Friedrich-Alexander-Universität
Erlangen-Nürnberg

KWARC research group

Theory Intersection

An automated approach to finding intersections of theories

Master Thesis in Computer Science

Michael Banken

Advisors:

Dennis Müller
Prof. Dr. Michael Kohlhase

Erlangen, October 22, 2020

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Nürnberg, October 22, 2020

[Michael Banken]

Contents

1 Introduction

Knowledge representation of mathematical theories

In recent years mathematics has been increasingly computerized. Many modern proofs are either verified by computational systems or in some cases even generated automatically. There are however still numerous problems with the previously existing solutions.

In order to address these problems the MMT project offers a framework to formalize, organize and represent mathematical knowledge in so called *theory graphs*.

Within such theory graphs mathematical knowledge is represented in so called *theories*. Once a theory T has been formalized in the MMT language, we can make use of it by either forming new theories that expand on T (e.g. extending first-order syntax with Natural Deduction inference rules) or by using T as the language to write a new theory in (e.g. a theory of monoids in the language of first-order logic). These relations between theories and other theories that make use of them make up the connections in the theory graph.

These connections between theories are called *theory morphisms* and they allow for knowledge to be transferred between different theories. Of particular interest to this work is the *view*, which functions as a language translation along similar structures inside two different theories by mapping the content of the codomain theory to the content of the domain theory.

Theory Intersections

One problem of transferring knowledge via theory morphisms is that in general this requires a total morphism. In the case of the view it may not always possible to formulate a total view, however it may be possible to formulate a partial view. While this does not in itself allow the same sort of knowledge transfer as a total view, we can limit the scope of a theory to those parts that can be translated. Along this line of thought we come to the concept of theory intersections. The concept is not new and indeed older than the implementation of MMT itself [1] and has later been proposed in its application to MMT by Kohlhase [2]. This work is also not the first time that this concept has been implemented, but seeks to expand on an earlier approach by Müller [3].

The concept of theory intersections is intuitively related to intersection on sets. If we simplify a theory to a set of declarations, then the intersection of two theories should correspond to the set intersection of these theory sets.

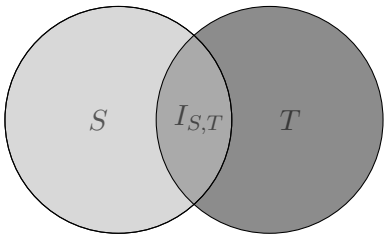


Figure 1.1: Intersection $I_{S,T}$ of two theory sets S and T

In order to move away from this simplification, a theory intersection itself consists of one or more theories and like an intersection between sets the content of the intersection is the content that is found in both S and T.

As an example monoid theory can be viewed as an intersection between two theories that describe monoid operations, such as addition on natural numbers and the or operator from boolean algebra. Both of these theories have a set, a binary operation on this set and a neutral element, as well as all the axioms we need for a monoid. By extracting these elements we can reconstruct monoid theory as an intersection of these theories.

Motivation and Goals

The goal of this thesis is to design and implement an automated approach for generating theory intersections. To achieve this we will also make use of an automated viewfinder to supply us with views, in order to further automate the process [4].

The benefit of intersecting theories is that by intersecting larger theories we generate smaller theories that fit more neatly into the Little Theories approach [5].

We also mentioned earlier, that the basis of the intersection is a partial view. By limiting the scope of the theory to those elements which share a similar structure we can make use of the same knowledge transfer that a total view would have offered, though only if it falls within the smaller limits of the new scope. This becomes immediately obvious if we consider the example above (Figure 2.9) of mathematical theory of monoids, which allows for reasoning that can be applied to any of the various different types of monoids based on the similarities they share.

Contribution

This thesis proposes an extended definition of theory intersections that allows the retaining of the inclusion structure of the original theory, as well as extend this structure to the intersections themselves.

It also provides an implementation of a tool that automates the process of generating these theory intersections with the extended definition in mind, while building on the previous solution implemented by Müller [3].

Structure of the thesis

In the following chapters we will first in Section 2 establish the context and terms of MMT theories and theory graphs, as found in the relevant literature, before taking a closer look at the existing approach to theory intersections.

Then in Section 3 we will establish the extended definition of theory intersections called deep theory intersections, which will form the theoretical core of this work.

In Section 4 we will discuss algorithms that can be used for the automated generation of these deep theory intersections.

In Section 5 we will establish some basic metrics that can be used to evaluate the worth of a theory intersection. This is useful for determining which intersections are worth adding to the theory graph.

Section 6 describes a tool, which implements the algorithms in Section 4, as well as the evaluation functions with which we can judge their results.

And finally in Section 8 we will draw our conclusions and discuss a few ways in which this project can be built upon.

2 Preliminaries

Before we move to the contribution of this thesis, let us first establish the basic terms of what we will be working with, as well as take a look at the state of previously existing approaches to theory intersections.

We will begin with a brief overview of theory graphs represented in the MMT-language.

Then we will take a look at the established definition of theory intersections, which we will refer to as flat theory intersections to distinguish them from the recursive approach of deep theory intersections proposed in Section 3.

2.1 Knowledge Representation in MMT

The MMT language allows us to describe theory graphs using the following syntax [6]:

Theory graph	TG	:==	(Th | Vw)*
Theory	Th	:==	S [: T] = {TBody}
View	Vw	:==	V : S $\to$ T = ({ LBody} | μ)
Theory Body	TBody	:==	(Dec | Str)*
Declaration	Dec	:==	c [: φ] [= ψ]
Structure	Str	:==	(s : S = { LBody}) | Inc
Include	Inc	:==	include S [= μ]
Link Body	LBody	:==	(ConAss | StrAss)*
Constant Assignment	ConAss	:==	c = φ
Structure Assignment	StrAss	:==	s = μ
Morphism	μ	:==	V | id_T | $\mu \circ \mu$
Term	φ, ψ	:==	T?c | x | $\alpha(\varphi 0, \varphi_1, ..., \varphi_n)$ | $\beta(\varphi_1, \Gamma, \varphi_2)$
Module identifier	S, T, V		
Local identifier	c, s		
Symbol identifier			S?c
Variable	x		
Context	Γ	:==	$(x : \varphi)*$

In this chapter we will take a closer look at each of these syntactic building blocks starting from the top down.

2.1.1 Theory graph

A theory graph consists of two building blocks.
The *theories* as its nodes, which contain the knowledge represented in the graph.
And the *theory morphisms* that allow to infer knowledge by transfering it between different

theories.

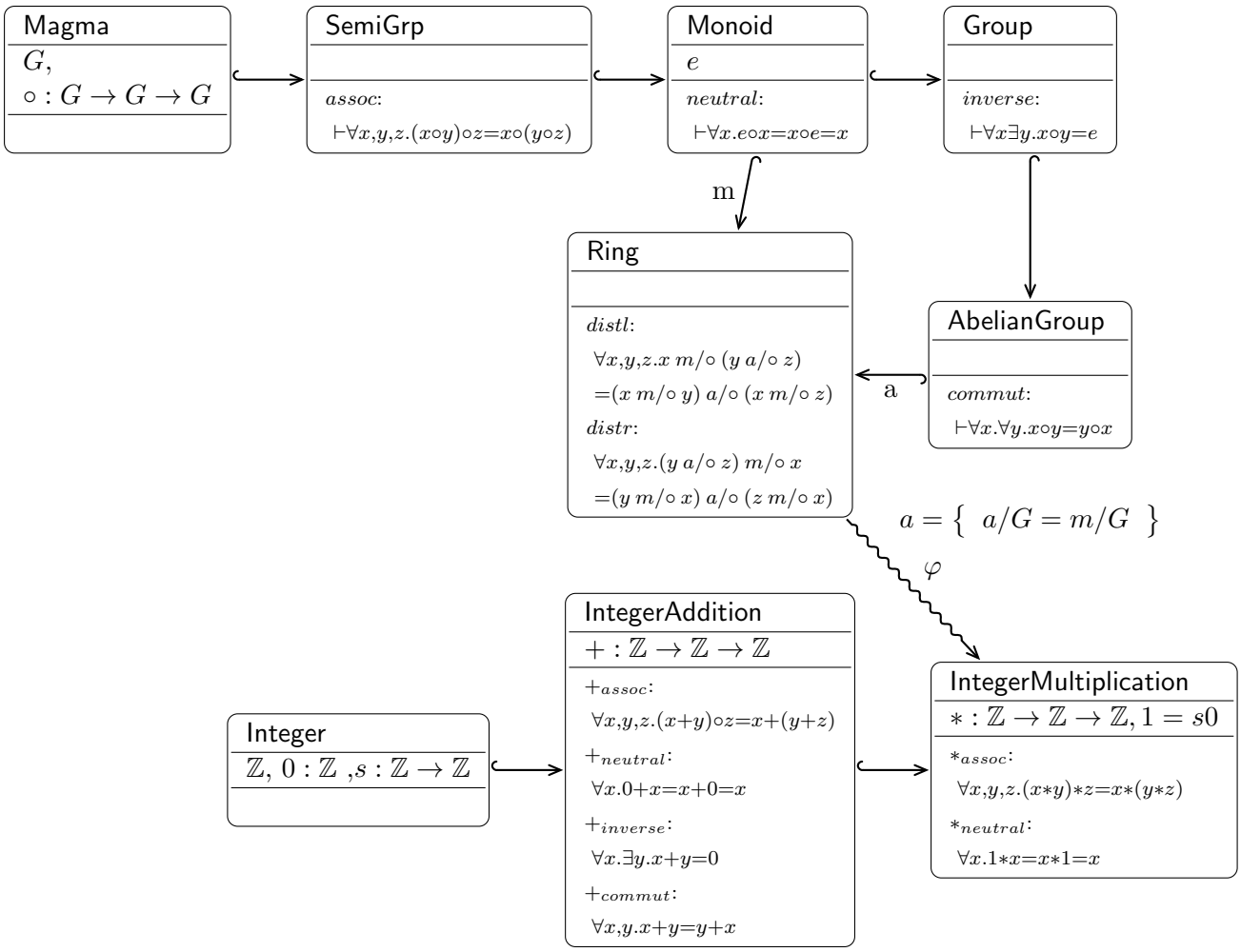


Figure 2.1: Theory graph of elemental algebra

In Figure 2.1 we can see three kinds of edges.

- S $\hookrightarrow$ T : inclusion
  copies content of S into T; functions as an implicit morphism
- S $\xhookrightarrow{\varphi}$ T : structure
  like an inclusion, but allows for mappings to existing terms;
  declarations from S in T are addressed $[\varphi]$/c
- S $\overset{\varphi}{\rightsquigarrow}$ T : view
  mapping of existing constants in S to existing Terms in T

All of these will be explained in detail below.

2.1.2 Theories

Theories can be seen as a list of declarations and structures. Declarations make up the content of a theory, while structures are used to describe the theory's relationship to other theories.

Theory	Th	:==	S [: T] = {TBody}
Theory Body	TBody	:==	(Dec | Str)*

S is the identifier of the theory.

T is the meta theory.

TBody is the body of the theory. It consists of any number of constant declarations and structures.

Theories can have optional meta theories. These function a lot like theory inclusions, but their intent is different. A theory's meta theory contains the language in which that theory is being described. For example group theory described in first order logic has the theory FOL as its meta theory.

Constant Declaration

Constant declarations are the building blocks of theories that make up its body. We earlier defined declarations with the following syntax.

Declaration	Dec	:==	c [: φ] [= ψ]

Such a constant declaration inside a theory S consists of 3 basic components.

c is a local identifier that will serve as the name of the declared constant

φ is an optional term over S that describes the constant's type

ψ an optional term over S that serves as the constant's definiens

Constant declarations also allow the addition of *aliases* and *notations*. However these are largely irrelevant for this work.[1].

The set of all declared symbols in a theory are its domain. We will refer to it as dom(S).

Terms

Terms are expressions over a theory S. They appear both as types and definiens of constants, as well as the right-hand side of constant assignments in theory morphisms.

Term	φ, ψ	:==	T?c | X | $\alpha(\varphi_0, \varphi_1, ..., \varphi_n)$ | $\beta(\omega, \Gamma, \varphi)$

The simplest expressions are just constant references, denoted by their identifier. MMT terms exist within the context of theories and can only use constants that are defined before them inside the theory.
For this reason we will refer to a term as $exp(S_c)$, if it uses those constants defined before c. $exp(S_c)$ are exactly those terms that can appear in a declaration of the constant c.[8]

We will refer to a term $exp(S)$ over a theory S, when it uses the constants declared in S. This means that

$$exp(S) = \bigcup_{c \in dom(S)} exp(S_c)$$

or more simply

$$exp(S_{c_{last}})$$

where c_{last} is the final declared constant in a theory, if such a constant exists.

[1] For more details consult the MMT documentation [7]

More complex terms can be constructed using the function application $\alpha(\varphi_0, \varphi_1, ..., \varphi_n)$, which takes an n-ary function term φ_0, as well as n terms as arguments for the function.

Another way to construct complex terms is via a binding $\beta(\varphi_1, \Gamma, \varphi_2)$, which take a binder φ_1, a context Γ of declared variables and the enclosed term φ_2.

For example the term $\beta(\forall, X : U, \alpha(P, X))$ could be used as a term to express the statement $\forall X.P(X)$, within a context where $\forall$ and X are defined.

Variables declared in an enclosing binder are also terms within the enclosed term.

2.1.3 Theory Morphisms

Theory Morphisms form the edges between the theory nodes in the theory graph.

A theory morphism $\sigma : S \to T$, is a mapping of all identifiers $c \in dom(S)$ to an expression over $dom(T)$. On the language syntax level they can be described as a list of assignments $c = Term_T$, where c is a constant identifier in S and $Term_T$ is an expression in T. [8]

We have to differentiate between two kinds of theory morphisms. Structures and views. Structures themselves can be subdivided into *includes* and *named structures*.

Includes

Includes are the simplest of structures. The inclusion of a theory is similar to inheritance from a class in object-oriented programming.

Include	Inc	:==	include S [= μ]

Semantically a theory inclusion from theory S to theory T makes the constants in its domain available to its codomain, as if copied to the codomain.

The result is that all symbols of a theory S that has been included in a theory T via an inclusion i are accessible in T. More specifically they are extending our understanding of domains so that we get

$$dom(T_{d_{i+1}}) = dom(T_i) \cup dom(S)$$

where d_{i+1} is the declaration after i and as a consequence

$$dom(S) \subseteq dom(T)$$

To better understand this it may be helpful to consider the *flattening* of the theory graph. During the flattening process structures are resolved by copying the content of the origin theory into the domain theory (the one in which the structure was declared). As a theory morphism the structure then simply maps all the constants in S to their copies in T. [6]
We call a theory that has gone through the process of flattening a *flattened theory*.

Named Structures

Named structures are similar to includes, but more powerful.

Structure	Str	:==	s : S = { LBody}
Link Body	LBody	:==	(ConAss | StrAss)*
Constant Assignment	ConAss	:==	c = φ
Structure Assignment	StrAss	:==	s = μ

By default the structure named s does map the constants inside theory S to copies in T, similar to an include, however the copies have changed local identifiers to [s]/c. More interestingly the body of the structure allows the assignment of constants to terms in T.

Named structures are useful when giving new notation inside the new theory or to avoid naming conflicts.

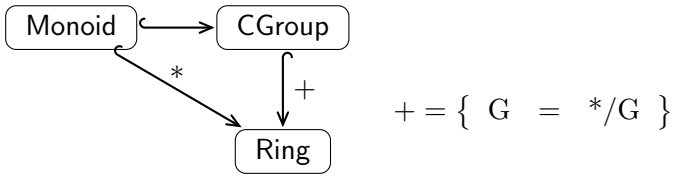


Figure 2.2: Ring theory using two different monoids via structures

An example for this is a theory describing algebraic rings, as seen in Figure 2.2. Since a Ring consists of both a monoid and a abelian group, we need to include both of those theories.

However while both the monoid and the abelian group in a ring share a common set, they have distinct operations. If we use simple includes for this, we have no way to distinguish between the two, since the theory CGroup, which describes the abelian group is already an extension of the Monoid theory.

By using named structures we can access both the content of Monoid and CGroup (including CGroups own access to Monoid) separately.

We can model the shared set of the operations by making the correct structure assignment to the + structure:

```
G = */G
```

Views

The *view* is probably the most interesting theory morphism for this work. While structures allow some knowledge transfer, on their own they are mostly a tool for structuring the MMT code to avoid undue duplication. Views on the other hand are much more flexible and connect any two theories in the graph.

View	Vw	:==	V : S → T = ({ LBody} | Morph)
Link Body	LBody	:==	(ConAss | StrAss)*
Constant Assignment	ConAss	:==	c = φ
Structure Assignment	StrAss	:==	s = μ

The declaration of a view is syntactically very similar to the Named Structure, except that it can be found outside a theory and requires an explicit target theory.

Another difference is that the view does not give access to the content of the origin theory to the target theory in the way that a structure does. Since we required a theory morphism σ to map all identifiers we similarly require the list of assignments to contain all constants in S, unless they have a definiens. Those that do have a definiens can be skipped, since their mapping can be inferred via homomorphic extension $\overline{\sigma}$ of σ. Due to this we will call σ total, even if it does not map the defined constants $def(S) \subset dom(S)$. [8]

A simple example for a view is mapping monoid theory *Monoid* to addition of natural numbers *Addition*. To formulate this view, we simply map the set of the monoid G to $\mathbb{N}$, the operation $\circ$ to $+$ and the appropriate axioms to their equivalent in *Addition*.

As another example of a view let us consider modal logic which is known to be a fragment of first order logic and the formulas of which can be translated to regular FOL-formulas via the standard translation [9].

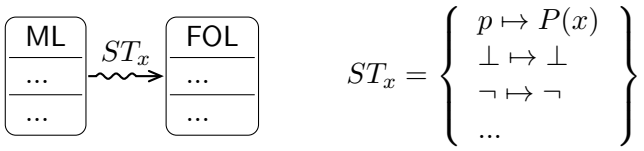


Figure 2.3: The standard translation as a view $ST_x : ML \to FOL$

Note that just like in these example a view is not necessarily invertible. However for the notion of theory intersections this will be a desirable property. Fortunately it is always possible to limit a (partial) view to a invertible (partial) view, though not necessarily a useful one.

If a view-like mapping σ is not total (not even when using its homomorphic extension), we will call it a partial view. Note that a partial is not strictly speaking a view, because we require theory morphisms and especially views to be total. This distinction makes sense, since partial views are of limited use compared to proper views, but they are of vital importance for creating theory intersections.

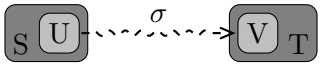


Figure 2.4: The partial view σ from S to T is defined on $U \subsetneq dom(S)$

2.2 Theory Intersection

2.2.1 Formulating the problem

As mentioned earlier it is not always possible to formulate a view between two theories that satisfies the totality requirement. However sometimes we can get reasonably close to formulating a view, by mapping at least some of the constants. In fact there is an automated tool for finding views, which creates a number of such partial views while searching for proper ones.[4]

In order to make use of these partial views the concept of theory intersections was introduced.[2][3]

However this does not work with just any partial view. We require specifically a view σ such that there is an inverse view σ^{-1}. A view is invertible if it injectively maps constants only to other constants. We call such a view a renaming.

Fortunately it is possible to convert any (partial) view into a partial renaming, by removing any assignment that is not to a constant. This means that our restrictions born from this requirement are really a lot less strict than they seem at first. However there is still the issue that the resulting partial view may be empty. While it is still possible to formulate an intersection over an empty view, it is not useful.

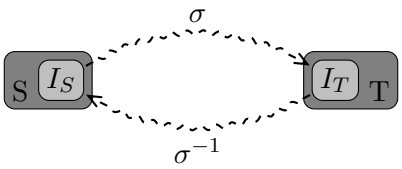


Figure 2.5: Partial invertible view exists between S and T

Alternatively it is possible to modify the original (partial) view σ by introducing a conservative extension T' of T, which is constructed by adding a constant c_φ for every complex term φ which σ assigns. The added constant's definiens is exactly φ.
We can then modify *sigma* by replacing all assignments to complex terms φ with assignments to the added constant c_φ.

We call either of these processes reducing σ to a (partial) renaming.[3]

2.2.2 Definition

For a first definition of theory intersection we will look at theory intersections as previously defined by Müller [3] before extending this definition in Subsection 3.2.1.

We should begin by stating that there are two kinds of theory intersections, the binary theory intersection and the unary theory intersection. In both cases the theory intersection is the result of intersecting over a partial renaming σ.
The binary theory intersection which as the name suggests consists of two theories, one for each of the intersected theories. The content of these new theories is the content of their original theory, which is mapped by σ (or it's inverse).
The unary theory intersection which consists of a single theory. The content of the new theory is again the content mapped by σ.

In both cases the theory intersection can be included by the remainder of the original theories, in order to preserve their integrity while integrating the newly gained connection within the intersection.
In the case of the binary intersection each remainder includes the theory containing the content of the original theory.
In the unary case both remainders include the same theory. Appropriate term substitutions have to be applied to keep the theory content consistent, since a single theory means we can at most keep the names from one original theory.

All theory intersections are constructed from a specific partial view $\sigma : S \to T$. We will refer to the result as a (theory) intersection of S and T over σ, or $S \overset{\sigma}{\cap} T$.

Definition 2.1 (Flat binary theory intersection)**.** A flat binary intersection $S \overset{\sigma}{\cap}_b T$ of two theories S and T over a invertible partial morphism $\sigma : S \to T$ is a new pair of theories I_S, I_T so that for any declarations d,e
$d \in dom(S) \wedge e \in dom(T) \wedge d \overset{\sigma}{\mapsto} e \Rightarrow d \in dom(I_S) \wedge e \in dom(I_T)$.

As seen in Figure 2.6 the binary intersection leaves us with a total view σ between I_s and I_T and it's inverse σ^{-1}. While this view σ (and it's inverse) is not strictly speaking identical to the original partial view σ, since they are views between different theories, we will still refer to them by the same name for notational purposes.

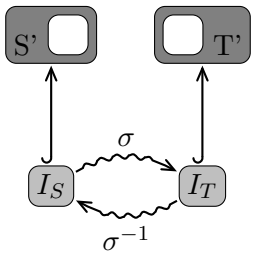


Figure 2.6: Binary Intersection between S and T

The advantage of a binary theory intersection is that it changes less about the theories themselves, since we do not need to worry about name changes.

Definition 2.2 (Flat unary theory intersection)**.** A unary intersection $S\overset{\sigma}{\cap}_u T$ of two theories S and T over an invertable partial morphism $\sigma : S \to T$ is a new theory I so that for any declarations d,e
$d \in dom(S) \wedge e \in dom(T) \wedge d \overset{\sigma}{\mapsto} e \Rightarrow d \in dom(I)$.

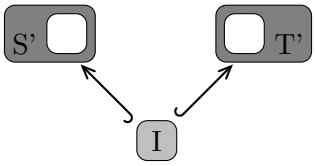


Figure 2.7: Unary Intersection between S and T

The unary theory intersection has the advantage that it reduces code duplication. However while the content of the original theories was structurally identical, the names were likely different and these naming differences are lost when creating the unary theory intersection. This means that unary intersections represent a more substantial change to the theory graph.

Binary theory intersections do not actually reduce code duplication, since all instances of constant declarations are merely moved into a different theory. However the changes made by binary intersections are less intrusive, as they merely encapsulate the content differently, making binary intersection a more conservative approach.

Note that the above definition assumes that since the above definition of unary theory intersections only takes constants from dom(S), constants in I will have the same name as in S. This may not be ideal, so name changes might be required. This is the primary reason why binary theory intersections are less intrusive and require less thought in their execution than unary intersections.

From now on we will call both of these *flat* theory intersections, because they only operate on flattened theories and do not take structures into account. This terminology change compared to previous works on the subject is needed to contrast them to *deep* theory intersections, which will be introduced in the next chapter (see Section 3).

To illustrate what a real theory intersection might look us consider a simple example from algebra:

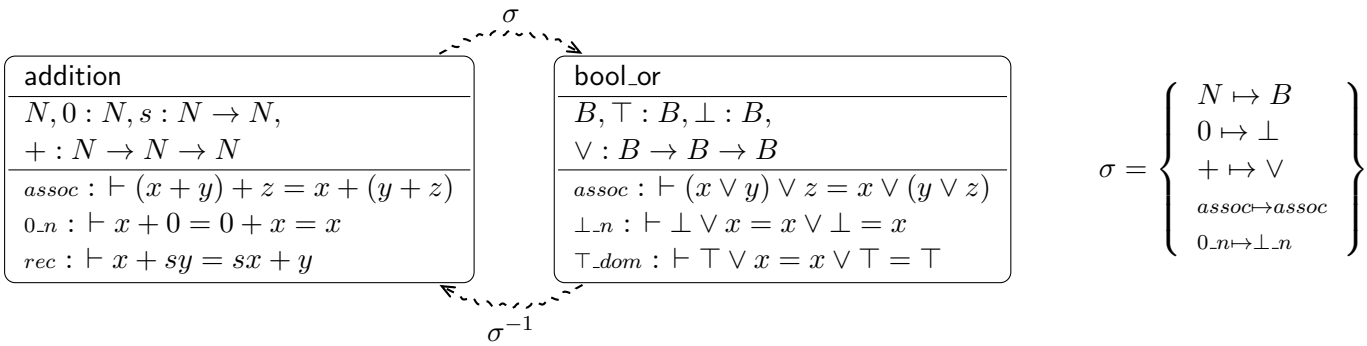


Figure 2.8: Partial view with inversion between addition and bool_or

The theories in the graph above (Figure 2.8) are *addition* and *bool_or*. They describe addition as we know it between natural numbers and the or operation from boolean logic. Both of these operations are binary functions, but there are more similarities than that. We can find these similarities by looking at the view ϕ and extract them into the unary theory intersection *monoid* as seen below in Figure 2.9.

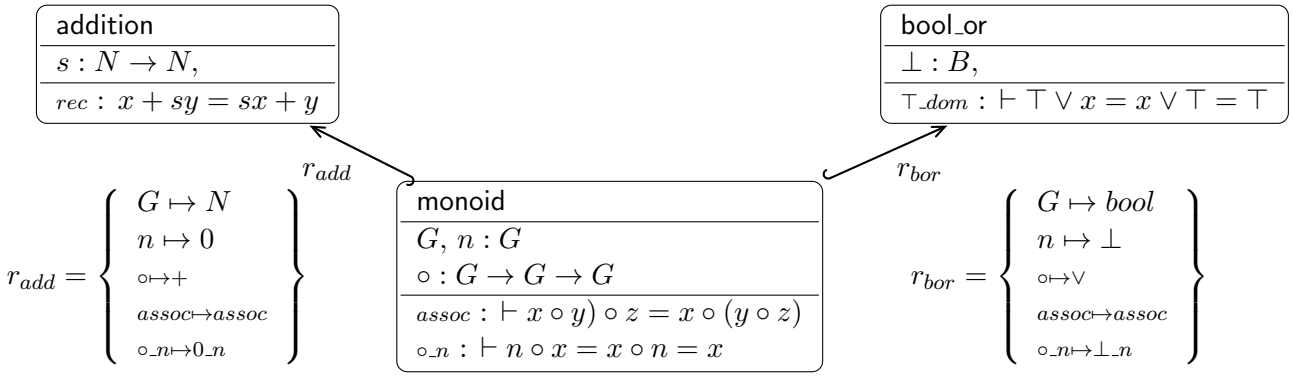


Figure 2.9: By intersecting over the partial view from Figure 2.8 we get the monoid theory.

As we can see this also changes the theories *addition* and *bool_or*, as we replace some of their content with a structure from the intersection to the original intersected theories.

Note that the names were changed according to what we would expect from a monoid. This work does not deal with this issue, but in practice appropriate naming changes are a likely necessity to preserve comprehension.

2.2.3 Creating Intersections from views

The process of creating a theory intersection involves the following steps:

1. Select a partial view σ
2. Reduce σ to a renaming
3. Create new theory with the constants $\{c \in dom(S) | c \in dom(\sigma)\}$

2.2.4 Inserting intersections into the theory graph

Introducing intersections into an existing theory graph of two theories S and T leaves us with two options, which change the theory graph in one of two ways.

The first way is to replace the theories entirely with the intersection and changed versions of S and T, which replace the intersected content with the inclusion of the appropriate theory from the intersection. The problem with this approach is that not only S and T would need changing, but also every theory that includes them and every morphisms that maps them. Even if measures are taken to ensure the same symbols can be used, they would at least need rebuilding.

The second way is to simply add the new versions to the theory graph and leave the old versions intact as legacy versions. This has the advantage that it doesn't break any pre-existing parts of the graph. However it does not offer any immediate benefit for any of those pre-existing parts either. Also it may cause additional confusion with conflicting names.

For this work we will go with the latter approach.

3 Deep Theory Intersections

Now that we have taken a look at what a flat theory intersection is, we will talk about some of their limitations and from there we will develop a new approach that overcomes at least some of these limitations.

We will refer to this approach as deep theory intersections, since the main difference is that it works by recursively intersecting the dependencies in depth, instead of operating on the already flattened theories.

3.1 Limitations of flat theory intersections

One of the fundamental aspects that makes the theory graph a graph is that theories within it are not universally flat. In order to properly intersect a theory, we need some way to address this issue.

In the following we will look at the example of the intersection between a pair of theories S_1 and T_1, which each have exactly one dependency called S_2 and T_2. Of course the exact structure of the graph can be a lot more expansive than this, both in breadth and depth. However as we will see even this simple example will already expand significantly. The general case of more complex graphs can quickly get confusing.

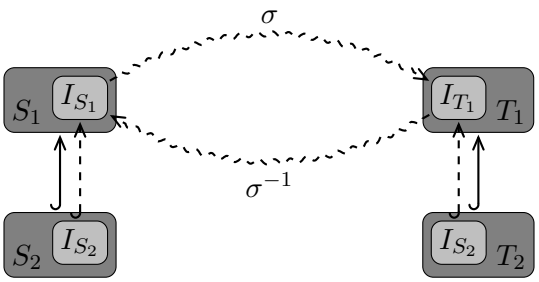


Figure 3.1: Invertible partial view between S_1 and T_1

Figure 3.1 shows us such a pair S_1,T_1 with a invertible view σ between them. Like in the previous chapter there is a proper subset of $\mathrm{dom}(S_1)$ called $\mathrm{dom}(I_{S_1})$ on which σ is defined, mapping it to the corresponding $\mathrm{dom}(I_{T_1}) \subsetneq \mathrm{dom}(T_1)$. The difference is that since this time S_1 is including a base theory S_2, there is also $I_{S_2} \subsetneq \mathrm{dom}(I_{S_1})$, which is located in S_2 and therefore not directly inside S_1.

The way flat theory intersections address this problem is by flattening the graph first and then working out the intersection between those flattened theories. Staying with our example graph, the resulting graph with a flat theory intersection looks something like seen in Figure 3.2.

The primary goal of theory intersections is to increase encapsulation and allow more morphisms to connect within the theory graph, through which knowledge can be transferred. This is directly at odds with the flattening that happens in flat theory intersections, as seen in Figure 3.2.

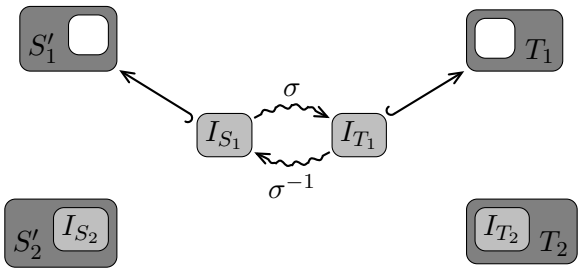


Figure 3.2: Existing structure is lost in the flattening process

A naive approach to solving the issue of losing this structure is to simply include the base theories in our changed theories S_1' and T_1', as seen in Figure 3.3.

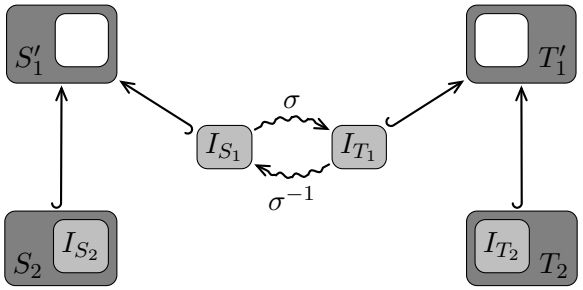


Figure 3.3: Simple attempt at restoring structure

The problem with this simple solution is that we cannot avoid doubling some constants in the resulting theories S_1' and T_1', namely those which are in I_{S_2} and I_{T_2}, since they also appear in I_{S_1} and I_{T_1}.

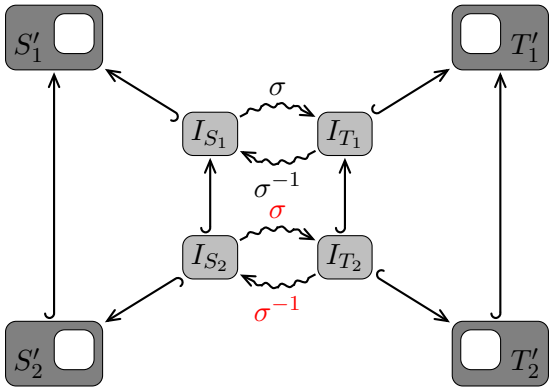


Figure 3.4: Naive deep intersection

We can solve this problem by further extracting all the constants that should be both in S_2' and I_{S_1} and encapsulate them in a separate theory I_{S_2} which can then be included by both S_2' and I_{S_1} (see Figure 3.4). The same can be done to the right side of the intersection.

Now all the constants declared in I_{S_2} and I_{T_2} are fully encapsulated within their own theories. Since the flattening provided by the semantics of MMT already ignores redundant inclusions, this means S_1' only has one copy of the constants in question.

However we now have a different problem, the graph in Figure 3.4 isn't a valid graph at all, at least not in the general case. The view $\sigma : I_{S_2} \to I_{T_2}$ does not necessarily exist.
Although we did select I_{S_2} specifically as those parts of S_2 that are mapped by σ, we did so using the original $\sigma : S_1 \to T_1$. That means while for every $c \in I_{S_2}$ there is a matching $d \in I_{T_1}$ so that $c \overset{\sigma}{\mapsto} d$, there is no guarantee that said d lies in I_{T_2}.

One solution is to simply remove the potentially non-existent view from the graph. This would leave us with a consistent theory graph, which allows us to use the view σ, which was after all our original goal. However this means giving up on the parts of σ that can be mapped between these two theories. Along these lines of thought we can see that the new problem is not entirely different to the one that brought us here in the first place.

For these reasons this thesis introduces the concept of deep theory intersections. Unlike flat theory intersections they retain the existing structure of inclusions within the theory graph, by intersecting not only the top level theories S and T, but also all of their dependencies against each other.

3.2 Deep Theory Intersections

3.2.1 Definition

Before we can define the deep theory intersection itself, we will first have to define a couple of helpful terms.

Definition 3.1 (dependency graph)**.** A dependency graph $\mathrm{dep}(S)$ of a theory S inside a theory graph γ is the smallest sub-graph of γ, so that

- $S \in \mathrm{dep}(S)$
- $\forall T, U \in \gamma.(T \hookrightarrow U \wedge U \in \mathrm{dep}(S)) \Rightarrow T \in \mathrm{dep}(S)$

Or defined as a recursive function

$$\mathrm{dep}(S) = \{S\} \cup (\bigcup_{I \in \mathrm{inc}(S)} \mathrm{dep}(I))$$

where $\mathrm{inc}(S) = \{I \in \gamma | I \hookrightarrow S\}$

We will call the theories inside dep(S) the dependencies of S.

Note that the way we have defined the dependencies of S, S is explicitly part of its own dependencies. It makes little difference whether we include it here, or explicitly add it to the set when needed.

Definition 3.2 (direct domain)**.** The direct domain $\overline{\mathrm{dom}}(S)$ of a theory S is the set of directly declared constants in S.

$$\overline{\mathrm{dom}}(S) = \{c \in \mathrm{dom}(S) | (c[: \varphi][= \psi]) \text{ is a declaration in S}\}$$

Note that by design any constant defined in a dependency of S that is not S itself is not part of $\overline{\mathrm{dom}}(S)$. Indeed this property is the primary purpose. Alternatively we can state that

$$\overline{\mathrm{dom}}(S) = \mathrm{dom}(S) \setminus \bigcup_{D \in \mathrm{dep}(S) \setminus \{S\}} \mathrm{dom}(D)$$

This allows us to make use of the desirable property that every constant c is found in the direct domain of exactly one theory S.

Like the flat theory intersections before them, deep theory intersections come both in a binary and a unary variant.

Definition 3.3 (Deep binary theory intersection)**.** A deep binary intersection of two theories S and T over a invertible partial morphism $\sigma : S \to T$ is a new theory sub-graph consisting of theory pairs and views between those pairs. The theories inside this intersection are called *intersection theories*.

The sub-graph of the intersection contains a *root*, which is a pair of intersection theories $(I_{S,T}, I_{T,S})$ with a pair of isomorphic views between them.

The contents of the intersection theory $I_{S,T}$ of the theories S and T are as follows:

- Every constant from S which is assigned in σ to a constant in T
- An inclusion of every theory I_{S,T_i} for any T_i such that $T_i \hookrightarrow T_y$
- An inclusion of every theory $I_{S_i,T}$ for any S_i such that $S_i \hookrightarrow S$

The theory $I_{S_i,T}$ is itself the root of the intersection between S_i and T and constructed accordingly. Similarly I_{S,T_i} is the root of the intersection between S and T_i. For notational purposes all intersection theories are indexed by the theories of which they are the root of their intersection.

The contents of $I_{T,S}$ are filled analogously using the inverse view σ^{-1} and recursing along the dependencies in the same manner.

The theories contained in the intersection are exactly the theories in the root and their dependencies.

The views between each pair are σ and σ^{-1} restricted to the accessible constants in each intersection theory.

Since a deep theory intersection is made up of multiple theories, which may be limited only by the depth of the original dependency graph, we will refer to the individual theories that make up the intersection as intersection theories. When referring to a specific intersection theory (or a specific pair of intersection theories) of S and T, we mean the intersection theory $I_{S,T}$ (when talking about pairs, we mean the pair ($I_{S,T}$,$I_{T,S}$)).

Just like there were two intersection theories in the flat binary intersection, deep binary intersections come in pairs, albeit in this case multiple pairs. When flattening the intersection, the resulting root is a flat binary intersection of S and T.

Since in the case of binary theories for every pair of intersected theories there is also a pair of intersection theories, we will differentiate between them by naming the theory from which the intersection theory was extracted as the first index.
For example the pair S_1, T_1 yields the intersection theories I_{S_1,T_1} and I_{T_1,S_1}, with I_{S_1,T_1} containing the intersected content of S_1 and I_{S_1,T_1} containing the intersected content of T_1.

Since these pairs are related by the property that there is an isomorphic view between the two, when talking about a pair of intersection theories $(I_{S,T}, I_{T,S})$, we will refer to $I_{T,S}$ as the *isomorphic partner* of $I_{S,T}$ and vice versa. The pair as a whole is called an *isomorphic pair*.

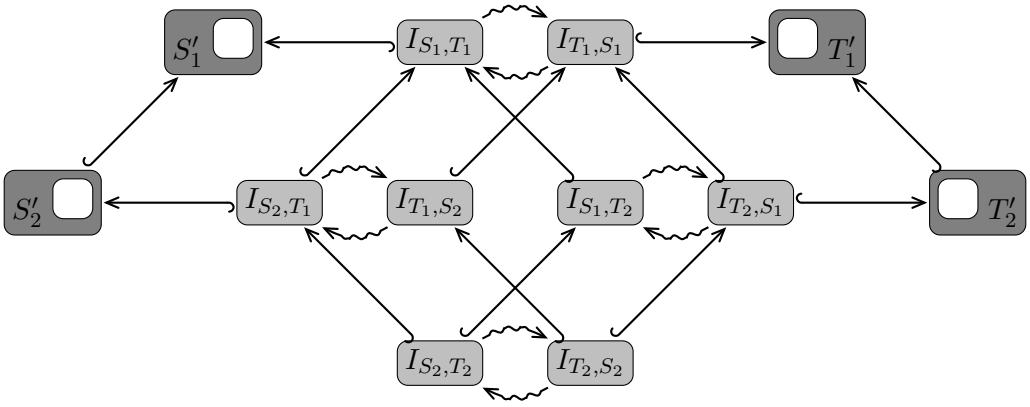


Figure 3.5: Deep binary Intersection between S_1 and T_1

Definition 3.4 (Deep unary theory intersection)**.** A deep unary intersection of two theories S and T over an invertible partial morphism $\sigma : S \to T$ is a new theory sub-graph consisting of theories. The theories inside this intersection are called *intersection theories*.

The sub-graph of the intersection contains a *root*, which is the intersection theory $I_{S,T}$).

The contents of $I_{S,T}$ of the theories S and T are as follows:

- Every constant from S which is assigned in σ to a constant in T
- An inclusion of every theory I_{S,T_i} for any T_i such that $T_i \hookrightarrow T_y$
- An inclusion of every theory $I_{S_i,T}$ for any S_i such that $S_i \hookrightarrow S$

The intersection theory $I_{S_i,T}$ is itself the root of the intersection between S_i and T and constructed accordingly. Similarly I_{S,T_i} is the root of the intersection between S and T_i. For notational purposes all intersection theories are indexed by the theories of which they are the root of their intersection.

The theories contained in the intersection are exactly the root and these dependencies.

Like the flat unary intersection, the deep unary intersection is the result of replacing the isomorphic pair in a deep binary intersection with a single theory. However like the deep binary version, the deep unary intersection consists of multiple dependent intersections between the pairings between the two sides' dependencies. This means we have to replace not just one, but every pair with a single intersection theory. When flattening the deep unary intersection, the resulting root is a flat unary intersection of S and T.

Since the intersection theories only take the constants from the domain of the view, it is very likely that the naming scheme does not fit the codomain, which will now include constants of the same name. For this reason it is strongly recommended to refactor the theory by renaming these constants in a more generic manner. It is also possible to add their old names as aliases by including them in a structure.[7]

In both cases we may recognize that the way a deep intersection $S \underset{deep}{\overset{\sigma}{\cap}} T$ is constructed is recursive. In fact this recursion means that an intersection theory I_{S_x,T_x} on a lower level of $S \underset{deep}{\overset{\sigma}{\cap}} T$ is itself the root of an intersection $S_x \underset{deep}{\overset{\sigma}{\cap}} T_x$.

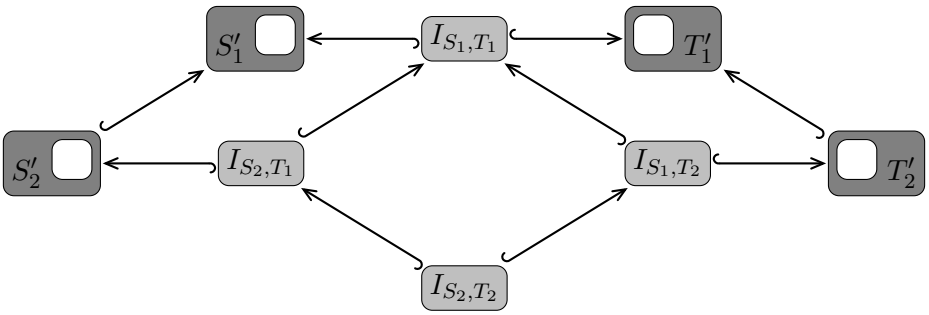


Figure 3.6: Deep unary Intersection between S_1 and T_1

3.2.2 Linking up the intersections to the theory graph

In the definition of a deep intersection $S\overset{\sigma}{\cap}T$ we have seen how the intersection theories themselves are formed, but we have skipped over what happens to the original dependencies of S (and T).

Once we have calculated all the intersection theories of $S\overset{\sigma}{\cap}T$ for every I_{S_x,T_y} we remove the declaration of every constant $c \in dom I_{S_x,T_y}$ from dep(S) and remove the corresponding declarations of $\sigma(c)$ from dep(T). Then replace all occurrences of c in dep(S) with the new c in I_{S_x,T_y}. In dep(T) replace all occurrences of $\sigma(c)$ with the $\sigma(c)$ declared in I_{T_y,S_x} for the binary case and the same c in I_{S_x,T_y} for a unary theory intersection. This means in the case of the unary intersection we use the constant assignments in σ as a term substitution on all terms in the remainder. We call a theory S_x that has been changed in such a way S_x' to differentiate it from the original.

Due to the fact that a such changed theory S_x' contains those constant declarations in S_x that remain after taking out all of those that are inside the intersection theories, we will call S_x the remainder (theory) of S.

In order to make sure that all of these replacements reach the correct theories, we have to make sure that $\forall S_x \in \text{dep}(S) : I_{S_x,T} \hookrightarrow S_x$. Accordingly in the binary case we ensure that $\forall T_y \in \text{dep}(T) : I_{T_y,S} \hookrightarrow T_y$ and in the unary case $\forall T_y \in \text{dep}(T) : I_{S,T_y} \hookrightarrow T_y$. This is the only direct inclusion of an intersection theory that S_x and T_y make. The end goal of this process is that after flattening the new theory S_x' has the same domain as the original theory S_x.

Notice that it is always the intersection between the theory itself and the top level theory on the other side of the intersection, which is included in the changed theory. Our intuition may fool us into assuming that the proper intersection to include in the remainder theory would have to be between the theories of the same level in the dependency subgraph of their respective sides, but this is not the case, since the dependencies are not necessarily structured the same. In fact one possible application of deep theory intersection is to impose an already existing structure in the dependencies of one theory onto a different theory via a partial view.

3.2.3 Other graph shapes

Until now we have exclusively looked at the case of $S\overset{\sigma}{\cap}T$ exactly one inclusion per S and T. Of course the exact structure of the dependency graph can be more complicated than that. It is relatively easy to imagine what examples with fewer dependencies look like, so we will instead take a look at more complex structures.

Again we will primarily look at the unary case, since it is easier read, but all observations

equally apply to the binary case, if the unary intersection theory is replaced with an isomorphic pair.

In the case of no inclusions we are de facto just dealing with flat theories and it is therefore no surprise that we do end up with exactly the same structure as if we were applying a flat theory intersection.

$$S \underset{deep}{\overset{\sigma}{\cap}} T = S \underset{flat}{\overset{\sigma}{\cap}} T \Leftrightarrow \neg\exists U, V : S \hookrightarrow U \wedge T \hookrightarrow V$$

More difficult is the case of more than one theory inclusion as seen with $S_1 \overset{\sigma}{\cap} T$ in Figure 3.7. Here we have a theory S_1 with N theory inclusions.

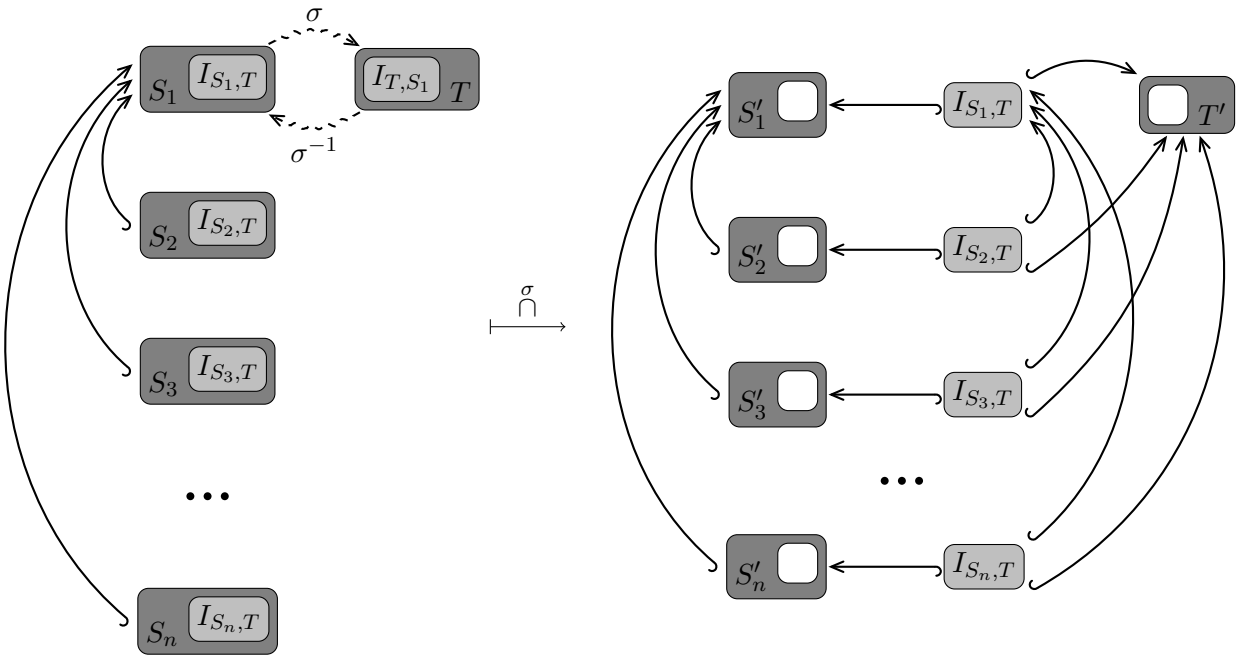


Figure 3.7: $S_1 \overset{\sigma}{\cap} T_1$ with S_1 having more than 1 inclusion

As we can see the result is again an intersection theory for every pair $(U, V) \in \text{dep}(S_1) \times \text{dep}(T)$, but this time all the lower level intersection theories are included directly in I_{S_x,T_x}.

3.3 Trivial and redundant intersection theories

Not all intersections are equally valid as improvements to the theory graph. Especially now that we have seen how quickly a rather simple theory graph can have quite complicated intersections, due to the potential blow up of $|\text{dep}(S) \times \text{dep}(T)|$ theory intersections, we have to worry about the cost of introducing intersection theories that are entirely worthless, even if the theory intersection in which they exist is not.

Fortunately some of these pathological cases are easy to recognize and are just as easily dealt with, as described below. While we will only look at the cases for unary theory intersections, all of these can be applied just the same to binary intersections. The only difference is that every change that is made to a intersected theory $I_{S,T}$ is done to both members of the pair $(I_{S,T}, I_{T,S})$.

3.3.1 Trivial intersection theories

Going back to the example graphs from above, even if σ is a meaningful morphism on which it makes sense to intersect, with σ not being total it could very well be that σ is mapping only constants in S_1 that are not in S_2. In this case the resulting intersection theory I_{S_1,T_2} would have an empty domain and since $\text{dom}(I_{S_1,T_1}) \subseteq (I_{S_1,T_2})$, so would be the domain of I_{S_1,T_1}. We refer to such theories as trivial intersection theories.

Trivial intersection theories can be omitted from the theory graph entirely, including all references to them (primarily inclusion in higher level intersection theories). This has no effect on the domain of the theories that would otherwise include them, because the domain of the trivial intersection theory is by definition empty.

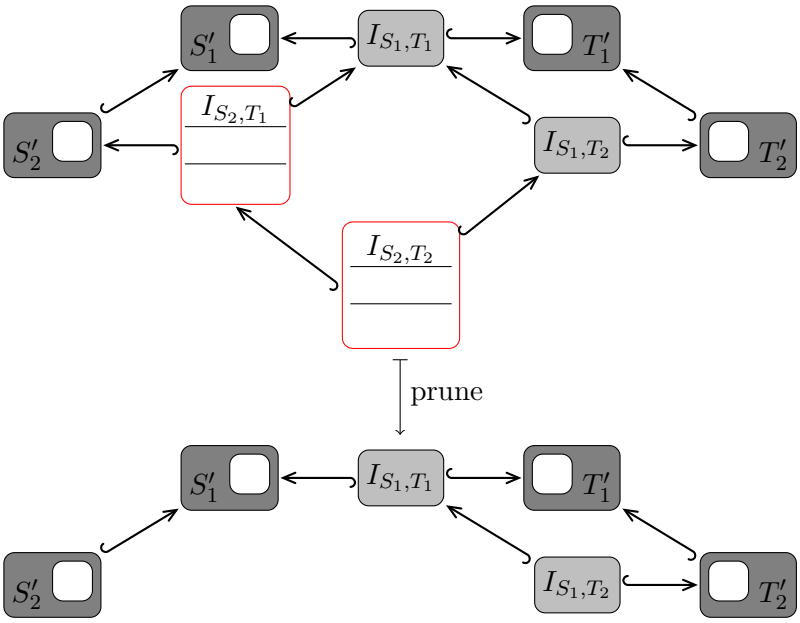


Figure 3.8: Deep unary Intersection between S_1 and T_1, with the trivial intersections I_{S_1,T_2} and I_{2_1,T_2} removed

Applying this idea to the scenario above and making the appropriate changes to the theory graph from Figure 3.6, we will get the more compact but equally useful theory graph seen in Figure 3.8.

The rationale behind removing trivial intersections should be obvious. The main reason why we want to intersect a theory in the first place is to exploit a common structure within the two theories. If this common structure does not exist then there is no point in making the intersection.

3.3.2 Redundant intersection theories

Similar to the trivial theories there are some theories, which do not contain any constant declarations of their own, meaning that their direct domain is empty. If these are entirely empty, they are trivial and already dealt with. However some theories contain only theory inclusions.

This is not necessarily a bad thing in itself. If a theory bundles multiple different theories, it

may convey meaning, which we can easily see when considering that such a theory may allow the formulation of a view that could not be formulated on any of its other dependencies.

However this is clearly not the case if the theory in question consists entirely of a single inclusion. Such a theory is with regards to semantics by flattening identical in all but name to the theory it includes. Since this is the case we can safely remove it, as long as we replace all occurrences of its inclusion with the redundant theory's own inclusion.

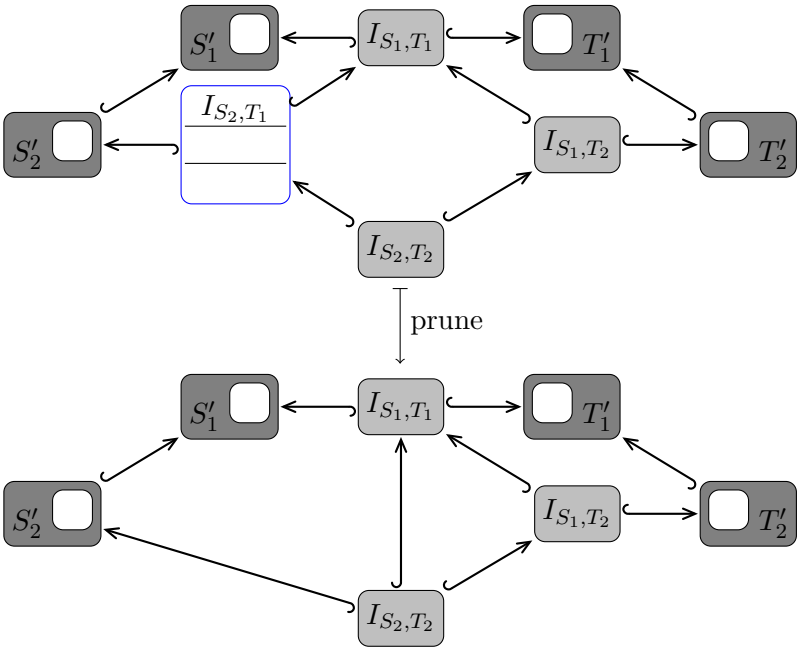


Figure 3.9: Deep unary Intersection between S_1 and T_1, with the redundant intersection I_{S_1,T_2} removed

Going back to the example graph in Figure 3.6, if I_{S_1,T_2} consists entirely of an inclusion of I_{S_2,T_2} and is otherwise empty (i.e. it has no other inclusions and $\overline{\text{dom}}(I_{S_1,T_2}) = \emptyset$), we can remove it as a redundant theory and replace its inclusion in I_{S_1,T_1} with an inclusion of I_{S_2,T_2}.

The rationale behind removing redundant intersection theories is less clear cut than in the case of trivial intersections. A redundant theory intersection does convey some interesting knowledge, but it doesn't provide any extra benefit over the theory that it includes.

3.3.3 Pruned theory graph

We can apply both of these newly developed insights about removable theories to our graph as a sort of pruning rule. Our end goal for a pruned theory intersection is that it contains no trivial or redundant theories.

Definition 3.5 (Fully pruned theory graph)**.** A theory graph γ is fully pruned, if

- There are no Theories $T \in \gamma$, so that

$$\text{dom}(T) = \emptyset \tag{3.1}$$

- There are no redundant Theories $R \in \gamma$, so that

$$\overline{\text{dom}}(R) = \emptyset \wedge \exists! S : S \hookrightarrow R \tag{3.2}$$

The simple way to go about pruning is to take a finished theory graph, see if there are any trivial or redundant theories, and if so remove them from the graph. However this may face the problem that some theories may only reveal themselves as such once we have already pruned other theories, such as the example in Example 3.6.

Example 3.6. Consider the example of the intersection $S_1 \overset{\sigma}{\cap} T_1$ as depicted in Figure 3.10.

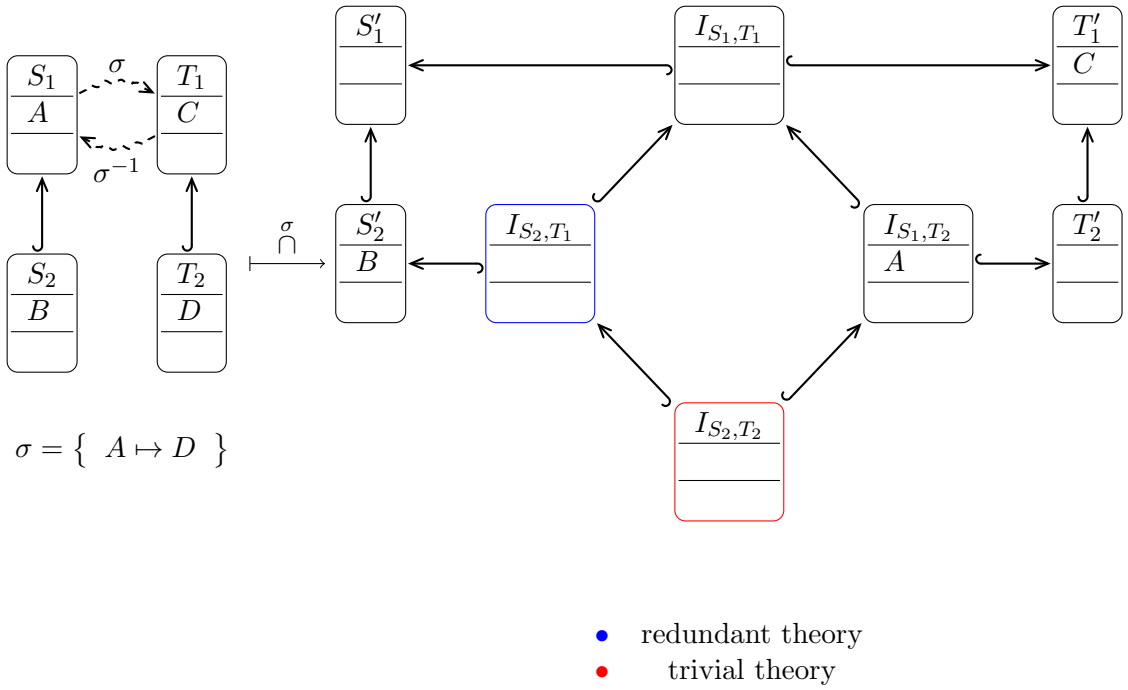


Figure 3.10: Unpruned theory graph of intersection $S_1 \overset{\sigma}{\cap} T_1$

It is immediately apparent that most of the intersection theories are empty and our intuition should tell us, that most of these theories are not exactly interesting and should probably be pruned.

Applying our two pruning rules for redundant and trivial intersections we find the following pruning candidates:

$I_{S_2,T2}$ has neither constant declarations nor inclusions. That makes $I_{S_2,T2}$ a trivial intersection.

$I_{S_2,T1}$ has no constant declarations and a single inclusion of $I_{S_2,T2}$, making it identical to $I_{S_2,T2}$ and is therefore a redundant intersection.

Before further pruning $I_{S_1,T1}$ also has no constants declared directly inside of it, however it is neither trivial nor redundant at this point, because it still has more than 1 inclusion.

After a first round of pruning we can see in Figure 3.11 that $I_{S_2,T2}$ and $I_{S_2,T1}$ have been removed.

However with both its inclusion of $I_{S_2,T1}$ and its replacement $I_{S_2,T2}$ removed from $I_{S_1,T1}$ this leaves the theory with only one remaining inclusion and still no direct constant declarations. This means $I_{S_1,T1}$ is now also redundant, since its contents are exactly the ones of $I_{S_1,T2}$. As such it can and will be removed in the next round of pruning. This proofs the assertion that some prunings are dependent on performing other prunings first.

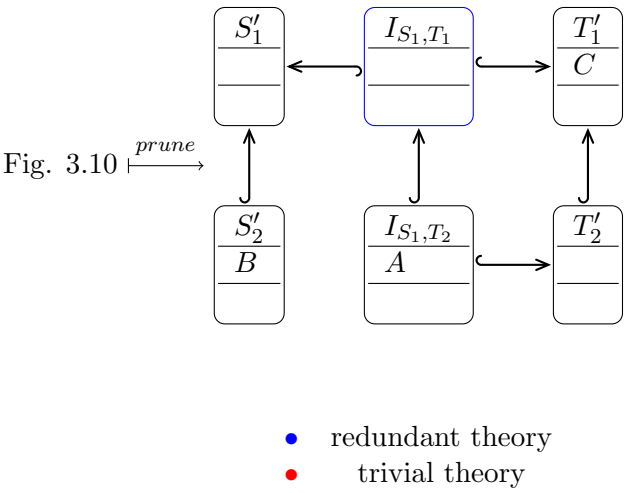


Figure 3.11: Pruning reveals more trivial and redundant theories, as it progresses

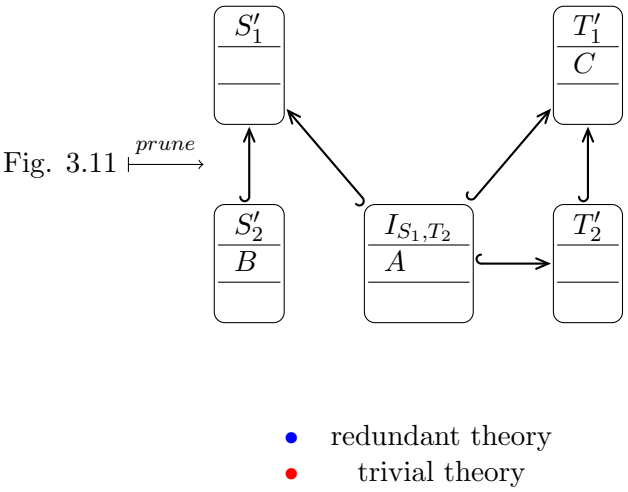


Figure 3.12: Intersection is fully pruned

After pruning $I_{S_1,T1}$ the intersection graph is finally fully pruned as seen in Figure 3.12. This took 2 iterations over the graph.

Note that T'_2 is also a redundant theory, since it consists entirely of an inclusion of the intersection $I_{S_1,T2}$ and has no remaining content after it has been extracted into $I_{S_1,T2}$. However we do not prune it at this stage, since it is not part of the intersection. In fact a better approach to dealing with this situation is treating the intersection as redundant due to having the same content as T_2 and keeping the original theory $T2$ intact. More on this topic in Subsection 3.3.4.

One method to solve this issue and fully prune a graph is to repeat the process, until there are no more changes to the graph. This would make a fully pruned graph a fixed point of pruning. A fully pruned graph has to be a fixed point of pruning, since our pruning rule is to remove any node that violates the rules for a fully pruned graph.

However such a change in a theory's prunability can only happen to a theory $S \in \gamma$, if the algorithm prunes a $T \in \text{dep}(S)$. It can never happen to a leaf of the graph (i.e. the base theories), they are either trivial or they are not.

For these reasons it is best if pruning happens bottom up. Fortunately, as we will see in Section 4

that is also the order in which we will build our theory intersection graph and since both of these are easily recognized at construction, we can begin the pruning process from the moment we start building the graph.

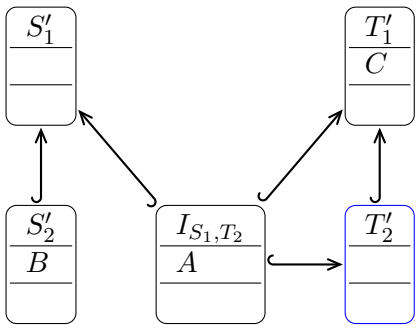
Theorem 3.7. *Any graph pruned from bottom up is a fully pruned theory graph.*

Proof. By induction over structure of graph.

Assumption :	Applying a pruning step to the root of a sub-graph S that has been applied to all nodes below it, leaves a pruned sub-graph dep(S).	
Induction base :	leaf node S :	$\neg\exists T : T \hookrightarrow S$
	case $\overline{\text{dom}}(S) \neq \emptyset$:	node is neither trivial nor redundant
	case $\overline{\text{dom}}(S) = \emptyset$:	node is trivial and pruned
Induction step :	node S :	$T_1, ..., T_n \hookrightarrow S$
	case $\overline{\text{dom}}(S) \neq \emptyset$:	node is neither trivial nor redundant and every dependent sub-graph is fully pruned by assumption
	case $\overline{\text{dom}}(S) = \emptyset$:	
	case n=0 :	as in induction base
	case n=1 :	is redundant and pruned and remaining dependent sub-graph is fully pruned by assumption
	case n=1 :	is neither trivial nor neither redundant and dependent sub-graph is fully pruned by assumption

□

3.3.4 Redundant remainders



- redundant theory

Figure 3.13: T'_2 is a redundant remainder

We saw in Example 3.6 that intersecting theories can result in remainder theories that are themselves redundant theories. For the same reason that we did not want redundant intersection theories, we may want to rid the graph of redundant remainders.

However unlike redundant intersections we may want to keep our redundant remainders intact for the reason, that these are original theories that were quite likely the result of human design instead of being machine generated. By definition their content would be mostly the same, but we lose the advantage of names that have been chosen to correspond to human intuition for exactly this particular theory.

There are multiple ways we can deal with this problem. The first and simplest is to keep the redundant remainder as it is seen in Figure 3.13.

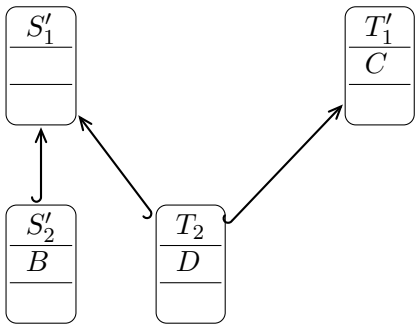


Figure 3.14: T_2 replaces I_{S1,T_2}

Alternatively we can reverse the approach of our usual pruning and instead replace the appropriate intersection theory with the original theory T_2 instead, as seen in Figure 3.14 for a unary intersection. For the unary intersection this would also mean that a reversal of the usual renaming is recommended, if the intersection is identical to the right side of the original theory pair. This way not only do we keep the human picked name for the theory, T_2, but we also get to keep the naming scheme of the theory content. In case of a binary intersection, the original theory would replace the side of the pair, which would be included by its remainder.

Fortunately there is no need to come up with a strategy to deal with trivial remainders. Since the intersection process and linking up of the remainders has the goal of keeping the domain of the theory intact. While this does not necessarily guarantee that there will not be a trivial remainder in our changed graph, it does mean that the remainder can only be trivial if the original theory has already been trivial before the intersection even took place.

Similar reasoning applies to the case of a redundant remainder theory which consists solely of a inclusion of a different remainder theory. This can also only be the case if the original theory corresponding to this remainder was already redundant itself.

3.4 Deep theory intersections and named structures

The deep intersection concept allows us to retain the structure of inclusions within the theory graph, which would have been lost with the flat intersection approach. However the deep theory concept still struggles with named structures.

Deep theory intersection relies that for any mapping of $c \mapsto d$, both c and d lie uniquely within a single intersection. By allowing for named structures this is no longer the case, since they might result in a single constant being mapped to two different terms. This would not usually be a problem, since within the theory which declares the structure these copies are in fact two distinct constants. However in the context of our deep theory intersections it brings up the problem that now a constant might have to be directly in two entirely distinct theories at the same time. When these theories come together through inclusion, we again have the problem of multiple declarations of the same constant and since they do not come as inclusions of the same theories, this will lead to an inconsistent theory.

This duplication could potentially be salvaged by utilizing a very drastic and highly interconnected version of the tiny theories approach [10]. It would in theory be possible given a theory S to introduce one constant $c \in \mathrm{dom}(S)$ at a time in tiny theories and create a net that would contain any theory I, so that $\mathrm{dom}(I)$ is equal to any set $P \in \wp(\mathrm{dom}(S))$.
However it should be immediately obvious that even in relatively small cases, this will explode the number of theories considerably. Extending a theory graph this way could very well make

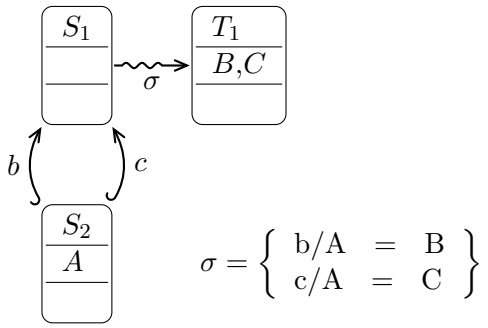


Figure 3.15: Structure creates ambigiuity between b/A and c/A in the intersection of S_1 and T_1

it utterly incomprehensible and therefore is not in fact an improvement.

In conclusion, although it is desirable to be able to process named structures correctly, it makes little sense to allow theory intersections on such structures, unless we find an adequate solution to deal with these issues. However since the problem with named structures is on the conceptual level of what a such an intersection would look like, it would likely require a substantially different theoretical foundation.

Note that this does not entirely preclude the use of structures within a theory that is intersected, but it does mean that the structure and the constants which it copies should not be part of the intersection.

4 Automated approach to generating intersections

In order to automatically refactor a graph via intersections we need an algorithm that is as hands off as possible. For this reason our algorithm starts with the theory graph as its input and returns us a list of suggested additions to the theory graph.

```
   Data:
   γ : theory graph
   Result: γ_1, ..., γ_n : theory graph
 1 Function findIntersections(γ) is
 2 |   views := viewfinder(γ);
 3 |   renamings := new List[morphism];
 4 |   for σ ← views do
 5 |   |   renamings += reduceToRenaming(view);
 6 |   end
 7 |   for i ← 1, ..., n do
 8 |   |   σ := renamings[i];
 9 |   |   S := σ.domain;
10 |   |   T := σ.codomain;
11 |   |   γ_i := intersectGraph(γ,S,T,σ);
12 |   end
13 |   res := select(γ_1, ..., γ_n);
14 |   return γ_1, ..., γ_n;
15 end
```

Algorithm 1: findIntersections(γ)

The program found in algorithm 1 can be divided into roughly three phases.

lines	phase	
2 to 6	1	Finding partial views
7 to 12	2	Translating partial views into intersections
13	3	Filtering the results

4.1 Finding partial views

As discussed in both definitions of theory intersections (see Section 2.2 and Subsection 3.2.1) the $S \overset{\sigma}{\cap} T$ is defined over the partial view σ, with σ being a partial renaming.

This means there are two things that need to happen if we want to start producing intersections on a theory graph γ.

First we need to find appropriate partial views. Fortunately a tool already exists that can find theory morphisms and in doing so it produces not only fully fledged views, but also partial views, which are more of a byproduct, but perfectly suited to our purposes.[4] This is what

happens in line 2 of algorithm 1.

```
2 views := viewfinder(γ);
```

Secondly we need to make sure that the partial views we gathered are indeed invertible, i.e. they are renamings. There are two ways to achieve this, one is conservative but produces smaller intersections, the other produces larger intersections, but at the price of altering the theories and the resulting intersection.

We can see that the conservative approach in algorithm 2 is relatively simple. We simply filter the constant assignments in the input theory morphism σ so that only those remain which map to a constant.

```
   Data:
   σ : theory morphism S → T
   Result: σ' : theory morphism
1  Function reduceToRenaming(σ) : theory morphism is
2  |  σ' := new theory morphism;
3  |  for (c ↦ τ) ← σ do
4  |  |  if τ is a constant then
5  |  |  |  σ' += c ↦ term;
6  |  |  end
7  |  end
8  |  return σ';
9  end
```

Algorithm 2: reduceToRenaming(σ)

The alternative is a little more involved, since it not only requires making changes to the view itself, but also to the mapped theories, before the intersection process even begins. We can keep the mappings to complex terms in $\sigma : S \to T$, if we make a conservative extensions T' of T, so that for every $(c \mapsto \tau) \in \sigma$ τ is a constant or there is a $c' \in T'$ with a definiens so that $c' = \tau$. This method has already been outlined by Müller.[3]

4.2 Translating partial views to intersections

The core of the algorithm is translating the partial views to intersections. This happens in the subroutine intersectGraph (seealgorithm 3).

This subroutine takes the graph, two theories S and T inside the graph and a partial renaming σ between them.

It returns a proposed graph change γ' which is a subgraph which could replace the subgraph of the original γ which contains $dep(S) \cup dep(T)$. This changed subgraph contains all the intersections of $dep(S) \times dep(T)$ as well as the appropriately changed theories in $dep(S) \cup dep(T)$ to include these intersections.

The subroutine is further subdivided into two parts: The creation of the intersection theories and the creation and linking up of the remainder theories.

Data:
γ : theory graph
S, T : theory in γ
σ : theory morphism $S \to T$
Result: γ' : theory graph
1 **Function** *intersectGraph(γ,S,T,σ) : theory graph* **is**
2 | intersectionTheories := intersect(γ,S,T,σ);
3 | γ' := createRemainder(S,T,γ);
4 | **return** γ';
5 **end**

Algorithm 3: intersectGraph(γ,S,T,σ)

4.2.1 Creating intersections from views

Data:
γ : theory graph
S, T : theory in γ
σ : theory morphism $S \to T$
Result: γ' : theory graph
1 **Function** *intersect(γ,S,T,σ) : theory graph* **is**
2 | γ' := new theory graph;
3 | **if** *exists* $I_{S,T}$ **then**
4 | | **return** γ';
5 | **end**
6 | **for** $i \leftarrow S.includes$ **do**
7 | | γ' += intersect(γ,i,T,σ);
8 | **end**
9 | **for** $j \leftarrow T.includes$ **do**
10 | | γ' += (intersect(γ,S,j,σ));
11 | **end**
12 | $I_{S,T}$:= createIntersection(S,T,σ);
13 | γ' += $I_{S,T}$;
14 | **return** γ';
15 **end**

Algorithm 4: intersect(γ,S,T,σ)

In order to keep track of the created intersections and to avoid duplication we need a set of variables $I_{S,T}$ for each possible combination of theories S and T which is globally accessible within the different subroutines. A suitable solution is a mutable map of $(theory \times theory) \to theory$.

It is also at the level of intersect(γ,S,T,σ) where the pruning happens, as discussed in Section 3.3.

The subroutine createIntersection (algorithm 5) is where the individual intersection theory itself is created. Its input are the actual theories that are being intersected S and T, and the theory morphism σ over which it is intersected. It returns the intersection theory $I_{S,T}$.

In order to include the correct intersections it also needs access to the variables where we store the already calculated intersection. Since we recurse over the theories inclusions before building the intersection itself, all of these will already be available.

Data:
S, T : theory in γ
σ : theory morphism $S \to T$
Result: I : theory
Function *createIntersection(S,T,σ) : theory* **is**
  I := new theory;
  for $i \leftarrow S.includes$ **do**
    I += include($I_{i,T}$);
  end
  for $j \leftarrow T.includes$ **do**
    I += include($I_{S,j}$);
  end
  for $dec \to S.declarations$ **do**
    if $(dec \mapsto X) \in \sigma$ **then**
      I+=rename(dec));
    end
  end
  return I;
end

Algorithm 5: createIntersection(S,T,σ)

4.2.2 Linking intersections to the graph

After the intersection theories have been built it is time to link them up to the graph. We do so, by creating remainder theories S'_x and T'_y for every $S_x \in dep(S)$ and $T_y \in dep(T)$ that conform to the rules we established in Subsection 3.2.1. This happens in the createRemainder function seen in algorithm 6.

Data:
γ : theory graph
S, T : theory in γ
Result: γ' : theory graph
Function *createRemainder(S, T, γ) : theory graph* **is**
  Left := createRemainderSide(S,T,γ);
  Right :=createRemainderSide(T,S,γ);
  return γ + *Left* + *Right*;
end

Algorithm 6: createRemainder(S, T, γ)

In order to reach all the dependencies of the original S and T, createRemainder requires a subroutine createRemainderSide that recurses over one side, determined by the order of theories passed in the argument (see algorithm 7).

The function createRemainderSide starts by making a new remainder theory S' which receives an inclusion of its top level intersection $I_{S,T} \hookrightarrow S'$. Then it recurses along the left side theory S along their inclusions. Each inclusion is then copied into the remainder theory with the domain changed to the newly created theory from the lower recursion level. Once all the theories are included, we pick all the constants from S which are not in the intersection.

In order to traverse the right hand side of the intersection, the arguments are switched.

Data:
γ : theory graph
S : theory in γ over which to recurse T : theory on the top level of the other side
Result: list : List[theory]

```
1  Function createRemainderS(S, T, γ) : theory graph is
2      S' := new theory;
3      list := new List[theory];
4      S' += include(I_{S,T});
5      for i ← {i ∈ γ | i ↪ S} do
6          list += createRemainderS(i, T, γ) S' += include(list.head);
7      end
8      for dec ← S.declarations do
9          if rename(dec) ∉ I_{S,T} then
10             S' += rename(dec);
11         end
12     end
13     return S' + list;
14 end
```

Algorithm 7: createRemainderSide(S, T, γ)

4.3 Filtering the results

Not all Intersections are desirable. We have already seen this in Section 3.3 and taken steps to get rid of these obviously unnecessary pruning candidates however sometimes the reasons for keeping or discarding an intersection are more ephemeral. We will take a closer look at these reasons in Section 5, but for now we will see how the algorithm can sort out these rejected candidates given a specific evaluation function.

4.3.1 Conflicting intersections

One of the reasons we want to get rid of at least some proposed changes is that not all intersections are compatible with each other. In fact any intersections $S \overset{\sigma}{\cap} T$ and $S \overset{\tau}{\cap} U$ so that there is a μ so that $I_{S,T} \overset{id_{S'}}{\cap} I_{S,U} \neq \emptyset$, are incompatible with each other in the same theory graph. This is because S' can only include one of the two intersection theories without having two instances of the same constant. Since this is again a conceptional problem of the intersection, it is not a matter that can be solved by simply making additional changes to the graph.

However even a case of $S \overset{\sigma}{\cap} T$ and $S \overset{\tau}{\cap} U$ without such an overlap in its intersections is slightly problematic to handle, because it will require additional changes to the graph structure and complicate any further considerations in this chapter. There are also additional problems that multiple intersections are not commutative, so the order matters.

For this reason if we do not want to return conflicting changes to the theory graph, it is easiest if we select just one intersection out of any pair of two intersections that share at least one common intersected theory.

In order to not just pick the first pair that the algorithm happens to find, the algorithm allows for a sort function to be applied. This can range from simply allowing more intersections to be applied all the way to filtering out intersections that aren't actually beneficial, as discussed in Subsection 4.3.2.

Data:
$\gamma_1, ..., \gamma_n$: theory graph
Result: $\gamma_1, ..., \gamma_m$: theory graph
Function *select*($\gamma_1, ..., \gamma_n$) **is**
  sorted := sort(γ);
  res := new list[theory graph];
  for $\gamma \leftarrow$ *sorted* **do**
    if $(\bigcup_{r \in res} r) \cap dep(\gamma.S) \wedge (\bigcup_{r \in res} r) \cap dep(\gamma.T)$ **then**
      res += γ;
    end
  end
  return *res*;
end

Algorithm 8: select($\gamma_1, ..., \gamma_n$)

After the list of alteration candidates has been suitably sorted, we simply pick any changed subgraph that does not conflict with a previous pick, until all changes are either thus picked or discarded.

4.3.2 Meaningful intersections

An additional problem facing the automation of the theory intersection process is the problem of finding intersections that are actually meaningful and improve the structure of the graph. The danger is that these new connections cannot be properly exploited, because they make the graph too hard to read for a human user.

We have already somewhat mitigated this problem with methods to cut out some of less useful intersection theories with the methods presented in Section 3.3. However this only helps us filter out some of the worst offenders and does not really guarantee the result would make a lot of sense to a human reader.

To further filter out bad intersections it is desirable to have measures of what makes a graph better than others. Unfortunately currently there are no established heuristics that would allow an accurate judgment of this, especially since the exact details of what makes a graph nicer can be highly subjective and may depend not only on the user working with the graph but also on what he is trying to accomplish at any given time.

Nevertheless we will look at a couple of options that lend themselves to automatically judging the quality of a graph in Section 5. With such measures we can fine tune the selection process in algorithm 8 by sorting the intersections in order of their quality.

If we sufficiently trust this automatic evaluation, it is also possible to filter out intersections that actively make the graph worse by inserting empty dummy intersections into the list of possible graphs, which only contain the original S and T, as well as their dependencies. If the dummy node is ranked above the intersection, the intersection would be filtered out by the select function.

However even without such evaluation methods it has been observed that intersecting over morphisms that aren't themselves meaningless tends to create meaningful intersections.[11]

5 Evaluation of theory graph quality

In algorithm 8 of Section 4 we made use of a sorting function with the underlying idea that this function would somehow sort the resulting subgraphs in order of their desirability. In order to make this work we need some sort of evaluation function $eval : theorygraph \to \mathbb{R}$ which rates theory graphs according to some kinds of quality criteria.

5.1 Knowledge

The aim of theory graphs is the representation of knowledge and implicitly expanding said knowledge via theory morphisms. Therefore the amount of contained knowledge makes for a natural starting point of judging a theory graph's quality. However there are multiple problems with this supposedly simple approach.

The biggest problem is that knowledge is not an easily quantified thing. Especially if we try to give different weights to specific instances of knowledge, such as the value of an axiom compared to the value of a theorem or the transformative knowledge contained in a theory morphism.

Definition 5.1 (Knowledge)**.** The knowledge contained in a theory graph γ is the set of all axioms, theorems, as well as all mappings $x \xrightarrow{\sigma} y$ where x and y are symbols in γ and σ is a view in γ.

Note that this definition does not treat all theory morphisms equally. It may seem counter-intuitive that we count all assignments inside a view separately, but only count structures like we would a single declaration. This is because when counting structures in this work, we are mainly concerned with simple inclusions. While it is possible for named structures to appear, they will never be part of the graph changes, since they are excluded from intersections (see Section 3.4).

One problem with this definition is that it does not take into account that it counts multiple instances of what is arguably the same knowledge. Consider for example the two graphs in Figure 5.1.

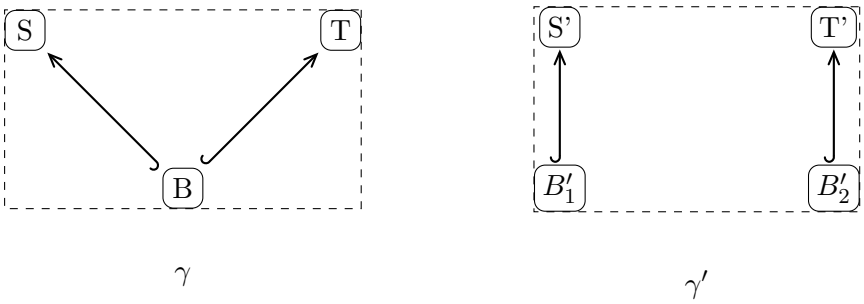


Figure 5.1: The graph on the left is erroneously counted as containing more knowledge

If we assume that B, B_1 and B_2 all are copies of what is de facto the same theory, with the

same being true for the pair (S,S') and (T,T'), then we would count still count γ as containing more knowledge than γ' despite the difference consisting entirely of copied knowledge.

However since drawing the distinction between knowledge that is simply copied is not easy, we will go with this definition regardless.

5.2 Representation

While an accurate measure of knowledge gives us a good idea of what you theoretically could do with a theory graph it is not the only measure that has an impact on the quality of a theory graph.

Intuition tells us that a bigger representation is not an advantage unless it actually results in more knowledge.

Defining what counts as a single instance of representation is not as difficult as defining what counts as knowledge, but it still requires some amount of thought.

One approach would be to go all the way down to the character level and convert the graph into its textual representation in the MMT language and measure the resulting volume of text. This is however a rather blunt approach, which implies that the quality of the graph is noticeably increased by using shorter identifiers, especially single character ones if at all possible. This is of course not what we want.

Counting the number of theories is going too far into the other direction. Since the objective of this process is the creation of new interesting theories, it lies in its very nature that the number of theories will increase.

With characters being too fine granular and theories being too broad, this suggest that the ideal level of abstraction is somewhere between those two levels. We can further make use of the fact that terms remain largely the same apart from some identifier substitutions and we arrive at the conclusion that the best level of abstraction is the symbol level. This means we count the constant declarations and structures.

Definition 5.2 (Representation)**.** The representation of a theory graph γ is the set of all declarations and structures in theories in γ, as well as all mappings $x \xrightarrow{\sigma} y$ where x and y are axioms in a theory in γ and σ is a view in γ.

If we compare the definition of knowledge to the definition of representation, we will notice that large parts of the definitions overlap.

Like the definition of knowledge, this one is not necessarily ironclad. However in the opinion of the author it should be at least slightly less contentious than the definition of knowledge, since any parts that are counted are undoubtably part of the representation.

For a more general application it may be desirable to expand the count of structures to any assignments made within a structure, but since deep theory intersections are not really compatible with named structures and with includes being very simple in their structure, we can simplify their representation to just a single count.

5.3 Possible evaluation functions

With these basic measures of graph content at least somewhat in place, we can look at derived measures to try and judge the quality of a theory graph.

5.3.1 Knowledge as an evaluation function

Definition 5.3 ($eval_k$)**.** The evaluation function $\underline{eval_k} : theorygraph \to \mathbb{R}$ maps a graph to the count of its contained knowledge.

$$eval_k(\gamma) = |\text{knowledge in } \gamma|$$

The fundamental problem with relying on knowledge as our sole measure to evaluate graph quality is that by intent theory intersections are at the very least knowledge preserving and in almost any sensible definition for a measure of knowledge they are knowledge increasing as long as they are not trivial.

If we can somehow make sure that performing no change wins in a tie-breaker, we could at least use this measure to ensure that there is some gain from performing a graph change. However in a non-trivial case it is almost guaranteed that knowledge will increase for most ways we would define knowledge.

While we could use this in order to remove trivial intersections from the graph, we are already doing this automatically by pruning the intersection graph, so this additional computation step is at best redundant.

5.3.2 Representation as an evaluation function

Definition 5.4 ($eval_r$)**.** The evaluation function $\underline{eval_r} : theorygraph \to \mathbb{R}$ maps a graph to the negative count of the elements in its representation.

$$eval_r(\gamma) = -|\text{representation of } \gamma|$$

The count of the representation is negative, because a higher amount of representation is not a desirable quality in itself.

The advantage of purely using representation as our measure is that it is by far the simplest measure to calculate.

However this is unfortunately the measures only real advantage. Like relying purely on knowledge this measure has the big downside that it is almost always irrelevant and subsumed by much simpler approaches which do not make use of a measure at all.

More specifically for most sensible definitions of representation a binary intersection will always be judged as worse than the unintersected theories. This is because a non-trivial binary intersection will create exactly one copy in an intersection theory for every declaration that is removed from the remainder, leaving the total number of declarations unchanged, but adds extra inclusions for the new dependent intersection theories.

However in the case of unary intersections this measure makes some amount of sense. Since the unary intersection takes two declarations in two different theories and replaces them with a single declaration in a common included theory there is an actual potential for saving on representation, while retaining at least morally the same amount of knowledge.

5.3.3 Difference of Knowledge and Representation as an evaluation function

Definition 5.5 ($eval_{k-r}$)**.** The evaluation function $\underline{eval_{k-r}} : theorygraph \to \mathbb{R}$ is the difference of knowledge and representation.

$$eval_{k-r}(\gamma) = eval_k(\gamma) + eval_r(\gamma)$$

Since $eval_r$ was already the negative count of the representation, it is added to calculate the difference.

Since the representation contains all the explicit knowledge with the addition of structures, this measure is roughly equivalent to measuring the implicit knowledge, while subtracting the aforementioned structures. We will refer to it as *knowledge gain*, as it measures the amount of extra that can be inferred compared to writing it all down explicitly.

The advantage of this measure for the purpose of evaluating intersections is that we rate them by the amount of implicit knowledge we gain over having to write that same knowledge down explicitly.

5.3.4 Quotient of Knowledge and Representation as an evaluation function

Definition 5.6 ($eval_{k/r}$)**.** The evaluation function $\underline{eval_{k/r}} : theorygraph \to \mathbb{R}$ is the quotient of knowledge and representation.

$$eval_{k/r}(\gamma) = \frac{eval_k(\gamma)}{-eval_r(\gamma)}$$

This evaluation function is effectively measuring how tightly knowledge is packed within the theory graph. It makes sense to refer to it as the *knowledge density*.

It has generally been observed that measures that simply count in absolute numbers are problematic, because they scale with the absolute size of the measured object. It is logical to assume that this is no different in the case of theory graphs.

The problem with applying this measure to our specific problem is that since the subgraph we evaluate is limited to the directly affected theories and their intersections, we do not actually get an accurate picture of the implicit knowledge these theories contain for the larger theory graph.

Furthermore the most relevant comparisons will happen between subgraphs of roughly similar size. This means that the argument of scale is somewhat diminished when it is applied to this particular problem.

Outside the scope of this work there may also be additional problems when blindly following this measure. Used as defined here it is possible to "improve" any theory graph under this measure by deleting any sections of the graph that have less knowledge density than the most dense cluster in the graph.

6 Implementation

As part of this work an implementation of the deep intersection algorithm is also provided. The full source code of this implementation can be found in the appendix.

It can also be found in the MMT git-repository [12] under the branch Intersections.

6.1 Practical difficulties and Solutions

6.1.1 Integrating the Intersecter with MMT

The Intersecter is integrated within the MMT system as an extension [13]. As such the Intersecter gains access to the theory graph in memory via the controller. It is at this level that the algorithm manipulate the theories.

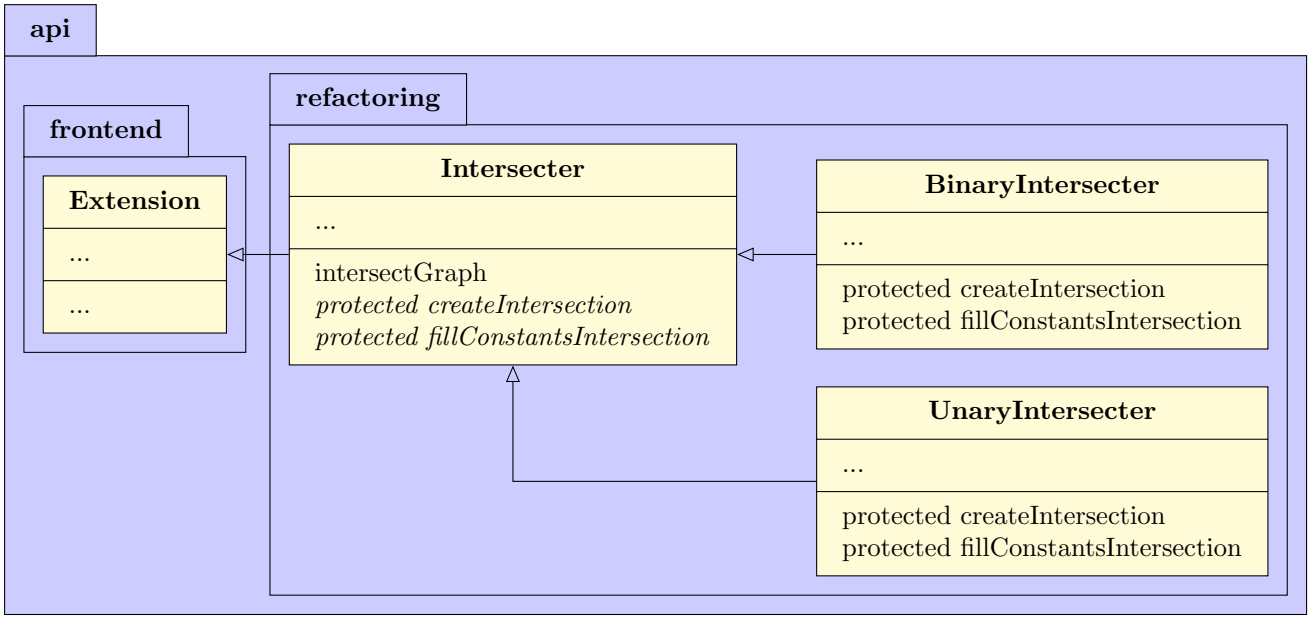


Figure 6.1: Class diagram of Intersecter and its subclasses

Since the theories are created in memory the algorithms themselves output theories in the same form. In order to get the output in MMT language form, the MMTSyntaxPresenter can be used.

The GraphEvaluator which handles the evaluation as discussed in Subsection 4.3.2 and Section 5 is organized in a similar manner, as seen in Figure 6.2.

Again there is a common GraphEvaluator class, with the different evaluation functions implemented by subclasses.

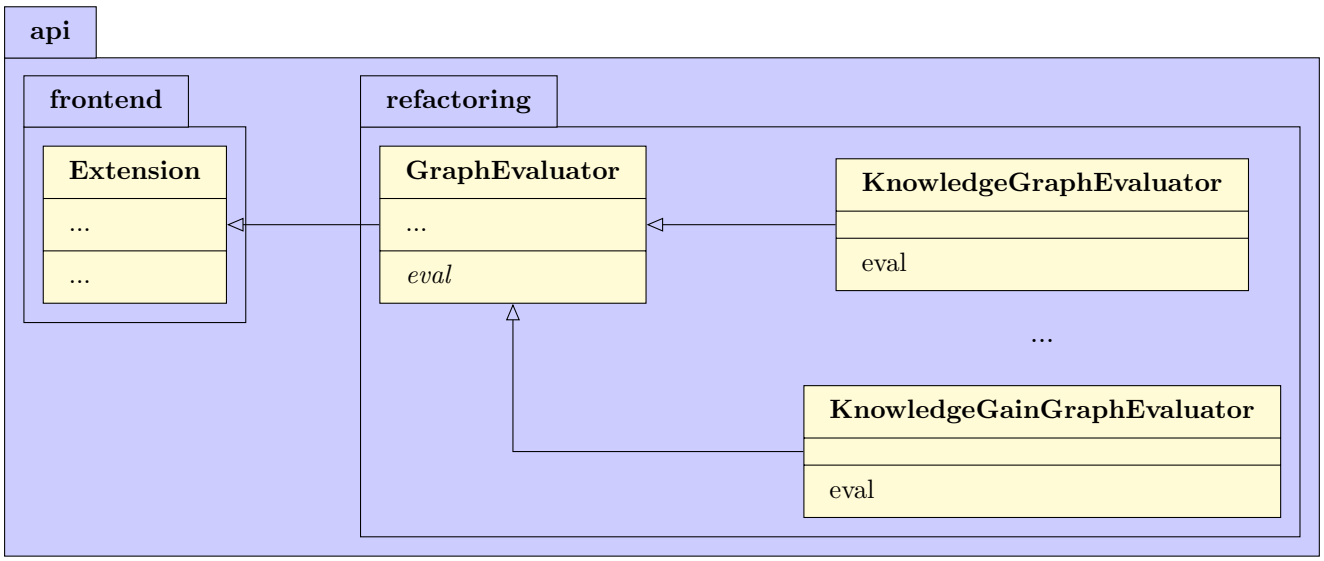


Figure 6.2: Class diagram of GraphEvaluator and its subclasses

class	eval-func	explanation
KnowledgeGraphEvaluator	$eval_k$	evaluates graph by amount of knowledge
RepresentationGraphEvaluator	$eval_r$	evaluates graph by amount of representation
KnowledgeGainGraphEvaluator	$eval_{k-r}$	evaluates graph by the amount of knowledge gained compared to written explicitly
KnowlDivRepGraphEvaluator	$eval_{k/r}$	evaluates graph by knowledge density

While the Graph Evaluator gets its arguments in the form of Scala level Theory objects, it may still occasionally make use of the controller (such as finding the theory which is the domain of a view). While it would be possible do this without using the controller, it makes sense for convenience. As such the GraphEvaluator requires theories within the input graph to be registered with the controller. The Intersecter takes care of this, so it should not be a problem in practice.

Both of these components come together in the FindIntersecter. The FindIntersecter implements algorithm 1 and is itself an Extension derived from the BuildTarget subclass (see Figure 6.3). In order to implement the different combinations of intersection types and evaluation functions, it is parameterized as a template over both an Intersecter and GraphEvaluator.

This component also makes use of the viewfinder [4] to find the views over which it intersects, hence the name FindIntersecter.

As a BuildTarget FindIntersecter implements both the build and clean method, allowing it to be called from the MMT-shell. For more details see Subsection 6.2.2.

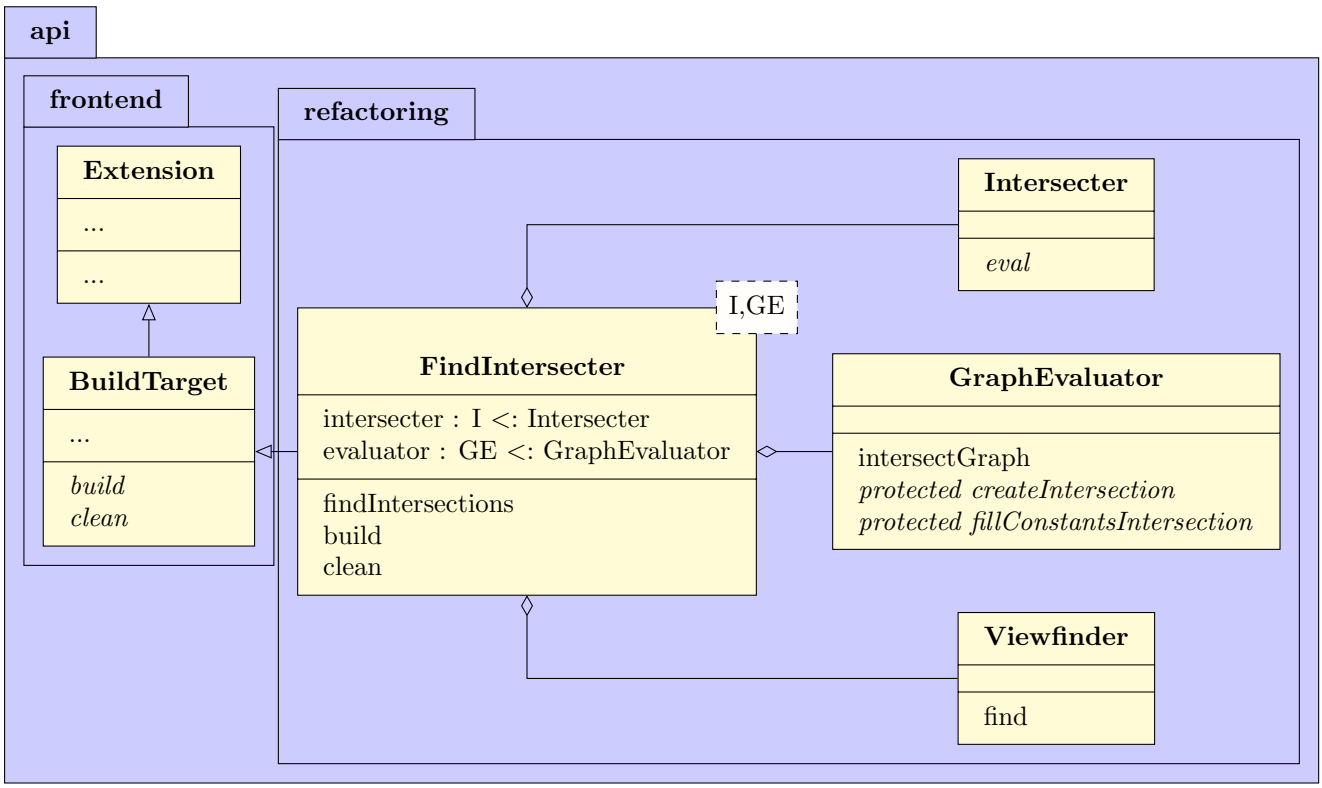


Figure 6.3: FindIntersecter combines an Intersecter with a GraphEvaluator

6.1.2 Avoiding redundancy in the intersection subgraph

While constructing the intersection theories there are two pitfalls we need to avoid. The first is constructing multiple instances of what is actually the same intersection and the second is multiple instances of what is de facto the same intersection.

The first case is the result of an input graph that looks something like the graph shown in Figure 6.4. Since S includes both S_1 and S_2, both of which include S_B, there are now two distinct inclusion paths for S to include S_B. Obviously we only want one instance of $S_B \cap T$.

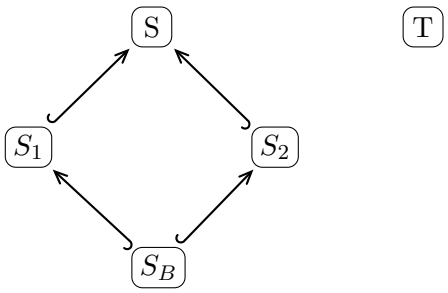


Figure 6.4: S transitively includes S_B via 2 different paths

The second case refers to trivial and redundant intersections, which were already discussed in Section 3.3. It was mentioned that these can be detected during the construction of the intersection theory in question, which is precisely what the implementation does to avoid them.

In order to deal with both of these issues we make use of a map to keep track of the already generated intersections, which remains persistent throughout the entire intersection process.

The map is implemented by a Map[(MPath, MPath), (Theory, Theory)]. This means that it maps a pair of theories identified by their path to a pair of theories which are the intersection of the input theories. In the case of a unary intersection it is a pair of 2 references to the same intersection theory.

This map is filled step by step, during the recursive construction of the intersections, according to the method of dynamic programming. When encountering an intersection that is already in the map it is skipped.

When encountering a theory that is trivial we simply do not add it to the map. We can recognize such a theory simply by checking whether it contains no declarations. This requires the small extra care when adding the inclusions to intersection theories, to check whether the specific inclusion is actually found in the map.

When encountering a redundant theory inclusion, instead of adding it to the map directly, we simply add the intersection theory it includes instead. We can recognize such a redundant theory by checking if it contains exactly one declaration and if it does, check whether this declaration is a theory inclusion. When the inclusions of this intersection is created in a theory higher up in the hierarchy it will simply add the replacement put in the map, thereby requiring no extra effort for resolving the elimination at that point.

Since the recursive steps of the algorithm first build the base theories and work their way up the inclusion hierarchy, this pruning process satisfies the criterion mentioned in Theorem 3.7 for it to result in a fully pruned theory graph.

6.1.3 Redundant inclusions

One unfortunate side effect of pruning the intersection theories is that the removal of redundant inclusions can lead to redundant theory inclusions. A redundant theory inclusion is a theory inclusion which is not needed, because it is already included transitively via a different theory inclusion, as show in Figure 6.5 where the inclusion $I_{S_2,T_2} \hookrightarrow I_{S_1,T_1}$ is redundant.

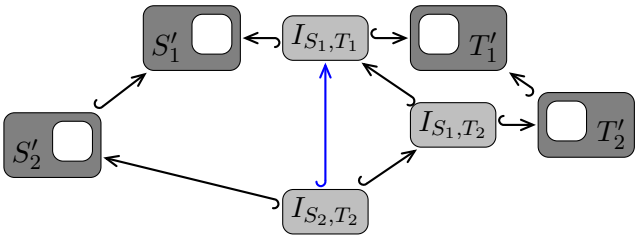


Figure 6.5: Example from Figure 3.9 creates a redundant inclusion

In order to prevent these redundant inclusions the implementation makes use of the GraphOptimizationTool [14] previously implemented by the author, in order to identify and remove them.

6.2 Interface and usage

6.2.1 API

Intersecter

The main interface method that Intersecter provides is intersectGraph, which takes a view over which to intersect and returns a changed subgraph of theories and views between those theories organized in a tuple.

```
/** Intersect two theories over view
  *
  * Creates intersection of domain and codomain of view over said view.
  * Lower level theories will be intersected recursively.
  *
  * @param view View over which theories are intersected
  * @return
  */
def intersectGraph(view: View): (List[Theory], List[(Theory, Theory)], List[Theory],
    List[(View,View)])
```

This method implements algorithm 3 from Section 4.

Notable differences are that this method only takes the view we earlier called σ as an argument. All other arguments are still needed, but they are passed implicitly. The graph is accessed via the controller and the target theories S and T can be extracted from the view itself.

On the other end of the routine, the returned graph is split into 4 parts. These are in the order they appear in the tuple:

res[x]	type	content
0	List[Theory]	list of changed dependencies of S
1	List[(Theory, Theory)]	list of intersection theories of $S \overset{\sigma}{\cap} T$ as isomorphic pairs
2	List[Theory]	list of changed dependencies of T
3	List[(View,View)]	list of pairs of views (σ,σ^{-1}) between isomorphic pairs

In the case of a unary intersection the the pairs in res[1] are two references to the same theory, since both sides include the same intersection theory. For the same reason the list of views res[3] is empty in this case. While it would be possible and valid to return identity morphisms, this is not particularly useful in practice.

GraphEvaluator

The main interface method that GraphEvaluator provides is eval, which takes a theory graph given as a list of theories and views between them and returns a numeric value.

```
/** Evalutes a graph given as a list of theories and views according to an evaluation
    function
    *
    * @param theories theories of the graph
    * @param views views of the graph
    * @return evaluation as integer value
    */
  def eval(theories : List[Theory], views : List[View]=List.empty[View]) : Int
```

The intended reading of the returned integer value is that high values are better than low values. It should be noted that these evaluations are only intended as a heuristic specifically for judging theory intersections compared with their unintersected counterparts. Applying tese GraphEvaluators to very dissimilar graphs is unlikely to reflect any meaningful judgement. For more details see Section 5.

FindIntersecter

Th main method FindIntersecter provides from the API perspective is the findIntersections method. This method implements algorithm 1. It takes an archive as its argument, which contains the parts of the theory graph which is searched for intersectable theories via the ViewFinder. It then creates those intersections using the Intersecter before evaluating them.

Its output is in the form of a list of pairs of theories and views, which are our intersections, together with a numeric value corresponding to their evaluation.

```
/** Searches for views over which to intersect and then does so
    *
    * views are found via the viewfinder
    * intersections done via intersecter of parameterized type I
    * intersection graphs are evaluated using evaluation function of parameterized
        type GE
    *
    * @param a Archive in which intersections are searched for
    * @return intersected graph as pair of theories and views and a numeric value of
        their evaluation
    */
  def findIntersections(a : Archive) : List[((List[Theory],List[View]), Int)]
```

This method is also what is called when using the FindIntersecter as a BuilTarget on the Archive a.

6.2.2 Interface via MMT-shell

As a BuildTarget the FindIntersecter can be used to build a file containing the intersections and intersected theories from the MMT-shell. However since it is parameterized over the used Intersecter and GraphEvaluator, we need a declared subclass of FindIntersecter . One such subclass is the FindBinaryIntersecter, which uses the BinaryIntersecter and the KnowledgeGainGraphEvaluator.

It can be loaded and used from the MMT-shell as follows:

```
mmt> extension info.kwarc.mmt.api.refactoring.FindBinaryIntersecter
mmt> build <ARCHIVENAME> intersections
```

Just replace <archivename> with the name of the archive that is to be built.

Since the FindIntersecter makes use of the viewfinder, it is important that a fitting preprocessor is loaded for the archive. If no preprocessor is loaded we will not find any views and thus the Intersecter will have nothing to intersect over.

The build target delivers its output into a file written in the MMT language which can be found inside the archive folder under the path "<ARCHIVEPATH>/export/intersections/<ARCHIVENAME>.mmt".

7 Example of Use

As an example we will take a look at the following collection of example theories located in a archive called testarchive, which consists of intersectable theories A3 and B3, as well as their included base theories.

```
namespace http://mydomain.org/testarchive/mmt-example

import mitm http://mathhub.info/MitM/Foundation

theory A1 : mitm:?Logic =
    A : type

theory A2 : mitm:?Logic =
    include ?A1
    C : type
    foobar : A →C

theory A3 : mitm:?Logic =
    include ?A2
    a : A
    foo : A →A
    Ax : ⊢a ≐a

theory B1 : mitm:?Logic =
    B : type

theory B2 : mitm:?Logic =
    include ?B1
    D : type
    b : B

theory B3 : mitm:?Logic =
    include ?B2
    d : D
    bar : B →D
    defined : D |
        = d
    barfoo : D →B
    Ax : ⊢d ≐d
```

With the proper preprocessors loaded we can build the testarchive from the MMT-shell.

```
mmt> build testarchive intersections
```

The viewfinder finds the following partial views over which the Intersecter could intersect.

```
view View1 : http://mydomain.org/testarchive/mmt-example?B3 ->
    http://mydomain.org/testarchive/mmt-example?A3 =
D
     = A ❙
d
     = a ❙
Ax
     = Ax ❙
❚

view View0 : http://mydomain.org/testarchive/mmt-example?A3 ->
    http://mydomain.org/testarchive/mmt-example?B3 =
A
     = D ❙
a
     = d ❙
Ax
     = Ax ❙
❚
```

It's not surprising that we find two views of which one is the inverse of the other, since we explicitly require invertible views for intersecting. However as we can see in the output below only one of these views is actually used, since we filter intersections to intersect over unique theories.

Also neither of these views is strictly speaking a proper view, but a partial view, which is exactly what we need for the intersection.

Since both intersection candidates are entirely mirrored and thus evaluate the same, the selection in this example ultimately comes down to the original order of the views as they came out of the viewfinder.

```

theory B2IntersectsA1 : mitm:/Foundation?Logic =
  D
       : type
❙
❚
theory A1IntersectsB2 : mitm:/Foundation?Logic =
  A
       : type
❙
❚
theory B3IntersectsA3 : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?B2IntersectsA1 ❙
  d
       : D
❙
  Ax
       : ⊢d≐d
❙
  defined
       : D
    | = d
❙
❚
```

```
theory A3IntersectsB3 : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 ❙
  a
       : A
❙
  Ax
       : ⊢a≐a
❙
❙
theory B1R : mitm:/Foundation?Logic =
  B
       : type
❙
❙
theory B2R : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?B1R ❙
  include http://mydomain.org/testarchive/mmt-example?B2IntersectsA1 ❙
  b
       : B
❙
❙
theory B3R : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?B2R ❙
  include http://mydomain.org/testarchive/mmt-example?B3IntersectsA3 ❙
  bar
       : B→D
❙
  barfoo
       : D→B
❙
❙
theory A1R : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 ❙
❙
theory A2R : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?A1R ❙
  include http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 ❙
  C
       : type
❙
  foobar
       : A→C
❙
❙
theory A3R : mitm:/Foundation?Logic =
  include http://mydomain.org/testarchive/mmt-example?A1R ❙
  include http://mydomain.org/testarchive/mmt-example?A3IntersectsB3 ❙
  foo
       : A→A
❙
❙view B2IntersectsA1toA1IntersectsB2 :
    http://mydomain.org/testarchive/mmt-example?B2IntersectsA1 ->
    http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 =
  D
       = A
  ❙
❙view A1IntersectsB2toB2IntersectsA1 :
    http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 ->
```

```
    http://mydomain.org/testarchive/mmt-example?B2IntersectsA1 =
  A
       = D
  ❙
❙view B3IntersectsA3toA3IntersectsB3 :
    http://mydomain.org/testarchive/mmt-example?B3IntersectsA3 ->
    http://mydomain.org/testarchive/mmt-example?A3IntersectsB3 =
  include http://mydomain.org/testarchive/mmt-example?B2IntersectsA1 |= COMPOSE
      (IDENTITY ?B2IntersectsA1);?B2IntersectsA1toA1IntersectsB2❙
  d
       = a
  ❙
  Ax
       = Ax
  ❙
❙view A3IntersectsB3toB3IntersectsA3 :
    http://mydomain.org/testarchive/mmt-example?A3IntersectsB3 ->
    http://mydomain.org/testarchive/mmt-example?B3IntersectsA3 =
  include http://mydomain.org/testarchive/mmt-example?A1IntersectsB2 |= COMPOSE
      (IDENTITY ?A1IntersectsB2);?A1IntersectsB2toB2IntersectsA1❙
  a
       = d
  ❙
  Ax
       = Ax
  ❙
❙
```

We can see in the output, that of the 9 pairs of intersection theories which should theoretically be part of the binary intersection $B3 \overset{View1}{\cap} A3$ only 3 pairs are actually part of the proposed subgraph. This is because most of these intersection theories were pruned during the creation of the intersection as they were either trivial or redundant.

8 Conclusion

We began this work by looking at theory intersections and the current state of implementing them. We observed that existing solutions do not preserve the original inclusion structure of the original theory graph.

We then extended the definition of theory intersection with the notion of deep theory intersections, which recursively intersect the dependencies of the intersected theories in order to preserve and extend the original structure. We also saw algorithms that are able to deliver such intersections.

To address the problem of intersections that are worse than their unintersected counterparts, we looked at both methods that clean up the resulting intersections by removing redundancies and empty theories, as well as methods to evaluate and filter the results.

To put this theory into practice we have also looked at an implementation that is integrated with the MMT system, which helps to identify intersection candidates and produces the proposed intersection theories.

Future Work

There are still multiple areas of the deep theory intersections which can be improved upon, both in theory and implementation.

A major gain would be to find a way to integrate the use of named structures in the intersection theories. However as discussed this is not merely an implementation challenge, but also challenges the assumptions of how a deep theory intersection is defined on a concept level.

Another aspect that needs to be examined more closely are the evaluation functions. Both for global application and specifically in this area. The evaluations provided in this work are mainly the result of what feels like it would make for a better graph with little practical verification. This means it is worth examining both new evaluation methods as looking at the ones presented here more closely.

On the implementation side of things there many things that could be done to improve the way the Intersecter can be applied to practical use. While the FindIntersecter BuildTarget already allows for the use of the tool from the console, it gives the user very little control over the process, other than selecting an archive to work with. It also means that the output is sitting in a file separate from the rest of the archive, requiring manual work if it is to be properly integrated. One possible way to address this would be a plugin for a tool that is already used for working with MMT documents, such as jEdit[15] or IntelliJ[16].

Appendices

A Source Code

```
1  package info.kwarc.mmt.api.refactoring
2
3  import info.kwarc.mmt.api.archives.{Archive, BuildTarget, Update}
4  import info.kwarc.mmt.api.frontend.{Controller, Extension}
5  import info.kwarc.mmt.api.modules.{ModuleOrLink, Theory, View}
6  import info.kwarc.mmt.api.notations.NotationContainer
7  import info.kwarc.mmt.api.objects.{Context, OMID, Term, Traverser}
8  import info.kwarc.mmt.api.presentation
9  import info.kwarc.mmt.api.presentation.{FileWriter, MMTSyntaxPresenter}
10 import info.kwarc.mmt.api.symbols._
11 import info.kwarc.mmt.api.utils.FilePath
12 import info.kwarc.mmt.api._
13
14 import scala.collection.mutable
15 import scala.util.{Success, Try}
16
17 abstract class Intersecter extends Extension {
18   val DEBUG = true
19   var optimizer = new GraphOptimizationTool()
20
21
22   override def start(args : List[String]): Unit = {
23     super.start(args)
24     controller.extman.addExtension(optimizer)
25   }
26
27   private def debugOut(arg: Any): Unit = if (DEBUG) println(arg)
28
29   /**
30     *
31     * @param view view over which the Intersecter intersects
32     * @return
33     */
34   def apply(view: View): (List[Theory], List[(Theory, Theory)], List[Theory],
       List[(View, View)]) = {
35     intersectGraph(view)
36   }
37
38   /** Intersect two theories over view
39     *
40     * Creates intersection of domain and codomain of view over said view.
41     * Lower level theories will be intersected recursively.
42     *
43     * @param view View over which theories are intersected
44     * @return
45     */
46   def intersectGraph(view: View): (List[Theory], List[(Theory, Theory)],
       List[Theory], List[(View,View)]) = {
```

```
47      implicit val intersections: mutable.HashMap[(MPath, MPath), (Theory, Theory)] =
            collection.mutable.HashMap[(MPath, MPath), (Theory, Theory)]()
48      val th1 = controller.getTheory(view.from.toMPath)
49      val th2 = controller.getTheory(view.to.toMPath)
50      implicit val renamings : mutable.HashMap[GlobalName, GlobalName] =
            mutable.HashMap[GlobalName, GlobalName]()
51      implicit val viewMap : mutable.HashMap[MPath, View] = mutable.HashMap[MPath,
            View]()
52      val views = recIntersect(th1, th2, view)
53      val rem1 = remainder1(th1, th2, intersections, renamings)
54      val rem2 = remainder2(th1, th2, intersections, renamings)
55      (rem1, intersections.values.toSet.toList, rem2, views)
56    }
57
58    /** Recursive method to generate intersections
59      *
60      * Recursively generates intersections for dependencies of th1 and th2
61      *
62      * @param th1 left side theory to intersect
63      * @param th2 right side theory to intersect
64      * @param view view to intersect over
65      * @param intersections map containing already computed theory intersections
66      * @param renamings map of renamings of GlobalNames
67      * @return
68      */
69    protected def recIntersect(th1: Theory, th2: Theory, view : View)
70                              (implicit intersections: mutable.HashMap[(MPath, MPath),
                                      (Theory, Theory)], renamings:
                                      mutable.HashMap[GlobalName, GlobalName], viewMap :
                                      mutable.HashMap[MPath, View])
71    : List[(View,View)] = {
72      var views = List[(View,View)]()
73      if (intersections.contains((th1.path, th2.path))) { //TODO optimize multiple
            calls for empty theory
74        return views
75      }
76      val includes1 = collect_structures(th1)
77      val includes2 = collect_structures(th2)
78
79      //Recurse through dependent Theories of th1 and th2
80      //th1:
81      includes1.foreach {
82        case PlainInclude(from, to) =>
83          views ++= recIntersect(controller.getTheory(from), th2, view)
84        case default =>
85      }
86      //th2:
87      includes2.foreach {
88        case PlainInclude(from, to) =>
89          views ++= recIntersect(th1, controller.getTheory(from), view)
90        case default =>
91      }
92      views ++ createIntersection(th1, th2, view, includes1, includes2)
93    }
94
95    protected def createIntersection(th1: Theory, th2: Theory, partialView : View,
        includes1 : List[Declaration], includes2 : List[Declaration])
```

```
96                                  (implicit intersections: mutable.HashMap[(MPath,
                                        MPath), (Theory, Theory)], renamings:
                                        mutable.HashMap[GlobalName, GlobalName], viewMap :
                                        mutable.HashMap[MPath, View])
97    : Option[(View,View)]
98
99    protected def addNonTrivialIntersection(th1 : Theory, th2 : Theory, int1 : Theory,
          int2 : Theory, view : Option[(View,View)])
100                                     (implicit intersections :
                                            mutable.HashMap[(MPath, MPath), (Theory,
                                            Theory)], viewMap : mutable.HashMap[MPath,
                                            View])
101   : Option[(View,View)] = {
102
103     var retview = view
104     //add to controller
105     controller.add(int1)
106     controller.add(int2)
107
108     //Check for redundancy
109     val redundant1 = optimizer.findRedundantIncludes(int1.path).toSet
110     val redundant2 = optimizer.findRedundantIncludes(int2.path).toSet
111
112     for (decl <- int1.getDeclarations) {
113       decl match {
114         case PlainInclude(from, to) if redundant1.contains(from) =>
              int1.delete(PlainInclude(from, to).name)
115         case default =>
116       }
117     }
118     for (decl <- int2.getDeclarations) {
119       decl match {
120         case PlainInclude(from, to) if redundant2.contains(from) =>
              int2.delete(PlainInclude(from, to).name)
121         case default =>
122       }
123     }
124     var res1 = int1
125     var res2 = int2
126     //check that intersection is not trivial
127     if (int1.getDeclarations.nonEmpty) {
128       if (int1.getDeclarations.length==1) int1.getDeclarations.head match {
129         case PlainInclude(from, to) =>
130           res1 = controller.getTheory(from)
131           controller.delete(int1.path)
132           retview = None
133         case default =>
134       }
135       if (int2.getDeclarations.length==1) {
136         int2.getDeclarations.head match {
137           case PlainInclude(from, to) =>
138             res2 = controller.getTheory(from)
139             controller.delete(int2.path)
140             retview = None
141           case default =>
142         }
143       }
144       intersections.put((th1.path, th2.path), (res1, res2))
```

```
145       retview match {
146         case Some((v1,v2)) =>
147           controller.add(v1)
148           controller.add(v2)
149           viewMap.put(int1.path, v1)
150           viewMap.put(int2.path, v2)
151         case None =>
152       }
153       retview
154     } else {
155       controller.delete(int1.path)
156       controller.delete(int2.path)
157       None
158     }
159   }
160
161   protected def intersectionDeclarations(th1 : Theory, th2 : Theory, view_map:
          collection.immutable.Map[FinalConstant, FinalConstant]): List[Declaration] = {
162     //debugOut(th1 + "\n" + th2)
163     val res = getSubDeclarations(th1).filter {
164       _ match {
165         case c: FinalConstant =>
166           view_map.contains(c) && (view_map(c).parent == th2.path ||
                th2.getDeclarations.collect{case subModule: ModuleOrLink =>
                subModule}.map(m => m.modulePath).contains(view_map(c).parent))
167         case PlainInclude(from, to) => false
168       }
169     }
170     res
171   }
172
173   protected def intersectionDefinedDeclarations(th : Theory, int : Theory, renamings
          : mutable.HashMap[GlobalName, GlobalName]) : List[Declaration] = {
174     getSubDeclarations(th).filter {
175       _ match {
176         case c: FinalConstant => isDefinedIn(c,
              isDefinedInTraverserState(int,renamings, mutable.HashSet[MPath]()))
177         case PlainInclude(from, to) => false
178       }
179     }
180   }
181
182   /** returns Declarations of theory including submodules
183     *
184     * returns List of all direct Declarations in a given Theory, including those that
          are declared in direct submodules
185     *
186     * @param th searched theory
187     * @return list of all Declarations
188     */
189   protected def getSubDeclarations(th : Theory): List[Declaration] = {
190     th.getDeclarations.flatMap {
191       _ match {
192         case c: FinalConstant => Some(c)
193         case PlainInclude(from, to) => Some(PlainInclude(from, to))
194         case s: Structure => s.getDeclarations
195         //all cases?
196         case default => None
```

```
197       }
198     }
199   }
200
201   protected def getRecIncludes(theory: Theory) : List[Theory] = {
202     val includes = theory.getIncludesWithoutMeta.map(controller.getTheory(_))
203     includes ++ includes.flatMap(getRecIncludes)
204   }
205
206   protected def getFlatDeclarations(theory : Theory) : List[Declaration] = {
207     getRecIncludes(theory).flatMap(_.getDeclarations)++theory.getDeclarations
208   }
209
210   /** Fill remainder of intersection with constants
211     *
212     * @param rem remainder theory to be filled
213     * @param th1 original theory from which to fill remainder
214     * @param th2 other theory
215     * @param intersections map of theory intersections
216     * @param renamings map of renamings of GlobalNames
217     * @return
218     */
219   protected def fillConstantsRemainder1(rem : Theory, th1 : Theory, th2 : Theory,
        intersections : mutable.HashMap[(MPath, MPath), (Theory, Theory)], renamings :
        mutable.HashMap[GlobalName, GlobalName]) : Theory = {
220     if (intersections.contains((th1.path, th2.path)))
          rem.add(PlainInclude(intersections.get(th1.path, th2.path).get._1.path,
          rem.path))
221     val intersected = intersections.get(th1.path, th2.path) match {
222       case Some((t1, t2)) => getFlatDeclarations(t1).map(d => d.name).toSet
223       case None => mutable.HashSet[LocalName]()
224     }
225     getSubDeclarations(th1).filter(_ match {
226       case c : FinalConstant => !(renamings.contains(c.path) &&
            intersected.contains(renamings(c.path).name))
227       case PlainInclude(from, to) => false //includes are handled separately
228       case s : Structure => true //structures are incompatible with deep intersections
229       case default => ???
230     }).foreach(addDeclaration(_, rem, renamings))
231
232     rem
233   }
234
235   /** Fill remainder of intersection with constants
236     *
237     * @param rem remainder theory to be filled
238     * @param th1 other theory
239     * @param th2 original theory from which to fill remainder
240     * @param intersections map of theory intersections
241     * @param renamings map of renamings of GlobalNames
242     * @return
243     */
244   protected def fillConstantsRemainder2(rem : Theory, th1 : Theory, th2 : Theory,
        intersections : mutable.HashMap[(MPath, MPath), (Theory, Theory)], renamings :
        mutable.HashMap[GlobalName, GlobalName]) : Theory = {
245     if (intersections.contains((th1.path, th2.path)))
          rem.add(PlainInclude(intersections.get(th1.path, th2.path).get._2.path,
          rem.path))
```

```
246     val intersected = intersections.get(th1.path, th2.path) match {
247       case Some((t1, t2)) => getFlatDeclarations(t2).map(d => d.name).toSet
248       case None => mutable.HashSet[LocalName]()
249     }
250     getSubDeclarations(th2).filter(_ match {
251       case c : FinalConstant => !(renamings.contains(c.path) &&
              intersected.contains(renamings(c.path).name))
252       case PlainInclude(from, to) => false //includes are handled separately
253       case s : Structure => true //structures are incompatible with deep intersections
254       case default => ???
255     }).foreach(addDeclaration(_, rem, renamings))
256
257     rem
258   }
259
260   /** Recursively create remainders of the right side theories of the intersection
261     *
262     * @param th1 left side theory
263     * @param th2 right side theory
264     * @param intersections map containing already computed theory intersections
265     * @param renamings map of renamings of GlobalNames
266     */
267   protected def remainder1(th1: Theory, th2: Theory, intersections:
        mutable.HashMap[(MPath, MPath), (Theory, Theory)], renamings:
        mutable.HashMap[GlobalName, GlobalName]): List[Theory] = {
268     val includes1 = collect_structures(th1)
269     val rem = Theory.empty(th1.parent, LocalName(th1.name.toString+"R"), th1.meta) //
          T'
270
271     //Recurse through dependent Theories of th1
272     val rem_list = includes1.flatMap(_ match {
273       case Include(IncludeData(home, from, args, df, total)) =>
274         val rem_pre = remainder1(controller.getTheory(from), th2, intersections,
              renamings)
275         addDeclaration(Include(rem.toTerm, rem_pre.last.path, args), rem, renamings)
276         rem_pre
277       case SimpleDeclaredStructure(home, name, from, isImplicit, istotal) =>
278         addDeclaration(SimpleDeclaredStructure(rem.toTerm, name, from, isImplicit,
              istotal), rem, renamings)
279         List()
280       case default => ???
281     })
282     //Fill remainder with remaining constants
283     fillConstantsRemainder1(rem, th1, th2, intersections, renamings)
284     controller.add(rem)
285     rem_list :+ rem
286   }
287
288   /** Recursively create remainders of the right side theories of the intersection
289     *
290     * @param th1 left side theory
291     * @param th2 right side theroy
292     * @param intersections map containing already computed theory intersections
293     * @param renamings map of renamings of GlobalNames
294     */
295   protected def remainder2(th1: Theory, th2: Theory, intersections:
        mutable.HashMap[(MPath, MPath), (Theory, Theory)], renamings :
        mutable.HashMap[GlobalName, GlobalName]): List[Theory] = {
```

```
296     val includes2 = collect_structures(th2)
297     val rem = Theory.empty(th2.parent, LocalName(th2.name.toString+"R"), th2.meta) //
            T'
298
299     //Recurse through dependent Theories of th2
300     val rem_list = includes2.flatMap(_ match {
301       case Include(IncludeData(home, from, args, df, total)) =>
302         val rem_pre = remainder2(th1, controller.getTheory(from), intersections,
                renamings)
303         addDeclaration(Include(rem.toTerm, rem_pre.head.path, args), rem, renamings)
304         rem_pre
305       case SimpleDeclaredStructure(home, name, from, isImplicit, istotal) =>
306         addDeclaration(SimpleDeclaredStructure(rem.toTerm, name, from, isImplicit,
                istotal), rem, renamings)
307         List()
308       case default => ???
309     })
310     //Fill remainder with remaining constants
311     fillConstantsRemainder2(rem, th1, th2, intersections, renamings)
312     controller.add(rem)
313     rem_list :+ rem
314   }
315
316   /** Move declaration to another theory
317     *
318     * Moves a declaration to a different theory, substituting all global names with
            the renaming
319     *
320     * @param dec declaration to be moved
321     * @param th new home theory
322     * @param renamings map of renamings of GlobalNames
323     * @return
324     */
325   protected def addDeclaration(dec : Declaration, th : Theory, renamings :
          mutable.HashMap[GlobalName, GlobalName]): dec.ThisType = {
326     val renamer = Renamer(name => renamings.get(name))
327     val translator = TraversingTranslator(renamer)
328
329     val renamed = dec.translate(th.toTerm, LocalName.empty, translator, Context())
330     try th.add(renamed) catch {
331       case err : AddError =>
332       case default => throw default
333     }
334     renamings.put(dec.path, renamed.path)
335     renamed
336   }
337
338   /** As above, but for collections
339     *
340     * @param decs collection of declarations
341     * @param th target theory
342     * @param renamings map of renamings of GlobalNames
343     * @return
344     */
345   protected def addDeclaration(decs : Iterable[Declaration], th : Theory, renamings :
          mutable.HashMap[GlobalName, GlobalName]) : List[Declaration] = {
346     decs.map(addDeclaration(_, th, renamings)).toList
347   }
```

```
348
349   /** collect all Includes in theory
350     *
351     * @param theory theory in which structures are to be collected
352     * @return
353     */
354   protected def collect_structures(theory : Theory): List[Declaration] =
          theory.getDeclarations.filter(_ match {
355     case PlainInclude(from, to) => true
356     case default => false
357   })
358
359   case class isDefinedInTraverserState(theory: Theory, renamings :
          mutable.HashMap[GlobalName, GlobalName], used : mutable.HashSet[MPath])
360
361   /** Traverser finding used theories
362     *
363     * This object is a traverser and searches a theory for all theories that are used
364     */
365   object isDefinedIn extends Traverser[isDefinedInTraverserState] {
366     /** Traverses terms
367       *
368       * Traverses over terms, finding any used theories
369       * @param t This is the current subterm
370       * @param con This is the current context
371       * @param state This is the traverser's state
372       * @return
373       */
374     def traverse(t: Term)(implicit con: Context, state: State): Term = t match {
375       // look for URIs
376       case OMID(path) =>
377         // cut path to module path
378         state match {
379           case isDefinedInTraverserState(_,renamings,used) =>
380             path match {
381               case gname: GlobalName => used.add(renamings.getOrElse(gname,
                      path).module)
382               case default => ??? //TODO?
383             }
384           case default =>
385         }
386         OMID(path)
387
388       // in all other cases, traverse
389       case t =>
390         Traverser(this, t)
391     }
392
393     /** Applies to Declaration
394       *
395       * Searches a Declaration for its used theories, adds them to state
396       * @param decl This is the Declaration to be searched
397       * @param state This is the traverser's state
398       */
399     def apply(decl: Declaration, state: State): Boolean = {
400       decl match {
401         case c: Constant =>
402           c.df match {
```

```
403             case Some(t) =>
404               traverse(t) (Context(), state)
405               state match {
406                 case isDefinedInTraverserState(theory,_,used) =>
                        used.subsetOf(getRecIncludes(theory).map(th =>
                        th.path).toSet+theory.path)
407                 case default => false
408               }
409             case _ => false
410           }
411         case _ => false
412       }
413     }
414   }
415 }
416
417 class BinaryIntersecter extends Intersecter {
418   /**
419     * Creates new (potentially partial) view by restricting it to the domain (and
          codomain) of the given theories
420     *
421     * renamings are used to translate the partial view into the correct domain.
422     *
423     * @param vm map of partial view that is to be restricted
424     * @param th1 domain theory
425     * @param th2 codomain theory
426     * @param renamings map of renamings of GlobalNames
427     * @return restricted view
428     */
429   def restrictVM(vm: scala.collection.immutable.Map[FinalConstant,FinalConstant],
          parent : DPath, th1: Theory, th2: Theory)
430               (implicit renamings: mutable.HashMap[GlobalName, GlobalName], viewMap
                      : mutable.HashMap[MPath, View])
431   : View = {
432     val res = new View(parent, LocalName(th1.name.toString+"to"+th2.name.toString),
          TermContainer(th1.toTerm), TermContainer(th2.toTerm), new TermContainer,
          false)
433     th1.getDeclarations.flatMap(_ match {case PlainInclude(from, to) =>
          Some(PlainInclude(from, to)) case default => None}).foreach(inc => {
434       res.add(Include(inc.home, inc.from.toMPath, List(),
            Some(viewMap(inc.from.toMPath).toTerm)))
435     })
436     for (c1 <- vm.keys) res.add(Constant(res.toTerm, ComplexStep(c1.parent)/c1.name,
          List(), TermContainer.empty, TermContainer(vm(c1).toTerm), None,
          NotationContainer.empty))
437     res
438   }
439
440   override def createIntersection(th1: Theory, th2: Theory, partialView : View,
          includes1 : List[Declaration], includes2 : List[Declaration])
441                                  (implicit intersections: mutable.HashMap[(MPath,
                                        MPath), (Theory, Theory)], renamings:
                                        mutable.HashMap[GlobalName, GlobalName], viewMap :
                                        mutable.HashMap[MPath, View])
442   : Option[(View,View)] = {
443     //Generate intersection of th1 and th2 over view and add them to intersections map
444     var int1 = Theory.empty(th1.parent,
          LocalName(th1.name.toString+"Intersects"+th2.name.toString), th1.meta)
```

```
445     var int2 = Theory.empty(th2.parent,
            LocalName(th2.name.toString+"Intersects"+th1.name.toString), th2.meta)
446
447     //Generate inclusions of dependent intersections
448     includes1.foreach {
449       case PlainInclude(from, to) =>
450         intersections.get((from, th2.path)).foreach { case (pre_int1, pre_int2) =>
451           addDeclaration(PlainInclude(pre_int1.path, int1.path), int1, renamings)
452           addDeclaration(PlainInclude(pre_int2.path, int2.path), int2, renamings)
453         }
454       case s: Structure => ??? //Ignored due inconsistent semantics
455       case default => ???
456     }
457     includes2.foreach {
458       case PlainInclude(from, to) =>
459         intersections.get((th1.path, from)).foreach { case (pre_int1, pre_int2) =>
460           addDeclaration(PlainInclude(pre_int1.path, int1.path), int1, renamings)
461           addDeclaration(PlainInclude(pre_int2.path, int2.path), int2, renamings)
462         }
463       case s: Structure => ??? //Ignored due inconsistent semantics
464       case default => ???
465     }
466     val view_map = ViewSplitter.getPairs(partialView, th1, th2)(controller).toMap
467     val view_map_inverse = ViewSplitter.getPairs(partialView, th1,
            th2)(controller).map(_.swap).toMap
468     fillConstantsIntersection(th1, th2, int1, int2, view_map, view_map_inverse,
            renamings)
469     val view = restrictVM(view_map, partialView.parent, int1, int2)
470     val view_inverse = restrictVM(view_map_inverse, partialView.parent, int2, int1)
471     //add intersections to map
472     addNonTrivialIntersection(th1, th2, int1, int2, Some((view,view_inverse)))
473   }
474
475   /** Creates intersection
476     *
477     * creates intersection of th1 and th2 over view
478     *
479     * @param th1 left side theory to intersect
480     * @param th2 right side theory to intersect
481     * @param view_map mapping of view to intersect over
482     * @param view_map_inverse reverse mapping of view to intersect over
483     * @param renamings map of renamings of GlobalNames
484     * @return
485     */
486   protected def fillConstantsIntersection(th1: Theory, th2: Theory, int1 : Theory,
          int2 : Theory, view_map: collection.immutable.Map[FinalConstant,
          FinalConstant], view_map_inverse: collection.immutable.Map[FinalConstant,
          FinalConstant], renamings: mutable.HashMap[GlobalName, GlobalName]): (Theory,
          Theory) = {
487     //Add constants from th1
488     intersectionDeclarations(th1, th2, view_map).map(addDeclaration(_, int1,
            renamings))
489     intersectionDefinedDeclarations(th1, int1, renamings).map(addDeclaration(_, int1,
            renamings))
490
491     //Add constants from th2
492     intersectionDeclarations(th2, th1, view_map_inverse).map(addDeclaration(_, int2,
            renamings))
```

```
493     intersectionDefinedDeclarations(th2, int2, renamings).map(addDeclaration(_, int2,
            renamings))
494
495     (int1, int2)
496   }
497 }
498
499 class UnaryIntersecter extends Intersecter {
500
501   override def createIntersection(th1: Theory, th2: Theory, partialView : View,
          includes1 : List[Declaration], includes2 : List[Declaration])
502                                  (implicit intersections: mutable.HashMap[(MPath,
                                       MPath), (Theory, Theory)], renamings:
                                       mutable.HashMap[GlobalName, GlobalName], viewMap :
                                       mutable.HashMap[MPath, View])
503   : Option[(View,View)] = {
504     //Generate intersection of th1 and th2 over view and add them to intersections map
505     var int = Theory.empty(th1.parent,
            LocalName(th1.name.toString+"Intersects"+th2.name.toString), th1.meta)
506
507     //Generate inclusions of dependent intersections
508     includes1.foreach {
509       case PlainInclude(from, to) =>
510         intersections.get((from, th2.path)).foreach { case (pre_int1, _) =>
511           addDeclaration(PlainInclude(pre_int1.path, int.path), int, renamings)
512         }
513       case s: Structure => ??? //Ignored due inconsistent semantics
514       case default => ???
515     }
516     includes2.foreach {
517       case PlainInclude(from, to) =>
518         intersections.get((th1.path, from)).foreach { case (pre_int1, _) =>
519           addDeclaration(PlainInclude(pre_int1.path, int.path), int, renamings)
520         }
521       case s: Structure => ??? //Ignored due inconsistent semantics
522       case default => ???
523     }
524     val view_map = ViewSplitter.getPairs(partialView, th1, th2)(controller).toMap
525     val view_map_inverse = ViewSplitter.getPairs(partialView, th1,
            th2)(controller).map(_.swap).toMap
526     fillConstantsIntersection(th1, th2, int, view_map, view_map_inverse, renamings)
527     //add intersections to map
528     addNonTrivialIntersection(th1, th2, int, int, None)
529   }
530
531   /** Creates intersection
532     *
533     * creates intersection of th1 and th2 over view
534     *
535     * @param th1 left side theory to intersect
536     * @param th2 right side theory to intersect
537     * @param view_map mapping of view to intersect over
538     * @param view_map_inverse reverse mapping of view to intersect over
539     * @param renamings map of renamings of GlobalNames
540     * @return
541     */
542   protected def fillConstantsIntersection(th1: Theory, th2: Theory, int : Theory,
          view_map: collection.immutable.Map[FinalConstant, FinalConstant],
```

```
      view_map_inverse: collection.immutable.Map[FinalConstant, FinalConstant],
      renamings: mutable.HashMap[GlobalName, GlobalName]): (Theory, Theory) = {
    //Add constants from th1
    intersectionDeclarations(th1, th2, view_map).map {
      case c: FinalConstant =>
        addDeclaration(c, int, renamings)
        renamings.put(view_map(c).path, renamings(c.path))
    }
    //No defined constants in unary intersection

    (int, int)
  }
}

/** ViewSplitter splits views into lists of pairs
  * Original code by Dennis Mueller
  * with some modifications
  */
object ViewSplitter {

  def apply(v : View) ( implicit controller : Controller) :
      List[(FinalConstant,FinalConstant)] = {
    getPairs(v, controller.getTheory(v.from.toMPath),
        controller.getTheory(v.to.toMPath))
  }

  /**
    * Splits a view up in Pairs of FinalConstants ; takes only those, that are
        directly elements
    * of (dom x cod)
    * @param v view
    * @param dom view's domain
    * @param cod view's codomain
    * @return
    */

  def getPairs(v:View, dom:Theory, cod:Theory) ( implicit controller : Controller) :
      List[(FinalConstant,FinalConstant)]= {
    val domconsts = v.getDeclarations.flatMap(d => d.name.head match {
      case ComplexStep(p) if p==dom.path => Some(d)
      case default => None
    }) collect {
      case c: FinalConstant if c.df.isDefined => c
    }
    (for {o <- domconsts} yield (o.name.tail,o.df.get)).filter(p =>
      p._2.head.get.module == cod.path
    ).map(p => (getConst(dom, p._1),getConst(cod, p._2.head.get.name))) collect {
      case (c1:FinalConstant,c2:FinalConstant) => (c1,c2)
    }
  }

  def getConst(th : Theory, name : LocalName): FinalConstant = {
    name.toList match {
      case head::Nil => th.get(LocalName(head)) match {
        case o : FinalConstant => o
      }
      case submoduleName::tail =>
        val submodule = th.get(LocalName(submoduleName)).asInstanceOf[ModuleOrLink]
```

```
595         getConstSub(submodule, tail)
596       case default => ???
597     }
598   }
599
600   def getConstSub(m : ModuleOrLink, name : LocalName): FinalConstant = {
601     m match {
602       case th : Theory => getConst(th, name)
603       case st: Structure =>
604         st.get(ComplexStep(st.from.toMPath)/name) match {case c : FinalConstant => c}
605     }
606   }
607
608   /**
609     * As above but takes TWO views
610     * @param v1 : dom -> cod
611     * @param v2 : cod -> dom and returns the paired declarations sorted by type:
612     *           (constant -> constant)
613     *           (constant -> term) and
614     *           (term <- constant)
615     * @return paired declarations as a triple of lists
616     */
617
618   def getPairs(v1:View, v2:Option[View], dom:Theory, cod:Theory)
619   :
       (List[(FinalConstant,FinalConstant)],List[(FinalConstant,Term)],List[(Term,FinalConstant)])
       = {
620
621     val consts1 = ((for {o <- v1.getDeclarations if
          o.name.head.toString=="["+dom.path+"]"} yield o) collect {
622       case c: FinalConstant if c.df.isDefined => c
623     }).map(c => Try({
624       val c1 = dom.get(c.name.tail)
625       val c2 = if (c.df.get.head.get.module==cod.path)
            cod.get(c.df.get.head.get.name) match {
626         case d:FinalConstant => d
627         case _ => c.df.get
628       } else c.df.get
629       (c1,c2)
630     })) collect {case Success((a,b)) => (a,b)}
631
632     val consts2 = v2 match {
633       case Some(v) => ((for {o <- v.getDeclarations if o.name.head.toString == "[" +
            cod.path + "]"} yield o) collect {
634         case c: FinalConstant if c.df.isDefined => c
635       }).map(c => Try({
636         val c1 = cod.get(c.name.tail)
637         val c2 = if (c.df.get.head.get.module == dom.path)
              dom.get(c.df.get.head.get.name) match {
638           case d: FinalConstant => d
639           case _ => c.df.get
640         } else c.df.get
641         (c2, c1)
642       })) collect {case Success((a,b)) => (a,b)}
643       case None => List()
644     }
645     val all = (consts1:::consts2).distinct
```

```
646     for(i <- 0 to all.length-2 ; j <- i + 1 until all.length) if
            (all(i)._1==all(j)._1 || all(i)._2==all(j)._2) return (Nil,Nil,Nil)
647     (all collect {case (c:FinalConstant,d:FinalConstant) => (c,d)},
648       all collect {case (c:FinalConstant,t:Term) => (c,t)},
649       all collect {case (t:Term, c:FinalConstant) => (t,c)})
650   }
651 }
652
653 /** Generic BuildTarget for creating intersections
654   *
655   * @tparam GE type of GraphEvaluator with which to measure quality of the resulting
          graph, eg counting declarations
656   * @tparam I type of the Intersecter to use, eg binary intersections
657   */
658 class FindIntersecter[I <: Intersecter, GE <: GraphEvaluator](intersecter : I,
      graphEvaluator : GE) extends BuildTarget {
659
660   var syntaxPresenter : MMTSyntaxPresenter = _
661   var outputPrefix = ""
662
663   override def start(args: List[String]): Unit = {
664     super.start(args)
665
666     controller.extman.addExtension(intersecter)
667     controller.extman.addExtension(graphEvaluator)
668     try {
669       syntaxPresenter = controller.extman.get(classOf[MMTSyntaxPresenter]).head
670     } catch {
671       case e : Exception =>
672         syntaxPresenter = new MMTSyntaxPresenter()
673         controller.extman.addExtension(syntaxPresenter)
674     }
675   }
676
677   /** Searches for views over which to intersect and then does so
678     *
679     * views are found via the viewfinder
680     * intersections done via intersecter of parameterized type I
681     * intersection graphs are evaluated using evaluation function of parameterized
          type GE
682     *
683     * @param a Archive in which intersections are searched for
684     * @return intersected graph as pair of theories and views and a numeric value of
          their evaluation
685     */
686   def findIntersections(a : Archive) : List[((List[Theory],List[View]), Int)] = {
687     //apply viewfinder
688     val viewFinder = new ViewFinder
689     if (controller.extman.get(classOf[Preprocessor]).exists(p => p.key != "" &&
          a.id.startsWith(p.key))) {
690       val preproc = (SimpleParameterPreprocessor + DefinitionExpander).withKey(a.id)
691       controller.extman.addExtension(preproc)
692     }
693     controller.extman.addExtension(viewFinder, List(a.id))
694     val theories = a.allContent.flatMap({p:MPath => try {controller.get(p) match {
          case dt : Theory => Some(p) case _ => None}} catch {case e : Exception =>
          None}})
695     while(!viewFinder.isInitialized) {
```

```
696       Thread.sleep(500)
697     }
698     val views = theories.flatMap(t => viewFinder.find(t, a.id).filter(v =>
            v.from!=v.to).filter(v => v.getDeclarations.nonEmpty))
699     /*
700     //export views to file
701     implicit val fw = new FileWriter(new java.io.File(a.root +
            "/export/intersections/"+a.id+"View.mmt"))
702     views.foreach(v => controller.add(v))
703     views.foreach(syntaxPresenter.apply(_))
704     fw.done
705     */
706
707     //phase 2 : intersect
708     val intersections = views.map(v => (intersecter.intersectGraph(v),
            (v.from.toMPath, v.to.toMPath)))
709
710     //phase 3 : sort
711     //This may need some cleaning up
712     val res = intersections.map { case ((l1, l2, l3, v), (t1, t2)) => ((l2.flatMap(p
            => List(p._1, p._2)).toSet.toList ++ l1 ++ l3, v.flatMap(p => List(p._1,
            p._2))), (t1, t2)) }
713       .map(i => (i, graphEvaluator.eval(i._1._1, i._1._2))).sortBy(_._2)
714     val filterSet = mutable.HashSet[Theory]()
715     res.filter(_._2>0 | true).filter(_ match {
716       case (((theories, views), (t1, t2)), int) =>
717         val dependencies = List(controller.getTheory(t1), controller.getTheory(t2))
718         !dependencies.exists(filterSet.contains) && {
719           filterSet ++= dependencies
720           true
721         }
722       case default => false
723     }).map(r => (r._1._1, r._2))
724   }
725
726   /** a string identifying this build target, used for parsing commands, logging,
            error messages */
727   override def key: String = "intersections"
728
729   /** clean intersections in a given archive */
730   override def clean(a: Archive, in: FilePath): Unit = {
731     val file = new java.io.File(a.root + "/export/intersections/"+a.id+".mmt")
732     file.delete()
733   }
734
735   /** build or update intersections in a given archive */
736   override def build(a: Archive, up: Update, in: FilePath): Unit = {
737     val res = findIntersections(a)
738     implicit val fw: FileWriter = new FileWriter(new java.io.File(a.root +
            "/export/intersections/"+outputPrefix+a.id+".mmt"))
739     res.foreach(r => {
740       r._1._1.foreach(syntaxPresenter.apply(_))
741       r._1._2.foreach(syntaxPresenter.apply(_))
742     })
743     fw.done
744   }
745 }
746
```

```
747 class FindBinaryIntersecter extends FindIntersecter(new BinaryIntersecter, new
        KnowledgeGainGraphEvaluator){
748   outputPrefix = "binary"
749 }
750
751 class FindUnaryIntersecter extends FindIntersecter(new UnaryIntersecter, new
        RepresentationGraphEvaluator){
752   outputPrefix = "unary"
753 }
754
755 /**
756   * Abstract class for GraphEvaluators.
757   * eval(g1)>eval(g2) <=> g1 is better than g2
758   */
759 abstract class GraphEvaluator extends Extension{
760   def apply(graph : List[Theory], views : List[View]=List.empty[View]): Int =
          eval(graph, views)
761
762   /** Evalutes a graph given as a list of theories and views according to an
          evaluation function
763     *
764     * @param theories theories of the graph
765     * @param views views of the graph
766     * @return evaluation as integer value
767     */
768   def eval(theories : List[Theory], views : List[View]=List.empty[View]) : Int
769 }
770
771 /**
772   * Graph Evaluator that counts the number of declarations and structures
773   */
774 class KnowledgeGraphEvaluator extends GraphEvaluator {
775   override def eval(theories : List[Theory], views : List[View]=List.empty[View]) :
          Int = {
776     var ind = 0
777     for (theory <- theories) {
778       for (dec <- theory.getDeclarations) {
779         dec match {
780           case const : Constant =>
781             ind +=1
782           case default =>
783         }
784       }
785     }
786     for (view <- views) {
787       controller.get(view.from.toMPath).getDeclarations.map(
788         {
789           case const : Constant =>
790             if (const.df.nonEmpty) {
791               ind +=1
792             }
793           case default =>
794         }
795       )
796     }
797     ind
798   }
799 }
```

```
800
801 /**
802   * Graph Evaluator that counts the number of declarations and structures
803   */
804 class RepresentationGraphEvaluator extends GraphEvaluator {
805   override def eval(theories : List[Theory], views : List[View]=List.empty[View]) :
        Int = {
806     var count = 0
807     for (theory <- theories) {
808       count -= theory.getDeclarations.length
809     }
810     count
811   }
812 }
813
814 /**
815   * GraphEvaluator that calculates #ind/#rep
816   */
817 class KnowlDivRepGraphEvaluator extends GraphEvaluator {
818   override def eval(theories : List[Theory], views : List[View]=List.empty[View]) :
        Int = {
819     var ind = 0
820     var rep = 1
821     for (theory <- theories) {
822       for (dec <- theory.getDeclarations) {
823         dec match {
824           case const : Constant =>
825             ind +=1
826             rep +=1
827           case default => rep +=1
828         }
829       }
830     }
831     for (view <- views) {
832       controller.get(view.from.toMPath).getDeclarations.map(
833         {
834           case const : Constant =>
835             if (const.df.nonEmpty) {
836               ind +=1
837             }
838           case default =>
839         }
840       )
841     }
842     ind/rep
843   }
844 }
845
846 /**
847   * GraphEvaluator that calculates Knowledge gain(#ind-#rep)
848   */
849 class KnowledgeGainGraphEvaluator extends GraphEvaluator {
850   override def eval(theories: List[Theory], views : List[View]=List.empty[View]): Int
        = {
851     var count = 0
852     for (theory <- theories) {
853       for (dec <- theory.getDeclarations) {
854         dec match {
```

```
855           case s :Structure => count -= 1
856           case default =>
857         }
858       }
859     }
860     for (view <- views) {
861       controller.get(view.from.toMPath).getDeclarations.map(
862         {
863           case const : Constant =>
864             if (const.df.nonEmpty) {
865               count +=1
866             }
867           case default =>
868         }
869       )
870     }
871     count
872   }
873 }
```

Bibliography

[1] Immanuel Normann. “Automated Theory Interpretation”. PhD thesis. 2009.

[2] Michael Kohlhase. “Mathematical Knowledge Management: Transcending the One-Brain-Barrier with Theory Graphs”. In: *EMS Newsletter* (June 2014), pp. 22–27.

[3] Dennis Müller and Michael Kohlhase. *Understanding Mathematical Theory Formation via Theory Intersections in MMT*. 2015.

[4] Dennis Müller, Michael Kohlhase, and Florian Rabe. “Automatically Finding Theory Morphisms for Knowledge Management”. In: *International Conference on Intelligent Computer Mathematics*. Springer. 2018, pp. 209–224.

[5] William M Farmer, Joshua D Guttman, and F Javier Thayer. “Little theories”. In: *International Conference on Automated Deduction*. Springer. 1992, pp. 567–581.

[6] Florian Rabe and Michael Kohlhase. “A scalable module system”. In: *Information and Computation* 230 (2013), pp. 1–54.

[7] *The MMT Language*. `https://uniformal.github.io//doc/language/`. Accessed:2020-8-20.

[8] Florian Rabe. “How to identify, translate and combine logics?” In: *Journal of Logic and Computation* 27.6 (Dec. 2014), pp. 1753–1798. ISSN: 0955-792X. DOI: `10.1093/logcom/exu079`.

[9] Patrick Blackburn and Johan van Benthem. “1 Modal logic: a semantic perspective”. In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, Johan Van Benthem, and Frank Wolter. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, 2007, pp. 1–84. DOI: `https://doi.org/10.1016/S1570-2464(07)80004-8`.

[10] Jacques Carette et al. “The MathScheme Library: Some Preliminary Experiments”. In: (June 2011).

[11] Dennis Müller. “Mathematical Knowledge Management Across Formal Libraries”. PhD thesis. Jan. 2019.

[12] *MMT git repository*. `https://github.com/UniFormal/MMT`. Accessed:2020-8-28.

[13] *MMT Extensions*. `https://uniformal.github.io/doc/api/extensions/`. Accessed:2020-8-26.

[14] *TGStyler*. `https://gl.kwarc.info/mathhub/MichaelBankenProjekt/-/blob/master/graphoptimization.pdf`. Accessed: 2020-08-28.

[15] *The jEdit IDE*. `https://uniformal.github.io//doc/applications/jedit.html`. Accessed:2020-8-26.

[16] *IntelliJ-MMT Plugin*. `https://uniformal.github.io//doc/applications/intellij/index.html`. Accessed:2020-8-26.