Structuring Theories with Implicit Morphisms

Florian Rabe[1,2] and Dennis Müller[2]

[1] LRI Paris
[2] FAU Erlangen-Nuremberg

Abstract. We introduce *implicit* morphisms as a concept in formal systems based on theories and theory morphisms. The idea is that there may be at most one implicit morphism from a theory S to a theory T, and if S-expressions are used in T their semantics is obtained by automatically inserting the implicit morphism. The practical appeal of implicit morphisms is that they hit the sweet-spot of being extremely simple to understand and implement while significantly helping with structuring large collections of theories.

1 Introduction

Motivation Theory morphisms have proved an essential tool for managing collections of theories in logics and related formal systems. They can be used to structure theories and build large theories modularly from small components or to relate different theories to each other [SW83,AHMS99,FGT92]. Areas in which tools based on theories and theory morphisms have been developed include specification [GWM+93,MML07], rewriting [CELM96], theorem proving [FGT93], and knowledge representation [RK13].

These systems usually use a logic L for the fine-granular formalization of domain knowledge, and a diagram D in the category of L-theories and L-morphisms for the high-level structure of large bodies of knowledge. This diagram is generated by all theories and morphisms occurring in a user's development.

For example, a user might reference an existing theory `Monoid`, define a new theory `Group` that extends `Monoid`, define a theory `DivGroup` (providing an alternative formulation of groups based on the division operation), and then define two theory morphisms `G2DG` : `Group` $\leftrightarrow$ `DivGroup` : `DG2G` that witness an isomorphism between these theories. This would result in the diagram on the right.[3]

$$\begin{tikzcd}
 & \texttt{Monoid} \arrow[dl, hook] & \\
\texttt{Group} \arrow[rr, bend left, "\texttt{G2DG}"] & & \texttt{DivGroup} \arrow[ll, bend left, "\texttt{DG2G}"]
\end{tikzcd}$$

The key idea behind implicit morphisms is very simple: We maintain an additional diagram I, which is commutative subdiagram of D and whose morphisms

[3] Note that we use the syntactic direction for the arrows, e.g., an arrow $m : S \to T$ states that any S-expression E (e.g., a sort, term, formula, or proof) can be translated to an T-expression $m(E)$. Models are translated in the opposite direction.

we call *implicit*. The condition of commutativity guarantees that I has at most one morphism i from theory S to theory T, in which case we write $S \overset{i}{\hookrightarrow} T$. Commutativity makes the following language extension well-defined: if $S \overset{i}{\hookrightarrow} T$, then any identifier c that is visible to S may also be used in T-expressions, with the semantics being that c abbreviates $i(c)$. For example, in the diagram above, we may choose to label `DG2G` implicit. Immediately, every abbreviation or theorem that we have formulated in the theory `DivGroup` becomes available for use in `Group` without any syntactic overhead. We can even label `G2DG` implicit as well if we prove the isomorphism property to ensure that I remains commutative, thus capturing the mathematical intuition that `Group` and `DivGroup` are just different formalizations of the same concept. While these morphisms must be labeled manually, any inclusion morphism like the one from `Monoid` to `Group` can be made implicit automatically.

Contribution At the highest level, our contribution is the observation that implicit morphisms form a sweet spot of a very simple language feature that has substantial practical uses. We recommend using implicit morphisms in all theory morphism–based formalisms.

More concretely, we present a formal system for developing structured theories with implicit morphisms. Our starting point is the MMT language [RK13], which already provides a very general setting for defining and working with theories and morphisms. MMT is foundation-independent in the sense that it allows embedding a large variety of declarative languages (logics, type-theories, etc.). Therefore, all our results are foundation-independent — users of a particular language L can use MMT directly as a structuring formalism for L or can easily transfer our results to a dedicated implementation of L.

We describe several example applications of implicit morphisms in detail: the identification of isomorphic theories, definitional extensions of theories, building large hierarchies of theories with many rarely used intermediate theories, and transparently refactoring theory hierarchies.

Overview In Sect. 2, we present the syntax and semantics of MMT. Even though the MMT language is not new, our presentation is an entirely novel contribution in itself: it is much simpler and more elegant than the original one in [RK13]. Crucially, this increase in simplicity allows spelling out the syntax and semantics of implicit morphisms, which we do in Sect. 3, within a few pages. In Sect. 4, we present applications. Finally we review related and future work in Sect. 5.

2 Theories and Theory Morphism

2.1 Overview

Flat Modules MMT provides formal syntax for both structured modules and module expressions and defines their semantics via **flattening**, which defines for every module M a flat module $M^\flat$.

Flat theories are lists of declarations $c : E[= e]$ where E and e are expression, and the latter is optional. We write $\mathtt{dom}(T)$ for the set of constant identifiers c in $T^\flat$ and $\mathtt{Obj}(T)$ for the set of closed expressions using only the symbols $c \in \mathtt{dom}(T)$ (see below for the definition of well-formed expressions). Constant declarations subsume virtually all basic declarations common in formal systems such as type/function/predicate symbols, axioms, theorems, inference rules, etc. In particular, theorems can be represented via the propositions-as-types correspondence as declarations $c : F = P$, which establish theorem F via proof P. Similarly, MMT expressions subsume virtually all objects common in formal systems such as terms, types, formulas, proofs.

Individual formal languages arise as fragments of MMT: they single out the well-formed expressions by defining the two MMT-**judgments** $\vdash_T e : e'$ (typing) and $\vdash_T e \equiv e'$ (equality) for every theory T and $e, e' \in \mathtt{Obj}(T)$. The details can be found in [Rab17].

Flat morphisms from a theory S to a theory T are lists of assignments $c := e$ where $c \in \mathtt{dom}(S)$ and $e \in \mathtt{Obj}(T)$. Every morphism M induces a **homomorphic extension** $M(-) : \mathtt{Obj}(S) \to \mathtt{Obj}(T)$, which replaces every $c \in \mathtt{dom}(S)$ in an S-expression with the T-expression e such that $c := e$ in M.

An MMT **diagram** consists of a set of named structured module declarations. For a given diagram, we write $\mathtt{Thy}$ for the set of theory names. And we write $\mathtt{Mor}(S,T)$ for the set of morphisms defined by

- for every declaration $m : S \to T = \{\sigma\}$, we have $m \in \mathtt{Mor}(S,T)$,
- for every $T \in \mathtt{Thy}$, we have $id_T \in \mathtt{Mor}(T,T)$,
- for every $M \in \mathtt{Mor}(R,S)$ and $N \in \mathtt{Mor}(S,T)$, we have $M;N \in \mathtt{Mor}(R,T)$.

$\mathtt{Thy}$ and $\mathtt{Mor}(S,T)$ form the category of theories and morphisms. Then flattening assigns

- to each $T \in \mathtt{Thy}$ the flat theory $T^\flat$,
- to each $M \in \mathtt{Mor}(S,T)$ the flat morphism $M^\flat$ from $S^\flat$ to $T^\flat$.

Logics and Well-Formed Expressions The logic and the definition of well-formed expressions are not a primary interest of this paper, and we only recap the essential structure needed in the sequel. We refer to [Rab17] for details.

MMT itself is independent of the base logic and provides a theory and theory morphism layer on top of an arbitrary declarative language. Our results in this paper are similarly logic-independent. The syntax and semantics of that base logic is provided by a set of rules for the judgments $\vdash_T e : E$ and $\vdash_T e = e'$ for $e, e', E \in \mathtt{Obj}(T)$. The declarations in theories and morphisms are subject to typing conditions using these judgments. The main theorem about MMT is that well-typed morphisms $M : S \to T$ preserve all judgments, e.g,., if $\vdash_S e : E$, then $\vdash_T M(e) : M(E)$. This includes the preservation of truth via the propositions-as-types principle if E is a proposition and e its proof.

The only aspect of the inference system for these judgments that is relevant to our purposes here is the typing rule for constants:

$$\frac{c : E[= e] \text{ in } T^\flat}{\vdash_T c[= e] : E}$$

Here, in order to avoid case distinctions for the case where a definition is present or not, we use a combined typing+equality judgment: $e_1 = e_2 : E$ represents the conjunction of $e_1 : E$, $e_2 : E$, and $e_1 = e_2$. Correspondingly, the definition of the homomorphic extension of a morphism with domain S includes the following case for constants:

$$M(c) = \begin{cases} e & \text{if } (c : E) \in S^\flat, (c := e) \in M^\flat \\ M(e) & \text{if } (c : E = e) \in S^\flat \end{cases}$$

Here if c has a definiens in S, we expand it before applying M.[4]

Note that these base cases introduce a mutual recursion between well-formedness and flattening: the well-formedness of a declaration in a theory depends on the flattening of all preceding declarations; correspondingly, the well-formedness of an assignment in a morphism depends on the homomorphic extension of the morphism obtained by flattening of all preceding assignments. Vice versa, well-formedness is a precondition for defining the flattening — the definition of flattening may become nonsensical if applied to ill-formed modules.

It is desirable to define well-formedness independently of flattening. But the mutual recursion makes sense from an implementation perspective: typical tools for formal systems first parse, check, and flatten a declaration entirely before moving on to the next declaration. Moreover, often the checking of well-formedness is as difficult or expensive as flattening anyway. In fact, inspecting practical systems with modular features (which often do not have a full rigorous formal specification) shows that our definition elegantly captures the essential commonalities between them.

Therefore, we will make flattening a partial function, i.e., $X^\flat$ is undefined if the module X is not well-formed.

2.2 Syntax

We start with the syntax for theories (which arises as a special case of the one given in [RK13]):

Definition 1 (Theory). *The grammar for theories and expressions is of the form*

$TDec$	$::=$	$T = \{Dec, \dots, Dec\}$	*theory declaration*
Dec	$::=$	$n : E[= E]$	*constant declaration*
c	$::=$	$t?n$	*qualified constant identifiers*
E	$::=$	$c \mid \dots$	*expressions built from constants*

In a theory declaration, each symbol ***name*** *n may be declared only once, and its* ***type*** *and* ***definiens*** *(if present) must be closed expressions over the*

[4] The Mmt tool accepts $c : e \in M^\flat$ even if $(c : E = e') \in S^\flat$ has a definition. In that case, Mmt checks $\vdash_T M(e') = e$ and puts $M(c) = e$. This is important for efficiency but not essential for our purposes here.

previously introduced constants only. We omit the remaining productions for expressions here, which allow forming complex expressions using application, binding, variables, literals, etc.

Example 1. For the purposes of our running examples, we assume a fixed base logic that provides a first-order logic a simple type system. type is the universe of types, A → B is the type of functions, and lambda abstraction is written [x:A] t x. We also use MMT notations, which are attached to constant declarations as # <not>; these are omitted from the formal grammar above because we only use them in the examples.

Then the (flat) theories `Group` and `DivGroup` from the introduction are:

```
Group =
  U          : type
  op         : U → U → U    # 1 ∘ 2
  unit       : U
  inverse    : U → U  # 1 ⁻¹
  // axioms omitted

DivisionGroup =
  U     : type
  div   : U → U → U # 1 / 2
  unit  : U
  // axioms omitted
```

At this point, this grammar only allows forming flat theories, i.e., `Thy` = `thy`. It is intentionally independent of the concrete choice of modularity principles, i.e., the kinds of module-structuring declarations (e.g., includes) and the module-expression-forming operators (e.g., union). But it already introduces all non-terminals for modularity: new declaration kinds for structured theories arise by adding productions for the non-terminal *Dec* and new theory expressions are formed by adding productions for the non-terminal T. This makes it easy to incrementally add new declaration kinds and operators to the language.

We proceed accordingly for flat morphisms:

Definition 2 (Morphism). *A **morphism** is of the form*

$MDec$	$::=$	$m : T \to T = \{Ass, \ldots, Ass\}$	*flat morphism declaration*
Ass	$::=$	$c := o$	*assignment to symbol*
M	$::=$	$m \mid id_T \mid M; M$	*morphism expressions*

such that a morphism declaration contains exactly one assignment $c := o$ *for each* $c \in \mathtt{dom}(S)$*, all of which satisfying* $o \in \mathtt{Obj}(T)$*.*

Example 2 (Morphisms). We give the morphism `DG2G` between the theories from Ex. 3:

```
DG2G : DivisionGroup −> Group =
  U := U
  div  := [a, b]  a ∘ (b^{-1})
  unit  := unit
  // assignments to axioms omitted
```

It maps the universe and unit of a division group to the corresponding notions of a group. And we have `DG2G`$(a/b) = a \circ b^{-1}$. Additionally, the morphism must maps every axiom of `DivGroup` to a proof in `Group` of the translated statement, but we omit those assignments.

Finally, we define diagrams as collections of modules:

Definition 3 (Diagram). *A **diagram** is a list of theory and morphism declarations:*

$$Dia \quad ::= \quad (TDec \mid MDec)^* \qquad \textit{diagrams}$$

A diagram is well-formed if each atomic theory/morphism declaration has a unique name and is well-formed relative to the diagram preceding it.

We give some examples how to add modularity principles to the grammar:

Example 3 (Includes). We extend the grammar with

Dec	::=	**include** S	include a theory into a theory
Ass	::=	**include** M	include a morphism into a morphism

This allows writing the theory `Group` from Ex. 1 by extending a theory of monoids. Again omitting all axioms, this looks as follows:

```
Monoid =
  U: type
  op :  U → U → U        # 1 ∘ 2
  unit   :  U
Group =
  include  Monoid
  inverse  :  U → U  # 1 ^{-1}
```

For each example, we will define the semantics in the next section.

2.3 Semantics

Flattening Just like the syntax, the definition of flattening is **compositional** in the sense that new modularity principles can be added later independently of each other. In particular, for every module expression X, we define its flattening $X^\flat$ by induction on the syntax. This follows the principle of compositional semantics, e.g., $(f(X,Y))^\flat$ depends only on $X^\flat$ and $Y^\flat$.

For theories T, at this point, this induction only has the base case of a reference to a declared theory t. Further cases arise when we add structuring declarations.

For the base case, of a theory $t = \{\Sigma\}$, we define $t^\flat$ by induction on the declarations in Σ:

$$t^\flat = \Sigma^\flat$$

where

$$\cdot^\flat = \varnothing$$

$$(\Sigma, D)^\flat = \Sigma^\flat \cup D^\flat$$

Here $D^\flat$ is the flattening of declaration d relative to $\Sigma^\flat$.

At this point, we only have one case for declarations D, namely constant declarations. Their flattening is trivial:

$$(n : E[= e])^\flat = \{t?n : E[= e]\}$$

where t is the name of the containing theory.

Correspondingly, for a declared morphism $m : S \to T = \{\sigma\}$, we have $m \in \mathtt{Mor}(S, T)$ and $m^\flat$ is defined by induction on the assignments in σ:

$$\cdot^\flat = \varnothing$$

$$(\sigma,\ A)^\flat = \sigma^\flat \cup A^\flat$$

with the trivial base case

$$(c := e)^\flat = \{c := e\}$$

Moreover, for $T \in \mathtt{Thy}$ we define

$$id_T^\flat = \{c := c \mid c : E \in T^\flat\}$$

And for $M; N \in \mathtt{Mor}(R, T)$, we define

$$(M; N)^\flat = \{c := N^\flat(M^\flat(c)) \mid c : E \in R^\flat\}$$

Note that in both cases, we only have to consider R-constants without definiens.

Example 4 (Includes (continued from Ex. 3)). Includes add new declarations D, so we have to add cases to the definition of $D^\flat$. We do this as follows

$$(\textbf{include}\ S)^\flat = S^\flat$$

This has the effect of copying over all declarations from the included into the including theory.

Note that the definition of $n : E = e^\flat$ qualifies the constant name n with the theory name t to form the identifier $t?n$ in the flattening. Therefore, includes can never lead to name clashes.

Also note, that because $S^\flat$ is a *set* of declarations, the include relation is transitive: if t includes s via two different paths, $t^\flat$ only contains one copy of the declarations of $s^\flat$. The situation is slightly more complicated for morphisms: if a morphism out of t includes two different morphisms out of s, these have to agree. Therefore, flattening and well-formedness must be distinguished:

$$(\textbf{include } M)^\flat = M^\flat$$

under the condition $M^\flat$ agrees with $\sigma^\flat$ on any constant that is in the domain of both.

The flat variant $\texttt{Group}^\flat$ of our refactored theory `Group` from Ex. 3 is hence given by copying all included symbols (from `Monoid`) over to `Group`, resulting in exactly the theory as presented earlier in Example 1 except for global identifiers from `Monoid` – i.e. the prefixes t in $t?n$.

3 Implicit Morphisms

3.1 Overview

The motivation behind implicit morphisms is to generalize the benefits of include declarations even further. Sometimes an atomic morphism is conceptually similar to an inclusion: For example, we might want to declare $m : \texttt{Division} \to \texttt{Group} = \{\texttt{divide} := \lambda x, y.x \circ y^{-1},\ \ldots\}$ to show how division is "included" into the theory of groups. Other times we have to use structures because we need to duplicate declarations but we would still like to use them like includes. For example, we have to declare the theory of rings using two structures (for the commutative group and the monoid). But we still want a ring to be implicitly converted into, e.g., a commutative group.

Our key idea is to use a commutative[5] subdiagram of the MMT diagram, which we call the *implicit-diagram*. It contains all theories but only some of the morphisms – the ones designated as *implicit*. Because the implicit-diagram commutes, there can be at most one implicit morphism from S to T – if this morphism is M, we write $S \overset{M}{\hookrightarrow} T$.

The implicit-diagram generalizes the inclusion relation $\hookrightarrow$. In particular, all identity and inclusion morphisms are implicit, and we recover $S \hookrightarrow T$ as the special case $S \overset{id_S}{\hookrightarrow} T$. Moreover, the relation "exists M such that $S \overset{M}{\hookrightarrow} T$" is also an order.

Consequently, many of the advantages of inclusions carry over to implicit morphisms.

- It is very easy to maintain the implicit-diagram, e.g., as a partial map that assigns to a pair of theories the implicit morphism between them (if any).
- We can generalize the visibility of identifiers: If $S \overset{M}{\hookrightarrow} T$, we can use all S-identifiers in T as if S were included into T. Any $c \in \texttt{dom}(S)$ is treated as a valid T-identifiers with definiens $M(c)$.
- We can still use canonical identifiers $s?n$. Because there can be at most one implicit morphism $s \overset{M}{\hookrightarrow} T$, using $s?n$ as an identifier in T is unambiguous. Crucially, we can use $s?n$ without bothering what M is.

[5] A diagram commutes if for any two morphisms $M, M' \in \texttt{Mor}(S, T)$, we have $M \equiv M'$.

3.2 Syntax

We introduce the family of sets $\mathtt{Mor}^{\mathtt{i}}(S,T)$ as a subset of $\mathtt{Mor}(S,T)$, holding the *implicit* morphisms. The intuition is that `Thy` and $\mathtt{Mor}^{\mathtt{i}}(S,T)$ (up to equality of morphisms) form a thin broad subcategory of `Thy` and $\mathtt{Mor}(S,T)$.

It remains to define which morphisms are implicit. For that purpose, we allow MMT declarations to carry *attributes*. Precisely, we add the following productions

MDec	::=	*Att MDec*	attributed morphism
Dec	::=	*Att Dec*	attributed declaration
Att	::=	**implicit** | ...	attributes

The set of attributes is itself extensible, and the above grammar only list the one that we use to get started. Additional attributes can be added when adding modularity principles.

Example 5. We can now change the declaration of the morphism `DG2G` from Ex. 2 by adding the attribute **implicit**.

3.3 Semantics

We only have to make two minor changes to the semantics to accommodate implicit morphisms. The first change govern how we obtain implicit morphisms, the second one how we use them.

Obtaining Implicit Morphisms Based on the above attributes, we define the set $\mathtt{Mor}^{\mathtt{i}}(S,T) \subseteq \mathtt{Mor}(S,T)$ of **implicit** morphisms to contain the following elements:

- all declared morphisms $m : S \to T = \{\sigma\}$ whose declaration carries the attribute **implicit**,
- all identity morphisms id_T,
- all compositions $M;N$ of implicit morphisms M and N,
- all morphisms that additional language features designate as implicit based on the use of additional attributes.

To define well-formedness for our extended syntax, we say that adding an attribute to a declaration is only well-formed if there is (up to equality of morphisms) at most one implicit morphism for any pair of theories, i.e., $\mathtt{Mor}^{\mathtt{i}}(S,T)$ has cardinality 0 or 1. Thus, the implicit morphisms form a thin broad subcategory of theories and morphisms – in other words, a commutative diagram.

Example 6 (Includes (continued from Ex. 4)). For include morphism, we add the following definition: if theory t contains the declaration **include** S, then the induced morphism from S to t is implicit.

Combined with composition morphisms, we see that all transitive includes between theories are implicit. That corresponds to the intuition that anything that is include should be available directly.

Example 7. For our morphism `DG2G` from Ex. 5, this means that all symbols declared in the theory `DivisionGroup` (see Ex. 1) are now visible in the theory `Group` with the definitions provided by `DG2G`.

Using Implicit Morphisms The intuition behind implicit morphisms is that all S-constants $c : E$ that can be mapped into the current theory via an implicit morphism $M : S \to T$ are directly available in T. We can practically realize this by adding new defined constants $c : M(E) = M(c)$ to T. However, physically adding definitions can be inefficient. It is more elegant to modify the typing rules such that $\vdash_T c : M(c) = M(E)$.

To do that, we only have to make a small modification to the original rules of MMT as presented in Sect. 2. To illustrate how simple the modification is, we repeat the original rule first for comparison:

$$\frac{c : E[= e] \text{ in } T^\flat}{\vdash_T c[= e] : E}$$

Now our modified rule is

$$\frac{c : E[= e] \text{ in } S^\flat \qquad S \overset{M}{\hookrightarrow} T}{\vdash_T c = M(c) : M(E)}$$

Note that the modified rule gives every constant a definiens. This is a technical trick to subsume the original rule: if c is already declared in T, we use $M = id_T$ and obtain $\vdash_T c = c : E$. We write $\overline{E}$ for the expression that arises from E by recursively replacing every c with its definiens.

The following theorem is our central theoretical result. It shows that the above modification has the intended properties:

Theorem 1 (Conservativity of Implicit Morphisms). *Consider any combination of modular features from the examples above that includes at least implicit identity morphisms.*

Then for well-formed diagrams and $S, T \in \texttt{Thy}$ and $M \in \texttt{Mor}(S, T)$

1. *Whenever the original system proves $\vdash_T e = e' : E$, so does the modified one.*
2. *Whenever the modified system proves $\vdash_T e = e' : E$, then the original inference system proves $\vdash_T \overline{e} = \overline{e'} : \overline{E}$.*

Proof. For the first claim, we proceed by induction on derivations. We only need to consider case where the original rule was applied. So assume it yields $\vdash_T c[= e] : E$, i.e., $(c : E[= e])$ in $T^\flat$. We apply the modified rule for the special case $T \overset{id_T}{\hookrightarrow} T$. The conclusion reduces to

– if e is absent: $\vdash_T c = c : E$, which is equivalent to $\vdash_T c : E$,
– if e is present: $\vdash_T c = e : E$ because $M(c) = M(e)$ according to the definition of $M(-)$.

For the second claim, we proceed by induction on derivations. We only need to consider the cases where the modified rule was applied. So assume it yields $\vdash_T c = M(c) : M(E)$ for $(c : E[= e])$ in $S^\flat$ and $S \overset{M}{\hookrightarrow} T$. We distinguish two cases:

- $M(c) = c$, i.e., c does not have a definiens: According to the definition of $M(-)$ this is only possible if e is absent. According to the definition of $T^\flat$, this is only possible if $c = S?n$ for $S \hookrightarrow T$ (including the special case $S = T$). In that case, $S^\flat \subseteq T^\flat$ and M is the include/identity morphism that maps all S-constants to themselves. Now applying the induction hypothesis to the well-formedness derivation of E yields $\vdash_T c : \overline{E}$ as needed.
- $M(c) \neq c$, i.e., c has a non-trivial definiens: Definition expansion eliminates c in favor of e. Clearly $\overline{c} = \overline{M(c)}$, and we only have to show that $\vdash_T \overline{M(c)} : \overline{M(E)}$ That follows from the judgment preservation of morphisms.

4 Applications

4.1 Identifying Theories via Implicit Isomorphisms

A common need in developing large libraries (both in formal and informal developments) is to identify two theories S and T via a canonical choice of isomorphisms. In these cases, it is often desirable to identify S and T, e.g., when working with S, we want to be able to use T-syntax directly without having to apply an isomorphism. One of the major short-comings of formal theories over traditional informal developments is that formal systems usually need to spell out these isomorphisms everywhere. In this section, we show that implicit morphisms elegantly allow exactly that kind of intuitive identification.

Note that because identity morphisms are implicit, our uniqueness requirement for implicit morphisms implies that two theories S and T must be isomorphic if there are implicit morphisms in both directions. Moreover, making a pair of isomorphisms implicit is well-formed if there are no other implicit morphisms between S and T yet.

Renamings We say that a named morphism $r : S \to T = \{\dots\}$ is a **renaming** if

- all assignments in its body are of the form $c := c'$ for T-constants c' without definiens
- every such T-constants c' occurs in exactly one assignment.

Clearly, every renaming is an isomorphism. The inverse morphisms contains the flipped assignments $c' := c$.

We make the following extension to syntax and semantics:

- A morphism declaration $r : s \to t = \{\dots\}$ may carry the attribute **renaming**.
- This is well-formed if there are no implicit morphism between s and t yet.
- In that case, we define $r \in \mathtt{Mor^i}(s,t)$ and $r^{-1} \in \mathtt{Mor}(t,s)$.

Example 8 (Renaming). Consider a variation of the theory `Monoid` from Ex. 3 in a different library:

```
Monoid2 =
  M: type
  connective  :  M → M → M        # 1 ∘ 2
  neutral        :  M
```

This theory is isomorphic to the previously introduced theory `Monoid` under the trivial renaming

```
renaming MonoidRen : Monoid2 −> Monoid =
  M              := U
  connective     := op
  neutral        := unit
```

Definitional Extensions We say that the named theory t is a **definitional extension** of S if $t = S$ or the body of t contains
– only constant declarations with definiens,
– only include declarations of theories that are definitional extensions of S.

If t is a definitional extension of S, it is easy to prove that t and S are isomorphic: both isomorphisms map all constants without definiens to themselves. In particular, the isomorphism $S \to t$ maps S-constants to themselves and expands the definiens of all other constants.

We make the following extension to syntax and semantics:
– An include declaration **include** s of a named theory s inside theory t may carry the attribute **definitional**.
– In that case, we define $id_s \in \texttt{Mor}^{\texttt{i}}(t, s)$ (in addition to the implicit morphism $id_t \in \texttt{Mor}^{\texttt{i}}(s, t)$ which is induced by the inclusion).

Example 9 (Extension with a Theorem). We extend the theory `Group` from Ex. 3 with a theorem

```
definitional   InverseInvolution   =
  include  Group
  inverse_invol    :  ∀[x]  (x^{−1})^{−1}≐x = (proof  omitted)
```

Remark 1 (Conservative Extensions). A definitional extension is a special case of a conservative extension. More generally, all retractable extensions are conservative, i.e., all extensions $S \hookrightarrow T$ such that there is a morphism $r : T \to S$ such that r is the identity on S.

But we cannot make the retractions implicit morphisms in general because they are not necessarily isomorphisms.

Canonical Isomorphisms If we have isomorphisms $m : s \to t$ and $n : t \to s$, we simply spell them out in morphism declarations and add the keyword **implicit** to both. This requires no language extensions.

Example 10. Having declared the morphism `DG2G` (as in Ex. 2) implicit, we do the same with the reverse morphism `G2DG`:

```
implicit   G2DG : Group −> DivisionGroup =
  U := U
  op := [a, b]  a/(unit /b)
  unit  := unit
  inverse  := [a]  unit /a
```

While the first one of these isomorphism declarations is straightforward, the second one requires checking that m and n are actually isomorphisms. Otherwise, the uniqueness condition would be violated. Thus, we have to check $m;n = id_s$ and $n;m = id_t$. In general, the equality of two morphisms $f, g : A \to B$ is equivalent to $\vdash_B f(c) = g(c)$ for all $c : E \in A^\flat$. Thus, if equality of expressions is decidable in the logic that MMT is instantiated with, then MMT can check this directly.

However, this does not work in practice. Already elementary examples require stronger, undecidable equality relations are used:

Example 11. Consider the isomorphism from Ex. 10. The result of mapping $x \circ y$ from `Group` to `DivGroup` and back is $x \circ (unit \circ y^{-1})^{-1}$. Clearly, the group axioms imply that this is equal to $x \circ y$. But formally that requires working with the undecidable equality of first-order logic.

Therefore, in our running example, we only make one of the two isomorphisms implicit.

In the sequel, we design a general solution to this problem. It allows systematically proving the equality of two morphisms and using that to make both isomorphisms implicit. This is novel work, but it requires significant prerequisites and is only peripherally related to implicit morphisms. Therefore, we only sketch the idea and leave the details to future work.

We add a language feature to MMT to prove equalities between morphisms: We add the productions

$$\begin{array}{lll} Dia & ::= & (TDec \mid MDec \mid MEq)^* \\ MEq & ::= & \textbf{equal}\, M = M : T \to T by \{Ass^*\} \end{array}$$

We define the declaration $M = N : S \to T by \{\sigma\}$ to be well-formed iff

- $M : S \to T$ and $N : S \to T$ are well-formed morphisms
- σ contains exactly one assignment $c := p$ for every $(c : E) \in S^\flat$
- for each of these assignments $c := p$, the term p is a proof of $\vdash_T M(c) = N(c)$.

To make m and n from above implicit isomorphisms, we have to do three things: define m and n, prove the equalities of $m;n = id_s$ and $n;m = id_t$, and make m and n implicit. Note that we cannot make both m and n implicit right away because that is only well-formed after proving the equalities.) Thus, we define a new attribute **implicit-later**, which states that a morphism should be considered implicit as soon as subsequent equality proves make it well-formed to do so.

Example 12 (Isomorphisms). We can now add declarations

$$\textbf{implicit} : \texttt{DG2G} : \texttt{DivGroup} \to \texttt{Group} = \text{(as above)}$$

$$\textbf{implicit-later} : \texttt{G2DG} : \texttt{Group} \to \texttt{DivGroup} = \text{(as above)}$$

$$\textbf{equal}\, \texttt{G2DG}; \texttt{DG2G} = id_{\texttt{Group}} : \texttt{Group} \to \texttt{Group} = \text{(omitted)}$$

$$\textbf{equal}\, \texttt{DG2D}; \texttt{G2DG} = id_{\texttt{DivGroup}} : \texttt{DivGroup} \to \texttt{DivGroup} = \text{(omitted)}$$

where the isomorphisms are as above and we omit all the equality proofs.

4.2 Fine-Granular and Flexible Theory Hierarchies

A common problem when defining modular theory hierarchies is that the most natural include-hierarchy for the most important theories is not necessarily the same as the most comprehensive hierarchy. For example, Ex. 3 defines `Group` with an include from `Monoid`. Instead, we could include `Monoid` into the intermediate theory `CancellationMonoid` and include that into `Group`. This change is not possible in retrospect — changing the theory hierarchy (which is one of the most fundamental structures of a library) usually presents an insurmountable refactoring problem. So instead we could systematically build a hierarchy that uses every intermediate theory (as done in [CFO11]). But this yields a very deep and complex hierarchy that is hard to navigate for casual users. Moreover, it does not protect us from later on discovering yet another intermediate theory that should have been added.

Implicit morphisms provide a simple solution to this problem because they behave effectively like inclusions but can be added later on. In the above example, we would

– define `CancellationMonoid` with an include from `Monoid`,
– keep `Group` as it is, i.e., also with an include `Monoid`,
– add an implicit morphism `CancellationMonoid` $\to$ `Group`.

As a more complex example, we have formalized a hierarchy of theories between `Band` and `Semilattice`:

$$\begin{tikzcd}
 & & z \circ x \circ y \circ z \doteq z \circ x \circ z \circ y \circ z \arrow[dl] \arrow[dr] & & \\
 & z \circ x \circ y \doteq z \circ x \circ z \circ y \arrow[dl] \arrow[dr] & & y \circ x \circ z \doteq y \circ z \circ x \circ z \arrow[dl] \arrow[dr] & \\
x \circ y \doteq x \circ y \circ x \arrow[dr] & & z \circ x \circ y \circ z \doteq z \circ y \circ x \circ z \arrow[dl] \arrow[dr] & & x \circ y \doteq y \circ x \circ y \arrow[dl] \\
 & z \circ x \circ y \doteq z \circ y \circ x \arrow[dr] & & x \circ y \circ z \doteq y \circ x \circ z \arrow[dl] & \\
 & & x \circ y \doteq y \circ x & &
\end{tikzcd}$$

All of these theories are of the form $t = \{$**include** `Band`, $a : F\}$, where F is the formula given in the diagram above.[6] In particular, `SemiLattice` $=$ $\{$**include** `Band`, $a : \forall[x]\forall[y]x \circ y \doteq y \circ x\}$.

There are various morphisms between them that describe the lattice structure of the corresponding varieties of bands. All of these map the constants from `Band` to themselves and the axiom a to a proof. In our formalization, we make all of these morphisms implicit.[7] It is straightforward to prove that the diagram

[6] All varieties of bands can be axiomatized in this way.

[7] Our formalization of the lattice of bands using implicit morphisms can be found at `https://gl.mathhub.info/MitM/smglom/blob/devel/source/algebra/bands.mmt`.

commutes: any two morphisms are identical except for assignment for the axiom, and these are provably equal due to proof irrelevance.

4.3 Transparent Refactoring

A major drawback of using modular theories is that it can preclude transparent refactoring. For example, consider a theory $t = \{\textbf{include } r, \textbf{include } s\}$, and assume we want to move a constant declaration D for the name n from r to s. Thus, the change to should be straightforward as it does not change the semantics of t.

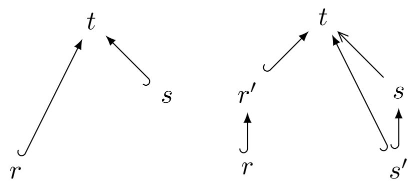
However, this is not a local change. It also requires updating every qualified reference from $r?n$ to $s?n$. Such references can occur anywhere where t is used. That may include theories that the person who does the refactoring does not know or does not have access to. Even if the source files always use the unqualified reference n (because the checker is smart enough to dynamically disambiguate them), this still requires a global rebuild to reach a consistent state again.

With implicit morphisms, we can solve this problem by making only the following local changes:

1. We rename s to s'.
2. We delete the declaration $n : E$ from s'.
3. We create new theories $r' = \{\textbf{include } r, D\}$ and $s = \{\textbf{include } s', D\}$.
4. We change t to $t = \{\textbf{include } r', \textbf{include } s'\}$.
5. We add an implicit morphism $s \to t$ that maps $s?n$ to $r'?n$.

Now t has the desired new structure. But, all old references to $s?n$ stay well-formed so that no global changes are needed, as in Fig. **??**.

The situation before (left) and after (right) refactoring is given below:



$$\begin{tikzcd}
 & t & & & t & \\
 & & s \arrow[ul, hook] & r' \arrow[ur, hook] & & s \arrow[ul] \\
r \arrow[uur, hook] & & & r \arrow[u, hook] & & s' \arrow[u, hook] \arrow[uul, hook]
\end{tikzcd}$$

5 Conclusion and Related Work

Implicit Conversions The need for implicit conversions has been recognized in many formal systems. In all cases, similar uniqueness constraints are employed as in ours.

Type-level conversions are functions between types such as the conversion from natural numbers to integers. **Theory-level** conversions morphisms between theories, like in this paper. The latter can be seen as a special case of the former: if every theory is seen as the type of its models (as in [MRK18]), then reduction along an implicit morphism $S \to T$ induces a conversion from T-models to S-models. Type-level conversions are present in many systems, e.g., the Coq proof assistant [Coq15] or the Scala programming language. The novelty in our approach is to restrict conversions two-fold: firstly to the theory level, secondly to those conversion functions that can be expressed as theory morphisms. This significantly reduces the complexity and permits an elegant logic-independent semantics, while still being practically useful.

Some formal systems support theory-level conversions without explicitly using theory morphisms. This is common in systems that use type classes as an analogue to theories. For example, the `sublocale` declarations of the proof assistant Isabelle [KWP99] or the `deriving` declarations of the programming language Haskell can be seen as **anonymous** implicit morphisms. Making theory morphisms primitive objects, as in our case, yields a simpler and more expressive implicit-conversion-system at the price of having an additional primitive concept.

Structuring Theories In systems that maintain large diagrams of theories, the problems solved by our approach have been recognized for some time. For example, the IMPS system [FGT93] allowed using theory morphisms to retroactively add defined constants to a previously declared theory. This corresponds to a definitional extension with an implicit retraction morphism as in Sect. 4.1.

In [CFK14], the idea of *realms* was introduced as a way to bundle a set of isomorphic theories and their definitional extensions into a single interface. The paper called for an implementation of realms as a new primitive concept in addition to theories and morphisms. In contrast, the much simpler feature of implicit morphisms achieves very similar goals: realms can be recovered by marking all isomorphisms as *implicit* and all extensions as *definitional*.

Scalability and Scoping Future work will focus on utilizing and evaluating implicit morphisms in large libraries, i.e., diagrams with thousands of theories and as many implicit morphisms as possible.

In doing so, we will pay particular attention to some problems that implicit conversions can cause at large scales. Users can be confused when implicit conversions are applied that they are not aware of. Different users may also have different preferences for which conversions should be implicit. And most critically, different developments may be incompatible if they introduce different implicit morphisms between the same theories. For those reasons, Scala, for example, only applies implicit conversions that are imported into the current namespace.

We anticipate that these problems will lead to an evolution of our solution that allows more localized control over which morphisms are implicit. Thus, instead of a single global diagram of implicit morphisms, every context may carry its own local one. But we defer this until the current implementation has been used to conduct very large case studies.

References

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

CELM96. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.

CFK14. J. Carette, W. Farmer, and M. Kohlhase. Realms: A Structure for Consolidating Knowledge about Mathematical Theories. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 252–266. Springer, 2014.

CFO11. J. Carette, W. Farmer, and R. O'Connor. Mathscheme: Project description. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, volume 6824, pages 287–288. Springer, 2011.

Coq15. Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

FGT92. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

FGT93. W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

GWM$^+$93. J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ.* Cambridge, 1993.

KWP99. F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.

MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *Tools and Algorithms for the Construction and Analysis of Systems 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.

MRK18. D. Müller, F. Rabe, and M. Kohlhase. Theories as Types. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning*, pages 575–590. Springer, 2018.

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

Rab17. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

SW83. D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.