$
     - For every successfully rewritten $(d' : D'[= \hat{d}]) \in G_{i-1}$ of $(d : D[= \hat{d}]) \in T$:
     $$\frac{d}{d'} \tag{5.2}$$

   Let δ' denote the fully rewritten declaration with all its components, then
   a) set $G_i := G_{i-1} \cup \delta'$,
   b) and set $g_i := g_{i-1} \cup (\delta' \mapsto \delta)$,

Unfortunately, even if App-Gen-0 successfully rewrites every declaration, it might happen that the generalized theory is not well-typed. We leave finding suitable conditions ensuring well-typedness to future work. However, once having well-typedness of the generated theory, we can conclude more properties:

Lemma 5.1.1 (Validity of App-Gen-0 under Assumption of Well-Typedness)**.** *If the theory G generated by* App-Gen-0 *is well-typed, then the morphism $g\colon G \nrightarrow T$ is well-typed as well and the square given in the algorithm definition commutes.*

Proof. The well-typedness of the morphism directly follows from the rewrite procedure as such, and the commutation is a direct result of the assignment $g(\text{include } R) = \varphi$. □

Corollary 5.1.0.1 (Behavior Preservation of App-Gen-0)**.** *If the theory G generated by* App-Gen-0 *is well-typed, then (G, g) is a behavior-preserving generalization of T.*

Proof. By construction, every T-declaration is in the image of well-typed g. □

Remark 5.1.1 (Termination and Rewrite Problems). In step 2 of App-Gen-0 we can face non-terminating behavior due to oscillating rewrite rules and/or the problem that declarations simply cannot be rewritten to an G_{i-1}-expression. We address these issues in Section 6.2.3 on our implementations. In particular, we give a termination criterion there.

Observe that in case of our merge sort on lists of natural numbers, the algorithm is able to successfully produce the expected merge sort on tosets, cf. the resulting code in Appendix A.2.

We conclude with a bit of terminology we already made use of, but which can now be stated formally.

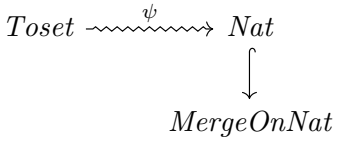
Definition 5.1.2 (App-Gen Terminology)**.** The following terminology applies to App-Gen-0 and derivatives we develop in next sections:

- An expression or declaration is called **generalizable** wrt. App-Gen if it survives step 2 of the algorithm above.
- By saying we **generalize an expression, declaration or theory T along $\varphi\colon R \nrightarrow S$** we mean the application of App-Gen on R, S, and T. In case of expressions, we assume the surrounding theory T to be evident from the context.

5.2. App-Gen-1: Generalizing App-Gen-0

In this section we show that even slight modifications of our running example's setting make the derived generalization algorithm fail. Consequently, we tackle a specific subset of such problems by making the algorithm aware of more rewrite rules.

Imagine we wanted to still generalize *MergeOnNat*, but now wrt. *Nat* forming a different toset, namely with the *reversed* standard ordering. That is, in the merge sort code from Figure 5.2a the expression `if (x ≤ y)` shall generalize to `if (y ≤ₓ x)`. Employing App-Gen-0, an obvious first try would be to modify the specialization morphism along we generalize such that we still have $X \mapsto \mathbb{N}$, but now $\leq_x \mapsto \geq$. For didactic reasons let us for a moment assume that *Nat* neither has the $\geq$ symbol nor



(a) The modified setting with the specialization morphism ψ now being more complex than a renaming. The theories are all taken from Figures 5.2a and 5.2b.

$$\frac{\mathbb{N}}{X} \qquad \frac{\lambda a, b : \mathbb{N}.\ \ b \leq a}{\leq_x}$$

(b) The initial set of rewrite rules given by ψ and Definition 5.1.1

```
28 merge (cons x xs) (cons y ys) ≐
29   if (x ≤ y) cons x (merge xs (cons y ys))
30   else cons y (merge (cons x xs) ys)
```

(c) Excerpt from the code in Figure 5.2a which cannot be rewritten to a declaration in context of *Toset*: No rewrite rule is applicable on $x \leq y$ and thus the dependency on $\leq$ remains.

Figure 5.3.: Modified setting of the running example where insufficient rewrite rules make App-Gen-0 fail on the shown code snippet

that it may be extended, then we would be forced to reflect the order flipping in that morphism as follows:

$$\psi\colon \mathit{Toset} \rightsquigarrow \mathit{Nat} \qquad \begin{array}{l} X \mapsto \mathbb{N} \\ \leq_x \mapsto \lambda a, b : \mathbb{N}.\ \ b \leq a \end{array}$$ [4]

For reference, Figure 5.3a shows the complete modified setting. Let us try App-Gen-0 from Definition 5.1.1 on it. The initial and relevant set of rewrite rules it uses is shown in Figure 5.3b. Evidently, they alone are unable to rewrite the expression $x \leq y$ occurring in *MergeOnNat* excerpted in Figure 5.3c. Thus, the surrounding declaration of that expression cannot be "abstracted away from its dependence on *Nat*" and is thus not generalizable with App-Gen-0. Hence, the algorithm *fails* in the modified setting.

Observe how the problem would not occur if we used a β-expanded form in the snippet exactly matching the $\psi(\leq_x)$:

```
1 if (x ≤ y) …                          /T Previously
2
3 if (([a: ℕ] [b: ℕ] b ≤ a) y x) …   /T Now
```

Since then we could apply the second rewrite rule from Figure 5.3b to get `if (≤ₓ y x)`,

[4] Note that for reasons of conciseness we employ the usual λ abstraction notation in the text, whereas we use the actually correct square bracket notation […] … solely in MMT surface syntax.

which is modulo notation[5] equivalent to the desired `if (y ≤ₓ x)`. Of course, requiring users to manually β-expand their terms is undesired and would partly defeat the whole purpose of automated generalization. Therefore, we tackle this problem from the other side and let the algorithm account for it.

Stepping back and ignoring the part on β-expansion, we see that $x \leq y$ is nothing else than an "instantiation" of the RHS of the morphism's assignment

$$\leq_x \mapsto \lambda a, b : \mathbb{N}.\ \ b \leq a$$

with (ordered) arguments y and x. Simultaneously, the desired rewritten form $y \leq_x x$ is an instantiation of the LHS of that assignment, namely $\leq_x$ instantiated with the *same* arguments in the *same* order. Thus, we realize that in addition to the previously generated rules of the form in Figure 5.4a, we now need rules of the form shown in Figure 5.4b as well.

$$\frac{c'}{c} \qquad\qquad \frac{\text{instantiation of } c' \text{ with arguments } \sigma}{\text{instantiation of } c \text{ with arguments } \sigma}$$

(a) Existing rule scheme (b) Desired new rule scheme

Figure 5.4.: The existing rule scheme next to the desired new rule scheme (conceptual sketch). They both are generated for all assignments $(c := c')$ in the specialization morphism.

On the formal level, this translates to the following: let ψ denote the specialization morphism, then for every $(c := \lambda x_1, \ldots, x_n.s) \in \psi$ we introduce the rewrite rule

$$\frac{s\sigma}{c\ \sigma(x_1) \ldots \sigma(x_n)} \quad \mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}.$$

In the example, we have $(\leq_x := \lambda a, b : \mathbb{N}.\ \ b \leq a) \in \psi$, hence generate the following rule:

$$\frac{(b \leq a)\sigma}{\leq_x\ \sigma(a)\ \sigma(b)} \quad \mathrm{dom}(\sigma) = \{a, b\} \tag{$*$}$$

which we can equivalently express as follows to see that it exactly fits our purpose:

$$\begin{aligned} &\Leftrightarrow \quad \frac{t_1 \leq t_2}{\leq_x\ t_2\ t_1} \\ &\Leftrightarrow \quad \frac{t_1 \leq t_2}{t_2 \leq_x t_1} \end{aligned} \tag{$**$}$$

[5] To be precise, modulo notation on the level of typesetting listings in the present thesis, *not* in terms of the MMT system's data structures.

By means of the last rule (∗∗), the previously problematic term $t = x \leq y$ can now be rewritten to $y \leq_x x$, which was the desired outcome. Arguing with the first rule (∗) we would consider t an instantiation of $\lambda a, b : \mathbb{N}.\ b \leq a$ via $\sigma = [a \mapsto y, b \mapsto x]$.

For reference, we explicitly state the second revision of the algorithm, which we developed in this section, together with assertions on its validity and behavior preservation:

Definition 5.2.1 (App-Gen-1: App-Gen-0 + surface β-awareness)**.** Let the theories R, S, T, and the morphism $\varphi\colon R \nrightarrow S$ be given. Build G and $g\colon G \nrightarrow T$ as follows:

$$\begin{array}{ccc} R & \overset{\varphi}{\rightsquigarrow} & S \\ \vdots\uparrow & & \downarrow\uparrow \\ \underline{G} & \overset{g}{\dashrightarrow} & T \end{array}$$

1. Set $G_0 := \{\text{include } R\}$, $g_0 := (\text{include } R \mapsto \varphi)$
2. In narrative order for every declaration $\delta \in T$ do the following:
   - In case of an inclusion `include U`: skip if $U = S$, else adopt unchanged.
   - In case of a constant declaration, try rewriting its type and possibly definiens component using the following rules until it is an G_{i-1}-expression:
     - For every $(c := c') \in \varphi$:
     $$\frac{c'}{c} \tag{5.3}$$
     - For every $(c := \lambda x_1, \ldots, x_n.s) \in \varphi$:
     $$\frac{s\sigma}{c\ \sigma(x_1) \ldots \sigma(x_n)} \quad \mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\} \tag{5.4}$$
     - For every successfully rewritten $(d' : D'[= \hat{d}]) \in G_{i-1}$ of $(d : D[= \hat{d}]) \in T$:
     $$\frac{d}{d'} \tag{5.5}$$

   Let δ' denote the fully rewritten declaration with all its components, then
   a) set $G_i := G_{i-1} \cup \delta'$,
   b) and set $g_i := g_{i-1} \cup (\delta' \mapsto \delta)$,

Again, we can state and prove properties completely analogous to the ones for App-Gen-0:

Lemma 5.2.1 (Validity of App-Gen-1 under Assumption of Well-Typedness)**.** *If the theory G generated by* App-Gen-1 *is well-typed, then the morphism $g\colon G \rightsquigarrow T$ is well-typed as well and the square given in the algorithm definition commutes.*

Corollary 5.2.0.1 (Behavior Preservation of App-Gen-1)**.** *If the theory G generated by* App-Gen-1 *is well-typed, then (G, g) is a behavior-preserving generalization of T.*

Remark 5.2.1. Rewrite scheme 5.4 from above can be understood as a guided form of β-expansion of T-terms.

More than guided β-expansion Recall that at the very beginning of this section, we wanted to generalize along the morphism $\psi\colon \mathit{Toset} \rightsquigarrow \mathit{Nat}$ with $\psi(X) = \mathbb{N}$ and $\psi(\leq_x) = \geq$. Then, we assumed without good reason that $\geq$ is not a symbol in *Nat*, which brought us to using a lambda expression to swap the arguments. Now we state the actual reason in hindsight for didactical reasons.

Let ψ be the morphism as just defined. Then App-Gen-1 would *fail* rewriting $x \leq y$ in *MergeOnNat*. To see this in detail, we have to think about how $\geq$ can be defined in *Nat*, when $\leq$ is already assumed to exist. Indeed sensible formalizations need to link $\leq$ and $\geq$ in one way or another. Some possibilies are shown in Figure 5.5. In the following, we shortly discuss why rewriting fails in every case and how the algorithm could be recovered – at least in theory. Note that in every case we have the same specialization morphism in mind along we generalize, namely ψ as defined above.

The first shown alternative is perceivedly equivalent to having used the morphism assignment $\leq_x \mapsto \lambda a, b : \mathbb{N}.\ b \leq a$ to begin with, which would be fine with App-Gen-1. However, in the eyes of our rule schemes this is different from using $\psi(\leq_x) = \geq$ since they only look at the "raw" RHS of assignments without going further with δ-expansion. Indeed, if they did so, we would be able to successfully rewrite the problematic term.

For the remaining alternatives we note that δ-expansion is unsatisfactory since $\geq$ does not even have a definiens in those cases. Accounting for the remaining alternatives would thus need to consider the equational theory given by the (dependently typed) axioms. Indeed, we can understand every symbol matching one of the forms below as a (bidirectional) rewrite rule between LHS and RHS:

$c\colon$	$\vdash \forall[y_1 \colon Y_1, \dots, y_n \colon Y_n].$	LHS $\Leftrightarrow$ RHS
$c\colon$	$\vdash \forall[y_1 \colon Y_1, \dots, y_n \colon Y_n].$	LHS $\doteq$ RHS
$c\colon \{y_1 \colon Y_1, \dots, y_n \colon Y_n\}$	$\vdash$	LHS $\Leftrightarrow$ RHS
$c\colon \{y_1 \colon Y_1, \dots, y_n \colon Y_n\}$	$\vdash$	LHS $\doteq$ RHS

```
theory Nat =
  ℕ: type ▌

  /T … ▌

  /T Declaration and definition of ≤, e.g. via subtraction ▌
  ≤: ℕ ⟶ ℕ ⟶ prop = ▌ = … ▌ # #1 ≤ #2 ▌

  /T Alternatives for definition of ≥
     ---------------------------------- ▌

  /T Alternative 1: intensional equality ▌
  ≥: ℕ ⟶ ℕ ⟶ prop = [a, b] b ≤ a ▌

  /T Alternative 2a: extensional equality with ∀ and ⇔ ▌
  ≥: ℕ ⟶ ℕ ⟶ prop ▌
  ax_reversed_order2a: ⊦ ∀[n] ∀[m] n ≤ m ⇔ m ≥ n ▌

  /T Alternative 2b: extensional equality with ∀ and ≐ ▌
  ≥: ℕ ⟶ ℕ ⟶ prop ▌
  ax_reversed_order2b: ⊦ ∀[n] ∀[m] n ≤ m ≐ m ≥ n ▌

  /T Alternative 2c: extensional equality with dependent types and ⇔ ▌
  ≥: ℕ ⟶ ℕ ⟶ prop ▌
  ax_reversed_order2c: {n: ℕ, m: ℕ} ⊦ n ≤ m ⇔ m ≥ n ▌

  /T Alternative 2d: extensional equality with dependent types and ≐ ▌
  ≥: ℕ ⟶ ℕ ⟶ prop ▌
  ax_reversed_order2d: {n: ℕ, m: ℕ} ⊦ n ≤ m ≐ m ≥ n ▌
▌
```

Figure 5.5.: Possible formalizations of $\geq$ in *Nat* on top of an existing $\leq$. For them to be sensible, they need to link both symbols either intensionally (alternative 1) or extensionally via axioms (alternatives 2a – d). Note that the equivalence of alternative 2a – d might very well depend on the chosen foundation and logic.

In fact, the MMT system already allows annotating a subset of such symbols with the keywords `role Simplify` to add a corresponding *unidirectional* left-to-right simplification rule to the set of rules for a theory.[6] However, considering such rules for App-Gen would open a Pandora's box. Had we before a rewriting system terminating in most sensible settings (cf. Section 6.2.3) and whose rule applications always bring one closer to a solution, we would now be faced with combinatorial explosion and possible cycles.

Overall, we conclude that it is easy to come up with sensible, non-contrived examples, which immediately break our derived App-Gen-1 algorithm. One key aspect is that for almost all practical purposes rewriting must be considered modulo some equational theory, which can come from both the chosen foundation and from arbitrary *user-defined* axioms. Note how App-Gen-1 touched the tip of the iceberg of managing the equational theory from LF by considering β-expansion on the very surface. Even when neglecting the variable part by the user, the overall problem of (efficiently) finding a suitable generalization is as complicated as higher-order rewriting and unification [Pre97; MN98], which is known to be undecidable.

5.3. Further Examples

5.3.1. Normed & Metric Spaces

In basic analysis every normed vector space induces a canonical metric space with the metric

$$d(a, b) := \|a - b\|$$

In MMT this corresponds to the following theories and morphism declarations:

```
1  theory MetricSpace =
2    X : type ❚
3    d: X ⟶ X ⟶ ℝ ❚
4
5    /T Metric space axioms omitted for brevity ❚
6  ❚
7
8  theory NormedVectorspace =
9    Y: type ❚
10   norm: Y ⟶ ℝ ❚
11   minus: Y ⟶ Y ⟶ Y ❙ # 1 - 2 ❚
12
13   /T Other vector space operations and axioms omitted for brevity ❚
14 ❚
15
16 view NormedAsMetricSpace : ?MetricSpace -> ?NormedVectorspace =
```

[6] See https://uniformal.github.io/doc/language/declarations.html and https://uniformal.github.io/apidoc/info/kwarc/mmt/lf/SimplificationRuleGenerator.html.

```
  X = Y ▮
  d = [a, b] norm (a - b) ▮
▮
```

We can now formalize some basic notions in the theory of normed vectorspaces:

```
theory NormedVectorspaceThms =
  include ?NormedVectorspace ▮

  cauchy
    : (ℕ ⟶ Y) ⟶ prop ▮
    = [f] ∀[ε: ℝ] ∃[N: ℕ] ∀[n: ℕ] ∀[m: ℕ] (n ≥ N ∧ m ≥ N) ⇒ ((norm ((f n) -
↪   (f m))) < ε) ▮
  convergent_to
    : (ℕ ⟶ Y) ⟶ Y ⟶ prop ▮
    = [f, y] ∀[ε: ℝ] ∃[N: ℕ] ∀[n: ℕ] (n ≥ N) ⇒ ((norm ((f n) - y)) < ε) ▮

  /T Lipschitz continuity of an endofunction ▮
  lipschitz
    : (Y ⟶ Y) ⟶ prop ▮
    = [f] ∀[y1: Y] ∀[y2: Y] norm ((f y1) - (f y2)) ≤ norm (y1 - y2) ▮

  f: ℕ ⟶ Y ▮
  f_is_cauchy: ⊢ cauchy f ▮

  my_y: Y ▮
  f_convergent: ⊢ convergent_to f my_y ▮

  /T A theory of balls ▮
  in_ball
    : Y ⟶ Y ⟶ ℝ ⟶ prop ▮
    = [y, center, radius] (norm (y - center)) < radius ▮

  /T Actually only for ε > 0 ▮
  center_always_in_ball
    : ⊢ ∀[y: Y] ∀[ε: ℝ] in_ball y y ε ▮

  ball_convergent_to
    : (ℕ ⟶ Y) ⟶ Y ⟶ prop ▮
    = [f, y] ∀[ε: ℝ] ∃[N: ℕ] ∀[n: ℕ] (n ≥ N) ⇒ in_ball (f n) y ε ▮
▮
```

Evidently from a mathematician's point of view, these declarations can be trivially lifted to metric spaces. Indeed, by employing App-Gen-1 we can do so as well. Especially

note the assignment to d by the morphism in the first code snippet above. It is very similar to to $\lambda a, b : \mathbb{N}.\ b \leq a$, which we discussed extensively in the last section. The generalized theory after applicatoin looks as follows:

```
theory NormedVectorspaceThmsGeneralized =
  include ?MetricSpace ❚

  cauchy
    : (ℕ ⟶ X) ⟶ prop ❚
    = [f] ∀[ε: ℝ] ∃[N: ℕ] ∀[n: ℕ] ∀[m: ℕ] (n ≥ N ∧ m ≥ N) ⇒ (d (f n) (f m))
  ↪ < ε ❚
  convergent_to
    : (ℕ ⟶ X) ⟶ X ⟶ prop ❚
    = [f,y]∀[ε:ℝ]∃[N:ℕ]∀[n:ℕ]n≥N⇒(d (f n) y)<ε ❚

  lipschitz
    : (X ⟶ X) ⟶ prop ❚
    = [f] ∀[y1: X] ∀[y2: X] d (f y1) (f y2) ≤ d y1 y2 ❚

  f : ℕ ⟶ X ❚
  f_is_cauchy: ⊢ cauchy f ❚

  my_y: X ❚
  f_convergent: ⊢ convergent_to f my_y ❚

  in_ball
    : X ⟶ X ⟶ ℝ ⟶ prop ❚
    = [y, center, radius] (d y center) < radius ❚

  center_always_in_ball
    : ⊢ ∀[y: X] ∀[ε: ℝ] in_ball y y ε ❚

  ball_convergent_to
    : (ℕ ⟶ X) ⟶ X ⟶ prop ❚
    = [f,y] ∀[ε: ℝ] ∃[N: ℕ] ∀[n: ℕ] n ≥ N ⇒ in_ball (f n) y ε ❚
❚
```

We omit the obvious view generated by App-Gen-1 since it effectively does nothing else than renaming.

5.3.2. Sequences & Nets

In topological spaces nets are a generalization of sequences where the indexing domain $\mathbb{N}$ is replaced by arbitrary upwards-directed posets [Cla16]. We can almost autogenerate

initial bits of the theory of nets by starting with the theory of sequences and applying App-Gen-1. Precisely, we start with the following formalization:

```
theory Topology =
  Y: type ▮

  /T Neighborhood arounds a point ▮
  neighborhood: {y: Y} type ▮
  in_neighborhood: {y: Y} Y → neighborhood y → prop ▮ # 2 ∈ 3 ▮
▮

theory SequencesInTopology =
  include ?Topology ▮
  sequence: type ▮ = ℕ → Y ▮

  eventually_constant
    : sequence → prop ▮
    = [s] ∃[c: Y] ∃[N: ℕ] ∀[n: ℕ] (n ≥ N) ⇒ s n ≐ c ▮

  convergent_to
    : sequence → Y → prop ▮
    = [s, y] ∀[U: neighborhood y] ∃[N: ℕ] ∀[n: ℕ] n ≥ N ⇒ (s n) ∈ U ▮

  has_limit_point
    : sequence → Y → prop ▮
    = [s, y] ∀[U: neighborhood y] ∀[n: ℕ] ∃[N: ℕ] N ≥ n ∧ (s N) ∈ U ▮
▮
```

We then define upwards directed posets and their incarnation as $\mathbb{N}$ in the theory of sequences:

```
theory UpwardsDirectedPoset =
  X : type ▮
  leq_poset: X → X → prop ▮ # 1 ⊑ 2 ▮

  /T Usual poset axioms (reflexivity, transitivity, antisymmetry) omitted ▮
  upwards_directed: ⊦ ∀[x1: X] ∀[x2: X] ∃[y: X] x1 ⊑ y ∧ x2 ⊑ y ▮
▮

view UpwardsDirectedPosetToNat : ?UpwardsDirectedPoset ->
 ↪  ?SequencesInTopology =
  X = ℕ ▮
  leq_poset = [a, b] b ≥ a ▮

```

```
38    /T By the maximum function on the natural numbers
39    upwards_directed = …
40
```

Generalizing `SequencesInTopology` along the given morphism gives:

```
42  theory SequencesInTopologyGeneralized =
43    include ?Topology
44    sequence: type  = X → Y
45
46    eventually_constant
47      : sequence → prop
48      = [s] ∃[c: Y] ∃[N: ℕ] ∀[n: ℕ] (n ≥ N) ⇒ s n ≐ c
49
50    convergent_to
51      : sequence → Y → prop
52      = [s, y] ∀[U: neighborhood y] ∃[N: ℕ] ∀[n: ℕ] n ≥ N ⇒ (s n) ∈ U
53
54    has_limit_point
55      : sequence → Y → prop
56      = [s, y] ∀[U: neighborhood y] ∀[n: ℕ] ∃[N: ℕ] N ≥ n ∧ (s N) ∈ U
57
```

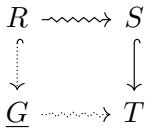
Note that in the listing above we retained the old declaration names. Indeed, discussion how users are asked to change names in the implementation is outside the scope of this thesis. As already remarked, the obtained theory of nets is only almost correct. Namely, it neglects the important aspect of nets that their indexing domain is left variable, even when the topological space is fixed. That is, in a topological space $(Y, \mathcal{O}_Y)$ a net is a function from an upwards-directed poset into Y.[7] It seems that nets are indeed not a purely syntactical generalization of sequences. Indeed, with the usual definition of a sequence as an $\mathbb{N}$-indexed sequence, they are not. However, if one defines a sequence as a functions from a countable toset into the respective topological space, then nets are again a purely syntactical generalization. The latter point can be justified insofar that $\mathbb{N}$-indexed sequences are in fact as powerful as sequences indexed by arbitrary countable tosets.[8]

Further observe how we were careful to express all sequence properties only in terms of quantifiers and the order relation without implicitly using the linearity or countability of $\mathbb{N}$. For example, we could have instead defined that a sequence s has a limit point y iff. for every neighborhood U of y the set $\{n \in \mathbb{N} \mid s\ n \in U\}$ is infinite. While this would generalize equally fine to upwards-directed posets, it would lead to a different notion than the *cofinality* notion in the generalization shown above and commonly accepted

[7] In particular, for a fixed topological spaces one needs to talk about the *class* of all nets.
[8] Arguably the usual $\mathbb{N}$-indexed sequences are just a sensible restriction without loss of generality.

for nets. The authors posit that careful formalization of proofs of basic theorems about sequences could also be purely syntactically generalized. As the MMT system does not sensibly support formalizing proofs, verification of the claim remains as future work.

5.4. Categorical Semantics



In the previous sections we only captured our App-Gen principle by specification of concrete algorithms. Thus it remained an open question whether the principle admits an elegant abstract definition. Our consistent use of the commuting square above might have already suggested to the reader that the principle possibly admits some categorical interpretation related to pullbacks and pushouts. In fact, in the category $\mathbb{Thy}$ of theories and theory morphisms, colimits are a sensible means to "identify, translate and combine logics" [Rab17]. Hence, this section intends to explore this direction for App-Gen on the very surface. Concretely, we motivate its relation and difference to pushouts and informally understand App-Gen as an *inverse* pushout. Dually, we glance at (inverse) pullbacks and their meaning in the broader context of refactoring as well. We leave proper treatment to future work.

Structure of the Section We first explain how pushouts are constructed in the category $\mathbb{Thy}$. Then, we will contrast those to the procedure of App-Gen and see the latter as inverse pushouts. Finally, we present the complete picture by dualizing (inverse) pushouts to (inverse) pullbacks.

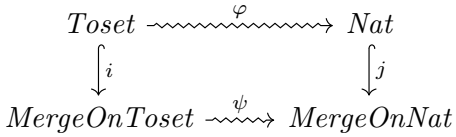
Pushouts Let us first informally explain the semantics and constructions of pushouts in $\mathbb{Thy}$. Recall the setting of our running example in Figure 5.6a. For the pushout we forget the lower right corner (Figure 5.6b). In the exposition below we will observe that the pushout operaton in fact recreates the forgotten theory *MergeOnNat* with exactly the same incoming morphisms.

Intuitively, to build the pushout, we can apply the same procedure as in many algebraic categories (Set, Mon, Grp, $\mathrm{Vect}_{\mathbb{F}}$, …), where pushouts can be thought of as *amalgamated sums*:

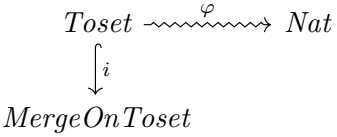
Pushouts in Algebraic Categories

The pushout of $B \overset{f}{\leftarrow} A \overset{g}{\rightarrow} C$ is $(B + C)/_{\sim}$, where $\sim$ is the smallest congruence relation identifying the elements in the coproduct which stem from a common source

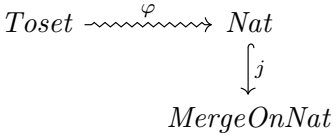
element in A. Formally, for every $a \in A$ we require $f(a) \sim g(a)$. The morphisms into the coproduct are simply given by the canonical embeddings of B and C into the set of equivalence classes $(B + C)/_{\sim}$.



(a) Complete square setting of the running example. Every other incomplete setting shall complete to this one via a categorical construction.



(b) The pushout setting



(c) The inverse pushout setting, which is completable by App-Gen

Figure 5.6.: The chapter's running example next to its restricted (inverse) pushout settings

Hence, let us first build the coproduct of *MergeOnToset* and *Nat*, which in $\mathbb{T}\mathbb{h}\mathbb{y}$ always amounts to a name clash free union with the obvious inclusion injections. Here, the summands do not share symbols to begin with, hence the coproduct theory is just a concatenation of both input theories. Next, we quotient out those declarations in the concatenation which stem from a common source declaration in *Toset*. In other words, for every declaration $\delta \in$ *Toset*, we follow the span's morphisms i and φ, and then identify $i(\delta)$ and $\varphi(\delta)$ in the concatenation. In particular we identify[9]

$$X = i(X) \sim \varphi(X) = \mathbb{N}$$
$$\leq_x = i(\leq_x) \sim \varphi(\leq_x) = \leq$$

In the pushout, these declarations from the concatenation are each amalgamated to a single declaration. In theory, pushouts are only unique up to isomorphism and at this point we could choose arbitrary names for the amalgamated symbols. However, users would rightly expect existing names to be chosen. For example, choosing $\mathbb{N}$ for the amalgamation of the pair X and $\mathbb{N}$ is indeed possible typewise as well as sensible UX-wise. Details on sensible selections of colimits in general and their existence can be found in [CMR17]. Continuing our specific choice, we also make $\leq$ stand for the identification of

[9]For brevity we left out the axioms contained in *Toset*. They do not contribute to the quotiening in any meaningful way.

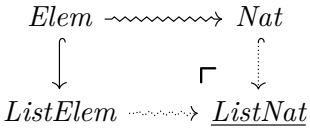


Figure 5.7.: Pushout of *elem*-lists to *Nat* yielding lists over natural numbers

$\leq_x$ and $\leq$. Neglecting the remaining three declarations of *Toset* representing its axioms, the quotiented concatenation so far with the coproduct's inclusion morphisms is already almost equal to the original *MergeOnNat* with its incoming morphisms (Figure 5.6a). The only difference is that our construction the inclusion of *Nat* has been flattened out. We neglect this exception to spare technical details.

The important conclusion is that the pushout operation *applied our existing specialization* φ to *MergeOnToset*.[10]

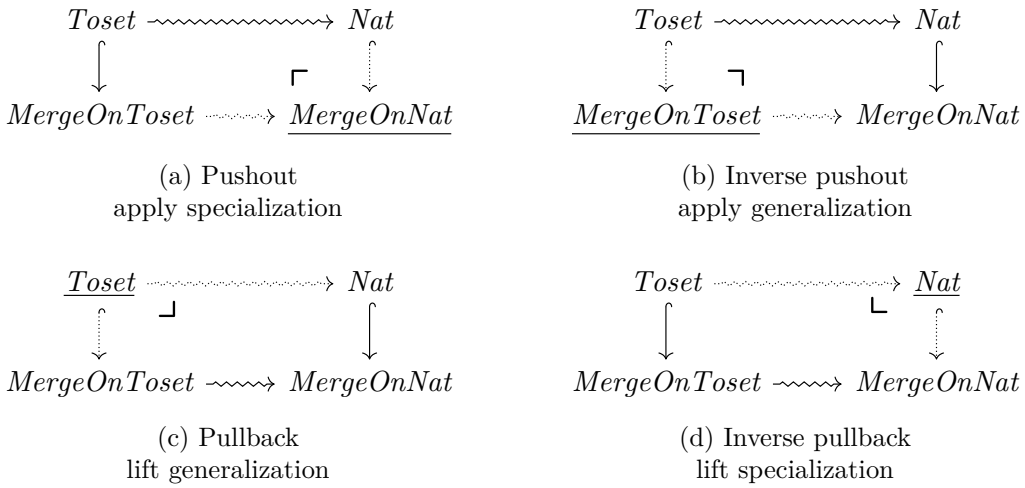
> *Remark* 5.4.1. The reader might convince themselves of their intuition by considering one more example taken from [CMR17]. The setting is sketched in Figure 5.7 and a more detailled formalization can be found in Appendix A.4. Consider a theory *Elem* declaring a single symbol $elem : type$. Building upon this theory, *ListElem* formalizes the notion of finite lists with entries of type *elem*. Orthogonally, the morphism φ: *Elem* $\rightsquigarrow$ *Nat* specializes *elem* to $\mathbb{N}$. Performing the "canonical" pushout *ListElem* $\hookleftarrow$ *Elem* $\rightsquigarrow$ *Nat* leads to the theory of lists of natural numbers *ListNat*.

Inverse Pushouts Comparing pushouts to our principle App-Gen, we realize that our principle begins its work from a different start configuration. Namely, it starts with the setting shown in Figure 5.6c, where our apparent pushout *MergeOnNat* already exists, but *MergeOnToset* has been forgotten. Performing the refactoring principle then creates that very forgotten theory. For this reason we informally call the procedure performed by App-Gen an *inverse pushout*. In fact, we conjecture that first performing this inverse pushout operation, then forgetting about the lower right theory, and finally computing the pushout yields exactly the forgotten theory.

Similar to pushouts above, we can record a characteristic semantic property: the inverse pushout operation *applied our existing generalization* φ to *MergeOnNat*.

The Full Picture We can dualize (inverse) pushouts to (inverse) pullbacks to get additional possibly interesting concepts. Whereas (inverse) pushouts semantically mean the application of either generalization or specialization, (inverse) pullbacks dually mean lifting either one. We depict all four concepts in Figure 5.8.

[10] Indeed, if ψ : *MergeOnToset* $\rightsquigarrow$ *MergeOnNat* denotes the indicated morphism generated by the pushout, we can understand ψ as a (hyper-)homomorphic extension of φ: *Toset* $\rightsquigarrow$ *Nat* to theory fragments of *Toset*, i.e. theories with a single inclusion, namely from *Toset* [Rab17, Notation 2.27].



(a) Pushout
apply specialization

(b) Inverse pushout
apply generalization

(c) Pullback
lift generalization

(d) Inverse pullback
lift specialization

Figure 5.8.: Various forms of categorical constructs with generalization or specialization meaning.

6. Implementation

I implemented the generalization principles described in the present thesis, namely Theory Splitting and App-Gen from Chapters 4 and 5, respectively, and contributed them to the MMT ecosystem[1]. Overall, this chapter showcases both implementations, discusses possible usage scenarios, and highlights design decisions. For Theory Splitting, we present a headless implementation, and overview some products which could be built on top of it. Focussing ourselves in particular on App-Gen, we present a headless implementation as well as a graphical user interface. Additionally, we state a termination criterion for rewriting and discuss how we deal with rewrite failures UX-wise. Finally, we conclude the chapter with a preliminary evaluation.

6.1. Theory Splitting

For Theory Splitting I implemented basic algorithmic, i.e. non-graphical, support for it on top of the MMT system.[2] In the course of doing so, I added APIs to deal with graph representations of theories and their declaration interdependencies as well.

(API) User Interface With our implementation, an API user can specify a theory and a valid partition on its declarations (cf. Definition 4.1.3) to obtain the respective declaration-grained split diagram. From there, the API user is free to further process the output, e.g. to export it to MMT surface syntax in the same way it is done below in the App-Gen implementation.

Intended Usage Currently due to the lack of a GUI, the implementation is not suited for end users. However, it can serve as the basis for (a) such a GUI and (b) batch refactorings. For (a), following the graphical plugin implementation for App-Gen below, a similar refactoring panel could be created for Theory Splitting. We imagine a prototypical user interface showing the interdependency graph of a theory, which the user can color according to their desired partition. Concerning (b), a batch refactorings (headless) tool could automatically search for sensible splits in libraries and suggest them to the user. In particular, the search space could be driven by *measures* on theories and morphisms. For example on theories, connectedness of the declaration interdepen-

[1] Concrete references for looking up the implementation code are given in the respective sections.

[2] While on top of the MMT syste, this has been done in a separate repository [ThySplit] for the reason that the implementation has a rather large graph library dependency whose adoption in the official MMT repository first needs careful evaluation.

```
1 theory ?Monoid
2 HasMeta ?Monoid http://cds.omdoc.org/urtheories?LF
3 Declares ?Monoid ?Monoid?X
4 constant ?Monoid?X
5 Declares ?Monoid ?Monoid?op
6 constant ?Monoid?op
7 DependsOn ?Monoid?op?type ?Monoid?X?type
```

Figure 6.1.: Relational information for a theory of monoids bookkept by the MMT system. It is a simple text format, from which we stripped some of the fully qualified URIs for brevity.

dency graph could be investigated. For morphisms, the authors briefly looked into and implemented very preliminarily surjectivity, totality and complexity measures.[3]

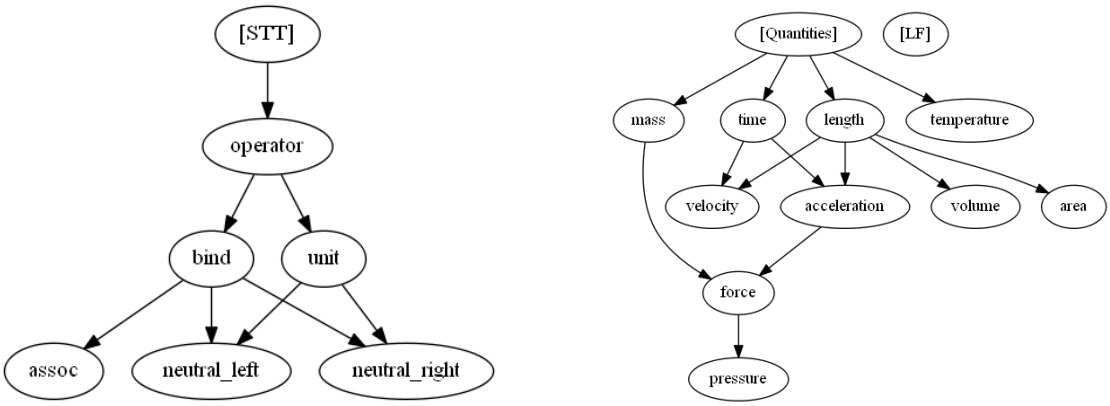
Rough Architecture Given a theory and a partition, the implementation first crucially needs the dependency relation on declarations (cf. Definition 2.2.2) to validate and perform the split. Fortunately, the relation is already bookkept by the MMT system in its so-called "relational information" (Figure 6.1), whose dependency information we directly feed as a graph into the third-party graph library jGraphT [Sic+18] for easier processing. In particular, the library allows transitive reduction with a one-liner. Validation and splitting is then straightforwardly performed close to Definition 4.1.3. Employing the graph library and its exporter to the graph rendering software GraphViz [Ell+03], we exemplarily rendered two declaration interdependency graphs in Figure 6.2. However, we note that our rendering was for prototypical reasons only. Especially, rendering of whole theory graphs in 2D as well as 3D has already been done in much more detail [RKM17; MKR] and those existing implementations should preferably be extended such that theories can also be looked into to show the declaration interdependency graphs.

6.2. App-Gen

The contribution for App-Gen is two-fold. On the one hand, it encompasses an implementation of the revised algorithm App-Gen-1 from Definition 5.2.1, which has been directly contributed to the core repository of the MMT system [MMTb].[4] On the other hand, it also adds graphical tool support for running it. The latter is based on top of

[3] Roughly, surjectivity measures how much a morphism's mapping (transitively) depends on the symbols in the codomain. Totality means non-partiality. Complexity describes the (tree) size of the assigned codomain expressions in the morphism. Renamings are minimally complex. We suggest a morphism be surjective, total and simple.

[4] The commits first publicly landed in release v16.0.0: https://github.com/UniFormal/MMT/releases/tag/v16.0.0.



(a) A theory of monads

(b) A theory of physical dimensions

Figure 6.2.: Rendered declaration interdependency graphs. Nodes with square brackets denote inclusions, all other denote ordinary constant declarations.

a preexisting MMT plugin for the IntelliJ IDEA editor [Intellij-MMT]. Again, the code has been directly contributed to that repository.

Structure of the Section In Section 6.2.1 we describe possible users of the tool. In Section 6.2.2 we sketch the placement of our tool in the MMT ecosystem and discuss initial design decisions of the algorithm. Subsequently, in Section 6.2.3 we reason about termination and success of generalization, which we have partly left unspecified in previous chapters. Whereas both previous sections focus on the headless algorithm itself, Section 6.2.4 finally presents the developed GUI and describes how it deals with generalization errors.

6.2.1. Intended User Base

We already described implementation-agnostic use cases in the introductory Chapter 1 as well as more concretely in the introduction to App-Gen in Chapter 5. Hence, we focus ourselves on describing use cases possible with the *current* state of the implementation and its GUI.

The implementation already allows end users to make existing knowledge more abstract by entering theories and morphisms into the graphical user interface and clicking a button. However, we remark that currently settings admitting successful generalization are limited by the set of rewrite rules App-Gen-1 considers, cf. Section 5.2 for an elaborated discussion. Nonetheless, we suppose that playing around with simple enough generalization tasks is possible, perhaps in the realm of algebraic hierarchies and their incarnations, cf. Section 5.3. Hence, the current state is more targetted at students instead of senior or research mathematicians. In particular, we detail an educational use case in Chapter 7.

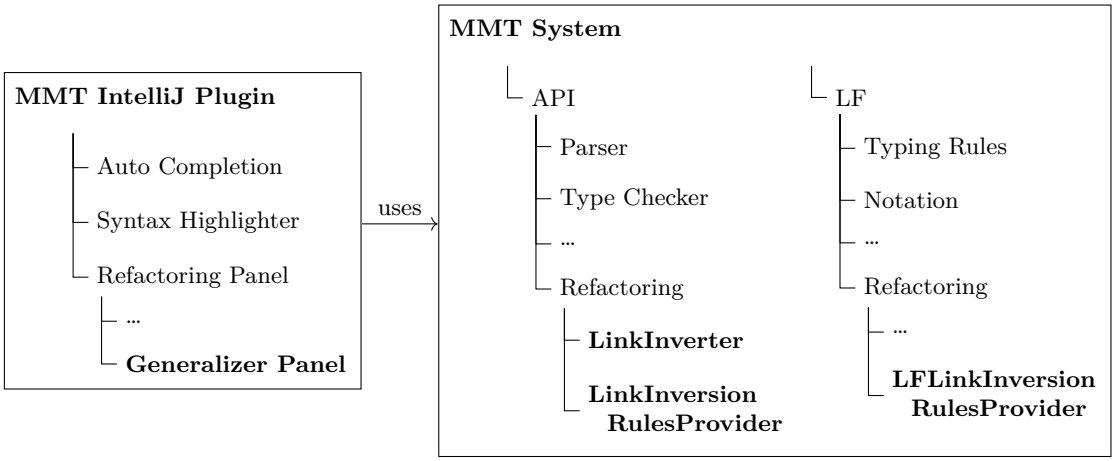


Figure 6.3.: Architectural landscape of the MMT ecosystem and my implementation (bold-faced)

6.2.2. Algorithm Implementation: Architectural Overview

A rough architectural overview can be seen in Figure 6.3, where the contribution is marked in bold. On the left, we see the plugin side accounting for the graphical user interface which we discuss below in Section 6.2.4. On the right, we see a sketch of the aggregated module/class/interface hierarchy of the MMT system. On the outermost level it is structured into multiple modules (API, LF, …). Especially, as foundation independence is a prescribed goal of MMT, most of the foundation-agnostic business logic is put into the API module, whereas the bits dependent on the "standard" foundation LF are put into its respective module.

The same holds true for my algorithm contribution, which is split across the API and LF modules as well. Evidently, App-Gen-1 from Definition 5.2.1 depends on LF only in one of its rewriting rule schemes, namely scheme eq. (5.4). In contrast, the other two rules schemes can be considered part of the "algorithmic skeleton" of the definition: Independent of which additional rules might get added in the future, these two rules are probably kept. Consequently, we split the logic as follows (Figure 6.3):

- a class `LinkInverter` in the API module
- an interface `LinkInversionRulesProvider` in the API module
- a class `LFLinkInversionRulesProvider` in the LF module implementing the former interface

The `LinkInverter` class precisely captures the algorithm's skeleton. As input, it takes theories R, S, T with $S \hookrightarrow T$, a morphism $\varphi\colon R \nrightarrow S$, and a `LinkInversionRulesProvider`. It then performs generalization by rewriting with the rule schemes eqs. (5.3) and (5.5), *and* those contributed by the passed rules provider. Eventually, assuming termination, it outputs the generalized theory together with a specialization morphism.

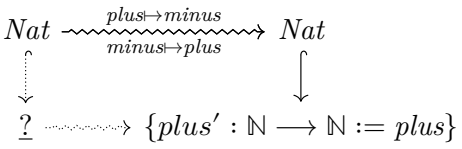


Figure 6.4.: Non-terminating generalization setting with App-Gen-1 and its implementation: The *plus* in the definiens of *plus′* will get indefinitely rewritten to *minus*, *plus*, *minus*, …. See Appendix A.3 in the appendix for elaborated code.

The employed `LFLinkInversionRulesProvider` class is currently the only rules provider. Provided a specialization morphism, it generates the rules given by the LF-dependent scheme eq. (5.4). To perform the matching happening in this scheme, it uses the already existing higher-order typed `Matcher` class from MMT's API module.

Employing this separation, multiple variants of App-Gen can coexist in a clean and DRY[5] way. In particular, future work might allow *user-defined* additional rewrite rules by means of a GUI. This becomes straightforward thanks to this separation.

6.2.3. Termination and Success of Generalization

Currently, the `LinkInverter` class recursively and exhaustively applies the gathered rewrite rules in unspecified order. This might be fine in most realistic circumstances fulfilling the termination criterion below, but may yield non-terminating behavior in some cases as exemplified by Figure 6.4. The root cause for non-termination in that example is that the output of a rewrite rule is not necessarily stable, but instead can be fed again as input to a second (possibly identical) rewrite rule. Thus, if we prevent applicability of a rule's output as a second rule's input, we can guarantee termination.[6] Do note that *void* rewrite rules, i.e. rules of the form $\frac{t}{t}$ for a term t, are *not* naïvely applied by the implementation. In fact, disregarding void rules preserves the implementation's semantics. As a consequence, generalizing along the identity morphism always terminates.

To capture the informal argument on applicability of a rule's output, we first introduce

Definition 6.2.1. Let R and S be two theories as well as R^* and S^* denote the sets of reflexive-transitively included theories of R and S, respectively. We define

[5] "Don't Repeat Yourself" – a common software engineering slogan

[6] A term is made up of only finitely many subterms on which only finitely many rules are once-only applicable.

the **intersection of two theories** by

$$R \cap S := \bigcup_{T \in R^* \cap S^*} T.$$

In other words, $R \cap S$ is the union of all common included theories.

Example 6.2.1.

- $R \cap R = R^*$
- $\{c : E\} \cap \{c : E\} = \emptyset$ as both symbols do not come from a common base theory and are thus treated as separate entities even though they share their respective theory-local name.
- Let $T := \{c : E\}$, $R = \{\text{include } T, c' : E := c\}$ and $S = \{\text{include } T, c' : E := c\}$. We have $R \cap S = \{c : E\} = T$. Indeed, in the MMT system the paths referring to c are equal from within both R and S. In fact, two symbols are identified in the intersection iff. their paths are equal.

We can now state the promised

Lemma 6.2.1 (Termination Criterion)**.** *Let $\varphi \colon R \rightsquigarrow S$ be a morphism along we generalize. If φ is the identity on $R \cap S$, then* App-Gen-1 *and the implemented algorithm both terminate.*

Proof. We show that rewritten terms are stable and cannot be subject to another rewrite anymore. For every assignment $(c := c') \in \varphi$, either c is a valid S-expression or not. In the first case, we have $c = c'$ per assumption and the rewrite rule becomes void. In the second case, any terms containing c cannot be valid S-expressions either and thus cannot be the input of any other rewrite rule by construction. □

In fact, the criterion applies to all possible (future) variants of App-Gen whenever the rewrite schemes' inputs are S-expressions.

We suppose without empirical evidence that in many practical generalization cases the termination criterion is fulfilled. Let us motivate this by considering the typical scenario for App-Gen. Namely, we posit it is a single formalization setting where different theories often share a bunch of common inclusions. For example, they might all share the theories used for logic, proof theory, and most importantly, the theories representing the building blocks of the overall mathematical concept being formalized. Concretely, in the setting of partial differential equations, the building blocks would exemplarily entail theories of topology, boundary conditions, functional analysis, and weak differentiation.[7]

[7] Actually, MMT theories as defined in [Rab18] and implemented in [MMTb] are allowed to specify a *meta theory*, whose use case is exactly to specify a theory aggregating other theories such as the named ones in this paragraph.

In such settings, we argue that most sensible specialization morphisms leave the common theories untouched and only act on the actual "lowest" theories. However, (non-identity) endomorphisms are notable exceptions.

Success of Generalization After exhaustive rewriting, according to Definition 5.2.1 we still ought to check if the final expression is well-typed in the to-be-generated theory G[8]. In the implementation, we neglect typechecking and only verify whether the rewritten expression is a closed, possibly ill-typed expression in G. To do so, we naïvely recurse the expression's subterms. For every referenced constant declaration, we check whether it originates from a theory already included in the current state of G. To be precise, we even allow it to come from a theory which has an implicit morphism to G.[9] That way, the implementation is future-proof to yet unforeseen usage in this direction.

To cope with failures, namely, if an expression fails to be a valid G-expression, we designed the core generalization algorithm in such a way that it accepts a dedicated `RewriteErrorHandler` object. This handler object can tell the algorithm to either skip the currently failing declaration (default), to forcefully assume it to be rewritable, or to rewrite it to another specified declaration altogether. The achieved flexibility can be used in multiple ways. For example, the GUI we present below uses the handler as a callback to actually display these errors. Another future use case would be to interactively query the user whether they desire to manually specify a generalized declaration. Recall that App-Gen tackles a fundamentally undecidable problem on its surface, where slight modifications make it fail, thus allowing human guidance would far outperform just showing a hardcoded failure message.

6.2.4. GUI on top of the IntelliJ IDEA MMT plugin

IntelliJ IDEA is an integrated development environment (IDE) for programming in Java and related languages [IDEA]. With its extensible plugin infrastructure, third-party vendors may add support for other languages as well. In particular, the MMT plugin [Intellij-MMT] adds the ability to use the IDE for formalization with the MMT system. It offers the usual features one would expect from a language integration in an IDE such as syntax highlighting, parsing, type checking as well as the ability to explore the hierarchical structure of MMT documents. Behind the scenes, the MMT plugin queries the MMT system in order to be able to provide all these features (Figure 6.3).

Within the scope of this thesis, I extended the plugin with a graphical interface to run the implementation of App-Gen-1 dicussed above. The code has been directly contributed to the repository in [Intellij-MMT].[10]

A screenshot can be seen in Figure 6.5. It shows an opened file formalizing normed vectorspaces from Section 5.3 on the right. On the left, we see the refactoring panel I contributed. It asks the user to enter the required parameters for App-Gen-1, namely

[8] To be precise, we should say G_{i-1} here and in the following where i denotes the current iteration.
[9] Implicit morphism subsume inclusions, see [RM18b] for details.
[10] The contribution has publicly landed in commit bf3e0f809283672582389df431af61cb23758e94, which represents the finishing commit for version v16.0.0 of the plugin.

paths to theories R, S, T and a morphism $R \rightsquigarrow S$.[11] Upon clicking of the "Generalize" button, the implemented algorithm is run. On success, its output – MMT surface syntax for the generalized theory and the created specialization morphism – is presented in a text box below the button, from which the user can directly paste its contents into the current document. On (partial) failure, namely if some declarations could not be successfully rewritten, they will be shown in a tree view left to the text box. The tree details which specific subterms caused the failure.

Concretely, in the screenshot we can see success and failure at the same time. On the one hand, generalizable declarations such as `cauchy` and `convergent_to` have been successfully incorporated in the output text box. On the other hand, the declaration `not_rewritable`, as evident from its definition shown in the editor on the right, is not generalizable and was thus added to the tree view. Specifically, the tree shows that the declaration caused failure in its definition and more precisely the subterm `norm` was not rewritable. Indeed, the expression `norm a` cannot even "morally" be lifted to metric spaces. Formally, we might say it does not rewrite rule pattern-matching `norm (_ - _)` and replacing it by `d(_, _)`. The reader is invited to look up the example's elaboration in Section 5.3 to convince themself of the details.

6.3. Preliminary Evaluation

> It's always too early to evaluate a technology, until, suddenly, it's too late!
>
> Martin Buxton (1987)

Due to several reasons we list below, we deem it to be too early for providing a thorough evaluation of the presented principles and implementations.

First and foremost, this thesis presents the refactoring principles without precisely defining criteria or measures determining when these are to be applied. Having such measures would enable intelligently performing the refactorings on larger libraries, whose outcomes could then be compared to refactorings suggested by humans. Additionally, the MMT system does currently not offer a practical way to formalize proofs at time of writing. On the one hand, this limits evaluability of measures determining good theory splits as useful dependency data is not available. On the other hand, especially for App-Gen the generalizability of proof terms would be a major concern in an evaluation.

Hence, we restrict ourselves to stating that the implemented principle App-Gen-1 successfully runs on all basic examples presented in this thesis and simple enough variations. This especially includes the merge sort running example from Chapter 5 and further examples mentioned in Section 5.3.

[11] Recall the commutative square structuring these parameters, which is used throughout this thesis, e.g. in Definition 5.2.1.

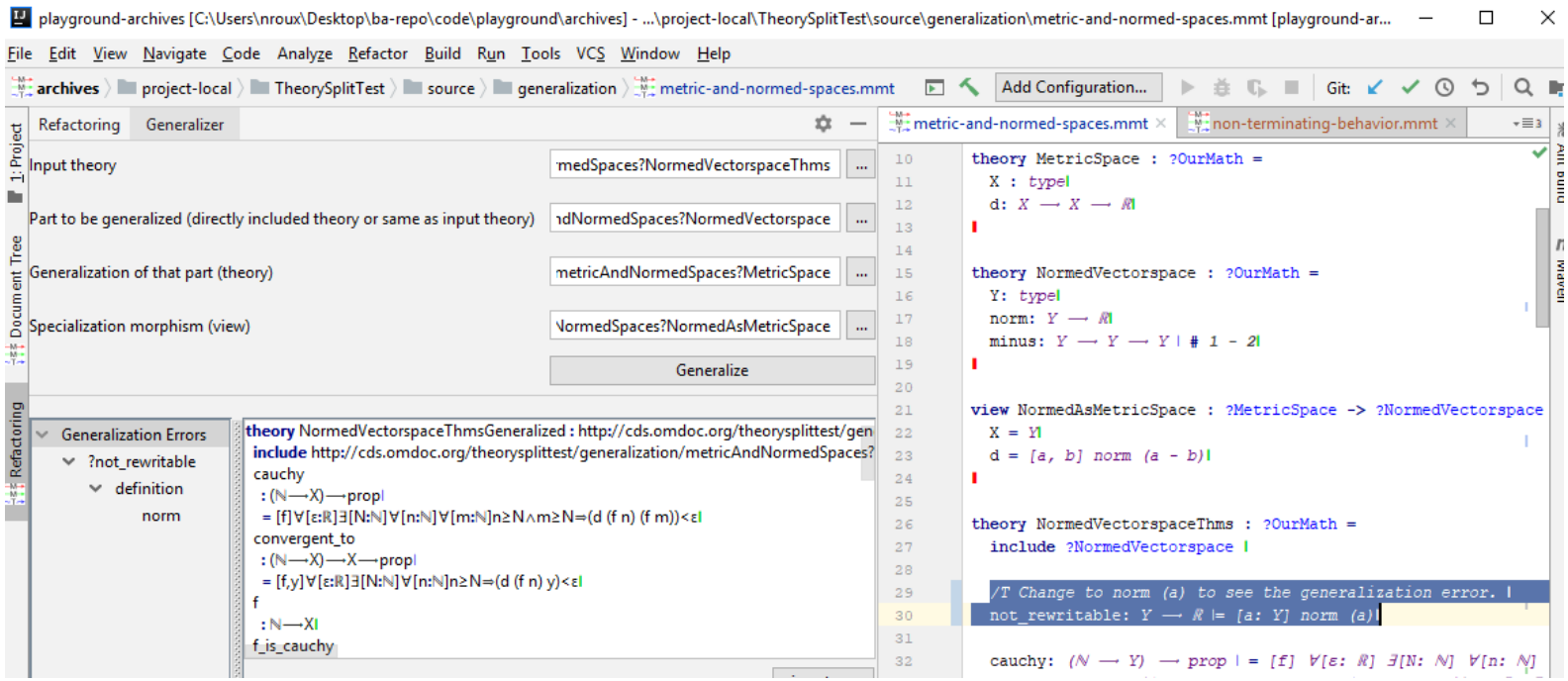


Figure 6.5.: Screenshot of the MMT plugin with App-Gen in action

7. Conclusion

Employing the MMT framework organizing formal knowledge into theories of typed declarations and connecting morphisms, we have given and motivated a definition of generalization refactorings in that system. Furthermore, we have shown a preliminary definition for behavior preservation. Central to both definitions is that they can be expressed and (partially) verified within the MMT system.

Based on this framework of generalization refactorings, we have presented the princiciples Theory Splitting and App-Gen. The former principle can be used for outsourcing declarations of an existing "large" theory onto multiple smaller, more coherent theories. We have focussed ourselves in particular on App-Gen in the present thesis, which generalizes theories along given morphisms. It does so by employing a set of rewrite rules it generates from the morphism's assignments. For example, it is able to generalize the definiton of convergence in normed vectorspaces to metric spaces by using the generated rewrite rule $\|\cdot - \cdot\| \to d(\cdot, \cdot)$. We have suggested App-Gen be used to help users in prototyping formalizations and to abstract existing knowledge, which is favorable from an MKM perspective. Furthermore, it may be strategically used in narrative presentation settings to primarily provide concrete knowledge while letting the abstract counterparts implicit. Last, we imagine an educational use case for students, which we detail below under future work.

We have presented and discussed early implementations for both principles. In particular for App-Gen, we have contributed the algorithm and a graphical interface to the MMT ecosystem [MMTb; Intellij-MMT]. Precisely, the GUI has been created on top of the MMT plugin for the IntelliJ IDEA IDE, which now shows a refactoring panel allowing users to run App-Gen after having entered the necessary parameters. On (partial) success, it outputs the generalized theory in MMT surface syntax, which users can directly paste into their existing formalization.

Future Work

Evaluation We have deferred proper evaluation of the presented theory, its implementations and possible use cases to future work. In particular, we think that being able to intelligently run the principles at a larger scale than manual execution is crucial for evaluation of their applicability on libraries.

Theory Our preliminary definitions of behavior preservation need to be tested in their usefulness with further generalization refactoring principles. Concerning App-Gen, it

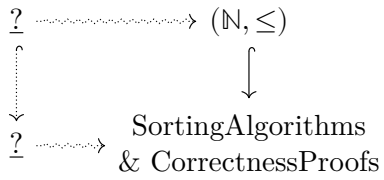


Figure 7.1.: Underlying "generalization problem" students are asked to explore.

remains an open question whether it can be adequately described by categorical semantics.

Implementation The MMT plugin could greatly benefit from graphical as well as functional extensions. We refer the reader to the issue list at the repository in [Intellij-MMT]. In particular, we believe showing generalization errors within the editor would greatly foster our conjectured App-Gen's use cases, especially the one about educational value, which we elaborate in more detail below.

Exploration of Educational Use Cases We imagine App-Gen could help students to become acquainted with the ubiquitous practice of generalization in mathematics and related subjects:

> Generalization is the heartbeat of mathematics, and appears in many forms. If teachers are unaware of its presence, and are not in the habit of getting students to work at expressing their own generalizations, then mathematical thinking is not taking place. ([Mas96])

Precisely, the authors would like to detail the following scenario they have in mind. Usual mathematics and computer science studies' curricula feature an introductory course to algorithms and data structures in the first semester, which typically contains a section on sorting algorithms. We propose students are assigned the task of finding a suitable generalization for sort algorithms defined on lists of natural numbers as depicted in Figure 7.1. In detail, the teaching process may go as follows:

1. The teacher introduces sorting algorithms on $(\mathbb{N}, \leq)$, while possibly hinting at natural numbers being pawns for abstract ordered objects. Also, the algorithms are proven to be correct for $(\mathbb{N}, \leq)$. The teacher provides a version of both the algorithms and proofs in a formalization system, cf. the lower right corner in Figure 7.1.

2. In tutorial sessions or as homework, students are asked how they would sort strings and consequently how the sorting algorithms can be generalized.

3. In computer rooms, parallel to those conceptual discussions, students are encouraged to enter and try their generalizations within the system. By "try" we mean

running App-Gen with the extension proposed above, namely that generalization errors are graphically highlighted in the editor.

For freshmen the formalization system would need to offer a sufficiently simple GUI frontend. Then, key takeaways could be:

- How many requirements do we have to put on the order such that the definition of a sorted list and a sorting algorithm still make sense? Students can be asked to try different flavors of ordered sets. For instance, the sortedness of a list $(a_i)_i$ could be formalized in different ways and proven equivalent in $(\mathbb{N}, \leq)$:

  $$\begin{aligned} \forall i.\quad & a_i \leq a_{i+1} && \text{(the order's transitivity is implicit)} \\ \forall i, j.\quad & a_i \leq a_j && \text{(the order's reflexivity is implicit)} \\ \forall i < j.\quad & a_i \leq a_j && \text{(generalizes previous ones)} \end{aligned}$$

  The students can be asked where they think the equivalence will fail with more weakly ordered sets before they try it out themselves and see those non-generalizable definitions and proofs highlighted by the system.

- Which algorithms are still correct, i.e. sort according to one of the sortedness definitions?

- Which algorithms are stable? This could be used to motivate strict weak orders.

A. Appendix

A.1. Behavior Preservation of Multiplicative and Dividive Groups

This section serves to show that our initial definition of behavior preservation in Definition 3.3.1 is too strong for practical purposes and needs further revision in future work. Consider the following axiomatization of groups via multiplication as usual and via division [Mat17]:

```
theory MultiplicativeGroup =
  U: type ❚
  e: U ❚
  op: U ⟶ U ⟶ U | # #1 ∘ #2 ❚
  inv: U ⟶ U ❚

  associative: ⊦ … ❚
  e_neutral: ⊦ … ❚
  inv_inverse: ⊦ … ❚
❚

theory DividiveGroup =
  U: type ❚
  e: U ❚
  div: U ⟶ U ⟶ U | # #1 / #2 ❚

  /T Axiomatization from
     "Matt F. (https://mathoverflow.net/users/44143/matt-f),
      Wiener's axiomatization of the group law based on division
      URL (version: 2017-08-20): https://mathoverflow.net/q/279161" ❚
  div_neutral1: ⊦ ∀[a: U] div(a, a) ≐ e ❚
  div_neutral2: ⊦ ∀[a: U] div(a, e) ≐ a ❚
  div_cancellation: ⊦ ∀[a: U] ∀[b: U] ∀[c: U] (a / c) / (b / c) ≐ a / b ❚
❚

view φ: MultiplicativeGroup -> DividiveGroup =
  U = U ❚
  e = e ❚
```

```
29    op = [a, b] a / (e / b) ▌
30    inv = [a] e / a ▌
31
32    associative = … ▌
33    e_neutral = … ▌
34    inv_inverse = … ▌
35  ▌
36
37  view ψ: DividiveGroup -> MultiplicativeGroup =
38    U = U ▌
39    e = e ▌
40    div = [a, b] a ∘ (inv b) ▌
41
42    /T this amounts to saying that (a ∘ (inv a)) ≐ e ▌
43    div_neutral1 = … ▌
44
45    /T this amounts to saying that (a ∘ (inv e)) ≐ a ▌
46    div_neutral2: = … ▌
47
48    /T this amounts to saying that (a ∘ (inv c)) ∘ (inv (b ∘ (inv c))) ≐ a ∘
      ↪  (inv b) ▌
49    div_cancellation: = … ▌
50  ▌
```

As intuition tells us, both theories should be behavior-preserving generalizations of each other. However, behavior preservation as from Definition 3.3.1 is only fulfilled if we consider some equational theory. Concretely, let us attempt to prove behavior preservation for ψ. For it we need to specify preimages in `DividiveGroup` for every declaration in `MultiplicativeGroup`. Doing so for `U` and `e` is straightforward, we just choose the corresponding identically named declarations. Let us consider `op`. There we can choose the preimage

$$t := [a, b]a/(e/b) \in \texttt{Obj}(DividiveGroup)$$

Indeed, applying the morphism ψ yields `op`

$$\begin{aligned}\overline{\psi}(t) &= [a, b]\ a \circ (\text{inv } \overline{\psi}(e/b)) \\ &= [a, b]\ a \circ (\text{inv } (e \circ (\text{inv } b))) \\ &= [a, b]\ a \circ b \\ &= op\end{aligned}$$

where we especially made use of the group axioms in `MultiplicativeGroup`.

A.2. Generalized Merge Sort on Tosets

```
theory MergeOnToset =
  include ?Toset ❚
  include ?PolymorphicLists ❚ /T Some imaginary theory of finite lists ❚

  /T Merge two pre-sorted lists ❚
  merge: List X ⟶ List X ⟶ List X ❚

  /T Axioms on merge (≈ inductive definition) ❚
  merge_base_case1: ⊢ ∀[l: List X] merge nil l ≐ l ❚
  merge_base_case2: ⊢ ∀[l: List X] merge l nil ≐ l ❚
  merge_ind_step: ⊢ ∀[x: X] ∀[y: X] ∀[xs: List X] ∀[ys: List X]
    merge (cons x xs) (cons y ys) ≐
      if (x ≤ₓ y) cons x (merge xs (cons y ys))
      else cons y (merge (cons x xs) ys) ❚

  /T Sort a list ❚
  mergesort: List X ⟶ List X ❚
  mergesort_base_case: ⊢ mergesort nil ≐ nil ❚
  mergesort_ind_step: ⊢ ∀[l: List X] ∀[k: List X]
    mergesort (l ::: k) ≐ merge (mergesort l) (mergesort k) ❚
❚
```

A.3. Non-Terminating Behavior with App-Gen

We show an exemplary formalization for the non-terminating setting with App-Gen from Figure 6.4:

```
theory Nat =
  ℕ: type ❚
  plus: ℕ ⟶ ℕ ❚
  minus: ℕ ⟶ ℕ ❚
❚

theory Nat' =
  include ?Nat ❚

  plus': ℕ ⟶ ℕ ❚ = plus ❚
❚

view φ : ?Nat -> ?Nat' =
  ℕ = ℕ ❚
```

```
15    plus = minus ▌
16    minus = plus' ▌
17  ▌
```

Generalizing Nat' along φ will indefinitely rewrite the *plus* expression in the highlighted line to *minus*, *plus*, *minus*, ... Indeed our termination Lemma 6.2.1 does not apply because φ is *not* the identity on $Nat \cap Nat' = Nat$.[1]

A.4. Pushout Example

Below we show an elaborated formalization for the example from Remark 5.4.1. The pushout $ListElem \hookleftarrow Elem \rightsquigarrow Nat$ results in the theory of lists over natural numbers

```
1   theory Elem =
2     elem: type ▌
3   ▌
4
5   theory Nat =
6     /T Natural numbers as usual with Peano axioms ▌
7
8     ℕ: type ▌
9     0: ℕ ▌
10    s: ℕ → ℕ ▌
11    plus: ℕ → ℕ → ℕ ▌ = … ▌ # #1 + #2 ▌
12
13    /T … ▌
14  ▌
15
16  view ElemToNat : Elem -> Nat =
17    elem = ℕ ▌
18  ▌
19
20  theory ListElem =
21    include ?Elem ▌
22
23    list: type ▌
24    nil: list ▌
25    cons: elem → list → list ▌
26
27    /T … ▌
28  ▌
```

[1] Beware that – speaking in the visual image of the commutative square with nodes R, S and T – here we have $R = Nat$ and $S = T = Nat'$.

Bibliography

[BC87] Thierry Boy de la Tour and Ricardo Caferra. *Proof Analogy in Interactive Theorem Proving: A Method to Express and Use It via Second Order Pattern Matching*. Jan. 1, 1987. 95 pp.

[BFS07] Markus Bach, Florian Forster, and Friedrich Steimann. "Declared Type Generalization Checker: An Eclipse Plug-In for Systematic Programming with More General Types". In: *Fundamental Approaches to Software Engineering*. Ed. by Matthew B. Dwyer and Antónia Lopes. Vol. 4422. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 117–120. ISBN: 978-3-540-71288-6 978-3-540-71289-3. DOI: 10.1007/978-3-540-71289-3_10. URL: http://link.springer.com/10.1007/978-3-540-71289-3_10 (visited on 03/12/2019).

[Car+11] Jacques Carette et al. "The MathScheme Library: Some Preliminary Experiments". In: *CoRR* abs/1106.1862 (2011). URL: http://arxiv.org/abs/1106.1862.

[Car+19] Jacques Carette et al. "Big Math and the One-Brain Barrier A Position Paper and Architecture Proposal". In: *arXiv:1904.10405 [cs, math]* (Apr. 23, 2019). URL: http://arxiv.org/abs/1904.10405 (visited on 06/04/2019).

[CF09] Jacques Carette and William M. Farmer. "A Review of Mathematical Knowledge Management". In: *Intelligent Computer Mathematics*. Ed. by Jacques Carette et al. Vol. 5625. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 233–246. ISBN: 978-3-642-02613-3 978-3-642-02614-0. DOI: 10.1007/978-3-642-02614-0_21. URL: http://link.springer.com/10.1007/978-3-642-02614-0_21 (visited on 06/04/2019).

[Cla16] Pete L. Clark. "Convergence". Oct. 18, 2016. URL: http://math.uga.edu/~pete/convergence.pdf (visited on 06/11/2019).

[CMR17] Mihai Codescu, Till Mossakowski, and Florian Rabe. "Canonical Selection of Colimits". In: *Recent Trends in Algebraic Development Techniques*. Ed. by Phillip James and Markus Roggenbach. Springer, 2017, pp. 170–188.

[DH82] Philip J. Davis and Reuben Hersh. *The Mathematical Experience*. Boston: Houghton Mifflin, 1982. 440 pp. ISBN: 978-0-395-32131-7 978-0-395-32157-7. URL: https://epdf.pub/the-mathematical-experience.html.

[Ell+03] John Ellson et al. “Graphviz and dynagraph – static and dynamic graph drawing tools”. In: *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[Fou19] Eclipse Foundation. *Eclipse documentation - Refactor Actions*. Mar. 2019. URL: https://help.eclipse.org/2019-03/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm (visited on 05/30/2019).

[Fow+99] Martin Fowler et al. *Refactoring: improving the design of existing code*. The Addison-Wesley object technology series. Reading, MA: Addison-Wesley, 1999. 1 p. ISBN: 978-0-201-48567-7.

[GM06] Alejandra Garrido and Jose Meseguer. “Formal Specification and Verification of Java Refactorings”. In: *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. Philadelphia, PA, USA: IEEE, Sept. 2006, pp. 165–174. ISBN: 978-0-7695-2353-8. DOI: 10.1109/SCAM.2006.16. URL: http://ieeexplore.ieee.org/document/4026866/ (visited on 05/27/2019).

[Gon+13] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Red. by David Hutchison et al. Vol. 7998. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 163–179. ISBN: 978-3-642-39633-5 978-3-642-39634-2. DOI: 10.1007/978-3-642-39634-2_14. URL: http://link.springer.com/10.1007/978-3-642-39634-2_14 (visited on 06/04/2019).

[Gon08] Georges Gonthier. “Formal proof–the four-color theorem”. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[Hal+17] Thomas Hales et al. “A FORMAL PROOF OF THE KEPLER CONJECTURE”. In: *Forum of Mathematics, Pi* 5 (2017), e2. ISSN: 2050-5086. DOI: 10.1017/fmp.2017.1. URL: https://www.cambridge.org/core/product/identifier/S2050508617000014/type/journal_article (visited on 06/19/2019).

[Har09] John Harrison. “HOL Light: An Overview”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 60–66. ISBN: 978-3-642-03358-2 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_4. URL: http://link.springer.com/10.1007/978-3-642-03359-9_4 (visited on 06/17/2019).

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.

[Ian+16] Mihnea Iancu et al. “Mixing Surface Languages for OMDoc”. 2016. URL: http://kwarc.info/kohlhase/submit/mmt-stex16.pdf.

[Ian17] Mihnea Iancu. “Towards Flexiformal Mathematics”. PhD thesis. Bremen, Germany: Jacobs University, 2017. URL: https://opus.jacobs-university.de/frontdoor/index/index/docId/721.

[IDEA] *IntelliJ IDEA – Capable and Ergonomic IDE for JVM*. URL: https://www.jetbrains.com/idea/ (visited on 06/13/2019).

[Ing18] Joseph Ingeno. *Software architect's handbook: become a successful software architect by implementing effective architecture concepts*. OCLC: 1055162429. 2018. ISBN: 978-1-78862-767-2. URL: http://proquest.safaribooksonline.com/?fpi=9781788624060 (visited on 06/17/2019).

[Intellij-MMT] *UniFormal/IntelliJ-MMT – An IntelliJ-Plugin for MMT*. URL: https://github.com/UniFormal/IntelliJ-MMT (visited on 06/13/2019).

[Jet19] JetBrains s.r.o. *Use Interface Where Possible - Help | IntelliJ IDEA*. 2019. URL: https://www.jetbrains.com/help/idea/use-interface-where-possible.html (visited on 05/30/2019).

[JL04] Einar Broch Johnsen and Christoph Lüth. “Theorem Reuse by Proof Term Transformation”. In: *Theorem Proving in Higher Order Logics*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Red. by David Hutchison et al. Vol. 3223. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–167. ISBN: 978-3-540-23017-5 978-3-540-30142-4. DOI: 10.1007/978-3-540-30142-4_12. URL: http://link.springer.com/10.1007/978-3-540-30142-4_12 (visited on 05/27/2019).

[Koh14] Michael Kohlhase. “Mathematical Knowledge Management: Transcending the One-Brain-Barrier with Theory Graphs”. In: *EMS Newsletter* (June 2014), pp. 22–27. URL: https://kwarc.info/people/mkohlhase/papers/ems13.pdf.

[KR14] Cezary Kaliszyk and Florian Rabe. “Towards Knowledge Management for HOL Light”. In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics (Coimbra, Portugal, July 7, 2014–July 11, 2014). Ed. by Stephan Watt et al. LNCS 8543. Springer, 2014, pp. 357–372. ISBN: 978-3-319-08433-6. URL: http://kwarc.info/frabe/Research/KR_hollight_14.pdf.

[LZ08] Hui Liu and Bin Zhu. “Refactoring Formal Specifications in Object-Z”. In: *2008 International Conference on Computer Science and Software Engineering*. 2008 International Conference on Computer Science and Software Engineering. Wuhan, China: IEEE, 2008, pp. 342–345. ISBN: 978-0-7695-3336-0. DOI: 10.1109/CSSE.2008.260. URL: http://ieeexplore.ieee.org/document/4722066/ (visited on 05/21/2019).

[Mas96] John Mason. "Expressing Generality and Roots of Algebra". In: *Approaches to Algebra*. Ed. by Nadine Bernarz, Carolyn Kieran, and Lesley Lee. Dordrecht: Springer Netherlands, 1996, pp. 65–86. ISBN: 978-0-7923-4168-0 978-94-009-1732-3. DOI: 10.1007/978-94-009-1732-3_5. URL: http://www.springerlink.com/index/10.1007/978-94-009-1732-3_5 (visited on 04/04/2019).

[Mat17] Matt F. (https://mathoverflow.net/users/44143/matt-f). *Wiener's axiomatization of the group law based on division*. URL: https://mathoverflow.net/q/279161 (visited on 08/20/2017).

[MKR] Richard Marcus, Michael Kohlhase, and Florian Rabe. "TGView3D System Description: 3-Dimensional Visualization of Theory Graphs". URL: https://kwarc.info/kohlhase/submit/tgview3D.pdf.

[MKR18] Dennis Müller, Michael Kohlhase, and Florian Rabe. "Automatically Finding Theory Morphisms for Knowledge Management". In: *Intelligent Computer Mathematics*. Conferences on Intelligent Computer Mathematics. Ed. by Florian Rabe et al. LNAI 11006. Springer, 2018. ISBN: 978-3-319-96811-7. DOI: 10.1007/978-3-319-96812-4. URL: http://kwarc.info/kohlhase/papers/cicm18-viewfinder.pdf.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lüttich. "The Heterogeneous Tool Set, Hets". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Vol. 4424. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 519–522. ISBN: 978-3-540-71208-4 978-3-540-71209-1. DOI: 10.1007/978-3-540-71209-1_40. URL: http://link.springer.com/10.1007/978-3-540-71209-1_40 (visited on 06/17/2019).

[MMTa] Florian Rabe. *The MMT Surface Syntax*. URL: https://uniformal.github.io/doc/language/delimiters.html (visited on 03/19/2016).

[MMTb] *UniFormal/MMT – The MMT Language and System*. URL: https://github.com/UniFormal/MMT (visited on 10/24/2017).

[MN98] Richard Mayr and Tobias Nipkow. "Higher-order rewrite systems and their confluence". In: *Theoretical Computer Science* 192.1 (Feb. 1998), pp. 3–29. ISSN: 03043975. DOI: 10.1016/S0304-3975(97)00143-6. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397597001436 (visited on 06/11/2019).

[NK07] Immanuel Normann and Michael Kohlhase. "Extended Formula Normalization for ϵ-Retrieval and Sharing of Mathematical Knowledge". In: *MKM/Calculemus*. Ed. by Manuel Kauers et al. LNAI 4573. Springer Verlag, 2007, pp. 266–279. ISBN: 978-3-540-73083-5.

[OMT] Michael Kohlhase and Dennis Müller. *OMDoc/MMT Tutorial for Mathematicians*. URL: https://gl.mathhub.info/Tutorials/Mathematicians/blob/master/tutorial/mmt-math-tutorial.pdf (visited on 10/07/2017).

[Opd92] William F. Opdyke. "Refactoring Object-oriented Frameworks". PhD thesis. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1992.

[Pre97] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. Secaucus, NJ, USA: Birkhauser Boston, Inc., 1997. ISBN: 3-7643-4032-0.

[Rab14] Florian Rabe. "A Logic-Independent IDE". In: *Workshop on User Interfaces for Theorem Provers*. Ed. by Cristoph Benzmüller and Bruno Woltzenlogel Paleo. Elsevier, 2014, pp. 48–60. DOI: 10.4204/EPTCS.167.7.

[Rab15] Florian Rabe. "The Future of Logic: Foundation-Independence". In: *Logica Universalis* 10.1 (2015). 10.1007/s11787-015-0132-x; Winner of the Contest "The Future of Logic" at the World Congress on Universal Logic, pp. 1–20.

[Rab17] Florian Rabe. "How to Identify, Translate, and Combine Logics?" In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[Rab18] Florian Rabe. "MMT: A Foundation-Independent Logical Framework". Online Documentation. 2018. URL: https://kwarc.info/people/frabe/Research/rabe_mmtsys_18.pdf.

[RK13] Florian Rabe and Michael Kohlhase. "A Scalable Module System". In: *Information & Computation* 0.230 (2013), pp. 1–54. URL: http://kwarc.info/frabe/Research/mmt.pdf.

[RKM17] Marcel Rupprecht, Michael Kohlhase, and Dennis Müller. "A Flexible, Interactive Theory-Graph Viewer". In: *MathUI 2017: The 12th Workshop on Mathematical User Interfaces*. Ed. by Andrea Kohlhase and Marco Pollanen. 2017. URL: http://kwarc.info/kohlhase/papers/mathui17-tgview.pdf.

[RM18a] Florian Rabe and Dennis Müller. "Structuring Theories with Implicit Morphisms". In: *24th International Workshop on Algebraic Development Techniques 2018*. 2018. URL: https://kwarc.info/people/frabe/Research/RM_implicit_18.pdf.

[RM18b] Florian Rabe and Dennis Müller. "Structuring Theories with Implicit Morphisms". Extended Abstract. 2018. URL: http://wadt18.cs.rhul.ac.uk/submissions/WADT18A43.pdf.

[Sch+12] Max Schafer et al. "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs". In: *IEEE Transactions on Software Engineering* 38.6 (Nov. 2012), pp. 1233–1257. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.13. URL: http://ieeexplore.ieee.org/document/6152131/ (visited on 05/27/2019).

[SH02] Axel Schairer and Dieter Hutter. "Proof Transformations for Evolutionary Formal Software Development". In: *Algebraic Methodology and Software Technology*. Ed. by Hélène Kirchner and Christophe Ringeissen. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2422. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 441–456. ISBN: 978-3-540-44144-1 978-3-540-45719-0. DOI: 10.1007/3-540-45719-4_30. URL: http://link.springer.com/10.1007/3-540-45719-4_30 (visited on 05/21/2019).

[Sic+18] John Sichi et al. *JGraphT - Graph Algorithms and Data Structures in Java (Version 1.3.0)*. http://www.jgrapht.org. 2018.

[SK18] Satwinder Singh and Sharanpreet Kaur. "A systematic literature review: Refactoring for disclosing code smells in object oriented software". In: *Ain Shams Engineering Journal* 9.4 (Dec. 2018), pp. 2129–2151. ISSN: 20904479. DOI: 10.1016/j.asej.2017.03.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S2090447917300412 (visited on 05/20/2019).

[SM07] Friedrich Steimann and Philip Mayer. "Type Access Analysis: Towards Informed Interface Design." In: *The Journal of Object Technology* 6.9 (2007), p. 147. ISSN: 1660-1769. DOI: 10.5381/jot.2007.6.9.a8. URL: http://www.jot.fm/contents/issue_2007_10/paper8.html (visited on 04/05/2019).

[SMM06] Friedrich Steimann, Philip Mayer, and Andreas Meißner. "Decoupling classes with inferred interfaces". In: *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*. the 2006 ACM symposium. Dijon, France: ACM Press, 2006, p. 1404. ISBN: 978-1-59593-108-5. DOI: 10.1145/1141277.1141604. URL: http://portal.acm.org/citation.cfm?doid=1141277.1141604 (visited on 04/05/2019).

[SPT02] Susan Stepney, Fiona Polack, and Ian Toyn. "Refactoring in Maintenance and Development of Z Specifications and Proofs". In: *Electronic Notes in Theoretical Computer Science* 70.3 (Nov. 2002), pp. 50–69. ISSN: 15710661. DOI: 10.1016/S1571-0661(05)80485-2. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066105804852 (visited on 05/21/2019).

[SS04] Mirko Streckenbach and Gregor Snelting. "Refactoring class hierarchies with KABA". In: *ACM SIGPLAN Notices* 39.10 (Oct. 1, 2004), p. 315. ISSN: 03621340. DOI: 10.1145/1035292.1029003. URL: http://portal.acm.org/citation.cfm?doid=1035292.1029003 (visited on 04/05/2019).

[ST00] Gregor Snelting and Frank Tip. "Understanding class hierarchies using concept analysis". In: *ACM Transactions on Programming Languages and Systems* 22.3 (May 1, 2000), pp. 540–582. ISSN: 01640925. DOI:

10.1145/353926.353940. URL: http://portal.acm.org/citation.cfm?doid=353926.353940 (visited on 04/05/2019).

[Sun+01] Gerson Sunyé et al. "Refactoring UML Models". In: *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gogolla and Cris Kobryn. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2185. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 134–148. ISBN: 978-3-540-42667-7 978-3-540-45441-0. DOI: 10.1007/3-540-45441-1_11. URL: http://link.springer.com/10.1007/3-540-45441-1_11 (visited on 05/27/2019).

[ThySplit] *Roux_Navid – Navid Roux's Thesis Repository*. URL: https://gl.kwarc.info/supervision/roux_navid/tree/master/code/playground (visited on 06/13/2019).

[Tip07] Frank Tip. "Refactoring Using Type Constraints". In: *Static Analysis*. Ed. by Hanne Riis Nielson and Gilberto Filé. Springer Berlin Heidelberg, 2007, pp. 1–17. ISBN: 978-3-540-74061-2.

[TK92] Thierry Boy de la Tour and Christoph Kreitz. "Building proofs by analogy via the Curry-Howard Isomorphism". In: *Logic Programming and Automated Reasoning*. Ed. by Andrei Voronkov. Red. by G. Goos, J. Hartmanis, and Jörg Siekmann. Vol. 624. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 202–213. ISBN: 978-3-540-55727-2 978-3-540-47279-7. DOI: 10.1007/BFb0013062. URL: http://link.springer.com/10.1007/BFb0013062 (visited on 05/28/2019).

[Whi13] Iain Johnston Whiteside. "Refactoring Proofs". PhD thesis. Edinburgh: The University of Edinburgh, 2013. URL: https://www.era.lib.ed.ac.uk/handle/1842/7970?show=full.

[Wie02] Freek Wiedijk. *Encoding the HOL Light logic in Coq*. Aug. 29, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.9077.

[WK00] Christoph Walther and Thomas Kolbe. "Proving theorems by reuse". In: *Artificial Intelligence* 116.1 (Jan. 2000), pp. 17–66. ISSN: 00043702. DOI: 10.1016/S0004-3702(99)00096-X. URL: https://linkinghub.elsevier.com/retrieve/pii/S000437029900096X (visited on 05/27/2019).