Management of Change in Declarative Languages

by

Mihnea Iancu

a thesis for conferral of a Master of Science in Smart Systems

Dr. Florian Rabe

Name and title of first reviewer

Prof. Dr. Michael Kohlhase

Name and title of second reviewer

Date of submission: August 20, 2012

School of Engineering and Science

Declaration I hereby declare that, to the best of my knowledge, this work contains original and independent results, and that it has not been submitted at another university for the conferral of a degree.

This thesis is a significantly revised and extended version of the material in [IR12a] published together with Dr. Florian Rabe. All contributions from other sources are explicitly marked and acknowledged.

Mihnea Iancu
Bremen, August 20, 2012

Acknowledgments First and foremost I would like to express my gratitude to Dr. Florian Rabe for his continuous guidance, support and encouragement as well as for suggesting this research topic in the first place.

I am also very grateful to Prof. Dr. Michael Kohlhase for his insightful comments which served to broaden my perspective and deepen my understanding of the topic.

Finally, I would like to thank the members of the KWARC group whose ideas and remarks proved often fruitful and always interesting.

Abstract

Due to the high degree of interconnectedness of formal mathematical statements and theories, human authors often have difficulties anticipating and tracking the effects of a change in large bodies of symbolic mathematical knowledge. Therefore, the automation of change management is often desirable. But, while computers can in principle detect and propagate changes automatically, this process must take the semantics of the underlying mathematical formalism into account. Therefore, concrete management of change solutions are difficult to realize.

The MMT language was designed as a generic declarative language that captures universal structural features while avoiding a commitment to a particular formalism. Therefore, it provides a promising framework for the systematic study of changes in declarative languages. We leverage this framework by providing, at the MMT level, a generic change management solution which is flexible and expressive and can be instantiated for arbitrary specific languages.

Contents

1 Introduction **5**

2 Related Work **6**

3 Preliminaries **7**

3.1 The MMT Language 7

3.2 The LATIN Library 12

4 A Theory of Changes **13**

4.1 A Data Structure for Changes 13

4.2 A Formal Treatment of Transactions 19

4.3 A Data Structure for Dependencies 27

4.4 Reasoning on Change 32

5 A Generic Change Management API **41**

5.1 Implementation 41

5.2 Evaluation 43

6 Applications and Future Work **44**

7 Conclusion **46**

1 Introduction

Mathematical knowledge is growing at an enormous rate. Even if we restrict attention to formalized mathematics, libraries are reaching sizes that users have difficulties overseeing. Since this knowledge is also highly interconnected, it is getting increasingly difficult for humans to anticipate and follow the effects of changes. Therefore, management of change (MoC) has received attention recently as a means for supporting the evolution, revision and maintenance of large collections of mathematical documents.

In this thesis, we focus on change management for formalized mathematics, which — contrary to traditional, semi-formal mathematics — permits mechanically computing and verifying declarations. In principle, this should allow change management tools to automatically identify and recheck those declarations that are affected by a change. Thus, during the evolution of a knowledge library, MoC could help improve efficiency, ensure correctness and provide support for advanced refactoring operations. However, current computer algebra and deduction systems have not been designed systematically with change management in mind. In fact, the question of how to do that is still open.

We contribute to the solution of this problem by studying change management for the MMT language [RK11]. Because it was introduced as a foundation-independent, modular, and scalable representation language for formal mathematical knowledge, it is a promising framework for change management. Firstly, **foundation-independence** means that MMT avoids a syntactical or semantic commitment to any particular formalism. Thus, an MMT-based change management system could be applied to virtually any formal system. Secondly, **modularity** is a well-known strategy to rein in the impacts of changes and has been the basis of successful change management solutions such as [AHMS02]. Thirdly, MMT deemphasizes sequential in-memory processing of declarations in favor of maintaining a **large scale** network of declarations that are retrieved on demand, a crucial prerequisite for revisiting exactly the affected declarations.

Mathematically, our result can be described as follows. Firstly, we formally define changes between MMT documents and systematically analyze their properties with respect to change detection and application. Secondly, we introduce an generic, abstract notion of semantic dependency relations. Thirdly, we combine the two in order to propagate differences along the dependency relation in such a way that validity is preserved. We only present our results for a fragment of MMT, but our treatment extends to the full language. Our algorithms are currently implemented, within the MMT system [Rab08], for the full MMT language, thus providing a generic change management system for formal mathematical languages.

In section 2, we refine our problem statement and compare it to related work. In section 3.1, we briefly describe the MMT language in order to be self-contained and in section 3.2 we introduce the LATIN project which servers as an important

motivation for this work. Then we develop the theory of change management in MMT in section 4 and give an overview of our implementation in section 5. Finally, we discuss applications and future work in section 6 and conclude in section 7.

2 Related Work

MoC has been applied successfully in a number of **domains** such as software engineering (e.g., [EG89]) or file systems ([Apa00, CVS, Git]). A typical MoC work flow in this setting uses *compilation units*, e.g., the classes of a Java program: These are compiled independently, and a compilation manager can record the dependency relation between the units. In particular, if compilation units correspond to source files, changes in a file can be managed by recompiling all depending source files.

Intuitively, this work flow can be applied to declarative languages for mathematics as well if we replace "compilation" with "validation" where the latter includes, e.g., type reconstruction, rewriting, and theorem proving. However, there are a few key differences. Firstly, the validation units are individual types and definitions (which includes assertions and proofs in MMT), of which there are many per source file (around 50 on average in the Mizar library [TB85]). Their validation can be expensive, and there may be many dependencies within the same theory and many little theories in the same source file. Therefore, validation units cannot be mapped to files so that the notions of change and dependency must consider fragments of source files. Moreover, since foundations may employ search with backtracking, the validation of a unit U may access more units than the validity of U depends on. Therefore, the dependency relation should not be recorded by a generic MMT validation manager but produced by the foundation. Recently several systems have become able to produce such dependency relations, in particular Coq and Mizar [AMU11].

MoC systems for mathematical languages can be classified according to the **nature** of changes. In principle, any change in a declarative language can be expressed as a sequence of *add* and *delete* operations on declarations. But using additional change natures is important for scalability. We use *update*s to change the type or definiens of a declaration without changing its MMT URI. We do not use *reordering* operations because MMT already guarantees that the order of declarations has no effect on the semantics. More complex natures have been studied in [BC08], which uses *split*s in ontologies to replace one concept with two new ones. Dually, we could consider *merge* changes, which identify two declarations. Instead, we introduce *pragmatic changes*, which extend the change language and are constructed from adds, deletes and updates, as a generic way to treat such complex changes (e.g. a *rename* can be formed from a delete and an add).

Moreover, MoC systems can be classified by the **abstraction level** of their document model. The most concrete physical and bit level are relatively uninteresting

and, there, standard MoC tools operate at the character level treating documents as arrays of lines [Apa00, CVS, Git]. More abstract document models such as XML are better suited for mathematical content [AM10, Wag10] and have been applied to document formats for mathematics [Wag10, ADD+11]. Our work continues this development to more abstract document models by using MMT, which specifically models mathematical data structures. For example, the order of declarations, the flattening of imports, and the resolution of relative identifiers are opaque in XML but transparent in MMT representations. Moreover, MMT URIs are more suitable to identify the validation units than the XPath-based URLs usually employed in generic XML-based change models.

The development graph model [AHMS99], which has been applied to change management in [Hut00, AHMS02, Sch06], is very similar to MMT: Both are parametric in the underlying formal language, and both make the modular structure of mathematical theories explicit. The main difference is that MMT uses a concrete (but still generic) declarative language for mathematical theories; modular structure is represented using special declarations. Somewhat dually, development graphs use an abstract category of theories using diagrams to represent modular structure; the declarations within a theory can be represented by refining the abstract model as done in [AHM10].

At an even more abstract level, document models can be specific to one foundational language. While foundation-independent approaches like ours can only identify potentially affected validation units, those could determine and possibly repair the impact of a change. That would permit treating even subobjects as validation units. However, presently no systems exists that can provide such foundation-specific information so that such MoC systems remain future work.

Finally we can classify systems based on how they **propagate** changes. Our approach focuses on the theoretical aspect of identifying the (potentially) affected parts. The most natural post-processing steps are to revalidate them, as, e.g., in [AHMS02], or to present them for user interaction as in [ADD+11]. The MMT system can be easily adapted for either one. A very different treatment is advocated in [KK11] based on using only references that include revision numbers so that changes never affect other declarations (because each change generates a new revision number).

3 Preliminaries

3.1 The MMT Language

MMT [RK11] is a generic, formal module system for mathematical knowledge. The MMT language is designed to be applicable to a large collection of declarative formal base languages and all MMT notions are fully abstract in the choice of the

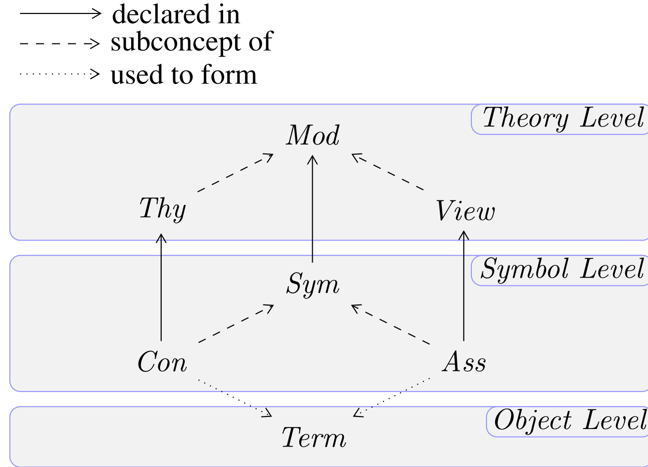

Figure 1: The MMT Ontology

base language.

Consequently, MMT focuses on foundation-independence, scalability and modularity.

MMT is meant to be applicable to all base languages based on *theories*. Relations between theories (theory morphisms) are represented as MMT *views*. Both theories and views form the MMT *module* level. A *theory morphism* $\overline{\sigma} : S \to T$ is a *signature morphism* $\sigma : S \to T$ interpreting all symbols of S in T but, in addition, $\overline{\sigma}$ translates all theorems of S to theorems of T. MMT uses the *Curry-Howard representation* to drop the distinction between symbols and axioms (and thus between signatures and theories). As a result, MMT needs only theories and theory morphisms. The MMT ontology is presented in figure 1.

The MMT Grammar We will only give a brief overview of MMT and refer to [RK11] for details. The fragment of the MMT grammar that we discuss in this thesis is given in figure 2. In particular, we have to omit the MMT module system for simplicity. The central notion is that of a **theory graph**, a list of modules, which are theories T or theory morphisms v.

A **theory** declaration $T = \{Sym^*\}$ introduces a theory with name T containing a list of symbol declarations. A **symbol** declaration $c : \omega = \omega'$ introduces a symbol named c with **type** ω and **definiens** ω'.

Notation 1. Both type and definiens are optional. However, in order to reduce the number of case distinctions, we use the special term $\bot$: If the type or definiens is omitted, we assume they are $\bot$.

Terms ω over a theory T are formed from constants $T?c$ declared in T, bound

Theory Graph	$\mathcal{G}$	$::=$	$\cdot \mid \mathcal{G},\ Mod$
Declaration	Dec	$::=$	$Mod \mid Sym$
Module Declaration	Mod	$::=$	$Thy \mid View$
Theory Declaration	Thy	$::=$	$T = \{Con^*\}$
View Declaration	$View$	$::=$	$v : T \to T = \{Ass^*\}$
Symbol Declaration	Sym	$::=$	$Con \mid Ass$
Constant Declaration	Con	$::=$	$c : \omega = \omega$
Assignment Declaration	Ass	$::=$	$c := \omega$
Term	ω	$::=$	$\bot \mid T?c \mid x \mid \omega\,\omega^+ \mid \omega X.\omega \mid \omega^v$
Variable Context	X	$::=$	$\cdot \mid X,\ x : \omega = \omega$
Module Identifier	M	$::=$	$T \mid v$
Theory Identifier	T	$::=$	MMT URI
Morphism Identifier	v	$::=$	MMT URI
Local Declaration Name	c	$::=$	MMT Name

Figure 2: Simplified MMT Grammar

variables x, application $\omega\,\omega_1\,\ldots\,\omega_n$ of a function ω to a sequence of arguments, bindings $\omega X.\omega'$ using a binder ω, a bound variable context X, and a scope ω', and morphism application ω^v. Except for morphism application, this is a fragment of the OPENMATH language [BCC+04], which can express virtually every object.

Theory morphism declarations (or views) $v : T \to T' = \{Ass^*\}$ introduce a morphism with name v from T to T' containing a list of assignment declarations. Such a morphism must contain exactly one assignment $c := \omega'$ for every undefined symbol $c : \omega = \bot$ in T and some term ω' over T'. Theory morphisms extend homomorphically to a mapping of T-terms to T' terms.

Intuitively, a theory morphism formalizes a translation between two formal languages. For example, the inclusion from the theory of semigroups to the theory of monoids (which extends the former with two declarations for the unit element and the neutrality axiom) can be formalized as a theory morphism. More complex examples are the Gödel-Gentzen negative translation from classical to intuitionistic logic or the interpretation of higher-order logic in set theory.

Notation 2. In order to have a fixed notation for symbols we unify the cases of constants in a theory and assignments in a morphism by using the following convention: A declaration $c := \omega'$ in a morphism v abbreviates a declaration $c : \omega = \omega'$ and the components `tp` and `def` are defined accordingly. The type ω is uniquely determined by MMT to ensure the type preservation of morphisms: Its value is τ^v where τ is the type of c in the domain of v. In this case, updates to assignments work in the same way as updates to symbols except that the component `tp` cannot be changed.

Every MMT declaration is identified by a canonical, globally unique URI. In par-

ticular, the URIs of symbol and assignment declarations are of the form $T?c$ and $v?c$.

MMT symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules. Their syntax and semantics is determined by the foundation, in which MMT is parametric. In particular, the validity of a theory graph is defined relative to a type system provided by the **foundation**:

Definition 1. *A* ***foundation*** *provides for every theory graph $\mathcal{G}$ a binary relation on terms that is preserved under morphism application. This relation is denoted by $\mathcal{G} \vdash \omega : \omega'$, i.e., we have $\mathcal{G} \vdash \omega : \omega'$ implies $\mathcal{G} \vdash \omega^v : \omega'^v$.*

Constant declarations $c : \omega = \omega'$ in a theory graph $\mathcal{G}$ are valid if $\mathcal{G} \vdash \omega' : \omega$. Thus, a foundation also has to define typing for the special term $\bot$: The judgment $\mathcal{G} \vdash \bot : \omega$ is interpreted as "ω is a well-typed universe, i.e., it is legal to declare constants with type ω". Similarly, $\mathcal{G} \vdash \omega : \bot$ means that ω may occur as the definiens of an untyped constant. This way the foundation can precisely control what symbol declarations are well-formed.

In order to talk efficiently about MMT theory graphs, we introduce a few definitions that permit looking up information in the theory graph:

Definition 2 (Lookup in Theory Graphs)**.** *For a theory graph $\mathcal{G}$, we write*

- $\vdash \mathcal{G}(M) = Mod$ *if a module declaration Mod with URI M is present in $\mathcal{G}$.*
- $\mathcal{G} \vdash T?c : \omega = \omega'$ *if T is a theory URI in $\mathcal{G}$ and the symbol declaration $c : \omega = \omega'$ exists in the body of T. We also define the corresponding notation for morphisms.*
- $\vdash \mathcal{G}(M?c) = Sym$ *if* $\vdash \mathcal{G}(M) = Mod$ *and Sym is the declaration with name c in the body of Mod.*
- $\vdash \mathcal{G}(M?c/o) = \omega$ *if* $\vdash \mathcal{G}(M?c) = Sym$ *and ω is the component o of Sym.*
- $\mathcal{G} \vdash u$ *if* $\vdash \mathcal{G}(u) = Dec$ *for some module or symbol declaration Dec, $\mathcal{G} \not\vdash u$ otherwise.*
- *if $\mathcal{G} \vdash u$, $\mathcal{G}(u)$ for the declaration in $\mathcal{G}$ with URI u and $\mathcal{G} \setminus u$ for the theory graph formed by removing $\mathcal{G}(u)$ from $\mathcal{G}$. The same notations function analogously for module bodies.*
- $T : S = \{b\}$ *for a theory T with meta S and body b and $v : S \to T : \{b\}$ for a view v from S to T with body b (to emphasize the constituent parts of a module).*

$$
\begin{aligned}
&PL = \{\\
&\quad bool : \mathtt{type} = \bot\\
&\quad \vee : bool \to bool \to bool = \bot\\
&\quad \wedge : bool \to bool \to bool = \bot\\
&\quad \Rightarrow : bool \to bool \to bool\\
&\qquad = \lambda x.\lambda y.y \vee (x \wedge y)\\
&\}
\end{aligned}
$$

(a) Theory PL

$$
\begin{aligned}
&NAT = \{\\
&\quad nat : \mathtt{type} = \bot\\
&\quad + : nat \to nat \to nat = \bot\\
&\quad * : nat \to nat \to nat = \bot\\
&\}
\end{aligned}
$$

(b) Theory NAT

$$
\begin{aligned}
&v : PL \to NAT = \{\\
&\quad bool := nat\\
&\quad \vee := +\\
&\quad \wedge := *\\
&\}
\end{aligned}
$$

(c) View $PL \to NAT$

Figure 3: The theory graph $\mathcal{G}_1$

Running Example 1. In figure 3 we present a simple MMT repository $\mathcal{G}_1$ representing an in-flux attempt to formalize propositional logic. $\mathcal{G}_1$ contains three modules, two theories (NAT and PL) and a view (v) between them.

For simplicity, we will assume that the MMT module system is used, that the symbols `type`, $\to$, and λ have been imported from a theory representing the logical framework LF, and that all theory graphs are validated relative to a fixed foundation for LF. Furthermore we assume declarations to have their usual notations and fixities.

With respect to MMT, the theory graph $\mathcal{G}_1$ is valid and the modules have the following content :

- NAT introduces a type nat of natural numbers and two binary operators $+$ and $*$.
- PL introduces a type $bool$ of formulas and three binary connectives ($\vee$, $\wedge$ and $\Rightarrow$), the last of which is defined in terms of the other two. Note that, while the definition for $\Rightarrow$ is valid (i.e. the type is correct), it is not adequate in the sense that it doesn't encode the expected, intuitive behavior of $\Rightarrow$ relative to $\wedge$ and $\vee$.
- v assigns $bool$ to nat, $\vee$ to $+$ and $\wedge$ to $*$.

MMT System The MMT System provides an API to the MMT data structures described above. The MMT API has operations for adding, validating and retrieving URI-identified knowledge items.

Retrieving knowledge items is done by taking an MMT URI and returning the MMT declaration identified by that URI.

With regard to validation we can distinguish three distinct strictness levels :

- XML-validation is quick but only guarantees that the input XML is well-formed
- Structural validation refers to the grammar in figure 2 and implements MMT-well formedness.
- Foundation-relative validation is carried out by plugins that check typing and equality constraints particular to each foundation.

There is an MMT implementation [Rab08, RK11] providing a Scala-based [OSV07] open source implementation of the MMT API. We will use (and extend) this implementation for achieving the change management services described in this thesis.

3.2 The LATIN Library

The LATIN project (**L**ogic **At**las and **In**tegrator) [KMR09, KMR] aims at developing tools for interfacing logics and proof systems. It focuses on developing a knowledge representation framework that is foundationally unconstrained and thus allows for the representation of most meta-theoretic foundations of mathematical knowledge in the same format.

The LATIN logic graph already contains work related to first-order logic (TPTP [SS98], CASL [ABKB+02, Mos04] and Mizar [TB85, UB06]), higher-order logic (PVS [ORS92], Isabelle/HOL [NPW02] and HasCASL [SM02]), logical frameworks (LF [Pfe91] and Isabelle [WBB+]) and mathematics (set theory, natural and rational numbers, algebraic structures and others). Furthermore, whenever possible, these representations are structured and related using logic morphisms.

The LATIN project is highly relevant to us for two reasons.

Firstly, a foundationally unconstrained knowledge representation framework allows us to discuss declarative languages in general and to develop our solutions generically. In this regard, we look at the MMT system, which is also used in the LATIN project. MMT is, by design, applicable to a large collection of declarative languages and parametric in its choice of foundation so we use the MMT language as a generic representation format for declarative languages and the MMT system as a basis for our implementation. Consequently, we discuss the various aspects of change management and design our solutions while considering their applicability to MMT structures and semantics.

Secondly, due to it's size, diversity and high level of modularity the LATIN logic graph can serve as a very useful case study for our change management solutions.

Furthermore, the level of heterogeneity with respect to the foundations encoded in it, allows for testing the genericity of our approach. It is important to note that the continued growth and development of the LATIN logic graph already highlighted the necessity for change management tools and, thus, represents an important motivation for this research.

4 A Theory of Changes

4.1 A Data Structure for Changes

Change Management can, in principle, be reduced to two components, detecting changes and reacting to changes, where the latter additionally requires accurate dependency information. Detecting changes, in the sense of only detecting *if* something was changed, is generally easy but appropriate reaction to those changes requires additional change semantics.

Firstly, the **change location** can be detected not only with respect to the syntax but also the semantics. In our MoC we can leverage the MMT document structure to localize changes relative to the MMT ontology. The makes our changes more complex (but also more useful) than line-based or file-based ones.

Secondly, the **change type** can be used to encode the expected semantic impact of each change and, in principle, the more fine grained the change taxonomy is the more precise the impact management can be. However, a large number of change types is not only difficult to manage but also challenging to establish rigidly, especially in the case of a generic MoC system. We solve this issue by defining a minimal core language of changes together with facilities that allow defining pragmatic extensions which map back to the core language.

4.1.1 Component Identifiers

Before we can develop our language of changes we need to enhance the MMT grammar given in section 3.1. Specifically, we add (or rather make explicit) another level of the MMT ontology to be able to refer to the components of individual declarations. Declaration components are relevant for MoC because they are the smallest units in the MMT ontology whose semantics is still given by MMT and not by the foundations and, therefore, are the smallest units that our generic MoC can accurately manage.

The components themselves are not new and the component identifiers simply make explicit the distinguished constituents of each declaration. At the module level, theories have a `meta` component identifier representing the (optional) meta theory while views have `from` and `to` representing references to the source and target theories. At the declaration level, constants have `tp` and `def` for the type

Component Identifier	O	$::=$	$T/o_{Thy} \mid v/o_{View} \mid T?c/o_{Con} \mid v?c/o_{Ass}$
Component	o	$::=$	$o_{Mod} \mid o_{Sym}$
Module Comp.	o_{Mod}	$::=$	$o_{Thy} \mid o_{View}$
Theory Comp.	o_{Thy}	$::=$	`meta`
View Comp.	o_{View}	$::=$	`from` | `to`
Symbol Comp.	o_{Sym}	$::=$	$o_{Con} \mid o_{Ass}$
Constant Comp.	o_{Con}	$::=$	`def` | `tp`
Assignment Comp.	o_{Ass}	$::=$	`def`

Figure 4: The Grammar for MMT Declaration Components

and definiens while assignments only have `def` since their type is automatically determined by MMT and not directly editable.

The grammar for components is given in figure 4.

4.1.2 Strict changes in MMT

Strict Changes We start by defining three primitive types of changes: adds ($\mathcal{A}$), deletes ($\mathcal{D}$) and updates ($\mathcal{U}$). While the necessity of adds and deletes is clear, updates may seem unnecessary because they can also be expressed as a pair of an add and a delete. However, we choose to make updates primitive in order to have our language of changes correspond with MMTs view on knowledge representation.

Specifically, URIs are a central notion in MMT and MMT modules or symbols can be seen as declarations indexed by URIs. Therefore, we consider MMT URIs to be central also to our MoC and define changes relative to URIs. Namely, upon comparing two documents D_1 and D_2, deletes represent URIs that are valid only in D_1, adds represent URIs that are valid only in D_2 and updates represent URIs that are valid in both D_1 and D_2. This has the added advantage that such a diff contains an unique reference to every changed MMT URI that is valid in D_1 or D_2.

We define the **strict MMT change language** based on the strict changes described above and the MMT grammar presented in figure 2. We use adds and deletes for the declaration level and updates for the object (component) level, basically recursing through the MMT document structure until we reach the components of declarations.

MMT documents have, at the declaration level, a fixed semantics given by the MMT language which induce semantic meaning to declaration level changes. Consequently, we define change types for each MMT declaration. However, at the object level, the semantics is foundation relative and is defined through the foundation plugins. This causes objects to appear as a black box to generic MMT services, in particular to our MoC system. Therefore we consider components as an atomic

Strict Diff	Δ^{s}	$::=$	$\cdot \mid \Delta^{\mathsf{s}} \bullet \delta^{\mathsf{s}}$
Strict Change	δ^{s}	$::=$	$\delta^{\mathsf{s}}_{Mod} \mid \delta^{\mathsf{s}}_{Sym}$
Symbol Change	$\delta^{\mathsf{s}}_{Sym}$	$::=$	$\delta^{\mathsf{s}}_{Con} \mid \delta^{\mathsf{s}}_{Ass}$
Constant Change	$\delta^{\mathsf{s}}_{Con}$	$::=$	$\mathcal{A}(T, c : \omega = \omega) \mid \mathcal{D}(T, c : \omega = \omega) \mid$ $\mathcal{U}(T, c, o_{Con}, \omega, \omega)$
Assignment Change	$\delta^{\mathsf{s}}_{Ass}$	$::=$	$\mathcal{A}(v, c := \omega) \mid \mathcal{D}(v, c := \omega) \mid$ $\mathcal{U}(v, c, o_{Ass}, \omega, \omega)$
Module Change	$\delta^{\mathsf{s}}_{Mod}$	$::=$	$\delta^{\mathsf{s}}_{Thy} \mid \delta^{\mathsf{s}}_{View}$
Theory Change	$\delta^{\mathsf{s}}_{Thy}$	$::=$	$\mathcal{A}(Thy) \mid \mathcal{D}(Thy) \mid \mathcal{U}(M, o_{Thy}, \omega, \omega)$
View Change	$\delta^{\mathsf{s}}_{View}$	$::=$	$\mathcal{A}(View) \mid \mathcal{D}(View) \mid \mathcal{U}(M, o_{View}, \omega, \omega)$

Figure 5: The Grammar for Strict MMT Changes

unit and we avoid recursing further into the document structure to identify changes inside terms.

Because components are parts of declarations and thus part of the MMT grammar, the individual role of each component is available at the generic MMT level. Therefore, using component identifiers in updates not only allows us to classify changes more precisely but also reason about changes more accurately by taking into account the function of each component. For instance, when a constant changes, we can differentiate between changes to the type ($o = \mathtt{tp}$) and changes to the definiens ($o = \mathtt{def}$). This is crucial because the definiens changes more often and is impacted by changes more often than the type of a declaration.

Note that in our implementation we locate changes even more precisely by recursing into the term level and identifying the exact subterms that changed. Nevertheless, as discussed above, change propagation is more practical if that information is not taken into account.

The grammar for our formal language of changes is given in figure 5.

The intuitive semantics of the change productions in figure 5 (with respect to an implicit theory graph $\mathcal{G}$) is as follows:

- $\mathcal{A}(Thy)$ **adds** a theory to the theory graph.
- $\mathcal{D}(Thy)$ **deletes** a theory from the theory graph.
- $\mathcal{U}(T, o, \omega, \omega')$ **updates** component o of theory T from ω to ω'.
- $\mathcal{A}(View)$ **adds** a view to the theory graph.
- $\mathcal{D}(View)$ **deletes** a view from the theory graph.
- $\mathcal{U}(v, o, \omega, \omega')$ **updates** component o of view v from ω to ω'.

- $\mathcal{A}(T, c : \omega = \omega')$ **adds** a constant declaration to the theory T.
- $\mathcal{D}(T, c : \omega = \omega')$ **deletes** a constant declaration from the theory T.
- $\mathcal{U}(T, c, o, \omega, \omega')$ **updates** component o of constant declaration $T?c$ from ω to ω'.
- $\mathcal{A}(v, c := \omega)$ **adds** a constant assignment to the view v.
- $\mathcal{D}(v, c := \omega)$ **deletes** a constant assignment from the view v.
- $\mathcal{U}(v, c, o, \omega, \omega')$ **updates** component o of constant assignment $T?c$ from ω to ω'.
- $\cdot$ does nothing.
- $\delta \bullet \Delta$ applies δ then Δ.

We can observe that the change productions for MMT are roughly obtained as the cartesian product between the add, delete, update change types discussed above and the MMT ontology. Thus, even with our minimalistic approach to change taxonomization we reach a rather high number of change productions. This highlights the importance of our simple approach, as any additional change type would lead to several new productions in our grammar and a more complex interaction between changes.

4.1.3 Pragmatic Changes in MMT

While simple, the strict changes give very little information regarding the semantics of the modifications in the context of the entire library. For instance, a delete of a declaration gives no information regarding what should happen to the declarations that depend on it so very little machine support can be provided for that beyond marking the affected declarations as invalid.

This is not the case for all possible change operations but, by reducing our change language to just three primitive types, we deconstruct all complex change operations to combinations of the primitive ones and lose their semantics. For example, in the case of a rename the propagation rule is clear, namely that all references to the renamed declaration should be replaced with the new name, but with our strict changes, renames would simply appear as an add and a delete.

Pragmatic Changes To obtain more expressivity in our change language we want to allow for the definition of new change types which we call pragmatic changes. Since we want to preserve the strict changes as a minimal standard version of our language we need the pragmatic changes to be reducible to sequences of strict changes. Furthermore, in order for the pragmatic changes to be truly more

expressive than the strict ones we want them to also encode information regarding their semantic impact. More precisely, we want to be able to predict the impact of pragmatic changes more accurately than for strict changes.

Therefore, we introduce a **change type constructor** which allows for the declaration of new pragmatic change types which express more complex changes as combinations of primitive changes. Since we desire the ability to correctly react to a change, we only permit the declaration of change types together with their propagation semantics (the changes that they entail to dependent knowledge items). In principle, the propagation semantics can be encoded as a function taking a knowledge item (representing the dependent item) and returning a strict diff (representing the changes that need to be applied to that item).

With respect to MMT, these knowledge items may be declarations (modules or symbols) or objects (terms).

In MMT, terms appear as components of declarations which correspond, in our strict change language, to updates. Therefore, their propagation semantics can be represented as a function $f_\omega : \omega \to \mathcal{U}$ because one update is enough to capture all the changes to one term. Moreover, since updates contain the old and new version of the term, the function can be simplified to $\omega \to \omega$ with an update being trivial to generate from this.

At the declaration level, we distinguish between symbol declarations and module declarations. In the case of symbol declarations (constants and assignments) all their components are fixed (e.g. a constant has exactly a `def` and a `tp`) so they can only be changed by changing their components (object level changes). Therefore, symbol level changes are already covered by object level ones.

Modules are different because they can be changed, not only by updating components, but also by adding or removing declarations. Consequently, we represent their propagation semantics accordingly as a function $f_{Mod} : Mod \to \Delta^{\mathsf{s}}$.

Following this intuition we first define pragmatic change types and then, based on that, pragmatic changes.

Definition 3. *A **pragmatic change type** is a tuple* $(N, P, F_\omega, F_{Mod})$ *where:*

- N *is a referable name.*
- P *is a predicate on sequences of primitive changes. If the* P *holds for a strict change sequence* Δ^{s} *it means the changes in* Δ^{s} *actually encode the semantic change* N*.*
- $F_\omega : \Delta^{\mathsf{s}} \to \omega \to \omega$ *is a function which represents the consequences of this change on impacted components.*
- $F_{Mod} : \Delta^{\mathsf{s}} \to Mod \to \Delta^{\mathsf{s}}$ *is a function which represents the consequences of this change on impacted modules.*

As we will see in section 4.4 F_ω is connected to foundational validity and F_{Mod} to structural validity.

Definition 4. *Given pragmatic change type* $(N, P, F_\omega, F_{Mod})$ *and a strict diff* Δ^{s} *such that* $P(\Delta^{\mathsf{s}})$ *holds, a* ***pragmatic change*** *is a pair of* N *and* Δ^{s} *(denoted* $N(\Delta^{\mathsf{s}})$*).*

Then, for a pragmatic change $N(\Delta^{\mathsf{s}})$ of type $(N, P, F_\omega, F_{Mod})$ its propagation semantics is given by $f_\omega = F_\omega(\Delta^{\mathsf{s}})$ and $f_{Mod} = F_{Mod}(\Delta^{\mathsf{s}})$.

The exact method how pragmatic changes are constructed is discussed in section 4.2 and their propagation mechanism is discussed in section 4.4.

With respect to change application, the intuitive application semantics of the production $N(\Delta^{\mathsf{s}})$ is the same as for Δ^{s}.

Running Example 2 (Continuing example 1). Consider the following definition for constant renames as a pragmatic change type.

- N is $rename$.
- P is the predicate such that $P(\Delta)$ holds if and only if Δ is of the form $\mathcal{D}(T, c : \omega_1 = \omega_2) \bullet \mathcal{A}(T, c' : \omega_1 = \omega_2)$.
- F_ω, when applied to Δ gives the function $f_\omega : \omega \to \omega$ which recurses into the given term and replaces all occurrences of $M?c$ with $M?c'$.
- F_{Mod} when applied to Δ gives the function $f_{Mod} : Mod \to \Delta^{\mathsf{s}}$ that renames assignments of $T?c$ from c to c' mirroring the rename of $T?c$ (with an add and a delete).

Example 1. Consider a foundation with primitives for datatype declaration and case based matching. Then, particular changes, such as adding or removing a case from a datatype definition, would have the precisely defined consequence of adding or removing a case everywhere that datatype is matched.

The foundation can then define, for instance the datatype remove change, using the change constructor as follows :

- N is $datatype\ remove$
- P is the predicate such that $P(\Delta)$ holds if and only if $\Delta = \mathcal{U}(M, c, o, \omega, \omega')$ and ω' and ω are both datatype declarations with ω having one (or more) fewer arguments.
- F_ω, when applied to Δ^{s} gives the function that takes a term ω and removes every occurence of the case distinction for each of the cases removed by Δ^{s}.
- F_{Mod}, when applied to Δ gives the trivial function returning the empty diff $\cdot$ for every module Mod.

Term	ω	$::=$	$\boxed{\omega}_\Delta \mid \boxed{\cdot}_\Delta \mid \bot \mid T?c \mid x \mid \omega\,\omega^+ \mid \omega X.\omega \mid \omega^v$
Variable Context	X	$::=$	$\cdot \mid X,\ x:\omega=\omega$

Figure 6: The Grammar for MMT Box Terms

Through pragmatics we allow for arbitrary mathematical extensions of our change language achieving a high level of power and flexibility. Examples 2 and 1 demonstrate two potential applications of pragmatic changes but our formalism can be used to capture many complex refactoring operations at both declaration and object level, such merge and split for declarations (symbols or modules) and alpha renaming for objects (terms). Therefore, change pragmatics lays the foundation for a **formal theory of refactoring**.

Pragmatic to Strict The translation from pragmatic to strict $\mathfrak{P}2\mathfrak{S}$ leaves strict changes as they are and unapplies purely pragmatic ones reducing them to the sequence of changes they encode.

More precisely $\mathfrak{P}2\mathfrak{S}$ is inductively defined as follows:

- $\mathfrak{P}2\mathfrak{S}(\cdot) = \cdot$
- $\mathfrak{P}2\mathfrak{S}(\delta^s \bullet \Delta) = \delta^s \bullet \mathfrak{P}2\mathfrak{S}(\Delta)$
- $\mathfrak{P}2\mathfrak{S}(N(\Delta^s) \bullet \Delta) = \Delta^s \bullet \mathfrak{P}2\mathfrak{S}(\Delta)$

4.1.4 Box Terms

We need one additional detail in our grammar: We add two special productions for terms ω, which we call **box terms**. The updated productions for terms are presented in figure 6. Box terms represent invalid terms that are introduced during change propagation.

$\boxed{\cdot}_\Delta$ represents a missing term caused by the sequence of changes in Δ. $\boxed{\omega}_\Delta$ represents a possibly invalid term ω caused by the sequence of changes in Δ. More sophisticated box terms can also record the required type, which gives users a hint what change is needed and permits applications to type-check a declaration relative to the box terms in it. We omit this here for simplicity.

4.2 A Formal Treatment of Transactions

Having formally defined an MMT change language in the previous section, we will now analyze the interaction between our chosen formalism and MMT theory graphs with respect to change detection and change application.

4.2.1 Change Detection

We see change detection as a two step process.

Firstly, we compare two theory graphs and automatically generate strict diffs representing the difference between them.

Secondly, we refine the strict diffs to pragmatic ones in a semi-automated process. Here, possible refinement steps are automatically detected and user input may be used to confirm their application.

Strict Change Detection Any change to an MMT theory graph can be described in terms of the strict changes we defined in section 4.1. We demonstrate this by providing a systematic way of computing changes between any two theory graphs and representing them as strict diffs.

We define a theory graph differencer $\mathfrak{Diff}$ that takes two theory graphs $\mathcal{G}$ and $\mathcal{G}'$ and computes the change sequence $\mathcal{G}' - \mathcal{G}$ representing the difference between them. $\mathfrak{Diff}\langle\mathcal{G} \mid \mathcal{G}'\rangle$ is defined inductively as :

- $\mathfrak{Diff}\langle\cdot \mid \cdot\rangle = \cdot$
- $\mathfrak{Diff}\langle\cdot \mid \mathcal{G}', M\rangle = \mathcal{A}(M) \bullet \mathfrak{Diff}\langle\cdot \mid \mathcal{G}'\rangle$
- $\mathfrak{Diff}\langle\mathcal{G}, M \mid \mathcal{G}'\rangle = \begin{cases} \mathcal{D}(M) \bullet \mathfrak{Diff}\langle\mathcal{G} \mid \mathcal{G}'\rangle & \text{if } \mathcal{G}' \not\vdash \pi(M) \\ \mathfrak{Diff}\langle M \mid \mathcal{G}'(\pi(M))\rangle \bullet \mathfrak{Diff}\langle\mathcal{G} \mid \mathcal{G}' \setminus \pi(M)\rangle & \text{if } \mathcal{G}' \vdash \pi(M) \end{cases}$
- $\mathfrak{Diff}\langle T : S = \{b\} \mid T : S' = \{b'\}\rangle = \mathfrak{Diff}\langle b \mid b'\rangle \bullet \begin{cases} \cdot & \text{if } S = S' \\ \mathcal{U}(T, \mathtt{meta}, S, S') & \text{if } S \neq S' \end{cases}$
- $\mathfrak{Diff}\langle v : S \to T = \{b\} \mid v : S' \to T' = \{b'\}\rangle = \mathfrak{Diff}\langle b \mid b'\rangle \bullet \begin{cases} \cdot & \text{if } S = S' \wedge T = T' \\ \mathcal{U}(v, \mathtt{from}, S, S') & \text{if } S \neq S' \wedge T = T' \\ \mathcal{U}(v, \mathtt{to}, T, T') & \text{if } S = S' \wedge T \neq T' \\ \mathcal{U}(v, \mathtt{from}, S, S') \bullet \mathcal{U}(v, c, \mathtt{to}, T, T') & \text{if } S \neq S' \wedge T \neq T' \end{cases}$
- $\mathfrak{Diff}\langle b, d \mid b'\rangle = \begin{cases} \mathcal{D}(d) \bullet \mathfrak{Diff}\langle b \mid b'\rangle & \text{if } b' \not\vdash d \\ \mathfrak{Diff}\langle d \mid b'(\pi(d))\rangle \bullet \mathfrak{Diff}\langle b \mid b' \setminus d\rangle & \text{if } b' \vdash \pi(d) \end{cases}$
- $\mathfrak{Diff}\langle\cdot \mid b', d\rangle = \mathcal{A}(d) \bullet \mathfrak{Diff}\langle\cdot \mid b'\rangle$

- $\mathfrak{Diff}\langle T?c : \omega_1 = \omega_2 \mid T?c : \omega_1' = \omega_2' \rangle =$
$$\begin{cases} \cdot & \text{if } \omega_1 = \omega_1' \wedge \omega_2 = \omega_2' \\ \mathcal{U}(T, c, \mathtt{tp}, \omega_1, \omega_1') & \text{if } \omega_1 \neq \omega_1' \wedge \omega_2 = \omega_2' \\ \mathcal{U}(T, c, \mathtt{def}, \omega_2, \omega_2') & \text{if } \omega_1 = \omega_1' \wedge \omega_2 \neq \omega_2' \\ \mathcal{U}(T, c, \mathtt{tp}, \omega_1, \omega_1') \bullet \mathcal{U}(T, c, \mathtt{def}, \omega_2, \omega_2') & \text{if } \omega_1 \neq \omega_1' \wedge \omega_2 \neq \omega_2' \end{cases}$$

- $\mathfrak{Diff}\langle v?c := \omega \mid v?c := \omega' \rangle =$
$$\begin{cases} \cdot & \text{if } \omega = \omega' \\ \mathcal{U}(v, c, \mathtt{def}, \omega, \omega') & \text{if } \omega = \omega' \end{cases}$$

The intuition behind the rigorous definition above is that the differencing algorithm views MMT theory graphs as URI-labeled trees up to declaration components. When differencing two theory graphs, $\mathfrak{Diff}$ traverses the respective trees checking which URIs exist in each of them and then separates URIs accordingly into old, new and preserved. Then, $\mathfrak{Diff}$ generate deletes for old URIs, adds for new ones and updates for those declaration components where the URI was preserved but the content has changed.

Running Example 3 (Continuing example 1). In figure 7 we present the theory graph $\mathcal{G}_2$ which represents an updated version of $\mathcal{G}_1$ from example 1.

More precisely, the author decides that *bool* is a bad choice of name for the type of formulas. Furthermore, noticing that the definition of $\Rightarrow$ doesn't work out, the author concludes that $\vee$ and $\wedge$ are a bad choice for PL primitives.

Therefore, compared to $\mathcal{G}_1$, in $\mathcal{G}_2$ $PL?bool$ is renamed to $PL?form$, $PL?\vee$ is deleted, and $PL?\neg$ is added. All other declarations remain unchanged, thus making the theory graph invalid.

We have $\mathfrak{Diff}\langle \mathcal{G}_1 \mid \mathcal{G}_2 \rangle = \Delta^{\mathsf{s}}$ where Δ^{s} is the strict diff: $\mathcal{D}(PL, bool : \mathtt{type} = \bot) \bullet \mathcal{A}(PL, form : \mathtt{type} = \bot) \bullet \mathcal{D}(PL, \vee : bool \to bool \to bool = \bot) \bullet \mathcal{A}(PL, \neg : form \to form = \bot)$.

Pragmatic Change Detection The theory graph differencer $\mathfrak{Diff}$ only generates strict changes, and thus strict diffs as a differencing result. We see pragmatic change detection as a distinct step in the change detection process. Pragmatic change detection acts on a pre-existing diff and replaces sets of strict changes with purely pragmatic changes.

Concretely, given a strict diff Δ^{s}, for any pragmatic change type $(N, P, F_\omega, F_{Mod})$ we generate the change sequences Δ^{s}_N that are sub-sequences of Δ^{s} and for which $P(\Delta^{\mathsf{s}}_N)$ holds. In other words, we compute all places where a change refinement step can be applied. At this point, the detected refinements could be applied automatically or user assistance could be required to confirm application of each refinement step.

$$
\begin{aligned}
&PL = \{\\
&\quad \mathit{form} : \mathtt{type} = \bot\\
&\quad \neg : \mathit{form} \to \mathit{form} = \bot\\
&\quad \wedge : \mathit{bool} \to \mathit{bool} \to \mathit{bool} = \bot\\
&\quad \Rightarrow : \mathit{bool} \to \mathit{bool} \to \mathit{bool}\\
&\qquad = \lambda x.\lambda y.y \vee (x \wedge y)\\
&\}
\end{aligned}
$$

(a) Theory PL

$$
\begin{aligned}
&NAT = \{\\
&\quad \mathit{nat} : \mathtt{type} = \bot\\
&\quad + : \mathit{nat} \to \mathit{nat} \to \mathit{nat} = \bot\\
&\quad * : \mathit{nat} \to \mathit{nat} \to \mathit{nat} = \bot\\
&\}
\end{aligned}
$$

(b) Theory NAT

$$
\begin{aligned}
&v : PL \to NAT = \{\\
&\quad \mathit{bool} := \mathit{nat}\\
&\quad \wedge := *\\
&\quad \vee := +\\
&\}
\end{aligned}
$$

(c) View $PL \to NAT$

Figure 7: The theory graph $\mathcal{G}_2$

There are several reasons to require user confirmation before generating a pragmatic change.

Firstly, the purpose of purely pragmatic changes is to encode the semantics of a change and use it for automatically propagating its consequences. Consequently, if change propagation is taken into account, generating a purely pragmatic change has consequences in the entire theory graph and so an incorrect such refinement would have global undesirable consequences. Therefore, we must ensure that each refinement step precisely captures the semantics intended by the user.

Secondly, since we allow arbitrary declarations of pragmatic change types, we cannot predict their interaction at the generic MoC level. Therefore we cannot assume that pragmatic changes are uniquely determined as we may have such changes that overlap in their application rules and, therefore, applying one would make the others inapplicable. This would cause any automated refinement process to be ambiguous with respect to the application order. While automatic disambiguation procedures, such as a precedence based system, can be imagined, we maintain that the correctness of this process can only be ensured with by requiring user confirmation.

Therefore, pragmatic diffs are constructed by (repeatedly) replacing matching sets of changes from a diff Δ with the corresponding pragmatic change.

Definition 5. *A diff Δ is called a* ***pragmatic refinement*** *of a strict diff Δ^s iff it can be obtained from Δ^s by constructing pragmatic changes as described above.*

Running Example 4 (Continuing examples 2 and 3). The first step of the computation of the difference $Rev_2 - Rev_1$ yields the diff Δ^s from example 3.

The next step is to refine the diff using $(\mathit{rename}, P, F_\omega, F_{Mod})$, the pragmatic

change for a rename from example 2.

We observe that $P(\Delta^{\mathsf{s}}_{\mathcal{R}})$ holds, where $\Delta^{\mathsf{s}}_{\mathcal{R}} = \mathcal{D}(PL, \mathit{bool} : \mathtt{type} = \bot) \bullet \mathcal{A}(PL, \mathit{form} : \mathtt{type} = \bot)$ is a sub-diff of Δ^{s}. Therefore, we can construct from it the pragmatic change $\mathit{rename}(\Delta^{\mathsf{s}}_{\mathcal{R}})$ representing a rename from $PL?\mathit{bool}$ to $PL?\mathit{form}$.

Finally, we get the pragmatic diff : $\Delta = \mathcal{D}(PL, \vee : \mathit{bool} \to \mathit{bool} \to \mathit{bool} = \bot) \bullet \mathcal{A}(PL, \neg : \mathit{form} \to \mathit{form} = \bot) \bullet \mathit{rename}(\Delta^{\mathsf{s}}_{\mathcal{R}})$

4.2.2 Change Application

Algebraically, the set of diffs Δ is the free monoid generated from changes δ. As we will see below, the operation of applying a diff to a theory graph can be regarded as this monoid acting on the set of theory graphs. In MoC tools, this process is sometimes called *patching*.

Same as for change detection we will treat the application of strict changes and pragmatic changes separately. We start by defining the application of strict changes and then extend it to the pragmatic case.

Strict Change Application We will now define the **application** of strict diffs Δ^{s} on theory graphs $\mathcal{G}$, which we denote by $\mathcal{G} \ll \Delta^{\mathsf{s}}$.

Clearly, it does not hold that any diff is applicable to any theory graph, but at the same time it also does not hold that a diff is only applicable to the theory graph based on which it was generated. Therefore, in order to adequately discuss change application we must first introduce the notion of applicability.

Definition 6. *A strict diff Δ^{s} is called **applicable** to the theory graph $\mathcal{G}$ (or $\mathcal{G}$-applicable) if $\mathcal{G} \vdash \Delta^{\mathsf{s}}$ according to the rules in figure 8.*

Intuitively a change or a diff is $\mathcal{G}$-applicable if and only if it can be applied to $\mathcal{G}$. Our definition yields a very weak notion of applicability in the sense that it only checks that the changes can be applied to the graph in accordance with their intuitive semantics. For instance, for a delete it checks that the deleted declaration exists in the theory graph, while for an add it checks that the used URI is valid and new to the graph. However, it doesn't check references or equality and typing constraints. Therefore, we allow for the case that the application of a $\mathcal{G}$-applicable diff will result in an inconsistent theory graph.

We now define, relative to a theory graph $\mathcal{G}$, the application of $\mathcal{G}$-applicable changes based on their intuitive application semantics discussed in section 4.1.

Definition 7 (Change Application)**.** *Given a theory graph $\mathcal{G}$ and a $\mathcal{G}$-applicable change δ, we define $\mathcal{G} \ll \delta$ as follows:*

- *If $\delta = \mathcal{A}(Mod)$ then $\mathcal{G} \ll \delta$ is the graph constructed from $\mathcal{G}$ by adding the module declaration Mod.*

$$\frac{\mathcal{G} \not\vdash \pi(Mod)}{\mathcal{G} \vdash \mathcal{A}(Mod)} \mathcal{A}_{mod} \qquad \frac{\mathcal{G} \vdash M \quad \mathcal{G} \not\vdash M?c}{\mathcal{G} \vdash \mathcal{A}(M, c : \omega = \omega')} \mathcal{A}_{sym}$$

$$\frac{\mathcal{G} \vdash Mod}{\mathcal{G} \vdash \mathcal{D}(Mod)} \mathcal{D}_{mod} \qquad \frac{\mathcal{G} \vdash M?c : \omega = \omega'}{\mathcal{G} \vdash \mathcal{D}(M, c : \omega = \omega')} \mathcal{D}_{sym}$$

$$\frac{\vdash \mathcal{G}(T/o) = \omega}{\mathcal{G} \vdash \mathcal{U}(T, o, \omega, \omega')} \mathcal{U}_{mod} \qquad \frac{\vdash \mathcal{G}(T?c/o) = \omega}{\mathcal{G} \vdash \mathcal{U}(T, c, o, \omega, \omega')} \mathcal{U}_{sym}$$

$$\frac{}{\mathcal{G} \vdash \cdot} \Delta^{\mathsf{s}}_{base} \qquad \frac{\mathcal{G} \vdash \Delta^{\mathsf{s}} \quad \mathcal{G} \ll \Delta^{\mathsf{s}} \vdash \delta}{\mathcal{G} \vdash \Delta^{\mathsf{s}} \bullet \delta} \Delta^{\mathsf{s}}_{dec}$$

Figure 8: Applicability of Changes

- *If* $\delta = \mathcal{D}(Mod)$ *then* $\mathcal{G} \ll \delta$ *is the graph constructed from* $\mathcal{G}$ *by deleting the module* Mod.
- *If* $\delta = \mathcal{U}(M, o, \omega, \omega')$ *then* $\mathcal{G} \ll \delta$ *is the graph constructed from* $\mathcal{G}$ *by updating the module component at* M/o *from* ω *to* ω'.
- *If* $\delta = \mathcal{A}(M, c : \omega = \omega')$ *then* $\mathcal{G} \ll \delta$ *is the graph constructed from* $\mathcal{G}$ *by adding the symbol declaration* $c : \omega = \omega'$ *to the module* M.
- *If* $\delta = \mathcal{D}(M, c : \omega = \omega')$ *then* $\mathcal{G} \ll \delta$ *is the graph constructed from* $\mathcal{G}$ *by deleting the symbol* $c : \omega = \omega'$ *from module* M.
- *If* $\delta = \mathcal{U}(M, c, o, \omega, \omega')$ *then* $\mathcal{G} \ll \delta$ *is the graph constructed from* $\mathcal{G}$ *by updating the symbol component at* $M?c/o$ *from* ω *to* ω'.

Moreover, we define $\mathcal{G} \ll \Delta$ *by* $\mathcal{G} \ll \cdot = \mathcal{G}$ *and* $\mathcal{G} \ll (\Delta \bullet \delta) = (\mathcal{G} \ll \Delta) \ll \delta$.

Pragmatic Change Application The notion of change applicability and change application for pragmatic changes comes naturally from the $\mathfrak{P}2\mathfrak{S}$ pragmatic to strict translation defined in section 4.1.

Definition 8. *Given a theory graph* $\mathcal{G}$*, a diff* Δ *is* ***applicable*** *(written* $\mathcal{G}$*-applicable) if and only if the strict diff* $\mathfrak{P}2\mathfrak{S}(\Delta)$ *is* $\mathcal{G}$*-applicable.*

Similarly, we can define the application of pragmatic diffs based on the application of their strict counterpart. The intuition is that pragmatic diffs exist as a higher

$\cdot^{-1}$	$=$	$\cdot$
$(\Delta \bullet \delta)^{-1}$	$=$	$\delta^{-1} \bullet \Delta^{-1}$
$\mathcal{A}(M, c : \omega = \omega')^{-1}$	$=$	$\mathcal{D}(M, c : \omega = \omega')$
$\mathcal{D}(M, c : \omega = \omega')^{-1}$	$=$	$\mathcal{A}(M, c : \omega = \omega')$
$\mathcal{U}(M, c, o, \omega, \omega')^{-1}$	$=$	$\mathcal{U}(M, c, o, \omega', \omega)$
$\mathcal{A}(Mod)^{-1}$	$=$	$\mathcal{D}(Mod)$
$\mathcal{D}(Mod)^{-1}$	$=$	$\mathcal{A}(Mod)$
$\mathcal{U}(M, o, \omega, \omega')^{-1}$	$=$	$\mathcal{U}(M, o, \omega', \omega)$

Figure 9: Inverses of Diffs and Changes

level semantic representation, similar to semantic annotations, and don't need to interact directly with the diff application service.

Definition 9 (Change Application)**.** *Given a theory $\mathcal{G}$ and a pragmatic diff Δ we define the application of Δ to $\mathcal{G}$ as $\mathcal{G} \ll \Delta = \mathcal{G} \ll \mathfrak{P}2\mathfrak{S}(\Delta)$.*

4.2.3 Formal Properties

Since every pragmatic change is reducible to a sequence of primitive changes and every pragmatic diff is reducible to a strict diff we can restrict attention to strict changes and strict diffs while analyzing their interaction with theory graphs.

Properties of Strict Diffs Having formally established change detection and change application we will now discuss the properties of our diffs with respect to their interaction with theory graphs.

Our diffs are **invertible** with respect to change application. This permits transactions where partially applied diffs are rolled back if they cause an error. It also makes it possible to easily offer undo-redo functionality in a user interface. The inverses are given in figure 9.

The following simple theorem permits **efficient lookup** in a hypothetical patched theory graph. This is important for scalability in the typical case where a large $\mathcal{G}$ should be neither changed nor copied:

Theorem 1. *Assume a theory graph $\mathcal{G}$ and a $\mathcal{G}$-applicable diff Δ. Then, for a*

symbol component path $M?c/o$:

$$\begin{array}{ll}
\vdash \ (\mathcal{G} \ll \cdot)(M?c/o) & = \mathcal{G}(M?c/o) \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M, c : \omega = \omega')))\,(M?c/o) & = \begin{cases} \omega & \text{if } o = \texttt{tp} \\ \omega' & \text{if } o = \texttt{def} \end{cases} \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M, c : \omega = \omega')))\,(M?c/o) & = \text{undefined} \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{U}(M, c, o, \omega, \omega')))\,(M?c/o) & = \omega' \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M = \{b\})))\,(M?c/o) & = b(c/o) \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M = \{b\})))\,(M?c/o) & = \text{undefined} \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet _))\,(M?c/o) & = (\mathcal{G} \ll \Delta)\,(M?c/o)
\end{array}$$

where $_$ is any change not covered by the previous cases. Similarly, for a module component path M/o :

$$\begin{array}{ll}
\vdash \ (\mathcal{G} \ll \cdot)(M/o) & = \mathcal{G}(M/o) \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M = \{b\})))\,(M/o) & = b(o) \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M = \{b\})))\,(M/o) & = \text{undefined} \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet \mathcal{U}(M, o, \omega, \omega')))\,(M/o) & = \omega' \\
\vdash \ (\mathcal{G} \ll (\Delta \bullet _))\,(M/o) & = (\mathcal{G} \ll \Delta)\,(M/o)
\end{array}$$

Proof. This is straightforward to prove using the definitions. □

We will now introduce and study an **equivalence relation** between diffs. Intuitively, two diffs are equivalent if their application has the same effect:

Definition 10. *Given a theory graph $\mathcal{G}$, two $\mathcal{G}$-applicable diffs Δ and Δ' are called $\mathcal{G}$-**equivalent** iff $\mathcal{G} \ll \Delta = \mathcal{G} \ll \Delta'$. We write this as $\Delta \equiv^{\mathcal{G}} \Delta'$.*

Our main theorem about change application is that diffs can be normalized. We need some auxiliary definitions first:

Definition 11. *The **referenced URI of a change** δ is defined as the URI that change affects. More precisely, for an add or delete is the path of the affected declaration, while for an update is the path of the affected component.*

We say that two MMT URIs clash if they refer to the same content element. Concretely, two MMT URIs clash if they are equal or if one is an ancestor of the other. For instance $T?c/\texttt{tp}$ has a path clash with $T?c$ or T but not with $T?c/\texttt{def}$. We extend this notion to changes, and we say that two changes have a path clash if their referenced URIs clash.

Definition 12. *A diff Δ is called **minimal** if there are no clashes between any two changes in Δ.*

Theorem 2. *Reordering the changes in a minimal diff yields an equivalent diff.*

Proof. In a minimal diff, each change affects a different declaration so the order of application is irrelevant. □

Theorem 2 yields our main result about minimal diffs. One we prove minimality of a diff, we ensure that application order is irrelevant and we can remove it from our concerns.

Theorem 3. *Given a theory graph $\mathcal{G}$, for any minimal $\mathcal{G}$-applicable strict diff it holds that* ***any pragmatic refinement yields a $\mathcal{G}$-equivalent diff****.*

Proof. The application of a pragmatic diff is defined as the application of its $\mathfrak{P}2\mathfrak{S}$ translation. But all refinement steps (strict to pragmatic translation) as well as the $\mathfrak{P}2\mathfrak{S}$ step (pragmatic to strict translation) preserve the individual changes. So the result of the $\mathfrak{P}2\mathfrak{S}$ translation will contain the same change as the initial diff, only possibly in a different order. Then the result follows immediately from Theorem 2. □

Theorem 4. *$\mathcal{G}' - \mathcal{G}$ is minimal, $\mathcal{G}$-applicable, and $\mathcal{G} \ll (\mathcal{G}' - \mathcal{G}) = \mathcal{G}'$.*

Proof. The proof is straightforward from the definition. □

Theorem 5. *If $\mathcal{G}' = \mathcal{G} \ll \Delta$, then there is a minimal diff Δ' such that $\Delta \equiv^{\mathcal{G}} \Delta'$.*

Proof. We put $\Delta' = (\mathcal{G} \ll \Delta) - \mathcal{G}$. Then the result follows from theorem 4. □

Relative to an existing theory graph $\mathcal{G}$ change detection can be thought as a function from the set of theory graphs to the set of diffs while change application can be thought as a function from the set of diffs to the set of theory graphs. The intuitive semantics of change detection and application then requires that the composition of these functions (in both directions) is the identity (up to $\mathcal{G}$-equivalence). Theorems 4 and 5 prove exactly that and can, therefore, be considered as **adequacy theorems** for our differencer $\mathfrak{Diff}$ and our diff application rules.

4.3 A Data Structure for Dependencies

Making the dependencies between knowledge items explicit is a central prerequisite for analyzing the impact of a change. Since impact analysis is tied to the validity of theory graphs, we have to introduce a formal notion of validity before we can introduce a formal notion of dependency.

4.3.1 Validity of theory graphs

Since MMT is a framework, validity of theory graphs involves not only checking with respect to the MMT grammar, but also with respect to declared foundations (via plugins). In addition, MMT offers support for presentation to content services that interpret presentation oriented representation into formal MMT documents. Therefore, we distinguish three levels of validity checking for theory graphs and discuss them individually. We will not focus on the validation itself but rather on those particular constraints that are non-local in the sense that a change in one part of the graph can cause another part to become invalid. These constraints are important for MoC because they induce a notion of dependency that we must capture in our formalism.

Structural Validity At the initial level of checking we define validity as MMT well-formedness ignoring object level notions that require a foundational commitment such as typing or equality constraints. Therefore, structural validation checks that terms, declarations and URIs are well-formed with respect to the MMT grammar. In particular, structural validity checks the totality of morphisms which requires that every morphisms $v : T \to S$ has exactly one assignment for each undefined constant in T. This structural constraint is important for our MoC because it is the only non-local one thus inducing a notion of (structural) dependency.

Foundational Validity Because MMT is foundation-independent, we must also define validity relative to a foundation. Intuitively, a theory graph is valid if every declaration is well-typed.

Definition 13. *A theory graph $\mathcal{G}$ is called* ***foundationally valid*** *if the following hold:*

- *for all symbol declarations $\mathcal{G} \vdash T?c : \omega = \omega'$ we have $\mathcal{G} \vdash \bot : \omega$ and $\mathcal{G} \vdash \omega' : \omega$*
- *for all assignment declarations $\mathcal{G} \vdash v?c := \omega$ where $v : S \to T$ and $\mathcal{G} \vdash T?c : \omega' = \bot$ we have $\mathcal{G} \vdash \omega'^{v} : \omega$.*

We will now extent the notion of foundational validity to individual paths in a theory graph so that we can precisely refer to the paths that break the validity of a theory graph.

Definition 14. *A* MMT *URI π is called foundationally valid with respect to $\mathcal{G}$ (written $\mathcal{G} \vdash^f \pi$) iff :*

$$\begin{cases} \mathcal{G} \vdash \bot : \mathcal{G}(T?c/\texttt{tp}) & \textit{if } \pi = T?c/\texttt{tp} \\ \mathcal{G} \vdash \mathcal{G}(T?c/\texttt{def}) : \mathcal{G}(T?c/\texttt{tp}) & \textit{if } \pi = T?c/\texttt{def} \\ \mathcal{G} \vdash \mathcal{G}(v?c/\texttt{def}) : \mathcal{G}(S?c/\texttt{tp})^v & \textit{if } \pi = v?c/\texttt{def} \textit{ and } v : S \to T \\ \mathcal{G} \vdash T/\texttt{meta} & \textit{if } \pi = T/\texttt{meta} \\ \mathcal{G} \vdash v/\texttt{from} & \textit{if } \pi = v/\texttt{from} \\ \mathcal{G} \vdash v/\texttt{to} & \textit{if } \pi = v/\texttt{to} \\ \mathcal{G} \vdash^f M?c/o\ \forall o.\mathcal{G} \vdash M?c/o & \textit{if } \pi = M?c \\ \mathcal{G} \vdash^f M/o\ \forall o.\mathcal{G} \vdash M/o \textit{ and} & \\ \mathcal{G} \vdash^f M?c\ \forall c.\mathcal{G} \vdash M?c & \textit{if } \pi \textit{ is a module declaration} \end{cases}$$

Definition 15. *Given a theory graph $\mathcal{G}$ and a $\mathcal{G}$-applicable diff Δ we say Δ is a foundationally valid diff with respect to $\mathcal{G}$ if all paths referenced in Δ are foundationally valid in $\mathcal{G} \ll \Delta$.*

Foundationally valid diffs are important because they do not directly introduce new invalid paths to the graph. They can negatively affect validity due to propagation, but the actually changed declarations are valid. This is naturally an important prerequisite for change propagation, because if the changes are invalid they should not be propagated anyway.

Now we introduce a definition to further discuss how foundational validity is affected by changes. A typical property of typing relations is that they satisfy a weakening property: Additional information can not invalidate a type inference. Formally, we can state this as:

Definition 16. *A foundation is called **monotonous** if the following rules are admissible for any $A = \mathcal{A}(M, c : _ = _)$ and for any $U = \mathcal{U}(M, c, o, \bot, _)$:*

$$\frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash A}{\mathcal{G} \ll A \vdash \omega : \omega'} \qquad \frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash U}{\mathcal{G} \ll U \vdash \omega : \omega'}$$

Almost all practical foundations for MMT are monotonous. This includes even substructural type theories like linear LF [CP02] because we only require weakening for the set of global declarations, not for local contexts. A simple counter-example is a type theory with induction in which constructors can be added as individual declarations: Then adding a constructor will break an existing induction. But most type theories introduce all constructors in the same declaration.

Change management can leverage monotony in combination with dependency relations: If it is known what declarations a declaration (may) depend on, only these can affect its validity.

Interpretation-relative validity An interpretor is a service which transforms presentation-oriented representations into MMTs concrete content-oriented XML

syntax. Such services may just offer only a parsing facility or may have more complex features such as automatic inference of omitted types or even (parts of) proofs. Then, a change in one declarations may affect the interpretation of other declarations without invalidating the actual content, thus inducing a notion of dependency.

For instance, an added declaration in an included theory could make interpretation of unqualified names fail if some name lookup becomes ambiguous (i.e. if the added declaration has the same unqualified name as another included declaration). However, this would not affect the content-based representation where fully qualified URIs will remain unambiguous. Still, from the perspective of a user that makes use of that interpretation service to interact with MMT this change consequence is important so automated detection is preferred. Clearly, re-interpreting the whole library on every change is not feasible so dependency information is needed.

4.3.2 Dependency relations

We can now define a dependency relation in MMT relative to a validation service $\mathfrak{S}$ which may be structural validation, foundational validation or the validation of an interpretation service.

Definition 17. *A* ***dependency relation*** *for a theory graph $\mathcal{G}$ and a validation service $\mathfrak{S}$ is a binary relation $\looparrowright^{\mathfrak{S}}$ between* MMT *URIs π_1 and π_2 such that the following rule is admissible:*

$$\frac{\mathcal{G} \vdash \delta \qquad \vdash \pi(\delta) \in \pi_1 \qquad \pi_1 \not\looparrowright^{\mathfrak{S}} \pi_2}{\mathcal{G} \ll \delta \vdash^{\mathfrak{S}} \pi_2}$$

where $\mathcal{G} \vdash^{\mathfrak{S}} \pi$ means that the declaration at path π in $\mathcal{G}$ is valid according to $\mathfrak{S}$.

The intuition behind definition 17 is that if an item doesn't depend on another then changing the latter will not affect the former. We choose to define the dependency relation relative to the negative impact relation rather than the impact relation to make our formalization more general. In a fully formal setting where the semantics of the declarations is fully specified the "impacts" and "does not impact" relations are duals. However, in a semi-formal setting, the coverage is not exhaustive and there can be declarations between which the dependency status is unknown. In that settings our definition of a dependency relation is based on the dual of the "does not impact" relation.

Moreover, note that dependency relations are not necessarily transitive. That way changes can be propagated one dependency step at a time, and intermediate revalidation can show that whether or not further propagation is necessary. Of course, the transitive closure (in fact: any larger relation) is again a dependency relation.

Another important observation is that, according to our definition, a dependency relation is between any two MMT URIs. However, our change language and differencing algorithm does not directly mention which URIs were changed. More

precisely, changes are not automatically propagated upwards through the document structure, from contained item to its container. For instance, an updated component induces an implicit change to its containing declaration, and an added or deleted symbol declaration induces an implicit change to its containing module.

Actually computing these induced changes is trivial given the simplicity of MMTs document structure, but we define our change language as is because our main focus here is foundational validity and, therefore, **foundational dependencies**. Foundational dependencies are dependencies that are generated by foundational validation and it holds that all foundational dependency relations are between components (given some reasonable assumptions about foundations).

Hence, we can specialize our generic definition of a dependency relation to the case of foundational dependencies.

Definition 18. *A* ***foundational dependency relation*** *for a theory graph* $\mathcal{G}$ *is a binary relation* $\looparrowright^f$ *between declaration components* $M?c/o$ *and* $M'?c'/o'$ *such that the following rule is admissible:*

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c'/o' \quad M?c/o \not\looparrowright^f M'?c'/o'}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \perp) \vdash^f M'?c'/o'}$$

This definition of a dependency relation was inspired by the one in [RKS11].

The intuition of definition 18 is the same as for definition 17 except this one pinpoints changes more precisely, by only considering dependencies between component paths.

The purpose of foundational dependencies is to keep track of dependencies introduced by the foundation plugins. This is important because leveraging dependency information is the only precise way to identify the consequences of changes. Revalidating the entire knowledge library is another possibility but it is not only much slower and computationally intensive but also inaccurate. It is possible for the semantics of a declaration to be altered by a change to one of its dependencies, without making the declaration invalid. In fact this is quite common, for instance, in the case of alias-type declarations where a declaration is directly defined in terms of another. Then the first will remain valid as long as the latter is valid, but its meaning may change beyond the intentions of the author. Therefore, we maintain that dependency information is the only accurate predictor of change propagation in the context of foundational validity.

We now introduce some definitions in order to be able to talk more precisely about impacts and dependencies.

Definition 19. *We define the* ***impacts*** *of a path* π *with respect to a validation service* $\mathfrak{S}$ *as the set of paths* π' *such that we have* $\pi \looparrowright^{\mathfrak{S}} \pi'$.

Consequently, define the ***foundational impacts*** *of a component path* $M?c/o$ *as the set of component paths* $M'?c'/o'$ *such that we have* $M?c/o \looparrowright^f M'?c'/o'$.

Definition 20. *We define the impacts of a change δ or sequence of changes as follows :*

- *if $\delta = \mathcal{U}(M, c, o, \omega, \omega')$ the union of the impacts of M, $M?c$ and $M?c/o$*
- *if $\delta = \mathcal{D}(T, c : \omega = \omega')$ the union of the impacts of T, $T?c$, $T?c/\texttt{tp}$ and $T?c/\texttt{def}$*
- *if $\delta = \mathcal{D}(v, c := \omega')$ the union of the impacts of v, $v?c$ and $v?c/\texttt{def}$*
- *if $\delta = \mathcal{D}(Mod)$ then it is the union of the impacts of $\pi(Mod)$ and all included symbol declarations*
- *if $\delta = \mathcal{A}(Dec)$ then it is $\emptyset$ since an add implies that the URI of the new declaration is fresh in the theory graph and, therefore, that there are no dependencies for it.*

This determines the following definition for foundational impacts :

Definition 21. *We define the impacts of a change δ or sequence of changes as follows :*

- *if $\delta = \mathcal{U}(M, c, o, \omega, \omega')$ the foundational impacts of $M?c/o$*
- *if $\delta = \mathcal{D}(T, c : \omega = \omega')$ the union of the foundational impacts of $T?c/\texttt{tp}$ and $T?c/\texttt{def}$*
- *if $\delta = \mathcal{D}(v, c := \omega')$ the union of the foundational impacts of $v?c/\texttt{def}$*
- *if $\delta = \mathcal{D}(Mod)$ then it is the union of the foundational impacts of all included declarations*
- *if $\delta = \mathcal{A}(Dec)$ then it is $\emptyset$ since an add implies that the URI of the new declaration is fresh in the theory graph and, therefore, that there are no dependencies for it.*

Running Example 5 (Continuing example 1). For the theory graph Rev_1, we obtain a foundational dependency relation by assuming a dependency whenever a constant occurs in a component. We also assume a dependency from each definiens to its type. The graph in figure 10 illustrates this relation.

4.4 Reasoning on Change

As discussed in the previous sections, any change has the potential of generating consequences in other parts of the library. In practice this makes editing in an interconnected library a two step process. After the initial edits, generating a change or set of changes, there is a **conflict resolution** step where the impacts are detected

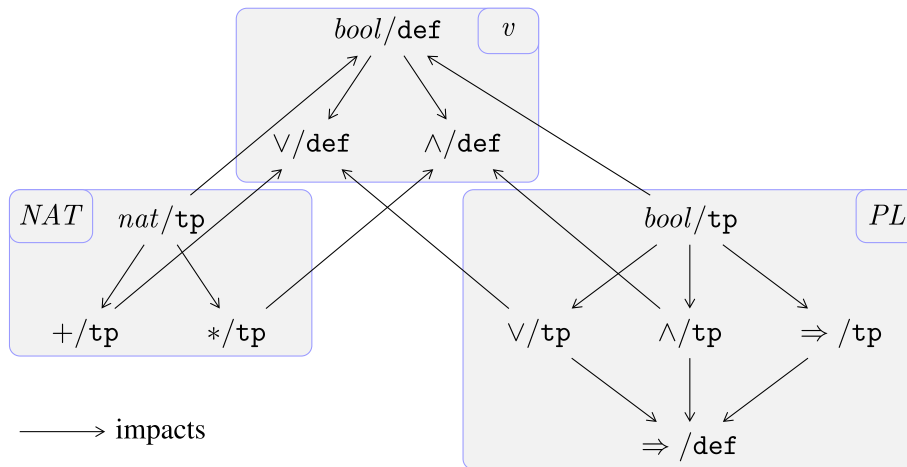

Figure 10: Foundational impact tree for Rev_1

and fixed. The aim of our change propagation is to make the change resolution process as easy and as automatic as possible. This involves detecting and marking the impacts of changes and, where possible, fix the conflicts automatically.

We distinguish two kinds of situations where reasoning on change is necessary, based on the relation to our formalization. Specifically, we classify situations considering whether we need to propagate only one change at a time or a sequence of changes.

These two cases are fundamentally different with respect to our formalization because, when propagating a transaction, several distinct changes may impact the same component causing several impact propagation functions to be applied to the same term. This requires handling the ambiguity introduces by the undefined order of the changes, as the result may be different depending on the application order. Therefore, propagating one change is significantly simpler than propagating an entire transaction.

With respect to use cases, the two situations we identified above are again distinct. Propagating one change at a time is rarely possible, except within an IDE with very tight integration with MMT, which would be capable to react "on edit" to the changes introduced by the author. Even in this case, propagating after every minor edit may be undesirable and distracting. A notable exception here are pragmatic changes which, as discussed in the previous sections, require user confirmation to be composed from a sequence of strict changes. Therefore, they can always be propagated one at a time, immediately after they are created. With respect to the second case, propagating transactions is useful in both IDE environments and in versioned repositories. Still, *if* pragmatic change are always propagated on the fly, transactions will *only contain strict changes*.

So, based on these use cases, we have two main concerns with respect to change propagation. On the one hand, propagating a single purely pragmatic change and, on the other hand, propagating an entire sequence of strict changes at once. These two feature requirements guide the design of the definitions and theorems concerned with change propagation. However, for the sake of flexibility of our system, we make our formalization generic enough to allow complete customization. In particular, we formally define propagation for any single change and for any sequence of changes.

We will first analyze propagating a single change, and then extend our coverage to transactions (sequence of changes).

Change Impact Propagation Our change language already achieves a great degree of flexibility through pragmatic changes. However, with respect to propagation, we can provide some extra configuration to cover additional use cases.

Firstly, while the propagation for pragmatic changes is already defined the one for strict changes is not and, therefore, it can (and should) be customizable. However, adds don't have any impacts anyway because they contain new declarations, so propagation functions only make sense for updates and deletes.

Secondly, authors rarely work on the entire library all at once, but rather do incremental work on a section until it is finished and then focus on integration with the rest of the library.

To accommodate these requirements we add the following definition:

Definition 22. *A change impact propagator configuration is a triple $\langle f_{\mathcal{D}}, f_{\mathcal{U}}, P \rangle$ where $f_{\mathcal{D}} : \omega \to \omega$ is the propagation function for deletes, $f_{\mathcal{U}} : \omega \to \omega$ is the propagation function for updates and P is a predicate on library paths representing the subset of the current repository where changes should be propagated.*

Note that in the following, if P is omitted, we assume it to be the tautological predicate meaning that propagation covers the entire theory graph.

With respect to validity, the notion of change propagation in MMT can be split into three distinct stages. As discussed in section 3.1 validation of MMT documents comes in three steps. First XML validation then structural validation and finally foundational validation. Obviously XML validation doesn't require any propagation but structural and foundational both do. In addition, we have validation done by external interpretors which convert presentation oriented syntax into semantic MMT documents. While the latter is not directly part of MMT it would be an important improvement to usability if our MoC can detect and mark conflicts introduced in the interpretors because that is how authors interact with MMT.

Relative to a theory graph, the condition of diff validity already guarantees that its changes contain (structurally and foundationally) valid terms so applying the diff will preserve validity locally (for the URIs it affects). Therefore, as discussed

in section 4.3 we are interested only in those constraints that are non-local, in the sense that a change to a declaration in one part of the graph can break a declaration in another part.

Structural Impact Propagation The only non-local constraint of structural validity is the totality of morphisms so structural impact propagation only needs to ensure that every morphism has exactly one assignment for each undefined constant in the domain theory.

Definition 23. *Given a theory graph $\mathcal{G}$ we define **structural impact propagation** for a $\mathcal{G}$-applicable diff Δ (denoted $\mathcal{SIP}(\mathcal{G},\Delta)$) is a diff constructed as follows :*

- *if $\delta = \mathcal{D}(Thy)$ then $\mathcal{SIP}(\mathcal{G},\delta)$ is the diff containing for each view $View$ in $\mathcal{G}$ with domain Thy the change $\mathcal{D}(View)$*
- *if $\delta = \mathcal{D}(T?c : \omega = \bot)$ or $\delta = \mathcal{U}(T?c, \mathtt{def}, \bot, \omega)$ then $\mathcal{SIP}(\mathcal{G},\delta)$ is the diff containing a $\mathcal{D}(v?c : \omega)$ for each view $v : T \to T'$ in $\mathcal{G}$*
- *if $\delta = \mathcal{A}(T?c : \omega = \bot)$ or $\delta = \mathcal{U}(T?c, \mathtt{def}, \omega, \bot)$ then $\mathcal{SIP}(\mathcal{G},\delta)$ is the diff containing a $\mathcal{A}(v?c : \boxed{\cdot}_{\mathcal{A}(T?c:\omega=\bot)})$ for each view $v : T \to T'$ in $\mathcal{G}$*
- *if $\delta = N(\Delta^{\mathsf{s}})$ where $(N, P, F_\omega, F_{Mod})$ is a pragmatic change type then $\mathcal{SIP}(\mathcal{G},\delta)$ is the diff composed from the diffs $F_{Mod}(\Delta^{\mathsf{s}})(Mod)$ for all modules Mod in $\mathcal{G}$ structurally impacted by Δ^{s}.*
- *if δ none of the above then $\mathcal{SIP}(\mathcal{G},\delta) = \cdot$*
- $\mathcal{SIP}(\mathcal{G},\cdot) = \cdot$
- $\mathcal{SIP}(\mathcal{G},\delta \bullet \Delta)\ \mathcal{SIP}(\mathcal{G},\delta) \bullet \mathcal{SIP}(\mathcal{G},\Delta)$

Theorem 6. *Given a valid theory graph $\mathcal{G}$, if Δ is a $\mathcal{G}$-applicable minimal diff, then the $\mathcal{SIP}(\mathcal{G},\Delta)$ is also $\mathcal{G}$-applicable and minimal. Moreover $\mathcal{G} \ll \Delta \bullet \mathcal{SIP}(\mathcal{G},\Delta)$ is structurally valid.*

Proof. Follows immediately from the definitions of structural impact propagation and minimal diffs. □

Foundational Impact Propagation Foundational validity is defined for each foundation via a MMT plugin that implements the semantics of that foundation. This means that the semantics given by the foundation is unavailable at the generic MMT level. Hence, foundational impact propagation is more difficult than the structural one because we need to account for all possible interactions between changes and their impacts.

Moreover, since the semantics is not available at the generic level, fixing the foundational conflicts introduced by changes is only possible for the cases where the impact propagation functions are already declared by users (e.g. for purely pragmatic changes). Therefore we define a default impact propagation function for strict changes which only marks the terms as impacted by wrapping them in a box term containing the old value of the term and the change that causes the conflict.

Formally, we define the default impact propagation function for strict changes f^{s} as follows :

- if ω is a box term $\boxed{\omega'}_{\Delta}$ then $f^{\mathsf{s}}(\delta)(\omega) = \boxed{\omega'}_{\Delta \bullet \delta}$
- if ω is not a box term then $f^{\mathsf{s}}(\delta)(\omega) = \boxed{\omega}_{\delta}$

It is clear that the terms which result from applying f^{s} are always foundationally valid, since box terms are by definition foundationally valid.

Now we can use f^{s} to define foundational impact propagation for a single change, while ensuring the preservation of validity. Note that since adds, by definition, introduce new declarations to a theory graph, there cannot be any foundational dependencies for them. Therefore, we skip adds when presenting our formalization for foundational impact propagation.

Definition 24. *Given theory graph $\mathcal{G}$, an propagator configuration $\langle f_{\mathcal{D}}, f_{\mathcal{U}}, P\rangle$ and a $\mathcal{G}$-applicable change δ, the **foundational impact propagation** of δ (denoted $\mathcal{FIP}(\mathcal{G}, \delta)$) is a diff generated as follows :*

First, let l be the list of all component paths $M?c/o$ in the foundational impact of δ (relative to the subgraph determined by P), ordered in their dependency order. The propagation function f is

- *if δ is a pragmatic change $N(\Delta)$ where $(N, P, F_{\omega}, F_{Mod})$ then $f = F_{\omega}(\Delta)$*
- *if δ is a delete then $f = f_{\mathcal{D}}(\delta)$*
- *if δ is an update then $f = f_{\mathcal{U}}(\delta)$*

Then $\mathcal{FIP}(\mathcal{G}, \delta)$ is $\mathcal{FIP}(\mathcal{G}, l, f)$ as defined in the following :

- $\mathcal{FIP}(\mathcal{G}, \cdot, f) = \cdot$
- $\mathcal{FIP}(\mathcal{G}, \langle M?c/o, l'\rangle, f) = \mathcal{U}(M, c, o, \omega, \omega') \bullet$
  $\mathcal{FIP}(\mathcal{G} \ll \mathcal{U}(M, c, o, \omega, \omega'), l', f)$ *where* $\omega = \mathcal{G}(M?c/o)$ *and*
  $\omega' = \begin{cases} f(\omega) & \text{if } \mathcal{G} \ll \mathcal{U}(M, c, o, \omega, f(\omega)) \vdash M?c/o \\ f^{\mathsf{s}}(\omega) & \text{otherwise} \end{cases}$

The intuition behind definition 24 is that each impacted component updated by applying the appropriate propagation function. In order to ensure validity against ill-defined propagation functions we additionally check if the user declared impact propagation function yields a foundationally valid component and, if not, we use the default propagation function f^{s} instead which we know to be correct.

Theorem 7. *Given a valid theory graph $\mathcal{G}$ and a propagator configuration $\langle f_{\mathcal{D}}, f_{\mathcal{U}}\rangle$, if δ is a $\mathcal{G}$-applicable change, then the $\mathcal{FIP}(\mathcal{G}, \delta)$ is a $\mathcal{G}$-applicable and minimal diff. Moreover if the impact relation $\looparrowright^f$ is transitive then $\mathcal{G} \ll \Delta \bullet \mathcal{FIP}(\mathcal{G}, \delta)$ is foundationally valid.*

Proof. Applicability and minimality follow immediately from the definitions of foundational impact propagation and minimal diffs. Foundational validity arises from the fact that each update in $\mathcal{FIP}(\mathcal{G}, \delta)$ replaces a (possibly) invalid term with a valid one. Then, if $\looparrowright^f$ is transitive then the subgraph affected by this process is dependency closed so there are no conflicts preserved or introduced by $\mathcal{FIP}$. □

Transaction Propagation Above we studied the propagation of only one change δ. But this does not cover the important case of transactions, where multiple changes are propagated together. This is typical in practice when an author makes multiple related changes. On the surface, the propagation mechanism appears to scale well from a single change to a sequence of changes, by just applying the same propagation rules to each change. However, the problem with that approach is that the resulting diff is not minimal anymore and, so, the final result is ambiguous with respect to application ordering.

Our first important insight is to focus on the **propagation of minimal diffs**. Theorem 4 guarantees that the diffs our differencing algorithm generates are indeed minimal, and theorem 3 guarantees that any pragmatic refinement also behaves as a minimal diff with respect to change application. Moreover, theorem 5 ensures that this is not actually a loss of generality.

The second insight is that the propagation generation process should be declaration oriented rather than change oriented. This means that we generate the required update for each impacted component relative to the entire diff rather than simply composing the propagation of each individual change. This has the result of creating only one change for each impacted component, so that the result of propagation is also a minimal diff.

However, this does not avoid the ambiguity problem, but merely move it into another place where it can be tackled more easily. In the case where a component is affected by several changes, we now have several impact propagation functions that must be applied to the same term. The application order for these propagation functions is still undefined and may still lead to ambiguity. Still, in this case we have more flexibility.

Firstly, we can ensure non-ambiguity by applying the functions in any order and checking if they commute by observing if the result is the same in each case.

Secondly, as discussed in the previous sections, pragmatic changes are to be confirmed by the user and, thus, can be propagated on the spot. This means that in principle, transactions of several changes can be assumed to have only strict changes. Since adds have no foundational dependencies we need only to ensure that $f_{\mathcal{D}}$ and $f_{\mathcal{U}}$ commute which is conceptually easy. This not only makes the runtime checking described above faster but also makes the process simple enough so that users who define changes and propagation rules can predict its result.

Definition 25. *Given theory graph $\mathcal{G}$ and a propagator configuration $\langle f_{\mathcal{D}}, f_{\mathcal{U}}, P\rangle$, the* ***foundational impact propagation*** *of a $\mathcal{G}$-applicable diff Δ is a diff generated as follows (and denoted $\mathcal{FIP}(\mathcal{G},\Delta)$) :*

First, for every component path $M?c/o$ in the foundational impact of Δ (relative to the subgraph determined by P) we construct the pair $(M?c/o, \Delta_{M?c/o})$ where $\Delta_{M?c/o}$ contains exactly those changes in Δ that impact $M?c/o$. Then, let l be the list of all the pairs defined above, ordered in the dependency order of the component paths. Now, the propagation function for a change δ is

- *$F_{\omega}(\Delta)$ if δ is a pragmatic change $N(\Delta)$ where $(N, P, F_{\omega}, F_{Mod})$ is a pragmatic change type.*
- *$f_{\mathcal{D}}(\delta)$ if δ is a delete*
- *$f_{\mathcal{U}}(\delta)$ if δ is an update*

Then $\mathcal{FIP}(\mathcal{G},\Delta)$ is $\mathcal{FIP}(\mathcal{G},l)$ as defined in the following :

- $\mathcal{FIP}(\mathcal{G},\cdot) = \cdot$
- $\mathcal{FIP}(\mathcal{G}, \langle (M?c/o, \Delta_{M?c/o}), l'\rangle, f) = \mathcal{U}(M,c,o,\omega,f(\omega))\bullet$
  $\mathcal{FIP}(\mathcal{G} \ll \mathcal{U}(M,c,o,\omega,f(\omega)), l', f)$
  where $\omega = \mathcal{G}(M?c/o)$ and f is :
  - *the composition of the propagation functions for all changes in $\Delta_{M?c/o}$ if they all commute and if $\mathcal{G} \ll \mathcal{U}(M,c,o,\omega,f(\omega)) \vdash M?c/o$*
  - *f^{s} otherwise*

Theorem 8. *Given a valid theory graph $\mathcal{G}$ and a propagator configuration $\langle f_{\mathcal{D}}, f_{\mathcal{U}}\rangle$, if Δ is a $\mathcal{G}$-applicable diff, then $\mathcal{FIP}(\mathcal{G},\Delta)$ is also a $\mathcal{G}$-applicable and minimal diff. Moreover if the impact relation $\looparrowright^{f}$ is transitive then $\mathcal{G} \ll \Delta\bullet\mathcal{FIP}(\mathcal{G},\Delta)$ is foundationally valid.*

Proof. Applicability and minimality follow immediately from the definitions of foundational impact propagation and minimal diffs. Foundational validity arises

from the fact that each update in $\mathcal{FIP}(\mathcal{G}, \Delta)$ replaces a (possibly) invalid term with a valid one. Then, if $\looparrowright^f$ is transitive the subgraph affected by $\Delta \bullet \mathcal{FIP}(\mathcal{G}, \delta)$ is dependency closed so there are no conflicts preserved or introduced by $\mathcal{FIP}$. Therefore, it does not have any impact so the entire graph

$$\mathcal{G} \ll \Delta \bullet \mathcal{FIP}(\mathcal{G}, \Delta)$$

is foundationally valid. □

Theorem 9. *Given a valid theory graph $\mathcal{G}$ and a propagator configuration $\langle f_{\mathcal{D}}, f_{\mathcal{U}} \rangle$, if Δ is a $\mathcal{G}$-applicable diff and $\looparrowright^f$ is transitive then it holds that $\mathcal{G}' = \mathcal{G} \ll \Delta \bullet \mathcal{SIP}(\mathcal{G}, \Delta) \bullet \mathcal{FIP}(\mathcal{G}, \Delta \bullet \mathcal{SIP}(\mathcal{G}, \Delta))$ is a valid theory graph. Moreover, replacing some (or all) box terms from $\mathcal{G}'$ with a valid term also yields a valid theory graph.*

Proof. Follows immediately from theorems 6 and 8. □

A typical situation where we would apply definition 25 is after a user made the changes Δ. Then propagation marks all terms that have to be revalidated and — if not well-typed — replaced interactively with well-typed terms. The theorem guarantees the resulting graph is valid again.

Running Example 6 (Continuing examples 4 and 5). Using pragmatic diff $\Delta = \mathcal{D}(PL, \vee : bool \to bool \to bool = \bot) \bullet \mathcal{A}(PL, \neg : form \to form = \bot) \bullet rename(\Delta^{\mathsf{s}}_{\mathcal{R}})$ from example 4 and the dependency relation from example 5, we compute the propagation of Δ.

Firstly we compute $\mathcal{SIP}(Rev_1, \Delta)$ by following definition 23 :

- $\mathcal{D}(PL, \vee : bool \to bool \to bool = \bot)$ generates the change $\mathcal{D}(v, \vee := +)$
- $\mathcal{A}(PL, \neg : form \to form = \bot)$ generate the change $\mathcal{A}(v, \neg := \boxed{\cdot}_{\mathcal{A}(PL, \neg : form \to form = \bot)})$
- $rename(\Delta^{\mathsf{s}}_{\mathcal{R}})$ generates the diff $\mathcal{D}(v, bool := nat) \bullet \mathcal{A}(v, form := nat)$

So, $\mathcal{SIP}(Rev_1, \Delta) = \mathcal{D}(v, \vee := +) \bullet \mathcal{A}(v, \neg := \boxed{\cdot}_{\mathcal{A}(PL, \neg : form \to form = \bot)}) \bullet \mathcal{D}(v, bool := nat) \bullet \mathcal{A}(v, form := nat)$

Then we compute $\mathcal{FIP}(Rev_1, \Delta \bullet \mathcal{SIP}(Rev_1, \Delta))$ by using the dependency relation from example 5 and following definition 25 :

- $\mathcal{D}(PL, \vee : bool \to bool \to bool = \bot)$ generates the change $\mathcal{U}(PL, \Rightarrow, \mathtt{def}, \lambda x.\lambda y.y \vee (x \wedge y), \boxed{\lambda x.\lambda y.y \vee (x \wedge y)}_{\Delta})$
- $\mathcal{A}(PL, \neg : form \to form = \bot)$ generates the empty diff $\cdot$

$$
\begin{aligned}
&PL = \{\\
&\quad form : \mathtt{type} = \bot\\
&\quad \neg : form \to form = \bot\\
&\quad \wedge : form \to form \to form = \bot\\
&\quad \Rightarrow : form \to form \to form\\
&\qquad = \boxed{\lambda x.\lambda y.y \vee (x \wedge y)}_{\Delta}\\
&\}
\end{aligned}
$$

(a) Theory PL

$$
\begin{aligned}
&NAT = \{\\
&\quad nat : \mathtt{type} = \bot\\
&\quad + : nat \to nat \to nat = \bot\\
&\quad * : nat \to nat \to nat = \bot\\
&\}
\end{aligned}
$$

(b) Theory NAT

$$
\begin{aligned}
&v : PL \to NAT = \{\\
&\quad form := nat\\
&\quad \neg := \boxed{\cdot}_{\mathcal{A}(PL, \neg : form \to form = \bot)}\\
&\quad \vee := +\\
&\}
\end{aligned}
$$

(c) View $PL \to NAT$

Figure 11: The theory graph after change propagation

- $rename(\Delta^{\mathtt{s}}_{\mathcal{R}})$ generates the diff
  $\mathcal{U}(PL, \wedge, \mathtt{tp}, bool \to bool \to bool, form \to form \to form) \bullet$
  $\mathcal{U}(PL, \Rightarrow, \mathtt{tp}, bool \to bool \to bool, form \to form \to form)$

Therefore, $\mathcal{FIP}(Rev_1, \Delta \bullet \mathcal{SIP}(Rev_1, \Delta)) = \mathcal{U}(PL, \Rightarrow, \mathtt{def}, \lambda x.\lambda y.y \vee (x \wedge y), \boxed{\lambda x.\lambda y.y \vee (x \wedge y)}_{\Delta}) \bullet \mathcal{U}(PL, \wedge, \mathtt{tp}, bool \to bool \to bool, form \to form \to form) \bullet \mathcal{U}(PL, \Rightarrow, \mathtt{tp}, bool \to bool \to bool, form \to form \to form)$

Finally, the graph $Rev_1 \ll \Delta \bullet \mathcal{SIP}(Rev_1, \Delta) \bullet \mathcal{FIP}(Rev_1, \Delta \bullet \mathcal{SIP}(Rev_1, \Delta))$ is shown in figure 11. As stated in theorem 9, it becomes foundationally valid after replacing the box term with a term of the right type.

Impact Propagation for Interpretation Services With respect to interpretors, change propagation is different because it does not affect the internal MMT content representation. Rather, when changes cause conflicts to be introduced at the interpretation level the induced changes required to fix the conflicts are at the level of the interpretor (i.e. at the presentation level). Therefore, MMT cannot directly manage those conflicts so it only provides a framework for detecting them and forwarding the conflict information to the individual interpretors which may then choose how to handle the conflicts.

Definition 26. *Given a theory graph $\mathcal{G}$ and an interpretation service $\mathfrak{S}$, the* ***interpretation relative impact propagation*** *for $\mathfrak{S}$ of a $\mathcal{G}$-applicable diff Δ is a set (denoted $\mathcal{IIP}^{\mathfrak{S}}(\mathcal{G}, \Delta)$) containing pairs of impacted paths and the changes that caused the impacts.*

Specifically, we filter the paths in the interpretation impact for $\mathfrak{S}$ *of* Δ *by checking if their* $\mathfrak{S}$*-interpretation in* $\mathcal{G} \ll \Delta$ *and* $\mathcal{G}$ *is identical to the one before* Δ *was applied (i.e. if it fails or yields a different result).*

Then, for each π *where the intepretation is not identical, we construct the pair* (π, Δ_π) *where* Δ_π *contains exactly those changes in* Δ *that impact* π*. Finally,* $\mathcal{IIP}^{\mathfrak{S}}(\mathcal{G}, \Delta)$ *is the set of all the pairs constructed above.*

Theorem 10. *Given a theory graph* $\mathcal{G}$*, an interpretation service* $\mathfrak{S}$ *and a* $\mathcal{G}$*-applicable diff* Δ*, if the presentation-oriented representation used by* $\mathfrak{S}$ *is edited at all paths returned by* $\mathcal{IIP}^{\mathfrak{S}}(\mathcal{G}, \Delta)$ *such that* $\mathfrak{S}$*-interpretation at those paths succeeds with the same result as the previous one (stored in the content library),* $\mathcal{G}$ *is valid relative to* $\mathfrak{S}$*.*

Proof. Follows immediately from the definitions. □

5 A Generic Change Management API

We have implemented MoC as a part of the MMT API [Rab08] by closely following the formalism established in section 4. Our implementation [API] covers the entire MMT language (not only the fragment presented in this thesis) and spans around 1500 lines of code. Consequently, MMT now serves as an API for generic change management.

5.1 Implementation

Changes The first step was the implementation of the language of changes described in section 4.1. In particular, we follow figure 4 for component identifiers, figure 5 for strict changes, definition 3 for pragmatic change types and definition 4 for pragmatic changes. Box terms can be represented at the MMT object level as OPENMATH error terms which are already in the MMT API we don't need to explicitly add them to the implementation. Note that, while our implementation does allow for individual pragmatic changes to be instantiated directly, in practice they are constructed by pragmatic change types from a matching diff during the refinement process (e.g. as shown in example 4. The class hierarchy implementing the change language described in section 4.1 is shown in figure 12.

Dependencies The second step was the implementation of the dependency relation discussed section 4.3. Our dependencies are built on top of the already existing MMT ontology, which MMT maintains together with the content [HIJ+11]. Once indexed, the ontology is persistently stored and available at any time. We generate dependency information from the lookups done by a plugin (foundation or

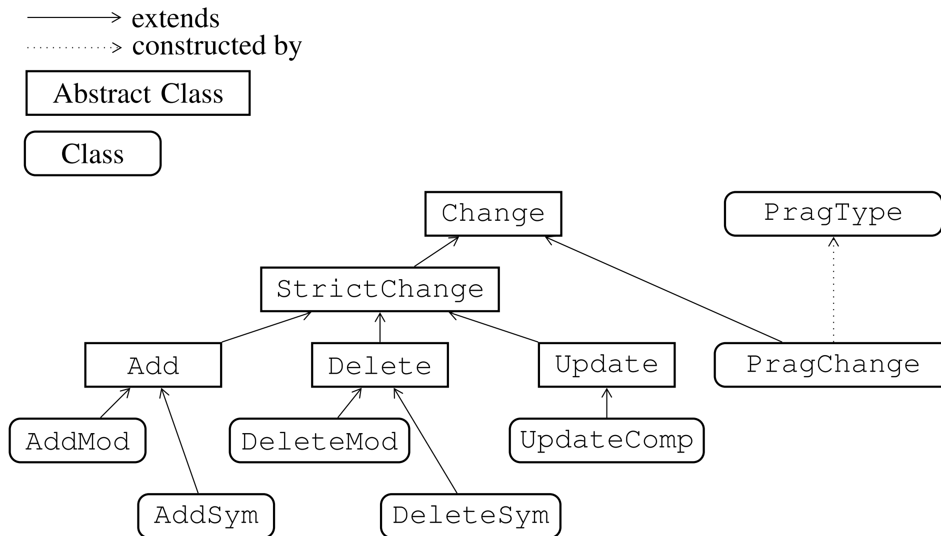

Figure 12: Diagram for Change Classes

interpretor) during processing of an item (by adding them to the ontology as dependencies of that item relative to the plugin). With respect to the implementation, this is done by de-coupling the content library from the plugins and connecting them only through an interface which does the heavy-lifting (tracing lookups and adding to the ontology). Alternatively, we can import a dependency relation, e.g., the ones from [AMU11].

Transactions and Propagation Finally, once we have changes and dependencies we can implement the notions described in sections 4.2 and 4.4 (i.e. change application, detection and propagation) to provide the desired functionality to our MoC API.

Firstly, the objects `Patcher` and `Differ` implement the formalism of $\ll$ and $\mathfrak{Diff}$ and provide the provide the ability to apply and detect changes. The two theory graphs that serve as arguments for differencing can either be provided directly or previous revisions can be pulled from an SVN repository.

Secondly, change refinement and change impact propagation are unified by the generic concept of diff enrichment which is represented in the implementation by the abstract class `Propagator`. Then, a `Refiner` is a special propagator that enriches diffs by generating pragmatic changes (given a diff and a set of available pragmatic types) and an `ImpactPropagator` is a propagator that adds changes to a diff based on a dependency relation and a propagation function. Then `SIP` and `FIP` implement the structural impact propagator ($\mathcal{SIP}$) and, respectively, the foundational impact propagator ($\mathcal{FIP}$) by extending `ImpactPropagator` with

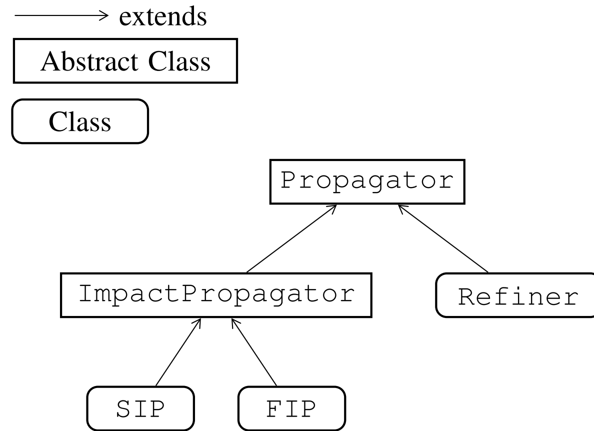

Figure 13: Diagram for Propagation Classes

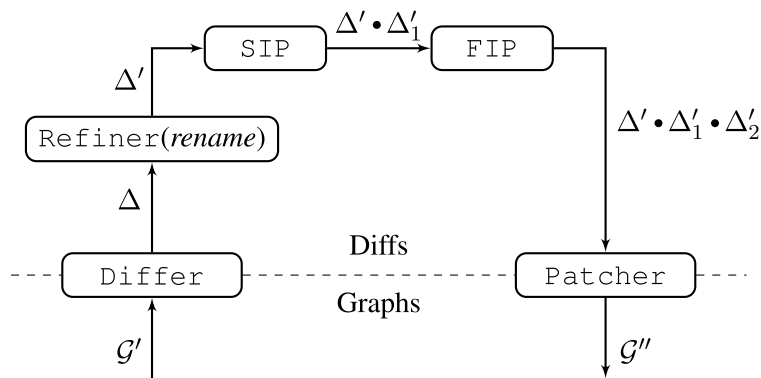

Figure 14: Workflow Diagram (relative to a graph $\mathcal{G}$)

the adequate dependency relation and propagation function.

A usage workflow is constructed from the differ and patcher together with a chain of propagators that enrich the diff generated by the differ before it is applied.

Running Example 7. Figure 14 shows the propagator chain that would produce the propagation result shown in examples 4 and 6.

5.2 Evaluation

We have applied the resulting system to obtain a change management API for the LATIN library.

Using the MMT plugins for LF [HHP93] — the language underlying the LATIN

dependencies	components (%)
$0-5$	1373 (79)
$6-10$	271 (15.6)
$11-15$	81 (4.7)
$16-26$	13 (0.7)

impacts	components (%)
$0-5$	1504 (86.5)
$6-10$	101 (5.8)
$11-25$	76 (4.4)
$26-50$	31 (1.8)
$50-449$	26 (1.5)

Figure 15: Components grouped by dependencies and impacts

library — we obtain a foundation that validates the library and computes a dependency relation for it. Figure 15 gives a summary of the dependency relation, where we include only the about 1700 components falling into the fragment of MMT treated in this thesis. The tables group the components by the number of components that they depend on (left) or that depend on them (right). This includes only direct dependencies — taking the transitive closure increases the numbers by about 20%.

The number of dependencies and impacts is generally low. This is a major benefit of our choice of using type and definiens as separate validation units, which avoids the exponential blowup one would otherwise expect. Indeed, on average a type has 3 times as many impacts as a definiens.

Our differencing algorithm can detect and propagate changes easily, and revalidating the impacted components is straightforward. The numbers show that even manual inspection (as opposed to automatic revalidation) is feasible in most cases: For example, changes to 86% of the components impact only 5 or less components. However, while the number of impacted components is small the variance is high (from 0 to 449) and it is usually very difficult for humans to identify exactly how many or which components are impacted. Our MoC infrastructure, on the other hand, does not only identify them automatically but also guarantees that all other components remain valid.

6 Applications and Future Work

We described in the previous section how our MoC can not only detect, apply and propagate changes but it supports a customizable change language, automatic conflict resolution (for pragmatic changes) and guarantees the preservation of validity. Therefore, it provides a powerful yet flexible framework for management of change in MMT. However, because it is a framework, the usefulness of our MoC depends on how many services make use of its features.

Outline of MMT Services Currently MMT has a web interface which provides support for browsing MMT repositories by using a notation system to render MMT documents in a web format using MATHML. It includes interactive features such as type inference, sub-term folding, and hiding/viewing reconstructed types, implicit arguments, implicit binders or redundant brackets. An instance of the MMT browser is available at [KMR09], where it serves the logic atlas of the LATIN project.

Furthermore, there is also a recently started effort geared towards adding editing support to MMT [IR12b] in two different user interfaces.

Firstly, it aims to connect the MMT API with the text editor jEdit [jEd] in order to obtain IDE-like authoring support.

Secondly, it focuses on extending the MMT webserver with interface components for wiki-like editing.

In the following we will discuss how our MoC can improve the usability of existing MMT services and describe some new useful applications that are made possible by our MoC.

Visualisation The information gathered by MoC is useful in getting an overview of a knowledge library or a module by providing information about changes, dependencies or conflicts. However, while this information is computed and stored by our MoC, we don't currently have a good way of making it available to users.

Nevertheless, the two MMT user interfaces provide a good platform for implementing such services.

Firstly, since MMT can use SVN [Apa00] as a backend and thus retrieve old versions of declarations, it can compute the evolution of a library or module as a sequence of changes. Then the MMT web interface can not only display MMT modules or declarations but also highlight the changes that occurred since, e.g., that users last session. Figure 16 shows an example visualization for two revisions similar to those in example 1 (there is in addition an update to the type of $\wedge$ to exemplify the visualization of updates) where the changes are color coded with yellow of renames, blue for updates and green for adds.

Secondly, the dependency information gathered by MoC can be visualized in a graph based interface. MMT already has a graph view [Rus11] for a theory graph showing theories as nodes and includes (or morphisms) as arrows. The same technology can be leveraged to also display dependencies between declarations.

Editing Clearly, MoC can provide substantial support while editing documents by keeping track of changes, dependencies and impacts. As before, this information is readily available due to our MoC and only needs to be integrated with editing interfaces to make this information visible. In addition to marking the conflicts

```
theory PL
  form : type
  ¬    : form → form
  ∧    : form → form → form
  ⇒    : bool → bool → bool
       = [x:bool][y:bool] y ∨ ( x ∧ y )
```

Figure 16: Change Visualization in MMT Web Interface

through impact propagation we can also provide other usability improvements.

For instance, since all dependency information is persistently maintained for each declaration our MoC can predict the potential impact of an edit before any change is made. Specifically, MoC can provide the number of dependencies of each declaration and the editing interface can provide an "impact meter" to inform the user of the potential conflicts introduced by changing that declaration.

Also, after one or several changes, the editor can display the potential pragmatic refinements detected by our MoC. Then, by sending back the answer to our MoC changes can be automatically propagated (if necessary). Thus, refactorings such as declaration renames or alpha-equivalent term changes can be automatically handled.

7 Conclusion

We have presented a theory of change management and refactoring based on the MMT language including difference, dependency, and impact analysis. As MMT is foundation-independent, our work yields a theory of change management for any arbitrary declarative language. It is implemented as a part of the MMT API and, thus, immediately applicable to any language that is represented in MMT. In particular, the latter includes the logical framework LF and, therefore, every logic encoded in it.

Because we use fine-grained dependencies, change propagation can identify individual type checking obligations (which subsume proof obligations) that have to be revalidated. The MMT API already provides a scalable framework for validating individual such obligations efficiently. Therefore, our work provides the foundation for a large scale change management system for declarative languages.

While our presentation has focused on a fragment of MMT, the results hold for the whole MMT language, in particular the module system.

Presently the most important missing feature is a connection between the MMT abstract syntax and the concrete syntax of individual languages. Therefore, change management currently requires an export into MMT's abstract syntax (which exists for, e.g., Mizar [TB85], TPTP [SS98], and OWL [W3C09]). Consequently, **future work** will focus on developing fast bidirectional translations between human-friendly source languages and their MMT content representation. If these include fine-grained cross-references between source and content, MMT can propagate changes into the source language; this could happen even while the user is typing.

References

[ABKB+02] E. Astesiano, M. Bidoit, B. Krieg-Brückner, H. Kirchner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL – the common algebraic specification language. *Theoretical Computer Science*, 286:153–196, 2002.

[ADD+11] S. Autexier, C. David, D. Dietrich, M. Kohlhase, and V. Zholudev. Workflows for the Management of Change in Science, Technologies, Engineering and Mathematics. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 164–179. Springer, 2011.

[AHM10] S. Autexier, D. Hutter, and T. Mossakowski. Change Management for Heterogeneous Development Graphs. In S. Siegler and N. Wasser, editors, *Verification, Induction, Termination Analysis, Festschrift in honor of Christoph Walther*. Springer, 2010.

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

[AHMS02] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002.

[AM10] S. Autexier and N. Müller. Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In M. Gormish and R. Ingold, editors, *Proceedings of 10th ACM Symposium on Document Engineering (DocEng2010)*, 2010.

[AMU11] Jesse Alama, Lionel Mamane, and Josef Urban. Dependencies in Formal Mathematics. *CoRR*, abs/1109.3687, 2011.

[Apa00] Apache Software Foundation. Apache Subversion, 2000. see http://subversion.apache.org/.

[API] MMT Source Code. Available at: `https://svn.kwarc.info/repos/MMT/src/mmt-api/branches/next-version`.

[BC08] A. Bundy and M. Chan. Towards Ontology Evolution in Physics. In W. Hodges and R. de Queiroz, editors, *Logic, Language, Information and Computation*, pages 98–110. Springer, 2008.

[BCC+04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See `http://www.openmath.org/standard/om20`.

[CP02] I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002.

[CVS] Concurrent Versions System: The open standard for Version Control. Web site at `http://cvs.nongnu.org/`. seen February 2012.

[EG89] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM, 1989.

[Git] Git. Web Site at: `http://git-scm.com/`.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HIJ+11] F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.

[Hut00] D. Hutter. Management of change in structured verification. In *Proceedings Automated Software Engineering, ASE-2000*, pages 23–34, 2000.

[IR12a] M. Iancu and F. Rabe. Management of Change in Declarative Languages. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, volume 7362 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2012.

[IR12b] M. Iancu and F. Rabe. Work-in-progress: An MMT-Based User-Interface. 2012.

[jEd] jEdit: Programmer's Text Editor. http://www.jedit.org/. seen May 2012.

[KK11] Andrea Kohlhase and Michael Kohlhase. Versioned links. In *Proceedings of the 29th annual ACM international conference on Design of communication (SIGDOC)*, 2011.

[KMR] Michael Kohlhase, Till Mossakowski, and Florian Rabe. Latin: Logic atlas and integrator. http://latin.omdoc.org.

[KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see https://trac.omdoc.org/LATIN/.

[Mos04] Peter D. Mosses, editor. *CASL Reference Manual*. Number 2960 in LNCS. Springer Verlag, 2004.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In D. Kapur, editor, *Proceedings of the 11th Conference on Automated Deduction*, number 607 in LNCS, pages 748–752, Saratoga Springs, NY, USA, 1992. Springer Verlag.

[OSV07] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard P. Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[Rab08] F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.

[RK11] F. Rabe and M. Kohlhase. A Scalable Module System. see http://arxiv.org/abs/1105.0548, 2011.

[RKS11] F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2011.

[Rus11] Mihaela Rusu. Interactive semantical graphs. Guided research thesis, Jacobs University Bremen, 2011.

[Sch06] Axel Schairer. *Transformations of specifications and proofs to support an evolutionary formal software development*. PhD thesis, Saarlndische Universitts- und Landesbibliothek, Postfach 151141, 66041 Saarbrcken, 2006.

[SM02] Lutz Schröder and Till Mossakowski. Hascasl: towards integrated specification and development of functional programs. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology — 9th International Conference AMAST 2002*, number 2422 in LNCS, pages 99–116. Springer Verlag, 2002.

[SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[UB06] Josef Urban and Grzegorz Bancerek. Presenting and explaining Mizar. In Serge Autexier and Christoph Benzmüller, editors, *Proceedings of the International Workshop "User Interfaces for Theorem Provers" 2006 (UITP'06)*, pages 97–108, Seattle, USA, 2006.

[W3C09] W3C. OWL 2 Web Ontology Language, 2009. http://www.w3.org/TR/owl-overview/.

[Wag10] M. Wagner. *A change-oriented architecture for mathematical authoring assistance*. PhD thesis, Universität des Saarlands, 2010.

[WBB+] Makarius Wenzel, Clemens Ballarin, Stefan Berghofer, Timothy Bourke, Lucas Dixon, Florian Haftmann, Gerwin Klein, Alexander Krauss, Tobias Nipkow, David von Oheimb, Larry Paulson, and Sebastian Skalberg. *The Isabelle/Isar Reference Manual*.