Mei – A Module System for Mechanized Mathematics Systems

(An extended version of PH.D. Thesis)

By
Jian Xu
McMaster University, Canada

January 30, 2008

Abstract

This thesis presents several module systems, in particular Mei and DMei, designed for mechanized mathematics systems. Mei is a λ-calculus style module system that supports higher-order functors in a natural way. The semantics of functor application is based on substitution. A novel coercion mechanism integrates a parameter passing mechanism based on theory interpretations with simple λ-calculus style higher-order functors. DMei extends Mei by supporting dependent functor types. Mei is the first module system that successfully supports both higher-order functors and a parameter passing mechanism based on theory interpretations.

Acknowledgments

I would first like to express my sincere thanks and appreciation to my supervisor, Dr. William M. Farmer, for his insight, thoughtful guidance and constant encouragement throughout my study.

Thanks to the members of my Supervisor Committee: Dr. Jacques Carette, Dr. Tom Maibaum, and Dr. Jeffery I. Zucker, for many valuable discussions and comments. Thanks to all my other professors for their help in these four years. Thanks to friends in ITC 206 for the time we shared together.

Thanks to my brother, Qin, for his encouraging words “You might not be born to be creative, however you can work hard to be creative.”. Thanks to my sister, Yanping, and my brother, Jianping for pushing me forward constantly.

Last but not least, many thanks to my parents for their support, my daughters, Katie and Kasidy, for the happiness they brought to me, and my wife, Yan Li, for her love and encouragement.

I am grateful to have been partially supported by postgraduate scholarships from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ontario Graduate Scholarships in Science and Technology (OGSST).

Contents

Chapter 1

Introduction

A mechanized mathematics system (MMS) is a computer system that supports and improves mathematical processing, by which we mean building mathematical models of the real world, and formulating and reasoning about properties of the real world within the context of the mathematical models. Examples of MMSs are theorem provers and computer algebra systems. Although it is important for an MMS to have an expressive and practical logic, an efficient proof engine, and a friendly user interface, it is barely useful without a powerful library. A powerful library for an MMS should (1) contain sufficient mathematical knowledge to support mathematical activities, and (2) be well organized so that new knowledge can be easily developed from existing knowledge. Currently, more and more mathematical knowledge is being formalized in different systems, which largely achieves the first goal. Thus, a good module system is necessary for MMSs in order to organize this mathematical knowledge. A good modularity mechanism aids the expressivity of MMSs. It helps to build up contexts and allows the user to reuse the theorems developed within one context in other contexts with similar structure. However, the development of the module systems for MMSs significantly lags behind the development of underlying logics, proof strategies, computational power, algorithms etc. Very few MMSs have a sophisticated module system that supports the development of large pieces of mathematical knowledge.

In this thesis, we will present several related module systems designed for MMSs. In this chapter, we will investigate some existing module systems, present our design goals, give an overview of our systems, and state our major contribution.

1.1 Background study

Most programming languages and specification languages have module systems for organizing large software developments and specifications. In this section we will review two families of module systems, namely, the ML-family module systems and the algebraic specification module systems.

Modular programming is a way to structure a large software system consisting of a collection of small pieces of codes. Though modular programming can be done in any programming language with sufficient discipline from programmers [75], many modern languages provide facilities for expressing and automatically checking their modular structures. The module systems of programming languages emphasize the decomposition of a large software development into smaller components. Code reuse and parallel software development are important for them. On the one hand, it is desirable to decompose programs into modules that are as independent as possible. On the other hand, these modules have to interact within some program, which requires an *interface* to express the essential information for communication. Among many modular mechanisms, the ML-family module systems are of particular interest because it treats parameterized modules as functors, i.e. functions from modules to modules. The module system itself is a small typed functional language, where structures and functors are objects and signatures are types. Hence researchers in this area enjoy a type-theoretic formalism approach to their module system [47, 57]. There are efforts to extend proof assistant systems with an ML-family module system, e.g. in *Coq* [18].

Algebraic specification languages are another area enjoying modularity. The module systems of specification languages stress a refinable way of specification developments. Theories and axioms are important for them; however, logical reasoning is not one of their concerns. A module of a specification language is an algebraic specification consisting of a signature and a set of axioms, roughly equal to our definition of a theory. The researchers in this area favour a category-theoretic approach where *specification morphism* (or simply *morphism*) plays a central role. A morphism is a homomorphic map from the sorts and operators of one specification to the sorts and operators of another such that axioms of the source specification are translated to axioms (or more generally, theorems) of the target specification. Most model specification languages provide some *specification-building operations* to form more complex specifications from smaller specifications, such as extension, union, renaming, and parameterized specification [71].

Both of these approaches developed a lot of great ideas to support modularity. Our work borrows ideas from both and integrates these ideas to build a module system that is suitable for MMSs. For instance, we like the idea of higher-order functors from ML-family module systems and the so-called fitting morphism style parameter passing mechanisms from algebraic specification languages. However, no current module system supports them both.

1.1.1 ML-family module systems

ML is a functional language, originally designed by R. Milner, with a family of descendants [59, 66]. Despite various extensions, every ML family language has a functional core language (with or without imperative features) and a module language. The core language is used to specify algorithms and data structures, and the module language is used to specify the structures of large software systems.

Signatures, structures, and functors. In an ML-family module language, the definitions of components in the core language are grouped into *structures*. *Signatures* are used to specify the interface of structures, by declaring or defining types and specifying functions with their types. A structure realizes a signature by defining the types and implementing all the specified components. In other words, a signature is a type representing a class of structures.

A *functor* is a parameterized structure that maps one structure to another structure, i.e. a function over structures. A signature is used to specify the parameter structures to which a functor can be applied. The body of a functor is defined with respect to an arbitrary structure, represented by a structure variable, satisfying the formal parameter signature. The functor can be applied to any structure that realizes its parameter signature. The semantics of instantiations of functors is defined via substitutions, as illustrated in Figure 1.1. The result is a concrete implementation of the functor body, which is a structure. Note that the body definition may or may not employ the formal parameter, although, in most cases, the body definition is an extension of the formal parameter. In other words, the body is defined in terms of arbitrary module expressions over the formal parameter. Since renaming is not supported, there is at most one occurrence of the formal parameter within the body definition. However, there are cases when we want to have more than one occurrence of the formal parameter within the body definition. For example, there are two monoid structures residing in a ring structure.

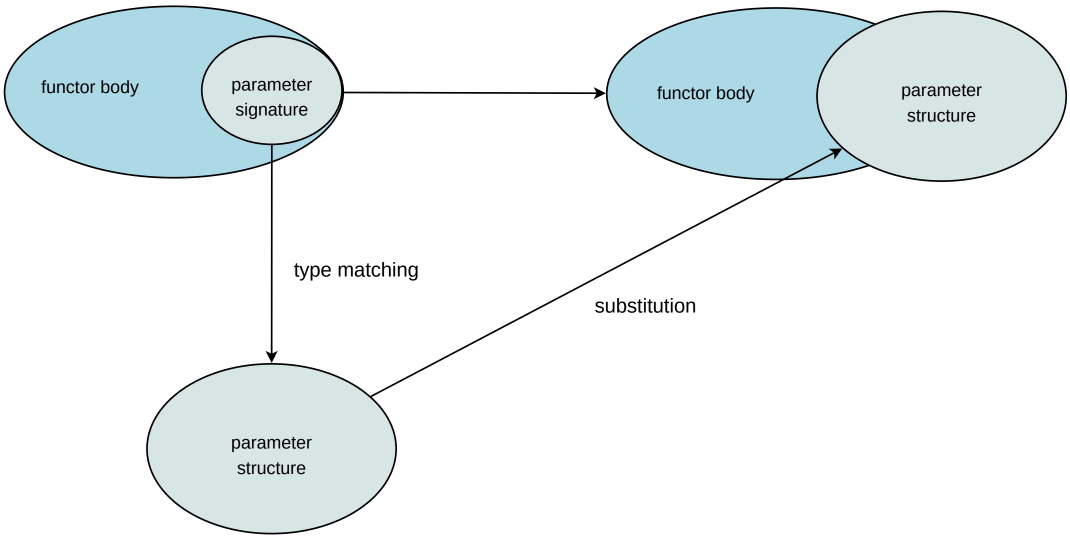

Figure 1.1: ML Style Functor Application

Parameter passing. The parameter passing mechanism of an ML-family module system is based on a matching between a formal signature and an actual structure, which is a purely syntactical issue. For example, if a signature S with an abstract type `t` specifies the input interface of a functor, only the structures, in which the type `t` is either declared or defined, can be used to instantiate the functor. The type `t` may have a concrete representation in an actual parameter structure, e.g. `t` = `int`. This effectively links the abstract type `t` and the concrete type `int` and separates the outside view of a structure from its concrete representation. By keeping the parameter matching purely syntactical, we can get a statically decidable type checker.

Looking from another angle, if we treat both signatures and structures as some kind of mathematical theories, type definitions such as `t` = `int` are a variant of symbol mappings from a theory to another theory, resembling theory translation as defined in Chapter 3.4.1. The differences are: (1) a mapping is embedded in a structure so that a structure can only be used for certain signatures, which is not flexible, and (2) a mapping is restricted to map type symbols, not function symbols (function symbols are matched purely by their names). A more general parameter passing mechanism should separate a mapping from a structure, which has an explicit theory translation style. This would allow a group structure (representing a group theory) to be used where structures of a monoid signature are required, even though they do not share the same names.

Higher-order functors. The functors introduced so far are first-order in the sense that they are functions over structures. The notion of *higher-order functor* is a generalized notion of functor such that (possibly high-order) functors can be used as the actual parameters and return values of other functors, just as higher-order functions can take other functions as input and return functions as output[1]. Extending the ML-family module systems to support higher-order functors involves the following issues:

(1) **Phase separation and separate compilation** A module is separated into a static part (type information) and a dynamic part (code). Type checking does not require any information from the dynamic code part. Typical solutions are Harper and Lillibridge's "translucent sums" and Leroy's "manifest types", in which a module system is a stratified layer on top of a core language. The major idea is that all information required by interactions between modules is captured by the signatures of modules. Concrete information required for interactions between modules is modeled by transparent (or manifest) types, which support separate compilation [47, 57].

Let us look at an example from [57]. In SML, assume that $\mathsf{S} : \Sigma$ and Σ specifies a type component t. Even though the signature Σ does not say anything about the implementation of t, another structure S' can rely on $\mathsf{S.t}$ being implemented as some particular type, say, int. This is because that type specifications in SML signatures are transparent and SML is defined as "an interactive language". The latter implies that users are expected to build their programs linearly in strict bottom-up order. Hence, the fact that t is implemented as int is available when defining S'. However, if we want to define S and S' in different compilation units, the implementation defining S' therefore cannot be typechecked until the implementation defining S has been written. The reason is that $\mathsf{S} : \Sigma$ does not suffice to determine whether it is correct to assume $\mathsf{S.t}$ to be int. The solution is to make type specifications in signatures opaque and enrich signatures with manifest type specifications, $\mathsf{type\ t} = \tau$.

This issue is not a problem for most MMSs because: (1) most MMSs are designed as an interactive language like SML, and (2) manifest types can be built in most MMSs. In MMSs, all types are abstract; their behaviours are defined by axioms. The concrete type representation $\mathsf{t} = \mathsf{int}$ can be expressed by two

[1] The usage of higher-order functors was shown in [14, 15] which employed Ocaml's higher-order functors extensively.

abstract type declarations and an axiom expressing their equality. Users can choose to expose only the abstract type `t` or both `t` and `int` and the axiom. The latter choice effectively simulates a manifest type.

(2) **Generativity** A functor is called *applicative* if two instantiations of it with the same actual parameter are compatible; otherwise it is called *generative* [62]. A generative functor creates a new type for each of its instantiations, i.e. instances of an abstract type are distinct. The question of which approach is better is still an open debate.

Note that the notion of generative functor is necessary only when the behaviour of a functor depends on a stateful object. In other words, the behaviour of an instantiation of a functor varies according to the value of an object, which varies according to the state of the system. A stateful object necessarily stands outside a functor and its actual parameter, i.e. it is global in some sense. As discussed later in §1.2, a module in an MMS is not only a name space mechanism, but also a reasoning context. Consequently, there is no global object that can be accessed by more than one theory. To access an object, one has to import it in some way. In this sense, there is no place for generative functors, unless a module is only used to solve name conflicts, as in *Coq*. We thus favour the notion of applicative functor.

(3) **Modules as first-class values** Some ML-family modules are first-class objects [62]. This means that a module has the full properties of any other object of the core language. For instance, modules can be used as an actual argument of a normal function of the core language. This requires building data types representing modules within the core language. For instance, in a system supporting constructive type theory, structures can be formalized as dependent records, signatures as dependent record types, and functors as functions over record types [21, 22]. Higher-order functors are natural, since functions are higher-order. One advantage of this approach is that users can exploit the computation mechanisms of the core language in the construction of modules. The biggest drawback of this approach is that we are forced to amalgamate a module language with a core langauge. This adds additional requirements on the expressivity of the core language and restricts the application of a module system.

In this thesis, we will not consider "modules as first-class values" for two reasons: (1) since a module language is merely used to express how to build theories from other theories (and/or functors), a simple notion of computation is enough for that purpose, and more importantly, (2) a module system should be orthogonal to underlying systems, so that it is applicable to arbitrary systems.

1.1.2 Algebraic specification systems

Specification languages are used to specify the software development process in terms of a sequence of progressively refined descriptions of a software system, resulting in a possibly executable program and its documentation. At each step, a description specifies what a software system must do, not how it will do it. Equational logic was extensively adopted as the basic formalism of specifications. The concern of modularity was presented in the early development of algebraic specification languages, since specifications are used to model abstract data structures.

Institutions. A notion of an *institution* [71], based on category theory, was introduced as the framework for formalizing logic-independent module systems.

Definition 1.1.1. Let $\boldsymbol{Set}$ be the category of sets and $\boldsymbol{Cat}$ the category of small categories. An institution consists of:

(1) a category $\boldsymbol{Sig}$ of signatures,

(2) a functor $\boldsymbol{Sen} : \boldsymbol{Sig} \to \boldsymbol{Set}$ giving, for each signature Σ, the set of sentences $\boldsymbol{Sen}(\Sigma)$, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the sentence translation map $\boldsymbol{Sen}(\sigma) : \boldsymbol{Sen}(\Sigma) \to \boldsymbol{Sen}(\Sigma')$,

(3) a functor $\boldsymbol{Mod} : \boldsymbol{Sig} \to \boldsymbol{Cat}$ giving, for each signature Σ, the category of models $\boldsymbol{Mod}(\Sigma)$, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct (forgetful) functor $\boldsymbol{Mod}(\sigma) : \boldsymbol{Mod}(\Sigma') \to \boldsymbol{Mod}(\Sigma)$,[2]

(4) for each $\Sigma \in \boldsymbol{Sig}$, a satisfaction relation $\models_\Sigma$, such that for each $\sigma : \Sigma \to \Sigma' \in \Sigma$, $M' \in \boldsymbol{Mod}(\Sigma')$ and $\varphi \in \boldsymbol{Sen}(\Sigma)$:

$$M' \models_{\Sigma'} \sigma(\varphi) \text{ if and only if } \boldsymbol{Mod}(\sigma)(M') \models_\Sigma \varphi.$$

[2] Notice the type of $\boldsymbol{Mod}(\sigma)$ is the reverse of that of $\boldsymbol{Sen}(\sigma)$ regarding the positions of Σ and Σ'.

Based on a particular institution, specifications can be defined in two ways. (1) A specification is a pair consisting of a signature Σ and a set $\boldsymbol{E} \subseteq \boldsymbol{Sen}(\Sigma)$ of equations. The *category of syntactical specifications* is the category whose objects are all such pairs $(\Sigma, \boldsymbol{E})$ and whose morphisms $\sigma : (\Sigma, \boldsymbol{E}) \to (\Sigma', \boldsymbol{E}')$ are signature morphisms $\sigma : \Sigma \to \Sigma'$ such that $\boldsymbol{Sen}(\sigma)[\boldsymbol{E}] \subseteq \boldsymbol{E}'$.[3] (2) A specification is a pair consisting of a signature Σ and a set $\boldsymbol{M} \subseteq \boldsymbol{Mod}(\Sigma)$ of models. The *category of semantics specifications* is a category whose objects are all such pairs $(\Sigma, \boldsymbol{M})$ and whose morphisms $\sigma : (\Sigma, \boldsymbol{M}) \to (\Sigma', \boldsymbol{M}')$ are signature morphisms $\sigma : \Sigma \to \Sigma'$ such that $\boldsymbol{Mod}(\sigma)[\boldsymbol{M}'] \subseteq \boldsymbol{M}$ [71].[4]

Specification building operations. Most algebraic specification languages support a set of specification building operations, such as extension, union, renaming, and first-order parameterized specification. Two approaches to define the semantics of these operations are based on the category of syntactical specifications and the category of semantics specifications respectively. (1) The operations do not add any expressive power to the underlying system, in the sense that a structured specification is equivalent to a large unstructured specification. The semantics is built at the syntactical level, i.e. over the category of syntactical specifications. However, there is nothing preventing one from defining the semantics of the operations over the category of semantics specifications. Some systems define both of them and show their compatibility, e.g. the specification language Casl [48] does this. (2) Other systems let extra these operations provide extra power, i.e. a specification built using these operations may not be equivalent to any unstructured specification. Their semantics is defined at the model level, since some set of models may not have any corresponding syntactical descriptions. Thus it is defined over the category of semantics specifications, for instance, in the specification language ASL [97].

Parameterized specifications. As functors in the ML-family module systems, parameterized specifications are the most important operation. There is no separate notion of interface in that specifications serve the role of interfaces as well. Only first-order parameterized specifications are supported in most algebraic specification languages. A parameterized specification is defined as an extension of its formal parameter specification. In other words, there is exactly one occurrence of formal parameter specification within the body definition of a parameterized specification,

[3] $\boldsymbol{Sen}(\sigma)[\boldsymbol{E}] = \{\boldsymbol{Sen}(\sigma)(\varphi) \mid \varphi \in \boldsymbol{E}\}$.

[4] $\boldsymbol{Mod}(\sigma)[\boldsymbol{M}'] = \{\boldsymbol{Mod}(\sigma)(M') \mid M' \in \boldsymbol{M}'\}$.

which is not adequate as indicated in §1.1.1.

Instantiation and parameter passing. To instantiate a parameterized specification, we need to provide an actual parameter specification. A matching between a formal parameter specification and an actual parameter specification is specified by a *fitting morphism* between them. Fitting morphisms are essentially theory interpretations, if we treat specifications as theories. Fitting morphisms are more flexible than the parameter passing mechanism of ML-family functor. In particular, the language of a specification does not matter, while the structure of a specification is crucial.

The semantics of an instantiation of a parameterized specification with a proper actual parameter and fitting morphism is defined as the *pushout* of the diagram consisting of the formal parameter specification, the actual parameter specification, the parameterized specification, the fitting morphism, and the embedding morphism between the formal parameter specification and the parameterized specification [71]. (See Figure 1.2.) The formal parameter specification identifies the shared portion of the actual parameter specification and the parameterized specification.

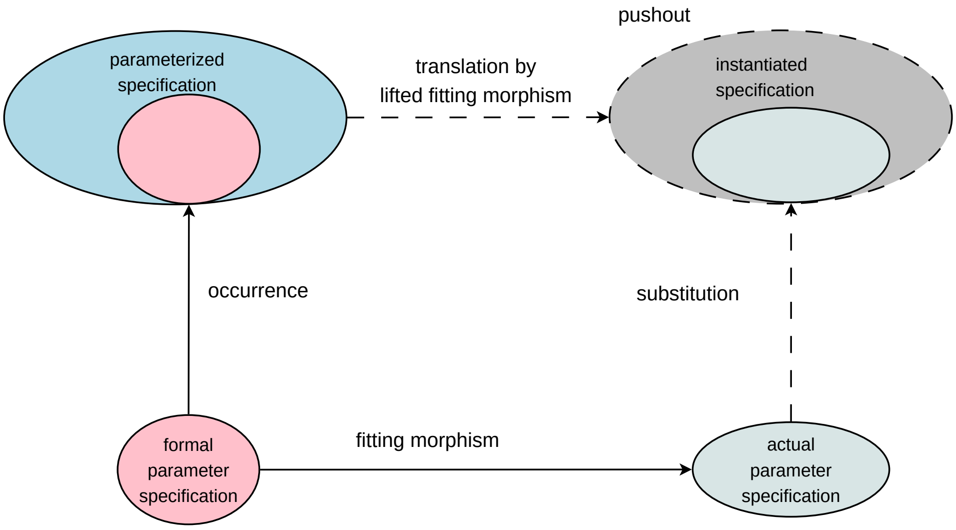
pushout
parameterized specification
translation by lifted fitting morphism
instantiated specification
occurrence
substitution
formal parameter specification
fitting morphism
actual parameter specification

Figure 1.2: Parameterized Specification Application

The pushout is unique up to isomorphism; it specifies the structural constraints of a specification. In other words, given a parameterized specification and an actual parameter specification, the instantiation may have a totally different signature from the signatures of the parameterized specification and the actual parameter specification, as long as the instantiation satisfies the structural constraints. This uncertainty of the resulting signature of an instantiated specification may not be a problem for most algebraic specification systems since they support only first-order parameterized specifications. It is even an advantage since it allows an implementation to choose a more meaningful signature for the instantiated specification following then that of the actual parameter specification. Figure 1.2 illustrates this reasonable implementation.

This uncertainty may, however, make higher-order parameterized specifications impractical. Assume that we are defining a specification by extending another specification, which is defined via a sequence of (higher-order) parameterized specification instantiations. In order to decide the symbols we can use to define the extension specification, it is necessary to evaluate the sequence of parameterized specification instantiations. While it is not a big job for first-order parameterized specification instantiations, it could be a huge effort for higher-order parameterized specification instantiations. Also, note that this is not a problem for the ML-family system, since types help users to figure out the set of symbols that can be used.

It is also not obvious if it is possible to define the semantics of higher-order parameterized specifications and instantiations purely within category theory. At least, the category of specifications is not enough, since there are no representations of higher-order parameterized specifications in that category.

1.2 Design goals

In this section, we present the major design goals we want to achieve. Some of them are addressed in §1.1 and are presented briefly here, while others are discussed in detail.

Design Goal 1 (DG1). *The module system should be independent of the language and logic of the underlying MMS.*

Although most languages and MMSs have their own module systems, we believe a module system should be independent of languages and logics. We want our module system to be orthogonal to the underlying MMS, i.e. it should be a stratified layer on top of the underlying MMS. Thus a module cannot be a first class object. However, we need to emphasize that there is nothing to prevent users from adopting ideas presented in this thesis to build a module system where modules are first class objects.

Design Goal 2 (DG2). *Mathematical knowledge is organized as theories.*

In our module systems, the basic building block is a *theory.* A theory[5] consists of a language L, a set Φ of axioms, and a set Δ of theorems that are proved from the axioms in a sound proof system. (Refer to §2.1.1 for the formal definitions.)

Note that a theory contains only those theorems that are already proved, in contrast to approaches (as in most algebraic specification languages) where a theory contains all theorems that are provable. In other words, each theory is a concrete representation of some mathematical theory. In order to extend a theory with a theorem, it is necessary to provide a proof together with the sentence that is proved.

Design Goal 3 (DG3). *The theories of the module system are organized according to the little theories method.*

Our notion of theory is based on the *little theories* approach [38], in contrast to the modules in most programming language paradigms which are based on the *big theory* approach [38]. In the big theory approach, one powerful set of axioms is used to model all objects of interest. Consequently, a module is a name scope mechanism, where an object developed in one module can be referred to from within another module by qualifying the object name. In the little theories approach, reasoning is carried out under different, separate contexts, modeled by theories. To use an object developed in another theory under the current context, we need to build a theory interpretation between them. Effectively, it will import a translated version of that theory implicitly or explicitly.

Design Goal 4 (DG4). *There are theory building operations that construct new theories from existing theories.*

Although we can always build a theory from scratch, it is better if we can reuse previously developed theories. Our system should support the usual theory building operations as supported in most algebraic specification systems, i.e. theory extension, union, and renaming. In particular, renaming helps users to express a theory in a different language in a simple way.

Design Goal 5 (DG5). *There is support for parameterized modules, called functors.*

The most important modular mechanism supporting reuse might be parameterized modules such as the functors of ML-family module systems and parameterized specifications of algebraic specification languages. Our module system should support parameterized modules, called *functors*, which are functions over theories.

[5] Our notion of a theory may be called a theory presentation in some other systems.

Design Goal 6 (DG6). *Functors can be higher-order.*

While first-order functors, where parameters may be theories but not other functors, are a very useful parameterized mechanism, higher-order functors will provide higher level abstractions. As shown in §1.1.1, higher-order functors are a generalization of first-order functors that manipulate functors as well as theories as values. A module system with higher-order functors should resemble the λ-calculus, in which theories are values and functors are functions. Higher-order functors are then handled in a uniform way as in the λ-calculus.

Design Goal 7 (DG7). *Functors can be defined in terms of arbitrary module expressions over the parameter theories.*

In most systems, a first-order parameterized specification is defined as an extension of its parameter specification. A generalization of that would be allowing a first-order functor to be defined as an arbitrary expression over the parameter theory as in the ML-family module systems. For examples, there may be no occurrence of the parameter theory in the expression or there may be two or more copies of the parameter theory in the expression. Module expressions can express more complicated theory constructions, e.g. functor application and renaming, because our systems support operations other than extension. This makes the functors behave more like functions.

Design Goal 8 (DG8). *Functors are applicative.*

As discussed in §1.1.1, our systems support applicative functors only.

Design Goal 9 (DG9). *There is a type system to classify module expressions by their values, which are theories or functors.*

A notion of a *type* of a theory is used to specify a class of theories. For instance, the class of theories that can be used to instantiate a functor is specified by a type. A type of a theory is also used to specify an interface of the theory, i.e. the exported language and sentences. Casting the type of a theory effectively changes its interface. In case *higher-order functors* are supported, a notion of *functor type* is used to specify a class of expressions, whose values are functors.

Design Goal 10 (DG10). *There is a subtyping mechanism that allows users to treat an object of one type (subtype) as an object of another type (supertype).*

By using a type system to classify module expressions by value, we are forced to define distinct functors with similar behaviour over different types. We thus need a way to define *polymorphic* functors. This can be handled by a subtyping relation in the sense that a functor defined over a type is applicable to any of its subtypes. In general, a subtype is more informative than its supertype. Thus, it is safe to use an object in the subtype in the places where an object of the supertype is required. Note that, because there is no state, subtyping does *not* involve all the problems associated with inheritance.

Design Goal 11 (DG11). *An actual parameter is passed to a functor via a mechanism similar to a fitting morphism.*

During a functor application, in order to pass an actual parameter to the functor, we need to match the actual parameter with the formal parameter. As shown in §1.1, one way is type matching, another way is providing a fitting morphism. While the former way is very simple, the latter way provides greater flexibility, in the sense that a theory expressed in a different language from the formal parameter can be used for instantiation. It is also desirable to generalize fitting morphisms so that a theory that is formalized in a different axiomatization can be used as a valid actual parameter.

1.3 Overview

In this thesis we present a hierarchy of module systems. A child system extends its parent system with additional mechanisms. In this section, we briefly describe these systems and state their major contributions to give readers the big picture.

Mei Basic. Mei Basic is the simplest system in the hierarchy. It satisfies all the design goals except **DG10** and **DG11**.

Mei Basic organizes mathematical knowledge as theories. It supports several operations to construct theories from those existing theories, including extension, renaming, and union. Mei Basic supports parameterized theories, called *functors*, which are applicative and can be higher-order. A notion of a *type* of a theory or a functor is used to specify a class of theories or functors. A first-order functor is defined in term of an arbitrary expression of its formal parameter. Basically, Mei Basic resembles an ML-family module system modeled on the typed λ-calculus.

Mei Core. Extending Mei Basic by a subtyping relation over (parameterized) theories, we get Mei Core, which satisfies **DG10** as well as the design goals that Mei Basic satisfies. For example, let *Group* be a theory of groups and be of type Group. Let Monoid be the type of theories of monoids. Let F be a functor whose formal argument is type Monoid. It is thus reasonable to allow the application of F to *Group*, since Group is a subtype of Monoid. This idea is generalized to functor types as well.

Mei. Mei[6] is Mei Core plus a powerful coercion mechanism. A notion of a *view* is supported in Mei. It is the smallest system satisfying all 11 design goals. A view shows how an object in one (target) type can be treated as an object in another (source) type in a more general way than that of a subtyping relation. The simplest views between theory types are similar to fitting morphisms (**DG11**). Views between functor types are defined in terms of views between their source and target types inductively.

Intuitively, a view shows how an object in its target type can be treated as an object in its source type by providing the information expressed by the source type but missed by the target type. The semantics of a view is then a functor in Mei Core, called a coercion functor, that adds the missing information to any objects in the target type. Effectively, a coercion functor generated from a view transfers an object in the target type to an object in the source type. By doing so systematically, expressions in Mei are transferred to those of Mei Core. Thus, the semantics of Mei is defined in terms of that of Mei Core.

Partial work on Mei was presented in Programming Languages for Mechanized Mathematics Workshop (PLMMW) [98].

DMei Core and DMei. In Mei, the result type of a functor type is fixed in the sense that it is independent of the argument type. The type of a functor application is then fixed no matter what parameter is used. In other words, some type information of the actual parameters is lost. There are cases when users might want to keep (or get back) the type information of actual parameters. This necessarily requires that the result type of a functor type is defined in terms of its argument type. Then, we can type functor applications by replacing each occurrence of the argument type in the result type by the actual type of the parameter. These are so-called dependent functor types.

[6] Mei is the Chinese name of *Prunus mume*, a species of Asian plum in the family *Rosaceae*. The tree flowers in late winter, typically late January or February in East Asia.

Generalizing Mei Core to support dependent functor typing, we get DMei Core. DMei Core gives more precise typing information for functor applications, though it necessarily complicates the typing system. Accordingly, extending DMei Core by the coercion mechanism, we get DMei.

1.4 Major contributions

The design goals we presented in §1.2 are not new. They are supported in various module systems. However, none of the current module systems supports all these goals. For instance, Leroy's manifest types is an ML-family module system [57], which is designed to be language independent (**DG1**). Signatures are types of structures and functors (**DG10**), and subtyping is supported (**DG11**). It has applicative higher-order functors (**DG5,6,7,8**). However, **DG4** and **DG9** are not fulfilled. It does not support module renaming, though other module building operations, such as extension and union, may be simulated by functors. More importantly, parameter passing is based on type matching, which is not as flexible. **DG2** and **DG3** are not necessary for a module system of programming languages. Another approach is to formalize modules as dependent records in type theory [62]. This approach is similar to the first one with the advantage that modules are first class objects, which is not a design goal. However, this formalism is not logic independent (**DG1**) because the underlying logic has to be powerful enough to formalize dependent records.

Most module systems of algebraic specification languages are logic independent (**DG1**), with the notion of a logic formalized as an institution. Normally, they follow the little theories approach (**DG3**), though usually not specified by the system. The specification building operations (**DG4**) are supported, as well as parameterized specifications (**DG5,8**). Specifications (**DG2**) are used as both objects and types (**DG10,11**), though there is not an explicit type system. Fitting morphisms are used for parameter passing (**DG9**). However, higher-order functors are usually not supported (**DG6**). In addition, parameterized specifications are defined as an extension of their formal parameter, not an arbitrary expression (**DG7**).

Our major achievement is to design module systems that fulfil all the design goals stated in §1.2: Mei and DMei. In particular, an ideal module system, in our opinion, is a system that integrates higher-order functors (**DG6**) and a parameter passing mechanism using fitting morphisms (**DG9**). Although there are a few unsuccessful efforts (we will investigate one incomplete proposal in Chapter 5), our systems are the first ones in which these mechanisms are fully integrated. This is achieved by a novel notion of a *view* and its coercion semantics. A view is a generalization of a fitting morphism which accounts for functors as well as theories. Instead of defining the semantics of functor application with views directly, we define the semantics of a view as a coercion functor, which transfers an actual parameter justified by the view to a new actual parameter (which should be equivalent to the original actual parameter in some sense) that can be justified by type matching.

Although Mei satisfies all the design goals, it is relatively simple. Moreover, there is a strong, rather beautiful analogy between Mei and the typed λ-calculus. It is easy to put Mei (in fact Mei Basic) into the typed λ-calculus frame as follows. The theory types and the functor types are the base types and function types respectively. Module expressions are terms, where functor abstractions (definitions) are λ-abstractions, and functor applications (instantiations) are function applications, whose semantics is given by β-reduction. The theory operators are analogous to extra term constructors, like the arithmetic operators, in many functional languages.

The rest of this thesis is organized as follows. Chapter 2 presents Mei Basic. Chapter 3 gives two extensions of Mei Core and Mei. Chapter 4 presents DMei Core and DMei. In Chapter 5, we show the power of a simple system by comparing Mei with the module systems of some specification languages and MMSs. Some possible extensions of Mei are given in Chapter 6. A simple implementation of Mei is presented in Chapter 7. Chapter 8 presents a skeleton of a module system designed for multi-logic MMS systems. The thesis ends with a short conclusion in Chapter 9.

Chapter 2

A simple module system – Mei Basic

In this section we present a simple ML-family module system, Mei Basic. We first informally present the features of Mei Basic, followed by a formal presentation of its syntax and semantics.

2.1 Informal presentation

As indicated in §1.3, Mei Basic is a module system enjoying the beauty of the typed λ-calculus. Mathematical knowledge is organized as modules called *theories*. Theories are objects and parameterized theories are functions. We will present the mechanisms supported by Mei Basic in this section.

2.1.1 Theories

Although there are different ways to present mathematical theories, we prefer the axiomatization approach. A theory in Mei Basic consists of a language (a set of symbols used by the theory), a set of axioms, and a set of theorems provable from the axioms in a sound proof system. In other words, a theory in Mei Basic is a finite concrete representation of a mathematical theory. In general, the set of axioms of a particular theory could be infinite. However, they must be finitely representable, for example, by a finite set of axiom schemes.

Both the syntax and the semantics of Mei Basic are defined in terms of *theories*. We define *language*, *sentence*, *proof*, and *theory* in a very abstract way and give examples to clarify them.

Definition 2.1.1 (Languages)**.** A *language* L is a set of symbols (often of various categories) that we can use to build syntactic objects. We will write $\mathcal{L}$ for the set of all L.

Definition 2.1.2 (Sentences)**.** The set of all *sentences* over a language $L \in \mathcal{L}$, denoted by $\boldsymbol{Sen}(L)$, is a set of syntactic objects constructed inductively via a set of sentence constructors.

Clearly, $\boldsymbol{Sen}(L_1) \subseteq \boldsymbol{Sen}(L_2)$ if $L_1 \subseteq L_2$. A set Φ of *axioms* is a set of sentences, i.e. $\Phi \subseteq \boldsymbol{Sen}(L)$. We will use $\mathcal{A}_L$ for the set of all such Φ, which is the power set of $\boldsymbol{Sen}(L)$. $\mathcal{A}$ is then the union of $\mathcal{A}_L$ for all $L \in \mathcal{L}$.

Definition 2.1.3 (Proofs)**.** Each MMS supports a set of proof strategies to derive sentences from sentences using certain (either predefined or user defined) inference rules. A sentence φ is *provable* from a set of sentences Φ if it is derivable from Φ. A *proof* of φ is then any derivation of φ from Φ. Given a language L, a set $\Phi \subseteq \boldsymbol{Sen}(L)$ of axioms, and a sentence $\varphi \in \boldsymbol{Sen}(L)$, we denote the set of proofs of φ by $\boldsymbol{Proof}(L, \Phi, \varphi)$.

Again the style of proofs stays unspecified, although we assume that there exists a set of inference rules such that a proof of a formula can be inductively derived following these rules. This necessarily implies that, if $L_1 \subseteq L_2$ and $\Phi_1 \subseteq \Phi_2$, then $\boldsymbol{Proof}(L_1, \Phi_1, \varphi) \subseteq \boldsymbol{Proof}(L_2, \Phi_2, \varphi)$.

Example 2.1.4. A language L for a many-sorted first-order logic is a set of symbols presented as a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{F}, \mathcal{P})$ where

(*a*) $\mathcal{S}$ is a set of *sort symbols* $s_1, s_2, \ldots$.

(*b*) $\mathcal{C}$ is a set of *constant symbols* $c_1, c_2, \ldots$. Each constant symbol is associated with a sort symbol $s \in \mathcal{S}$.

(*c*) $\mathcal{F}$ is a set of *(primitive or basic) function symbols* $\mathsf{f} : s_1 \times \cdots \times s_m \to s$ $(m \geq 1)$. Each symbol f has a *type* $s_1 \times \cdots \times s_m \to s$, where $m \geq 1$ is the *arity* of f.

(*d*) $\mathcal{P}$ is a set of *(primitive or basic) predicate symbols* $\mathsf{p} : s_1 \times \cdots \times s_m$ $(m \geq 1)$. Each symbol p has a *type* $s_1 \times \cdots \times s_m$, where $m \geq 1$ is the *arity* of p.

In this case, there are four categories of symbols, namely sort symbols, constant symbols, function symbols, and predicate symbols. We can define terms and formulas in the usual way. A sentence is then a closed formula, i.e., a formula in which no free variable occurs. An example of a proof system is a Hilbert-style proof system. A proof of a sentence is a derivation of the sentence starting from the axioms (logical and non-logical axioms) and then applying the inference rules (such as modus ponens).

Definition 2.1.5 (Specification environment). A *specification environment* θ is a pair (L_θ, Φ_θ) where:

(*a*) $L_\theta \in \mathcal{L}$ is a language.

(*b*) $\Phi_\theta \subseteq \boldsymbol{Sen}(L_\theta)$ is a set of axioms.

A specification environment records the symbols and axioms currently visible. It grows when a symbol or an axiom is declared.

Definition 2.1.6 (Specifications). Let $\theta = (L_\theta, \Phi_\theta)$ be a specification environment. A *specification* S under θ is a 4-tuple $(L, \Phi, \Delta, \theta)$ where:

(*a*) L is a set of symbols such that $L \cap L_\theta = \emptyset$ (the empty set) and $L \cup L_\theta \in \mathcal{L}$ is a language .

(*b*) $\Phi \subseteq \boldsymbol{Sen}(L \cup L_\theta)$ is a set of axioms where $\Phi \cap \Phi_\theta = \emptyset$.

(*c*) Δ is a set of theorems, i.e. pairs $\langle \varphi, \mathsf{pf} \rangle$, where $\varphi \in \boldsymbol{Sen}(L \cup L_\theta)$ is the sentence of the theorem and $\mathsf{pf} \in \boldsymbol{Proof}(L \cup L_\theta, \Phi \cup \Phi_\theta, \varphi)$ is its proof.

S is called *closed* if the symbols used in Φ and Δ are in L and the axioms used in the proofs of Δ are in Φ, i.e. θ is empty.

A specification is always defined under some specification environment.

Definition 2.1.7 (Theories). A *theory* T is a closed specification, abbreviated as (L, Φ, Δ).

We will write $\mathcal{TH}$ for the class of all theories, $\mathbf{sen}(\Delta) = \{\varphi \mid \langle \varphi, \mathsf{pf} \rangle \in \Delta\}$ for the set of all sentences in Δ, and $\mathbf{lang}(T)$ for the language L of the theory T.

Remark 2.1.8. Note that a theory in Mei Basic contains only those theorems that are already proved, in contrast to approaches where a theory contains all the theorems that are provable. A theory in Mei Basic may be called a theory representation in those approaches. However, our notion of a theory is widely adopted in many theorem proving systems. Consequently a theory in Mei Basic is a syntactic object in terms of the underlying MMS. In other words, our system manipulates only syntactic representations of mathematical theories.

Remark 2.1.9. As indicated in §1.2, a theory is a closed reasoning context. An object defined in one theory cannot be referred to from within another theory. There are no global objects. In other words, an object is always *local* to a theory, i.e. it is developed and employed within a theory. As a result, the meaning of a theory is totally determined by its definition, not something outside the theory.

Remark 2.1.10. In terms of implementation, languages, sentences, and proofs are abstract types whose concrete representations are determined by the underlying MMS. The type for theories is then defined in terms of the abstract types for languages, sentences, and proof.

Remark 2.1.11. Note that we make very few assumptions about the underlying logic. The underlying logic will be specified by how a sentence is built on top of a language which is not part of our module system. For instance, we do not assume that the underlying logic is classical or constructive. However, our module system does work better for some particular logics. For instance, although our module system can be built on top of a logic without the CRI property, it will possess the Modularization Property only if the underlying logic possesses the CRI property (see §2.5 for details).

2.1.2 Theory extension, renaming, and union

Mei Basic supports several operations to construct new theories from existing theories. As in most algebraic specification languages, Mei Basic supports theory extension, renaming, and union.

Extension. Extending an existing theory by adding new symbols is the simplest way to form a structured theory hierarchy. To develop a new theory, instead of starting from scratch, we can start from an existing theory and extend it by adding new language symbols and axioms. For instance, to build a theory of groups, we might rather start by inheriting the language and axioms of a theory of monoids.

Renaming. There are two basic reasons that we need a *renaming* mechanism: (1) to avoid name conflict for use in the union operation below, and (2) to name symbols in a meaningful way according to the intended semantics. For instance a theory of rings may be built from a theory of groups and a theory of monoids. It is quite possible that both theories use $\circ$ as the name of their binary operators, therefore, one of them has to be renamed. In another example, one may use "mm" as the name of the sort in a theory of monoids, representing the carrier set of a monoid. A theory of groups can be defined by extending the theory of monoids, and it is natural to rename "mm" to "gg", representing the carrier set of a group.

Example 2.1.12. Let $L = (\mathcal{S}, \mathcal{C}, \mathcal{F}, \mathcal{P})$ be the language of the many-sorted first-order logic defined in 2.1.4. A renaming of L is a pair (L, ρ), where $\rho : L \to L'$ is a symbol mapping consisting of four one-to-one functions ρ_s, ρ_c, ρ_f, ρ_p:

(1) ρ_s maps each sort symbol of L to another sort symbol.

(2) ρ_c [ρ_f, ρ_p, respectively] maps each constant symbol [function symbol, predicate symbol] in L to a constant symbol [function symbol, predicate symbol] respecting its type (and arity).

This forms a new language L', written $\rho(L) = (\rho_s(\mathcal{S}), \rho_c(\mathcal{C}), \rho_f(\mathcal{F}), \rho_p(\mathcal{P}))$.

ρ is a mapping from L to its image language $\rho(L)$. We can define, inductively over the structure of sentences, the sentence translation ρ_{sen}, the mapping from $\boldsymbol{Sen}(L)$ to $\boldsymbol{Sen}(\rho(L))$, which maps each sentence $\varphi \in \boldsymbol{Sen}(L)$ to a sentence $\rho_{sen}(\varphi) \in \boldsymbol{Sen}(\rho(L))$ (refer to the example in §3.4.1 for details). Similarly we can define inductively the proof translation ρ_{pf}, the mapping from $\boldsymbol{Proof}(L, \Phi, \varphi)$ to $\boldsymbol{Proof}(\rho(L), \rho_{sen}(\Phi), \rho_{sen}(\varphi))$, where $\Phi \subseteq \boldsymbol{Sen}(L), \varphi \in \boldsymbol{Sen}(L)$, and $\rho_{sen}(\Phi) = \{\rho_{sen}(\psi) \mid \psi \in \Phi\}$. A theory translation ρ_{thy} of a theory $T = (L, \Phi, \Delta)$ is a triple $(\rho, \rho_{sen}, \rho_{pf})$, which maps T to its image theory, written $\rho_{thy}(T) = (\rho(L), \rho_{sen}(\Phi), \rho_{thm}(\Delta))$, where $\rho_{thm}(\Delta) = \{\langle \rho_{sen}(\varphi), \rho_{pf}(\mathsf{pf})\rangle \mid \langle \varphi, \mathsf{pf}\rangle \in \Delta\}$. When there is no ambiguity, we may drop all subscripts.

Note that a renaming is just a special case of a *translation* defined in §3.4.1, where ρ is one-to-one and the target language is unspecified.

Union. To build a theory based on two or more existing theories, we need the *union* operation. For instance, a natural way to build a theory of rings is to extend the combination of a theory of groups and a theory of monoids.

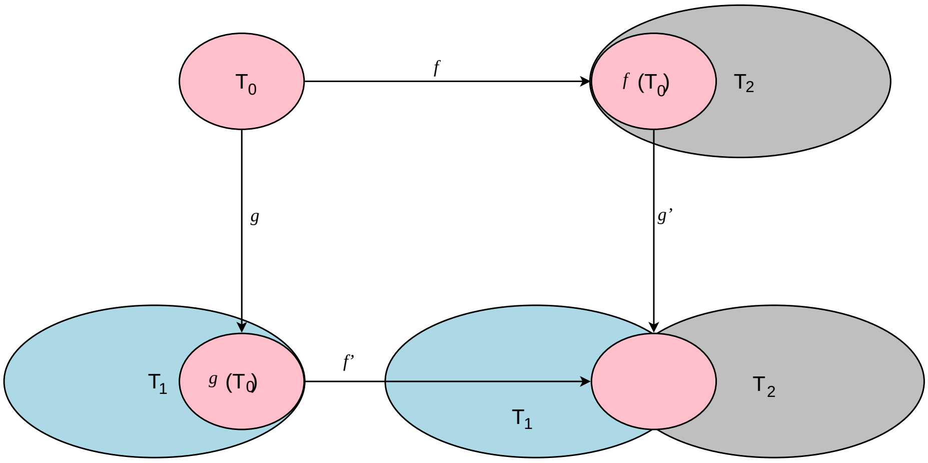

Figure 2.1: Amalgamated union

There are two approaches we can follow to construct a theory from existing theories, set-theoretic union and amalgamated union. Figure 2.1 illustrates the idea of amalgamated union. Let $T_0 = (L_0, \Phi_0, \Delta_0)$, $T_1 = (L_1, \Phi_1, \Delta_1)$, and $T_2 = (L_2, \Phi_2, \Delta_2)$ be theories. The amalgamated union of T_1 and T_2 with respect to T_0 is the disjoint union of T_1 and T_2 modulo the equivalence relation induced by f and g. Note that the images of T_0 in T_1 and T_2 may have totally different syntactic representations, though they must share the same structure. In other words, T_0 identifies the common parts of T_1 and T_2 via f and g, and thus, they will be mapped to the same images by f' and g'. Information that is not identified by T_0 must be distinguished in the amalgamated union regardless of its original syntactic representation in T_1 and T_2. For example, assume that both T_1 and T_2 have a symbol a, which is not in $f(T_0)$ or $g(T_0)$, then a has to be mapped to two distinct symbols, say a_1 and a_2, in the amalgamated union by f' and g'. This is a so-called pushout in the category theory, where theories are objects and theory morphisms are morphisms. Clearly, the amalgamated union approach is more general than the set union approach. In the simplest case, $L_0 = L_1 \cap L_2$ and the amalgamated union of T_1 and T_2 w.r.t. T_0 is equivalent to the set-theoretic union $T_3 = (L_1 \cup L_2, \Phi_1 \cup \Phi_2, \Delta_1 \cup \Delta_2)$.

However, it seems that, with the help of renaming, we can simulate amalgamated union by the set-theoretic union. By using the renaming mechanism, we can force the common parts to be syntactically identical and the other parts to be syntactically distinct. In fact, we can regard the renamings as an explicit representation of the morphisms f and g. This idea is illustrated in the following example:

Example 2.1.13.

$$\begin{array}{lll}
\mathit{Monoid} & \equiv & \texttt{language}\ \texttt{sort}\ mm \\
 & & \qquad\qquad \texttt{const}\ e : mm \\
 & & \qquad\qquad \texttt{func}\ \ \circ : mm^2 \rightarrow mm \\
 & & \texttt{axioms}\ \ \forall x_1, x_2, x_3 : mm.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\
 & & \qquad\qquad \forall x : mm.\ x \circ e = x \wedge e \circ x = x \\
 & & \texttt{theorems}\ \ \ldots
\end{array}$$

$$\begin{array}{lll}
\mathit{Group} & \equiv & \mathit{Monoid}\ \texttt{with}\ \{mm \mapsto gg\}\ \texttt{extended by} \\
 & & \texttt{language}\ \texttt{func}\ \ \ ^{-1} : gg \rightarrow gg \\
 & & \texttt{axioms}\ \ \forall x : gg.\ x \circ x^{-1} = e \\
 & & \qquad\quad\ \ \forall x : gg.\ x^{-1} \circ x = e \\
 & & \texttt{theorems}\ \ \ldots
\end{array}$$

$$\begin{array}{lll}
\mathit{Ring} & \equiv & (\mathit{Group}\ \texttt{with}\ \{gg \mapsto rr,\ e \mapsto 0, \circ \mapsto +\} \oplus \\
 & & \mathit{Monoid}\ \texttt{with}\ \{mm \mapsto rr,\ e \mapsto 1, \circ \mapsto *\})\ \texttt{extended by} \\
 & & \texttt{language} \\
 & & \texttt{axioms}\ \ \forall x, y, z : rr.\ x * (y + z) = (x * y) + (x * z)\ \ \ldots \\
 & & \texttt{theorems}\ \ \ldots
\end{array}$$

By renaming, we identify the sorts mm and gg as rr, separate e as 0 and 1, and separate $\circ$ as $+$ and $*$. Although we do not define the syntax of the underlying core language and symbol mapping, the meaning of the above example should be clear.

There is still the danger that unintended name collapses may happen when the union operation is embedded in a parameterized theory body (see §2.1.3), e.g. the actual parameter theory and the parameterized theory body may share some symbols that are not supposed to be identical. The solution is to use types (see §2.1.3) to identify the amalgamated part of the two theories, i.e. T_0 in Figure 2.1. The main idea is: (1) the union of two types is the set-theoretic-union of them, (2) two symbols are considered identical if they are identical in the type specification, and (3) two symbols are considered distinct otherwise (and therefore have to be renamed). Our approach is a variant of the amalgamated union where the amalgamated part T_0 is identical with its images in both T_1 and T_2. The formal definition of the union operation is in §2.4.2. Also note that this is just a side-use of types. Since the types are not only used for specifying the amalgamated part of theories, they contain more information as defined in §2.1.3.

Remark 2.1.14. Name conflict is always an issue for large system development. There are basically two approaches to solve this issue: (1) Names from different modules are distinct. Sharing of names is expressed by equational constraints [57]. (2) Names from different modules are not distinct. Sharing is expressed by names [3]. Renaming is used to solve unintended name conflict. A side-effect of the little theories approach is that a theory is a simple name scope which isolates the names in one theory from those in other theories. However, name spaces can not be nested. Since we have a renaming mechanism, we prefer approach (2) within a theory. When a theory is constructed from two or more component theories, we adopt the convention that the same name refers to the same object regardless where it comes from. In the above example, we give the same name to the carrier sets of the group theory and the monoid theory to force them to be identical in their union. The renaming mechanism then can be used to distinguish their respective binary operators as the addition and the multiplication operators in the ring theory[1].

2.1.3 Parameterized theories

Parametrization is a key concept of a module system for structuring and reusability of modules. A parameterized theory is a generic theory with respect to its formal parameter theory (the argument type). It can be instantiated provided that the actual parameter theory matches the argument type. We use the notion *functor* for a parameterized theory. It is illustrated best by the following example. Note that the language of the underlying MMS we use in the following examples is not defined in this thesis. We hope the meaning is clear from the context.

Example 2.1.15. A functor *Comm* may be defined as follows to add the commutative

[1] There is a question here: "Does the amalgamated union exist?" In terms of category theory: "Do pushouts exist?" Pushouts may not always exist when we require the morphisms preserving some properties [49]. However, since a renaming gives an isomorphism that will preserve almost all properties, it is reasonable to assume the the existence of the pushout corresponding to the above example. For example, it exists for the order-sorted algebraic specifications as shown in Theorem 1 in [49].

axiom to any theory with a binary operator.

$$\begin{array}{lll} \mathit{Comm} & \equiv & \texttt{functor}\ \mathsf{X}\ : \\ & & \texttt{language}\ \texttt{sort}\ \mathit{ele} \\ & & \qquad\qquad \texttt{func}\ \circ : \mathit{ele}^2 \to \mathit{ele}. \\ & & \mathsf{X}\ \texttt{extended by} \\ & & \texttt{axioms}\quad \forall x, y : \mathit{ele}.\ x \circ y = y \circ x \\ & & \texttt{theorems}\quad \ldots \end{array}$$

Let *Monoid* be a monoid theory defined as follows:

$$\begin{array}{lll} \mathit{Monoid} & \equiv & \texttt{language}\ \texttt{sort}\ \mathit{ele} \\ & & \qquad\qquad \texttt{const}\ e : \mathit{ele} \\ & & \qquad\qquad \texttt{func}\ \ \circ : \mathit{ele}^2 \to \mathit{ele} \\ & & \texttt{axioms}\ \ \forall x_1, x_2, x_3 : \mathit{ele}.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\ & & \qquad\quad\ \ \forall x : \mathit{ele}.\ x \circ e = x \wedge e \circ x = x \\ & & \texttt{theorems}\ \ \ldots \end{array}$$

The application of *Comm* to the actual parameter theory *Monoid* results in the commutative monoid theory by substituting *Monoid* for X. It is equivalent to the following flat theory:

$$\begin{array}{lll} \mathit{CommMonoid} & \equiv & \texttt{language}\ \texttt{sort}\ \mathit{ele} \\ & & \qquad\qquad \texttt{const}\ e : \mathit{ele} \\ & & \qquad\qquad \texttt{func}\ \ \circ : \mathit{ele}^2 \to \mathit{ele} \\ & & \texttt{axioms}\ \ \forall x_1, x_2, x_3 : \mathit{ele}.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\ & & \qquad\quad\ \ \forall x : \mathit{ele}.\ x \circ e = x \wedge e \circ x = x \\ & & \qquad\quad\ \ \forall x, y : \mathit{ele}.\ x \circ y = y \circ x \\ & & \texttt{theorems}\ \ \ldots \end{array}$$

Figure 2.2 illustrates the instantiation of a first-order functor. It is similar to the ML-family functor application as illustrated in Figure 1.1, except that, with the help of theory operations described in §2.1.2, more than one occurrence of the formal parameter theory can be in the functor body. It is also similar to the instantiation of a parameterized specification as illustrated in Figure 1.2, except that the parameter passing mechanism is not as general, i.e. type matching is used instead of a fitting morphism.

Remark 2.1.16. In the case when there is an axiom declared for X, the parameter theory of *Comm*, any reference to the axiom in a proof of a theorem of *Comm* will be replaced by the corresponding axiom or theorem in *Monoid*. This correspondence is expressed by the type matching as shown later.

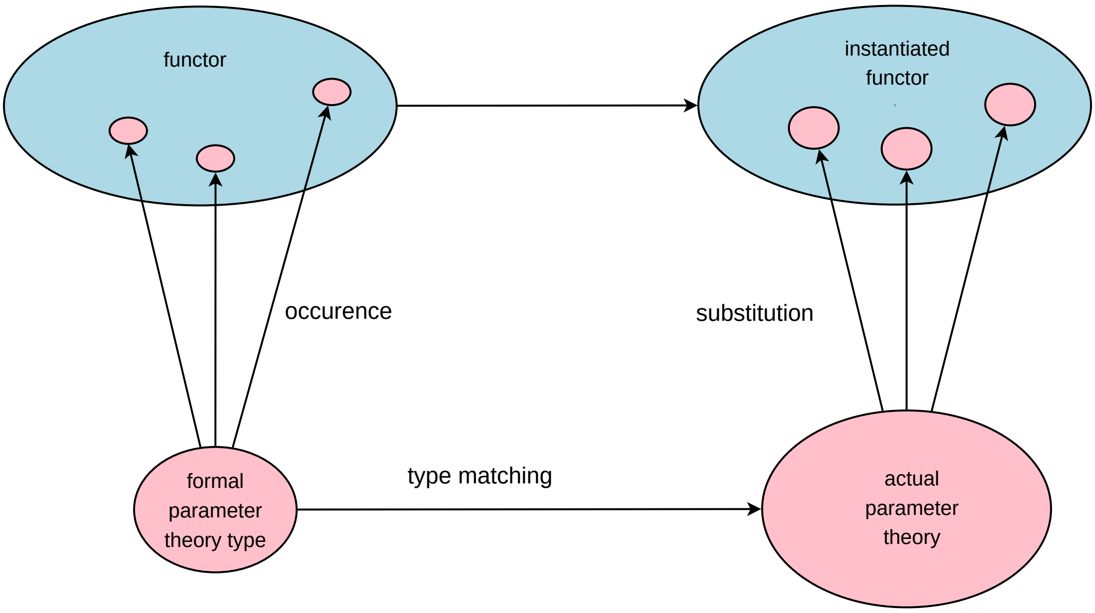

Figure 2.2: First-order Functor Application

Types of theories. Note that we use a notion of a *type* of a theory to specify the class of theories that can be used to instantiate a functor.

Definition 2.1.17 (Types). Let $\mathit{Thy} = (L, \Phi, \Delta)$ be a theory. A *type* T of Thy is a pair $(L_\mathsf{T}, \Phi_\mathsf{T})$ such that:

(a) $L_\mathsf{T} \subseteq L$.

(b) $\Phi_\mathsf{T} \subseteq (\Phi \cup \mathbf{sen}(\Delta))$.

The extension of the type T is $\mathbf{Ext}(\mathsf{T}) = \{(L, \Phi, \Delta) \mid L_\mathsf{T} \subseteq L \text{ and } \Phi_\mathsf{T} \subseteq (\Phi \cup \mathbf{sen}(\Delta))\}$.

Note that, by definition, a theory can have more than one type. This is natural, since the type of a theory provides an outside view of the theory, which is not unique.

In fact, we use the type of a theory as an interface for that theory which determines what information is exposed to the outside world. However, the entire language and all facts (axioms and theorems) are not necessarily exported. For example, auxiliary theorems may be hidden.

However, a module expression which represents a theory may have only one type, called the *principal* type. The principal type of a module expression is decided by the typing rules. For example, if the theory *CommMonoid* is defined from scratch, its principle type may be

$$\begin{aligned}\mathsf{CommMonoid} \;\equiv\; & \texttt{language}\ \texttt{sort}\ ele\\ & \qquad\quad \texttt{const}\ e : ele\\ & \qquad\quad \texttt{func}\ \circ : ele^2 \to ele\\ & \texttt{axioms}\ \forall x_1, x_2, x_3 : ele.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3\\ & \qquad\quad \forall x : ele.\ x \circ e = x \wedge e \circ x = x\\ & \qquad\quad \forall x, y : ele.\ x \circ y = y \circ x\end{aligned}$$

However, if it is defined from an instantiation of *Comm* as in Example 2.1.15, its principle type will be

$$\begin{aligned}\mathsf{CommMult} \;\equiv\; & \texttt{language}\ \texttt{sort}\ ele\\ & \qquad\quad \texttt{func}\ \circ : ele^2 \to ele\\ & \texttt{axioms}\qquad \forall x, y : ele.\ x \circ y = y \circ x\end{aligned}$$

This is decided by the typing rules defined in §2.3.2.

Higher-order functors. Mei Basic supports higher-order functors, in the sense that functors can be used as parameters as well as return values of other functors. This provides more flexible reusability of theories as illustrated in the following example:

Example 2.1.18. Let Ring be the type of ring theories, AbsPoly be the type of abstract polynomial theories, and Ring → AbsPoly be the type of functors that take any theory that is a ring and return an abstract polynomial theory. *MakePoly*, defined as follows, is a second-order functor that takes a theory of rings and a concrete representation theory and returns a concrete polynomial theory with the indicated representation.

$$\begin{aligned}MakePoly \;\equiv\; & \texttt{functor}\ Coe : \mathsf{Ring}.\\ & \texttt{functor}\ MakeAbsPoly : \mathsf{Ring} \to \mathsf{AbsPoly}.\\ & MakeAbsPoly\ Coe\end{aligned}$$

Let $\mathit{Real} :$ Ring be the theory of real numbers and $\mathit{MakeDensePoly} :$ $\mathsf{Ring} \rightarrow \mathsf{AbsPoly}$ be a functor taking a theory of type Ring that returns a polynomial theory over that theory with the dense representation. A polynomial theory over reals with dense representation is derived by applying *MakePoly* to *Real* and *MakeDensePoly* as follows:

$$\mathit{DenRealPoly} \equiv \mathit{MakePoly}\ \mathit{Real}\ \mathit{MakeDensePoly}$$

The semantics of the instantiation of higher-order functors is based on substitutions as for first-order functors, except that functors can be used in substitutions. As a result, we need a notion of a *type of a functor* to specify the class of functor parameters or the class of resulting functors.

Remark 2.1.19. We use the curried version of higher-order function notation. We assume that functor application associates to the left, i.e. $\mathit{MakePoly}\ \mathit{Real}\ \mathit{MakeDensePoly} \equiv (\mathit{MakePoly}\ \mathit{Real})\ \mathit{MakeDensePoly}$.

Types of functors. Since functors are functions from theories to theories, or more generally functors to functors, we define *functor types* for functors as usual. For example, the functor *Comm* has the functor type $\mathsf{Mult} \rightarrow \mathsf{CommMult}$, where Mult is defined as follows and $\mathsf{CommMult}$ is defined as above.

$$\begin{array}{lll} \mathsf{Mult} & \equiv & \texttt{language}\ \texttt{sort}\ ele \\ & & \qquad\qquad \texttt{func}\ \circ : ele^2 \rightarrow ele \\ & & \texttt{axioms} \end{array}$$

The intuition of this functor type is that the functor *Comm* takes any theory that has a single sort *ele* and a binary operator $\circ$ as input and returns a theory with an additional commutative axiom. This justifies the reason that *CommMonoid* has the type $\mathsf{CommMult}$ if it is derived from the instantiation of *Comm*.

Remarks 2.1.20.

(1) It is straightforward to extend the Mei Basic to support higher-order functors by defining functor types for functors as in typed λ-calculus. The substitution mechanism used for defining the semantics of first-order functor applications can be easily generalized to account for higher-order functors. The simple parameter passing mechanism, type matching, makes the generalization easy. However, there is no obvious way to generalize the pushout semantics to account for

instantiations of higher-order parameterized specifications. This is the main reason that we start our system from an ML-family module system which makes the generalization of higher-order functors easy. The remaining question is how to generalize our system to allow a parameter passing mechanism similar to the fitting morphism that fits our higher-order functors well.

(2) The type Mult → CommMult reveals no information about how a result theory is constructed from a parameter theory. For example, both the functor *Comm* and a functor that defines a theory of CommMult from scratch without actually making use of the parameter are of type Mult → CommMult.

Remark 2.1.21. As shown in §1.4, it is easy to put Mei Basic into the λ-calculus framework. The theory types and functor types are the base types and function types respectively. Module expressions are terms, where functor abstractions are λ-abstractions and functor applications are function applications.

2.2 Syntax

The following is the concrete syntax of Mei Basic. A module expression represents a theory (or functor). It specifies the way a theory (or functor) is constructed. For example, they can be defined from scratch, from the operations over theories such as extension, union, and renaming, and from functor applications.

```
EXPR  ::=  MOD-CONST
       |   TYPE-SPEC THY-SPEC
       |   TYPE-SPEC EXPR
       |   EXPR extended by SPEC
       |   EXPR ⊕ EXPR
       |   EXPR with MAPPING
       |   functor VAR : TYPE. EXPR
       |   EXPR EXPR

TYPE  ::=  TYPE-CONST
       |   TYPE-SPEC
       |   TYPE → TYPE
```

$\texttt{THY-SPEC} ::= (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS})$

$\texttt{TYPE-SPEC} ::= (\texttt{LANG}, \texttt{AXIOMS})$

$\texttt{MOD-CONST} ::= \texttt{IDENTIFIER}$

$\texttt{TYPE-CONST} ::= \texttt{IDENTIFIER}$

$\texttt{VAR} ::= \texttt{IDENTIFIER}$

An object in EXPR is called a module expression, which represents a theory (or functor), denoted by a sans-serif E with subscripts when necessary. It describes the way in which a theory (or functor) is constructed. An object in TYPE is called a module type, which represents a class of theories (or functors), denoted by a sans-serif T.

We assume that a module expression and a module type can be named. Therefore a module identifier [type identifier, respectively] is a module expression [module type]. They act as abbreviations for defined modules or types respectively. In fact we keep an environment of named module expressions and module types, defined as follows, which is used to search for the module expression or module type associated with a module identifier or a type identifier respectively. A module identifier [type identifier, respectively] is then nothing more than a constant that has a module expression [module type] as its value.

Definition 2.2.1 (Environment)**.** An *environment* σ is a set of pairs $(\mathsf{C}, \mathsf{E}/\mathsf{T})$ where (1) C is a unique module identifier in σ, and (2) E (or T) is a module expression (or a module type) associated with the identifier C. We will write $\sigma(\mathsf{C})$ for the module expression E (or module type T) if E (or T) is associated with C in σ.

We leave the syntax of the categories LANG, AXIOMS, THMS, IDENTIFIER, and MAPPING unspecified, since it should be part of the underlying mechanized mathematics system. However, we write **source**(ρ) [**target**(ρ)] for the set of source [target] symbols of a mapping ρ.

Some other name conventions: $\mathsf{S}, \mathsf{S}_1, \ldots$ range over SPEC, $\rho, \rho_1, \ldots$ range over MAPPING, $L, L_1, \ldots$ range over LANG, $\Phi, \Phi_1, \ldots$ range over AXIOMS, $\Delta, \Delta_1, \ldots$ range over THMS.

2.3 Rules

We define the *proper* module expressions and module types via sets of rules.

2.3.1 Rules for types

type(T) is read as "T is a module type". **closed**(L, Φ) asserts that every symbol used in Φ is in L.

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \mathbf{closed}(L, \Phi)}{\mathbf{type}(\mathsf{T})} \qquad \textbf{(THY-TYPE)}$$

$$\frac{\mathbf{type}(\mathsf{T}_1) \quad \mathbf{type}(\mathsf{T}_2)}{\mathbf{type}(\mathsf{T}_1 \rightarrow \mathsf{T}_2)} \qquad \textbf{(FUNC-TYPE)}$$

$\equiv$ means syntactically identical. We have two categories of module types: theory types and functor types, for theory objects and functor objects respectively. Functor types associate to the right, i.e. $\mathsf{T}_1 \rightarrow \mathsf{T}_2 \rightarrow \mathsf{T}_3$ means $\mathsf{T}_1 \rightarrow (\mathsf{T}_2 \rightarrow \mathsf{T}_3)$.

2.3.2 Rules for typing module expressions

$\Gamma \vdash \mathsf{E} : \mathsf{T}$ is read as "E is a module expression of type T under the context Γ". $\Gamma = \{\mathsf{X}_1 : \mathsf{T}_1, \ldots\}$ is a *context* that binds type variables with module types. A context is built from functor abstractions which express the type assumptions of variables. If Γ is empty, we write $\vdash \mathsf{E} : \mathsf{T}$. We will write $\Gamma,\ \mathsf{X} : \mathsf{T}$ for $\Gamma \cup \{\mathsf{X} : \mathsf{T}\}$. We assume that theory variables are distinct, since they can be represented as identifiers internally which are freshly generated for each variable declaration (see §7.2 for details).

We use a sans-serif X for variables and sans-serif C for constants. **closed**(L, Φ, Δ) asserts that every symbol used in Φ is in L and every axiom used in a proof in Δ is in Φ. **map**(ρ) asserts that ρ is a mapping, i.e. a set of symbol pairs, where **source**(ρ) [**target**(ρ)] is the domain [range] of ρ. $\rho(L)$ is the image of L via ρ and $\rho(\Phi)$ is the translation of Φ in which each symbol in L is replaced by its image in $\rho(L)$.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \qquad \textbf{(ASSUMP)}$$

$$\frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \qquad \textbf{(CONST)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad L \subseteq L_{\mathsf{E}} \quad \Phi \subseteq \Phi_{\mathsf{E}}}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \qquad \textbf{(CAST)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(EXT)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \qquad \textbf{(UNION)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : (\rho(L), \rho(\Phi))} \qquad \textbf{(REN)}$$

$$\frac{\Gamma, \mathsf{X} : \mathsf{T}_1 \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \textbf{(ABS)}$$

$$\frac{\Gamma \vdash \mathsf{E}_{\mathsf{f}} : \mathsf{T}_1 \to \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_{\mathsf{p}} : \mathsf{T}_1}{\Gamma \vdash \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}} : \mathsf{T}_2} \qquad \textbf{(APP)}$$

We will drop parentheses whenever there is no loss of meaning. Functor applications associate to the left, i.e. $\mathsf{E}_1\ \mathsf{E}_2\ \mathsf{E}_3$ means $(\mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{E}_3$. Also note that the type checking rules defined above are syntax-directed.

Theorem 2.3.1. *Type checking of Mei Basic is decidable.*

Proof. By induction on the syntax of module expressions. Since the type checking rules are syntax-directed, type-checking of a module expression can always be reduced to that of a subexpression of it that is strictly smaller than it. Since the size of a module expression is finite, the problem will be reduced to the type-checking of one of the base cases, such as theory definition. □

Corollary 2.3.2. *Type inference of Mei Basic is decidable.*

Remark 2.3.3. We choose not to use a more general **(BASIC)** rule,

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta))}{\vdash (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC}'\textbf{)}$$

since it is not deterministic. It returns a set of possible types, which makes the type-checking algorithm inefficient. In fact, the algorithm has to guess a type for a theory definition expression, which could easily fail. Therefore, we choose to explicitly cast every theory definition with its intended type.

Remark 2.3.4. Although the rule **(BASIC)** is concerned with casting module expressions, it does not compromise the rule **(CAST)**, because (1) **(BASIC)** is only applicable to theory definitions, and (2) it allows one to define a theory with the same language but a different set of axioms from its type.

2.4 Semantics

2.4.1 Substitution of module expression

We can see that the syntax of module expressions follows the λ-calculus style. It is thus not a surprise that we need a notion of substitution to define the semantics. The *substitution function*, $\mathsf{E}[\mathsf{X} := \mathsf{E_p}]$, is defined as follows:

$$\begin{array}{rcl}
\mathsf{Y}[\mathsf{X} := \mathsf{E_p}] & = & \begin{cases} \mathsf{E_p} & \text{if } \mathsf{X} \equiv \mathsf{Y} \\ \mathsf{Y} & \text{otherwise.} \end{cases} \\
\mathsf{C}[\mathsf{X} := \mathsf{E_p}] & = & \mathsf{C} \\
(L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)[\mathsf{X} := \mathsf{E_p}] & = & (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) \\
((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ \mathsf{E})[\mathsf{X} := \mathsf{E_p}] & = & (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ \mathsf{E}[\mathsf{X} := \mathsf{E_p}] \\
(\mathsf{E}\ \texttt{extended by}\ \mathsf{S})[\mathsf{X} := \mathsf{E_p}] & = & \mathsf{E}[\mathsf{X} := \mathsf{E_p}]\ \texttt{extended by}\ \mathsf{S} \\
(\mathsf{E}_1\ \oplus\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E_p}] & = & \mathsf{E}_1[\mathsf{X} := \mathsf{E_p}]\ \oplus\ \mathsf{E}_2[\mathsf{X} := \mathsf{E_p}] \\
(\mathsf{E}\ \texttt{with}\ \rho)[\mathsf{X} := \mathsf{E_p}] & = & \mathsf{E}[\mathsf{X} := \mathsf{E_p}]\ \texttt{with}\ \rho
\end{array}$$

$$
\begin{aligned}
(\texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E_p}] &= \begin{cases}
\texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2[\mathsf{X} := \mathsf{E_p}] & \text{if } \mathsf{Y} \not\equiv \mathsf{X} \text{ and} \\
& \mathsf{Y} \text{ is not free in } \mathsf{E_p} \\
\texttt{functor}\ \mathsf{Z} : \mathsf{T}_1.\ \mathsf{E}'_2[\mathsf{X} := \mathsf{E_p}] & \text{if } \mathsf{Y} \not\equiv \mathsf{X} \text{ and} \\
& \mathsf{Y} \text{ is free in } \mathsf{E_p} \\
\texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2 & \mathsf{Y} \equiv \mathsf{X}.
\end{cases} \\
(\mathsf{E}_1\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E_p}] &= (\mathsf{E}_1[\mathsf{X} := \mathsf{E_p}])\ (\mathsf{E}_2[\mathsf{X} := \mathsf{E_p}])
\end{aligned}
$$

where, in the second case[2] for functor abstraction, $\mathsf{E}'_2 = \mathsf{E}_2[\mathsf{Y} := \mathsf{Z}]$ and Z is a fresh variable. The most important cases are the variable case and the third subcase of the functor abstraction case. They are the base cases for the inductive definition.

Definition 2.4.1. Two module expressions are *α-equivalent*, written $\mathsf{E}_1 =_\alpha \mathsf{E}_2$, if they are only different in the names of the bound variables.

We will consider two module expressions to be equal if they are α-equivalent.

2.4.2 Operational semantics

Following the normal operational semantics of the λ-calculus, we define the operational semantics of Mei Basic via evaluation rules, which transfer a module expression to another preserving its type. The evaluation eventually *converges* in the sense that it terminates in finitely many steps at a *normal form* (*NF*): a module expression that is either a theory definition (similar to a primitive value) or a functor abstraction (similar to a λ abstraction).

$$
\begin{aligned}
\mathit{NF} \ ::= \ & \texttt{TYPE-SPEC THY-SPEC} \\
\mid \ & \texttt{functor VAR : TYPE. EXPR}
\end{aligned}
$$

[2] We rename the bound variable to avoid the variable capture problem. It is not a problem for the implementation since identifiers, which are all distinct, are used to represent variables as shown in §7.2.

For convenience, we will write N, N_1, $\ldots$ for module expressions in *NF*, the set of normal forms.

Let $\cup$ be the standard union and $\sqcup$ be the disjoint union.

Definition 2.4.2. $\uplus$ is an amalgamated union operation with respect to types polymorphically defined as follows:

(1) Let $(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_1,\ \Delta_1)$, $(L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2})\ (L_2,\ \Phi_2,\ \Delta_2)$ be two theory definitions.

  (*a*) $L_1 \uplus L_2 = (L_{\mathsf{T}_1} \cup L_{\mathsf{T}_2}) \sqcup (L_1 \backslash L_{\mathsf{T}_1} \sqcup L_2 \backslash L_{\mathsf{T}_2})$.

  (*b*) $\Phi_1 \uplus \Phi_2 = (\Phi_1^{\mathsf{T}_1} \cup \Phi_2^{\mathsf{T}_2}) \sqcup (\Phi_1 \backslash \Phi_1^{\mathsf{T}_1} \sqcup \Phi_2 \backslash \Phi_2^{\mathsf{T}_2})$, where $\Phi_i^{\mathsf{T}_i} = \{\varphi \in \Phi_i \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}_i}\}$, for $i = 1, 2$, and $\mathbf{lang}(\varphi)$ refers to the set of symbols occurring in φ.

  (*c*) $\Delta_1 \uplus \Delta_2 = (\Delta_1^{\mathsf{T}_1} \cup \Delta_2^{\mathsf{T}_2}) \sqcup (\Delta_1 \backslash \Delta_1^{\mathsf{T}_1} \sqcup \Delta_2 \backslash \Delta_2^{\mathsf{T}_2})$, where $\Delta_i^{\mathsf{T}_i} = \{(\varphi, \mathsf{pf}) \in \Delta_i \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}_i}\}$, for $i = 1, 2$.

(2) Let $(L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)$ be a theory definition and $(L_{\mathsf{S}},\ \Phi_{\mathsf{S}},\ \Delta_{\mathsf{S}})$ be the extensions as in rule **(EXT)**.

  (*a*) $L \uplus L_{\mathsf{S}} = (L_{\mathsf{T}} \cup L_{\mathsf{S}}) \sqcup (L \backslash L_{\mathsf{T}})$.

  (*b*) $\Phi \uplus \Phi_{\mathsf{S}} = (\Phi^{\mathsf{T}} \cup \Phi_{\mathsf{S}}^{\mathsf{T}}) \sqcup (\Phi \backslash \Phi^{\mathsf{T}} \sqcup \Phi_{\mathsf{S}} \backslash \Phi_{\mathsf{S}}^{\mathsf{T}})$, where $\Phi^{\mathsf{T}} = \{\varphi \in \Phi \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$ and $\Phi_{\mathsf{S}}^{\mathsf{T}} = \{\varphi \in \Phi_{\mathsf{S}} \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$.

  (*c*) $\Delta \uplus \Delta_{\mathsf{S}} = (\Delta^{\mathsf{T}} \cup \Delta_{\mathsf{S}}^{\mathsf{T}}) \sqcup (\Delta \backslash \Delta^{\mathsf{T}} \sqcup \Delta_{\mathsf{S}} \backslash \Delta_{\mathsf{S}}^{\mathsf{T}})$, where $\Delta^{\mathsf{T}} = \{(\varphi, \mathsf{pf}) \in \Delta \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$ and $\Delta_{\mathsf{S}}^{\mathsf{T}} = \{(\varphi, \mathsf{pf}) \in \Delta_{\mathsf{S}} \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$.

The idea is that two symbols are considered identical in the union if they are identical in the type specifications and are considered distinct otherwise. For example, let Mult be a type declaring a sort *ele* and a binary operator $\circ$ and *Monoid* be a theory of a monoid with an additional constant e. According to the evaluation rule (*) below, the language of (Mult) *Monoid* $\oplus$ (Mult) *Monoid* contains one *ele*, one $\circ$, but two copies of e, since e is not in Mult. The two e's have to be systematically or heuristically renamed to be distinct.

Definition 2.4.3. Let $(L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)$ be a theory definition. $\rho[\,]$ is a translation operation via a mapping $\rho : L_{\mathsf{T}} \to L'_{\mathsf{T}}$ with respect to types polymorphically defined as follows:

(*a*) $\rho[L] = \rho(L_{\mathsf{T}}) \sqcup (L \backslash L_{\mathsf{T}})$.

(*b*) $\rho[\Phi] = \rho(\Phi^{\mathsf{T}}) \sqcup (\Phi \backslash \Phi^{\mathsf{T}})$, where $\Phi^{\mathsf{T}} = \{\varphi \in \Phi \,|\, \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$.

(c) $\rho[\Delta] = \rho(\Delta^{\mathsf{T}}) \sqcup (\Delta \backslash \Delta^{\mathsf{T}})$, where $\Delta^{\mathsf{T}} = \{(\varphi, \mathsf{pf}) \in \Delta \mid \mathbf{lang}(\varphi) \subseteq L_{\mathsf{T}}\}$.

The purpose is to avoid accidental name collisions between the translated symbols and the "local" symbols that should not be affected by the translation representing a renaming.

Evaluation rules. The following evaluation rules define the evaluation of module expressions:

$$\frac{}{\mathsf{C} \longrightarrow \sigma(\mathsf{C})} \tag*{\textbf{(CONST)}}$$

$$\frac{}{\mathsf{T}\ (\mathsf{T}'\ (L,\ \Phi,\ \Delta)) \longrightarrow \mathsf{T}\ (L,\ \Phi,\ \Delta)} \tag*{\textbf{(THY-CAST)}}$$

$$\frac{\mathsf{E} \longrightarrow \mathsf{E}'}{\mathsf{T}\ \mathsf{E} \longrightarrow \mathsf{T}\ \mathsf{E}'} \tag*{\textbf{(CAST)}}$$

$$\frac{}{\begin{array}{l}((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta))\ \texttt{extended by}\ (L_{\mathsf{S}},\ \Phi_{\mathsf{S}},\ \Delta_{\mathsf{S}}) \\ \qquad \longrightarrow (L_{\mathsf{T}} \cup L_{\mathsf{S}},\ \Phi_{\mathsf{T}} \cup \Phi_{\mathsf{S}} \cup \mathbf{sen}(\Delta_{\mathsf{S}}))(L \uplus L_{\mathsf{S}},\ \Phi \uplus \Phi_{\mathsf{S}},\ \Delta \uplus \Delta_{\mathsf{S}})\end{array}} \tag*{\textbf{(THY-EXT)}}$$

$$\frac{\mathsf{E} \longrightarrow \mathsf{E}'}{\mathsf{E}\ \texttt{extended by}\ \mathsf{S} \longrightarrow \mathsf{E}'\ \texttt{extended by}\ \mathsf{S}} \tag*{\textbf{(EXT)}}$$

$$\frac{}{\begin{array}{l}(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_1,\ \Delta_1) \oplus (L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2})\ (L_2,\ \Phi_2,\ \Delta_2) \\ \qquad \longrightarrow (L_{\mathsf{T}_1} \cup L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_1} \cup \Phi_{\mathsf{T}_2})\ (L_1 \uplus L_2,\ \Phi_1 \uplus \Phi_2,\ \Delta_1 \uplus \Delta_2)\end{array}} \tag*{\textbf{(THY-UNION)}}$$

$$\frac{\mathsf{E}_1 \longrightarrow \mathsf{E}_1'}{\mathsf{E}_1 \oplus \mathsf{E}_2 \longrightarrow \mathsf{E}_1' \oplus \mathsf{E}_2} \tag*{\textbf{(UNION1)}}$$

$$\frac{\mathsf{E}_2 \longrightarrow \mathsf{E}_2'}{(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_1,\ \Delta_1) \oplus \mathsf{E}_2 \longrightarrow (L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_2,\ \Delta_3) \oplus \mathsf{E}_2'} \tag*{\textbf{(UNION2)}}$$

$$((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta))\ \texttt{with}\ \rho \longrightarrow (\rho(L_{\mathsf{T}}), \rho(\Phi_{\mathsf{T}}))\ (\rho[L],\ \rho[\Phi],\ \rho[\Delta]) \qquad \textbf{(THY-REN)}$$

$$\frac{\mathsf{E} \longrightarrow \mathsf{E}'}{\mathsf{E}\ \texttt{with}\ \rho \longrightarrow \mathsf{E}'\ \texttt{with}\ \rho} \qquad \textbf{(REN)}$$

$$\frac{}{(\texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2)\ \mathsf{E}_{\mathsf{p}} \longrightarrow \mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]} \qquad \textbf{(APP-SUB)}$$

$$\frac{\mathsf{E}_{\mathsf{f}} \longrightarrow \mathsf{E}'_{\mathsf{f}}}{\mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}} \longrightarrow \mathsf{E}'_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}} \qquad \textbf{(APP)}$$

Note that **(APP-SUB)** is β-reduction. By using **(EXT)**, **(UNION1)**, **(UNION2)**, **(REN)**, and **(APP)**, we are following the normal order reduction of λ-calculus.

Remarks 2.4.4.

(1) The evaluation rules are syntax-directed in the sense that, given a module expression, at most one rule is applicable.

(2) The evaluation rules implement lazy evaluation (i.e. parameters are evaluated only as needed). It is not hard to change the evaluation rules to implement eager evaluation.

Soundness of the type system. A sound type system guarantees that a well-typed expression cannot get into a wrong state. This can be shown by the following two theorems. Theorem 2.4.5 (progress) says that either a well-typed module expression is already in its normal form or there is an evaluation rule applicable to it, i.e. it will not get stuck. Theorem 2.4.6 (preservation) states that the type of a module expression is kept by all evaluation rules, i.e. evaluation does not break well-typedness of a module expression.

Theorem 2.4.5. *If* $\vdash \mathsf{E} : \mathsf{T}$, *then either* $\mathsf{E} \in NF$ *or else there is an* $\mathsf{E}' \neq \mathsf{E}$ *such that* $\mathsf{E} \longrightarrow \mathsf{E}'$.

Proof. By induction on a typing derivation of E. □

Theorem 2.4.6. *If* $\vdash \mathsf{E} : \mathsf{T}$ *and* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\vdash \mathsf{E}' : \mathsf{T}$.

Proof. By induction on a typing derivation of E. □

Normalization of well-typed module expressions. We want to show that the evaluation of a well-typed module expression is guaranteed to halt in finitely many steps, i.e. well-typed expressions are normalizable. Since there is at most one evaluation rule applicable to a module expression, there is only one notion of normalization[3].

Definition 2.4.7. A *normalizable* module expression E, written $\mathsf{E}\downarrow$, is a module expression that is derivable from the following rules:

$$\frac{\mathsf{N} \in \mathit{NF}}{\mathsf{N}\downarrow} \qquad \frac{\mathsf{E} \longrightarrow \mathsf{E}' \quad \mathsf{E}'\downarrow}{\mathsf{E}\downarrow}$$

Lemma 2.4.8. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$*, then* $\mathsf{E}\downarrow$ *iff* $\mathsf{E}'\downarrow$*.*

Proof. ($\Leftarrow$) trivial. ($\Rightarrow$) follows from the fact that there is at most one evaluation rule applicable for E. □

To prove that every well-typed module expression is normalizable, we will follow the routine of normalization proofs in two steps: (1) construct a set SN of module expressions that are normalizable, and (2) show that every well-typed module expression is an element of SN.

Definition 2.4.9. For each type T, the set SN_{T} of *strongly normalizable*[4] module expressions is defined inductively as follows:

$$\frac{\mathsf{E} : \ (L, \Phi) \quad \mathsf{E}\downarrow}{\mathsf{E} \in \ \mathit{SN}_{(L,\,\Phi)}} \qquad \frac{\forall \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_1}.\ \mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2}}{\mathsf{E} \in \mathit{SN}_{\mathsf{T}_1 \to \mathsf{T}_2}}$$

Then, $\mathit{SN} = \{\mathit{SN}_{\mathsf{T}} \mid \mathsf{T} \text{ is a type}\}$

Lemma 2.4.10. *If* $\mathsf{E} \in \mathit{SN}$*, then* $\mathsf{E}\downarrow$*.*

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L, \Phi)$. Directly follows from Definition 2.4.9.

(2) $\mathsf{T} \equiv \mathsf{T}_1 \rightarrow \mathsf{T}_2$ and $\mathsf{E} \in \mathit{SN}_{\mathsf{T}_1 \to \mathsf{T}_2}$. Let $\mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_1}$ be an arbitrary module expression. By Definition 2.4.9, $\mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2}$, because $\mathsf{E} \in \mathit{SN}_{\mathsf{T}_1 \to \mathsf{T}_2}$. By the induction hypothesis, $(\mathsf{E}\ \mathsf{E_p})\downarrow$, which implies $\mathsf{E}\downarrow$.

□

[3] λ-calculus distinguishes weak and strong normalization with respect to reduction paths.

[4] Here, we are defining "strong" normalization with respect to functor applications.

Note that a functor expression E is strongly normalizable if E itself is normalizable and each of its instantiations $\mathsf{E}\ \mathsf{E_p}$ is strongly nomalizable when $\mathsf{E_p}$ is strongly normalizable. This stronger assumption over strongly normalizable expressions will be crucial in the following proofs.

Lemma 2.4.11. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$*, then* $\mathsf{E} \in SN$ *iff* $\mathsf{E}' \in SN$*.*

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L, \Phi)$. Directly follows Lemma 2.4.8 and Definition 2.4.9.

(2) $\mathsf{T} \equiv \mathsf{T}_1 \rightarrow \mathsf{T}_2$.

($\Rightarrow$) Let $\mathsf{E_p} \in SN_{\mathsf{T}_1}$ be an arbitrary module expression. Since $\mathsf{E} \in SN_{\mathsf{T}_1 \rightarrow \mathsf{T}_2}$, by the definition of SN, $\mathsf{E}\ \mathsf{E_p} \in SN_{\mathsf{T}_2}$. Since $\mathsf{E}\ \mathsf{E_p} \longrightarrow \mathsf{E}'\ \mathsf{E_p}$, by the induction hypothesis, $\mathsf{E}'\ \mathsf{E_p} \in SN_{\mathsf{T}_2}$. Since the choice of $\mathsf{E_p}$ is arbitrary, definition of SN gives the result.

($\Leftarrow$) Analogous to ($\Rightarrow$).

□

The following lemma will be used to prove Theorem 2.4.13 below. This is the typical case where, in order to prove a theorem, we need a stronger induction hypothesis, which is embodied in the lemma.

Lemma 2.4.12. *If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$*,* $\Gamma \equiv \mathsf{X}_1 : \mathsf{T}_1, \ldots, \mathsf{X_n} : \mathsf{T_n}$*, and* $\mathsf{E}_1 \in SN_{\mathsf{T}_1}, \ldots, \mathsf{E_n} \in SN_{\mathsf{T_n}}$*, then* $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in SN_{\mathsf{T}}$*.*

Proof. By induction on the derivation of $\Gamma \vdash \mathsf{E} : \mathsf{T}$. Let $\Gamma \equiv \mathsf{X}_1 : \mathsf{T}_1, \ldots, \mathsf{X_n} : \mathsf{T_n}$ in the following proof.

ASSUMP. $\mathsf{E} \equiv \mathsf{X_i}$ and $\mathsf{T} \equiv \mathsf{T_i}$. Trivial.

CONST. $\mathsf{E} \equiv \mathsf{C}$. Assume $\sigma(\mathsf{C}) = \mathsf{E}'$. Since $\mathsf{C}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \equiv \mathsf{C}$, it is sufficient to show $\mathsf{C} \in SN_{\mathsf{T}}$. By the typing rule **CONST**, $\vdash \mathsf{E}' : \mathsf{T}$ since $\mathsf{C} : \mathsf{T}$. By the induction hypothesis, $\mathsf{E}' \in SN_{\mathsf{T}}$. But $\mathsf{C} \longrightarrow \mathsf{E}'$, so by Lemma 2.4.11, $\mathsf{C} \in SN_{\mathsf{T}}$.

BASIC. $\mathsf{E} \equiv (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)$. Since E is of a theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}]{\downarrow}$. But $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \equiv \mathsf{E}$ is already in normal form.

CAST. $\mathsf{E} \equiv (L, \Phi)\ \mathsf{E}'$. Since E is of a theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L',\Phi')}$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rcl} ((L,\Phi)\ \mathsf{E}')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] & = & (L,\Phi)\ (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\ & \longrightarrow & (L,\Phi)\ ((L',\Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})) \\ & \longrightarrow & (L,\Phi)\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \end{array}$$

which is in normal form.

EXT. $\mathsf{E} \equiv \mathsf{E}'$ `extended by` $(L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})$. Since E is of a theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L',\Phi')}.$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rl} & (\mathsf{E}'\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}))[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\ = & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}])\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\ \longrightarrow & ((L',\Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\ \longrightarrow & (L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))\ (L_{\mathsf{E}'} \uplus L_\mathsf{S},\ \Phi_{\mathsf{E}'} \uplus \Phi_\mathsf{S},\ \Delta_{\mathsf{E}'} \uplus \Delta_\mathsf{S}) \end{array}$$

which is in normal form.

UNION. $\mathsf{E} \equiv \mathsf{E}' \oplus \mathsf{E}''$. Since E is of a theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$ and $\Gamma \vdash \mathsf{E}'' : (L'', \Phi'')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L',\Phi')}$$

and

$$\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L'',\Phi'')}.$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'})$ and $\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \longrightarrow (L'', \Phi'')\ (L_{\mathsf{E}''},\ \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}''})$.

$$\begin{array}{rl} & (\mathsf{E}' \oplus \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \\ = & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]) \oplus (\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]) \\ \longrightarrow & (L', \Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}) \oplus (L'', \Phi'')\ (L_{\mathsf{E}''},\ \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}''}) \\ \longrightarrow & (L' \cup L'',\ \Phi' \cup \Phi'')\ (L_{\mathsf{E}'} \uplus L_{\mathsf{E}''},\ \Phi_{\mathsf{E}'} \uplus \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}'} \uplus \Delta_{\mathsf{E}''}) \end{array}$$

which is in normal form.

REN. $\mathsf{E} \equiv \mathsf{E}'\ \texttt{with}\ \rho$. Since E is of a theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \in SN_{(L', \Phi')}.$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rl} & (\mathsf{E}'\ \texttt{with}\ \rho)[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \\ = & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}])\ \texttt{with}\ \rho \\ \longrightarrow & ((L', \Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \texttt{with}\ \rho \\ \longrightarrow & (\rho(L'),\ \rho(\Phi'))\ (\rho[L_{\mathsf{E}'}],\ \rho[\Phi_{\mathsf{E}'}],\ \rho[\Delta_{\mathsf{E}'}]) \end{array}$$

which is in normal form.

ABS. $\mathsf{E} \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}''$ and $\mathsf{T} \equiv \mathsf{T}' \rightarrow \mathsf{T}''$. In order to prove

$$(\texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]\ \in SN_{\mathsf{T}' \rightarrow \mathsf{T}''},$$

it is sufficient to prove

$$(\texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]\ \mathsf{E}' \in SN_{\mathsf{T}''},$$

where E' is an arbitrary module expression of type T'. Without lose of the generality, we assume $\mathsf{X} \not\equiv \mathsf{X_i}$ for $1 \leq \mathsf{i} \leq \mathsf{n}$. We have

$$\begin{array}{rl} & (\texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]\ \mathsf{E}' \\ = & (\texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E}' \\ \longrightarrow & \mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]\ [\mathsf{X} := \mathsf{E}']. \end{array}$$

Since $\Gamma, \mathsf{X} : \mathsf{T}' \vdash \mathsf{E}'' : \mathsf{T}''$ is in the premise, by the induction hypothesis (in the reverse direction)

$$\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\ [\mathsf{X} := \mathsf{E}'] \in SN_{\mathsf{T}''}.$$

By lemma 2.4.11,

$$(\texttt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\ \mathsf{E}' \in SN_{\mathsf{T}''}.$$

APP. $\mathsf{E} \equiv \mathsf{E}'\ \mathsf{E}''$. Assume $\Gamma \vdash \mathsf{E}' : \mathsf{T}'' \rightarrow \mathsf{T}$ and $\Gamma \vdash \mathsf{E}'' : \mathsf{T}''$ are in the premise. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{\mathsf{T}'' \rightarrow \mathsf{T}}$$

and

$$\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{\mathsf{T}''}.$$

By Definition 2.4.9,

$$\begin{array}{rl} & (\mathsf{E}'\ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\ = & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}])\ (\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\ \in & SN_{\mathsf{T}}. \end{array}$$

□

Theorem 2.4.13. *If* $\vdash \mathsf{E} :\ \mathsf{T}$, $E\!\downarrow$.

Proof. By Lemma 2.4.12, $\mathsf{E} \in SN_{\mathsf{T}}$. By Lemma 2.4.10, $\mathsf{E}\!\downarrow$. □

2.4.3 Denotational semantics

We will define the denotational semantics of module types and module expressions via a set of valuation functions, one for each category of syntactic objects.

Semantics of module types. The valuation function for module types, $[\![\,]\!]_{\mathsf{t}}$, is a total function on module types such that:

(1) $[\![(L, \Phi)]\!]_{\mathsf{t}} = \{(L, \Phi)\ (L_T, \Phi_T, \Delta_T) \mid (L_T, \Phi_T, \Delta_T) \in \mathcal{TH}$ and $L \subseteq L_T$ and $\Phi \subseteq \Phi_T \cup \mathbf{sen}(\Delta_T)\}$.

(2) $[\![\mathsf{T}_1 \to \mathsf{T}_2]\!]_{\mathsf{t}} = [\![\mathsf{T}_1]\!]_{\mathsf{t}} \to [\![\mathsf{T}_2]\!]_{\mathsf{t}}$, that is the function space from the valuation of T_1 to the valuation of T_2.

Definition 2.4.14.

(1) A *variable assignment* φ is a function that maps each variable $\mathsf{X} : \mathsf{T}$ to a value v of $[\![\mathsf{T}]\!]_{\mathsf{t}}$.

(2) Given a variable assignment φ, a variable $\mathsf{X} : \mathsf{T}$, and a value $\mathsf{v} \in [\![\mathsf{T}]\!]_{\mathsf{t}}$, $\varphi[\mathsf{X} : \mathsf{T} \mapsto \mathsf{v}]$ is the variable assignment such that a substitution of φ is,

$$\varphi[\mathsf{X} : \mathsf{T} \mapsto \mathsf{v}]\ (\mathsf{Y} : \mathsf{T}') \ = \begin{cases} \mathsf{v} & \text{if } \ \mathsf{X} \equiv \mathsf{Y} \text{ and } \mathsf{T} \equiv \mathsf{T}' \\ \varphi(\mathsf{Y} : \mathsf{T}') & \text{otherwise.} \end{cases}$$

Semantics of module expressions. The valuation function for typed module expressions, $[\![\,]\!]_{\mathsf{te}}^{\sigma,\Gamma,\varphi}$, maps each typed module expression to an assertion that $[\![\mathsf{E}]\!]_{e}^{\sigma,\Gamma,\varphi}$ is a member of $[\![\mathsf{T}]\!]_{\mathsf{t}}$:

$$[\![\mathsf{E} :\ \mathsf{T}]\!]_{\mathsf{te}}^{\sigma,\Gamma,\varphi} = [\![\mathsf{E}]\!]_{e}^{\sigma,\Gamma,\varphi} \in [\![\mathsf{T}]\!]_{\mathsf{t}}$$

Note that the valuation function for types, $[\![\,]\!]_{\mathsf{t}}$, is defined above, while the valuation function for module expressions, $[\![\,]\!]_{e}^{\sigma,\Gamma,\varphi}$, is not yet defined. Therefore, we will define $[\![\,]\!]_{e}^{\sigma,\Gamma,\varphi}$ next. Given a module expression E of type T, $[\![\mathsf{E}]\!]_{e}^{\sigma,\Gamma,\varphi}$ returns a value in the domain $[\![\mathsf{T}]\!]_{\mathsf{t}}$.

Remarks 2.4.15.

(1) Both $[\![\,]\!]_{\mathsf{te}}^{\sigma,\Gamma,\varphi}$ and $[\![\,]\!]_{e}^{\sigma,\Gamma,\varphi}$ are defined with respect to an environment σ, a context Γ, and a variable assignment φ.

(2) $[\![\,]\!]_{e}^{\sigma,\Gamma,\varphi}$ is a partial function. We will use $\uparrow$ to represent the undefined (divergent) value. For instance, the value for a wrongly typed module expression should be undefined.

ASSUMP. $\mathsf{E} \equiv \mathsf{X}$:

$$[\![\mathsf{X}]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} \varphi(\mathsf{X} : \mathsf{T}) & \text{if } \mathsf{X} : \mathsf{T} \in \Gamma \\ \uparrow & \text{otherwise.} \end{cases}$$

CONST. $\mathsf{E} \equiv \mathsf{C}$:

$$[\![\mathsf{C}]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} [\![\mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi} & \text{if } \sigma(\mathsf{C}) = \mathsf{E}' \\ \uparrow & \text{otherwise.} \end{cases}$$

BASIC. $\mathsf{E} \equiv (L_\mathsf{T}, \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta)$:

$$[\![(L_\mathsf{T}, \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta)]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} (L_\mathsf{T}, \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta) & \text{if } (L, \Phi, \Delta) \in \mathcal{TH} \\ \uparrow & \text{otherwise.} \end{cases}$$

CAST. $\mathsf{E} \equiv (L_\mathsf{T}, \Phi_\mathsf{T})\ \mathsf{E}'$:

$$[\![(L_\mathsf{T}, \Phi_\mathsf{T})\ \mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} (L_\mathsf{T}, \Phi_\mathsf{T})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) & \text{if } [\![\mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi} = (L_{\mathsf{T}'}, \Phi_{\mathsf{T}'})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \\ & \text{and } L_\mathsf{T} \subseteq L_{\mathsf{T}'} \text{ and } \Phi_\mathsf{T} \subseteq \Phi_{\mathsf{T}'} \\ \uparrow & \text{otherwise.} \end{cases}$$

EXT. $\mathsf{E} \equiv \mathsf{E}'$ `extended by` S:

$$[\![\mathsf{E}' \texttt{ extended by } \mathsf{S}]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} \mathsf{E}_{\mathsf{eval}} & \text{if } [\![\mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi} = (L_{\mathsf{T}'}, \Phi_{\mathsf{T}'})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \\ & \text{and } \mathsf{S} = (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \text{ and } \mathsf{E}_{\mathsf{eval}} \in \mathcal{TH} \\ \uparrow & \text{otherwise,} \end{cases}$$

where $\mathsf{E}_{\mathsf{eval}} \equiv (L_{\mathsf{T}'} \cup L_\mathsf{S}, \Phi_{\mathsf{T}'} \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))\ (L_{\mathsf{E}'} \uplus L_\mathsf{S}, \Phi_{\mathsf{E}'} \uplus \Phi_\mathsf{S}, \Delta_{\mathsf{E}'} \uplus \Delta_\mathsf{S})$.[5]

[5] $\mathsf{E}_{\mathsf{eval}}$ may not be in $\mathcal{TH}$ because Φ_S may use symbols that are neither in $L_{\mathsf{T}'}$ nor in L_S.

UNION. $\mathsf{E} \equiv \mathsf{E}_1 \oplus \mathsf{E}_2$:

$$[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} \mathsf{E}_{\mathsf{eval}} & \text{if } [\![\mathsf{E}_1]\!]_e^{\sigma,\Gamma,\varphi} = (L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_{\mathsf{E}_1}, \Phi_{\mathsf{E}_1}, \Delta_{\mathsf{E}_1}) \\ & \text{and } [\![\mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi} = (L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2})\ (L_{\mathsf{E}_2}, \Phi_{\mathsf{E}_2}, \Delta_{\mathsf{E}_2}) \\ \uparrow & \text{otherwise,} \end{cases}$$

where $\mathsf{E}_{\mathsf{eval}} \equiv (L_{\mathsf{T}_1} \cup L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_1} \cup \Phi_{\mathsf{T}_2})\ (L_{\mathsf{E}_1} \uplus L_{\mathsf{E}_2}, \Phi_{\mathsf{E}_1} \uplus \Phi_{\mathsf{E}_2}, \Delta_{\mathsf{E}_1} \uplus \Delta_{\mathsf{E}_2})$.

REN. $\mathsf{E} \equiv \mathsf{E}'$ `with` ρ:

$$[\![\mathsf{E}' \ \texttt{with}\ \rho]\!]_e^{\sigma,\Gamma,\varphi} \simeq \begin{cases} \mathsf{E}_{\mathsf{eval}} & \text{if } [\![\mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi} = (L_{\mathsf{T}'}, \Phi_{\mathsf{T}'})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \\ & \text{and } \mathbf{source}(\rho) = L_{\mathsf{T}'} \\ \uparrow & \text{otherwise.} \end{cases}$$

where $\mathsf{E}_{\mathsf{eval}} \equiv (\rho(L_{\mathsf{T}'}), \rho(\Phi_{\mathsf{T}'}))\ (\rho[L_{\mathsf{E}'}], \rho[\Phi_{\mathsf{E}'}], \rho[\Delta_{\mathsf{E}'}])$.

ABS. $\mathsf{E} \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2$:

$$[\![\texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi}$$

is a function $F : [\![\mathsf{T}_1]\!]_{\mathsf{t}} \to [\![\mathsf{T}_2]\!]_{\mathsf{t}}$ such that

$$\forall \mathsf{v} \in [\![\mathsf{T}_1]\!]_e^{\sigma,\Gamma,\varphi}.\ F(\mathsf{v}) = [\![\mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi[\mathsf{X}\mapsto\mathsf{v}]}.$$

In particular,

$$\forall \mathsf{E}_{\mathsf{p}} : \mathsf{T}_1.\ F([\![\mathsf{E}_{\mathsf{p}}]\!]_e^{\sigma,\Gamma,\varphi}) = [\![\mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]]\!]_e^{\sigma,\Gamma,\varphi},$$

where $\mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]$ is the *substitution function* defined in §2.4.1.

APP. $\mathsf{E} \equiv \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}$:

$$[\![\mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}]\!]_e^{\sigma,\Gamma,\varphi} = [\![\mathsf{E}_{\mathsf{f}}]\!]_e^{\sigma,\Gamma,\varphi}([\![\mathsf{E}_{\mathsf{p}}]\!]_e^{\sigma,\Gamma,\varphi})$$

Correctness of the operational semantics. The following theorem shows the correctness of the denotational semantics of Mei Basic with respect to the operational semantics:

Theorem 2.4.16. *If* $\mathsf{E} : \mathsf{T}$ *and* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $[\![\mathsf{E}]\!]_e^{\sigma,\Gamma,\varphi} = [\![\mathsf{E}']\!]_e^{\sigma,\Gamma,\varphi}$ *for all environments* σ, *all contexts* Γ, *and all variable assignments* φ.

Proof. By induction on the operational evaluation rules of E. □

We would also like to prove that the operational semantics is correct with respect to the denotational semantics. However, the following statement is not true:

Statement 2.4.17. *If $[\![\mathsf{E}_1]\!]_e^{\sigma,\Gamma,\varphi} = [\![\mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi}$ for all environments σ, all contexts Γ, and all variable assignments φ, there exist module expressions E_1' and E_2' such that $\mathsf{E}_1' =_\alpha \mathsf{E}_2'$, $\mathsf{E}_1 \longrightarrow \mathsf{E}_1'$, and $\mathsf{E}_2 \longrightarrow \mathsf{E}_2'$.*

The counterexample is $\mathsf{E}_1 \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T}.\ \mathsf{E}_{11}$ and $\mathsf{E}_2 \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T}.\ \mathsf{E}_{22}$ where $\mathsf{E}_{11} \neq_\alpha \mathsf{E}_{22}$. Although it is possible that all the instantiations of E_1 and E_2 can be reduced to the same module expression which means that they share the same functionality (denotational semantics), they are already in the normal form and cannot be reduced further and they are not α-equivalent.

Instead of Statement 2.4.17, we present the following conjecture:

Conjecture 2.4.18. *If $\vdash \mathsf{E} : (L, \Phi)$ is a module expression of theory type, then $[\![\mathsf{E}]\!]_e^{\sigma,\Gamma,\varphi} \in NF$ and $\mathsf{E} \longrightarrow [\![\mathsf{E}]\!]_e^{\sigma,\Gamma,\varphi}$ for all environments σ, all contexts Γ, and all variable assignments φ.*

Remark 2.4.19. To prove Conjecture 2.4.18, there are two cases we need to consider: (1) E does not have subexpressions of functor type and (2) E has subexpressions of functor type. Case (1) is easy to prove. Case (2) is the hard part, since we need a hypothesis not only over module expressions of theory type but also over module expressions of functor type. Intuitively, the hypothesis should state that an instantiation of a module expression of functor type (well typed) either can be reduced to a normal form or is a module expression of functor type (well typed). It is likely that we need another conjecture for module expressions of functor type, which possibly has to be proved mutually with Conjecture 2.4.18.

The following conjecture follows directly Conjecture 2.4.18 showing that, if two module expressions of theory type have the same denotational semantics, they can be reduced to the same normal form, namely their denotational semantics, by the evaluation rules.

Conjecture 2.4.20. *If $\vdash \mathsf{E}_1 : (L, \Phi)$, $\vdash \mathsf{E}_2 : (L, \Phi)$, and $[\![\mathsf{E}_1]\!]_e^{\sigma,\Gamma,\varphi} = [\![\mathsf{E}_2]\!]_e^{\sigma,\Gamma,\varphi}$ for all environments σ, all contexts Γ, and all variable assignments φ, then $\mathsf{E} \equiv [\![\mathsf{E}_1]\!]_e^{\sigma,\Gamma,\varphi} \in NF$, $\mathsf{E}_1 \longrightarrow \mathsf{E}$, and $\mathsf{E}_2 \longrightarrow \mathsf{E}$.*

2.5 Issues about conservative extensions of theories

Definition 2.5.1. A theory T_2 is a *conservative extension* of another theory T_1, if T_2 is an extension of T_1 and every provable sentence of T_2 that involves only symbols of T_1 is also provable in T_1.

The notion of conservative extension of theories is interesting due to

Theorem 2.5.2. *If T_2 is a conservative extension of T_1 and T_1 is consistent, then T_2 is consistent as well.*

Hence, it will be useful if we can infer that a theory is a conservative extension of another theory, since conservative extensions will not introduce new inconsistencies. For instance, let $F \equiv$ `functor X` : (L, Φ). `X extended by S` be a first-order functor and $T_1 \equiv (L, \Phi, \Delta)$ where Δ is empty. Assume that $F\ T_1 = (L, \Phi, \Delta)$ `extended by S` is a conservative extension of T_1. We would expect that, for an arbitrary theory T of type (L, Φ) that is consistent, $(F\ T)$ would be consistent and a conservative extension of T. This property is formalized as the *Modularization Property* in [28, 31, 92]. Not all logics possess the Modularization Property, but many useful logics do, for example, many-sorted first-order logic possesses the Modularization Property [92]. [31] shows that the Modularization Property is equivalent to the Craig Robinson Interpolation (CRI). Formally, this result is

Theorem 2.5.3. *Let $F \equiv$ `functor X` : (L, Φ). `X extended by S` be a first-order functor and $T_1 \equiv (L, \Phi, \Delta)$ where Δ is empty. Assume that $F\ T_1 = (L, \Phi, \Delta)$ `extended by S` is a conservative extension of T_1. For an arbitrary consistent theory $T : (L, \Phi)$ in some logic $\mathbf{L}$, if $\mathbf{L}$ has CRI, then $(F\ T)$ is a conservative extension of T.*

Proof. Since $T : (L, \Phi)$, there is a trivial interpretation from T_1 to T. The rest of the proof follows directly from the Modularization Theorem in [31]. □

Theorem 2.5.3 states that in the case when the underlying logic possesses the Modularization Property, if a Mei Basic functor is defined *properly*, its instantiation will give a *good* result.

Although it is good for the user to acknowledge this information, it will greatly reduce the flexibility of Mei Basic functors if only this kind of functor is admitted because:

(1) The body of a Mei Basic functor might not be an extension of its parameter. For instance, the body may not have any occurrence of the parameter or the parameter may be renamed in the functor body. For the latter case, we can still talk about conservative extension on the model level but not on the syntax level as in Definition 2.5.1.

(2) Some useful functors produce an extension of its parameter, but not a conservative extension, e.g. the functor *Comm* produces an extension of its parameter of type Mult (refer to §2.1.3 for definitions of *Comm* and Mult) but not a conservative extension.

Chapter 3

A module system with subtyping and coercion – Mei

The module system we built so far is very close to the ML-family module system. Both systems support higher-order functor abstractions and applications. The semantics of functor application is defined via a substitution function. However, a renaming operator is supported by Mei Basic but not by ML's module system. As in an ML-family module system, parameter matching of functor applications in Mei Basic is based purely on syntax. The advantage is that type checking is decidable. However, the parameter passing mechanism is rigid in the sense that some reasonable functor applications are ruled out by the typing rules. We address this problem in this chapter and extend Mei Basic with two new mechanisms, subtyping and coercion, to solve it.

3.1 Subtyping

By Definition 2.1.17, every theory in Mei Basic can have more than one type. For instance, the theory *Monoid* is of both type Monoid and Mult. Thus we can use *Monoid* as the actual parameter of *Comm*, which requires a theory of type Mult as input. However, this idea does not work for functors as shown in the following two examples:

Example 3.1.1.

$$\begin{aligned} \mathit{CommAssoc} \;\equiv\; & \texttt{functor}\ \mathsf{X}\ : \\ & \texttt{language sort}\ ele \\ & \qquad\qquad \texttt{func}\ \circ : ele^2 \rightarrow ele. \\ & \mathsf{X}\ \texttt{extended by} \\ & \texttt{axioms}\ \ \forall x, y : ele.x \circ y = y \circ x \\ & \qquad\qquad \forall x_1, x_2, x_3 : ele.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \end{aligned}$$

Clearly, *CommAssoc* is a functor of type Mult → CommAssocMult, where CommAssocMult is defined as follows:

$$\begin{aligned} \mathsf{CommAssocMult} \;\equiv\; & \texttt{language sort}\ ele \\ & \qquad\qquad \texttt{func}\ \ \circ : ele^2 \rightarrow ele \\ & \texttt{axioms}\ \ \forall x, y : ele.x \circ y = y \circ x \\ & \qquad\qquad \forall x_1, x_2, x_3 : ele.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \end{aligned}$$

However, it is reasonable that *CommAssoc* can be used in the place where a functor of type Mult → CommMult is required, since it does provide the functionality that Mult → CommMult requires and even more. In other words, *CommAssoc* should be considered as an element of Mult → CommMult in some sense.

Example 3.1.2. Let F be a functor of type Mult → T, where T is an unspecified type. It is reasonable that F can be used in the place where a functor of type Monoid → T is required, since any theory of type Monoid is also of type Mult and the instantiation does output a theory of T. In other words, F should be considered as an element of Monoid → T in some sense.

Now, it is clear that we need a uniform mechanism for both theories and functors that can treat an object of one type as an object of another type when it is reasonable to do so. One natural approach will be subtyping, which gives a way to regard an object of subtype T_1 as an object of its supertype T_2. In general, the subtype T_1 is more informative than the supertype T_2. Thus, it is safe to use an object of T_1 in the place where an object of T_2 is required. Subtyping is interpreted as a subset relation in many languages. Another interpretation of subtyping is coercion, i.e. T_1 is a subtype of T_2 if there is a coercion function that maps every object in T_1 to an object in T_2. Both Coq and Lego implement coercive subtyping [5, 44]. At the module level, a coercion function corresponds to a projection function over record types. Clearly, there is only one way to coerce a record encoding the theory of rings to a record encoding the theory of monoids by syntactic projection. In terms of Amokrane Saïbi's Inheritance Graph [86], there is at most one path between two classes.

3.1.1 Naive subtyping rules

The following are the most natural subtyping rules we can find in many languages.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_2 \subseteq L_1 \quad \Phi_2 \subseteq \Phi_1}{\mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(THY-SUB)}$$

$$\frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}} \qquad \textbf{(FUNC-SUB)}$$

$$\frac{\mathsf{T}_1 <: \mathsf{T}_2 \quad \mathsf{T}_2 <: \mathsf{T}_3}{\mathsf{T}_1 <: \mathsf{T}_3} \qquad \textbf{(TRAN)}$$

Note that, in the premises of rule **(FUNC-SUB)**, the subtype relation between the argument types is reversed (contravariant), while that between result types has the same direction (covariant) as that between the functor types. If $F : \mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1}$ is a functor, it can be seen as a functor in $\mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}$ as well. Because F accepts any object in $\mathsf{T}_{\mathsf{s}_1}$ as input, it will thus also accept any object in $\mathsf{T}_{\mathsf{s}_2}$, since $\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1}$. In addition, an instantiation of F that returns an object in $\mathsf{T}_{\mathsf{t}_1}$ can thus also return an object in $\mathsf{T}_{\mathsf{t}_2}$, since $\mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}$. Refer to Examples 3.1.1 and 3.1.2 for concrete examples.

3.1.2 Algorithmic subtyping rules

We can observe that the subtyping rules in §3.1.1 are not syntax-directed, thus they are not algorithmic[1], i.e. they cannot give a subtype checking algorithm directly. The

[1] By "algorithmic rules", we mean that we can derive a type checking algorithm from the rules directly. We assume that types are not annotated for all subexpressions and thus the rules are used bottom-up to infer the types of the subexpressions.

problem is rule **(TRAN)**. Since both T_1 and T_3 are metavariables, they can match any types. Another problem is that T_2 is in both premises of **(TRAN)** but not in the conclusion. Thus we have to *guess* a value for it and check the premises, and there is little chance to succeed. The good news is that we can eliminate rule **(TRAN)** by the following theorem:

Theorem 3.1.3. *If we can derive* $\mathsf{T}_1 <: \mathsf{T}_2$*, we can derive it without using the rule* **(TRAN)***.*

Proof. Prove by induction on the subtyping derivation. Let us consider the cases of the last step of the derivation.

(1) Case **(THY-SUB)** trivial.

(2) Case **(FUNC-SUB)** $\mathsf{T}_1 \equiv \mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1}$ and $\mathsf{T}_2 \equiv \mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}$.

$$\frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}}$$

Since both the size of $\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1}$ and $\mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}$ are strictly smaller than that of $\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}$, the latter can be derived without using rule **(TRAN)** directly from the induction hypothesis.

(3) Case **(TRAN)**

$$\frac{\mathsf{T}_1 <: \mathsf{T} \quad \mathsf{T} <: \mathsf{T}_2}{\mathsf{T}_1 <: \mathsf{T}_2}$$

By the induction hypothesis, we can assume that both $\mathsf{T}_1 <: \mathsf{T}$ and $\mathsf{T} <: \mathsf{T}_2$ can be derived without using rule **(TRAN)**. There are thus two subcases[2] for the derivation of $\mathsf{T}_1 <: \mathsf{T}$ and $\mathsf{T} <: \mathsf{T}_2$ as follows:

(*i*) $\mathsf{T}_1 \equiv (L_1, \Phi_1)$, $\mathsf{T}_2 \equiv (L_2, \Phi_2)$, and $\mathsf{T} \equiv (L, \Phi)$.

$$\frac{\dfrac{L \subseteq L_1 \quad \Phi \subseteq \Phi_1}{\mathsf{T}_1 <: \mathsf{T}} \quad \dfrac{L_2 \subseteq L \quad \Phi_2 \subseteq \Phi}{\mathsf{T} <: \mathsf{T}_2}}{\mathsf{T}_1 <: \mathsf{T}_2}$$

can be replaced by

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_2 \subseteq L_1 \quad \Phi_2 \subseteq \Phi_1}{\mathsf{T}_1 <: \mathsf{T}_2}.$$

[2] The case in which **(THY-SUB)** and **(FUNC-SUB)** are used together is not possible because T_1 and T_2 must be either both theory types or both functor types.

(ii) $\mathsf{T}_1 \equiv \mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1}$, $\mathsf{T}_2 \equiv \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}$, and $\mathsf{T} \equiv \mathsf{T}_\mathsf{s} \rightarrow \mathsf{T}_\mathsf{t}$.

The derivation

$$\dfrac{\dfrac{\mathsf{T}_\mathsf{s} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_\mathsf{t}}{\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_\mathsf{s} \rightarrow \mathsf{T}_\mathsf{t}} \quad \dfrac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_\mathsf{s} \quad \mathsf{T}_\mathsf{t} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_\mathsf{s} \rightarrow \mathsf{T}_\mathsf{t} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}}}{\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}}$$

can be reorganized as the following derivation

$$\dfrac{\dfrac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_\mathsf{s} \quad \mathsf{T}_\mathsf{s} <: \mathsf{T}_{\mathsf{s}_1}}{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1}} \quad \dfrac{\mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_\mathsf{t} \quad \mathsf{T}_\mathsf{t} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}}{\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}}.$$

Again, since both the size of $\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1}$ and that of $\mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}$ are strictly smaller than that of $\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}$, the latter can be derived without using rule **(TRAN)** directly from the induction hypothesis.

□

Theorem 3.1.3 shows that the algorithmic subtyping rules are both sound and complete with respect to the naive subtyping rules. Also notice that, the algorithmic subtyping rules are syntax-directed and checking a subtype relation can always be reduced to checking smaller subtype relations. This justifies the following proposition.

Proposition 3.1.4. *Subtype checking is decidable for the algorithmic subtyping rules.*

3.1.3 New typing rule

We need a new typing rule to say that, if E is an object of T_1 and $\mathsf{T}_1 <: \mathsf{T}_2$, then E is an object of T_2. Instead of the most general subsumption rule,

$$\dfrac{\Gamma \vdash \mathsf{E} : \ \mathsf{T}_1 \quad \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{E} : \ \mathsf{T}_2},$$

we choose to use an enhanced functor application rule

$$\dfrac{\Gamma \vdash \mathsf{E}_\mathsf{f} : \mathsf{T}_1 \rightarrow \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_\mathsf{p} : \mathsf{T}_\mathsf{p} \quad \mathsf{T}_\mathsf{p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E}_\mathsf{f}\ \mathsf{E}_\mathsf{p} : \mathsf{T}_2} \qquad \textbf{(APP)}.$$

The reason is that the subsumption rule is not algorithmic in that E in the conclusion is a metavariable that can match arbitrary expressions. Also, the only place we need the subtype relation in typing a module expression is in the functor application when the actual parameter type does not match the argument type of the functor. The justification of using rule **(APP)** instead of the subsumption rule is similar to that of the elimination of rule **(TRAN)** in §3.1.2, but more tedious. Similar justification for simply typed λ-calculus with subtyping can be found in §16.2 in [80].

3.1.4 Instantiation of functors

With the help of subtype relations, a functor can accept more objects as input as shown in Figure 3.1.

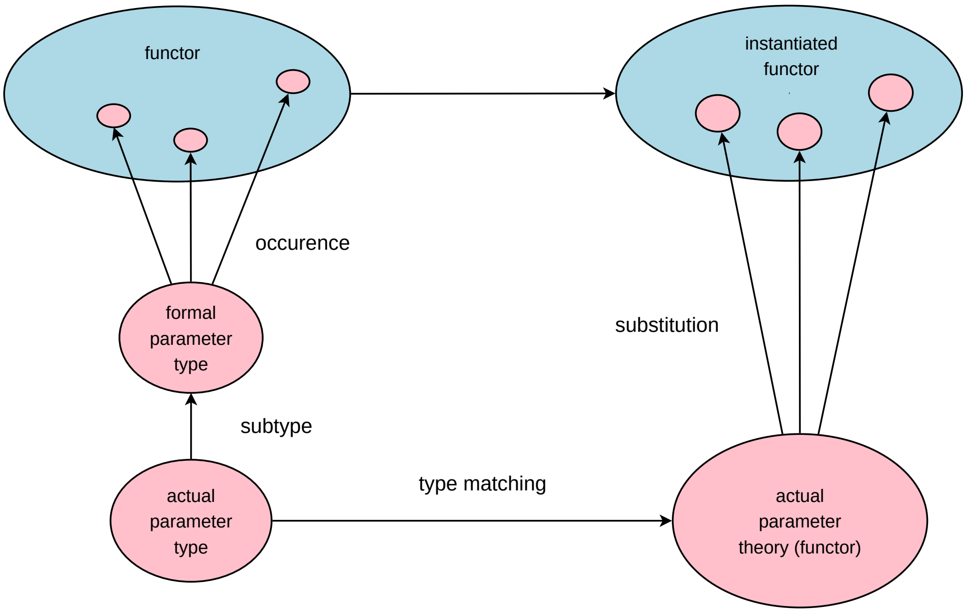

Figure 3.1: Functor Application with Subtyping

3.2 Syntax of Mei Core

The syntax and rules of Mei Core are similar to those of Mei Basic except for the subtyping rules (§3.1.2) and the new **(APP)** rule (§3.1.3). This subsection presents the concrete syntax and rules of Mei Core. The parts that are different from Mei Basic are put in boxes. We will use this “box” convention in the following chapters too.

$$\begin{array}{lcl}
\texttt{EXPR} & ::= & \texttt{MOD-CONST} \\
 & | & \texttt{TYPE-SPEC}\ \texttt{THY-SPEC} \\
 & | & \texttt{TYPE-SPEC}\ \texttt{EXPR} \\
 & | & \texttt{EXPR extended by SPEC} \\
 & | & \texttt{EXPR} \oplus \texttt{EXPR} \\
 & | & \texttt{EXPR with MAPPING} \\
 & | & \texttt{functor VAR} : \texttt{TYPE.}\ \texttt{EXPR} \\
 & | & \texttt{EXPR}\ \texttt{EXPR}
\end{array}$$

$$\begin{array}{lcl}
\texttt{TYPE} & ::= & \texttt{TYPE-CONST} \\
 & | & \texttt{TYPE-SPEC} \\
 & | & \texttt{TYPE} \rightarrow \texttt{TYPE}
\end{array}$$

$$\texttt{THY-SPEC} \ ::= \ (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS})$$

$$\texttt{TYPE-SPEC} \ ::= \ (\texttt{LANG}, \texttt{AXIOMS})$$

$$\texttt{MOD-CONST} \ ::= \ \texttt{IDENTIFIER}$$

$$\texttt{TYPE-CONST} \ ::= \ \texttt{IDENTIFIER}$$

$$\texttt{VAR} \ ::= \ \texttt{IDENTIFIER}$$

Rules for types.

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \mathbf{closed}(L, \Phi)}{\mathbf{type}(\mathsf{T})} \tag*{\textbf{(THY-TYPE)}}$$

$$\frac{\mathbf{type}(\mathsf{T}_1) \quad \mathbf{type}(\mathsf{T}_2)}{\mathbf{type}(\mathsf{T}_1 \rightarrow \mathsf{T}_2)} \tag*{\textbf{(FUNC-TYPE)}}$$

Subtyping rules.

$$\boxed{\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_2 \subseteq L_1 \quad \Phi_2 \subseteq \Phi_1}{\mathsf{T}_1 <: \mathsf{T}_2}} \qquad \boxed{\textbf{(THY-SUB)}}$$

$$\boxed{\frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{s}_1} \longrightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \longrightarrow \mathsf{T}_{\mathsf{t}_2}}} \qquad \boxed{\textbf{(FUNC-SUB)}}$$

Rules for typing module expressions.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \qquad \textbf{(ASSUMP)}$$

$$\frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \qquad \textbf{(CONST)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) <: (L, \Phi)}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \qquad \textbf{(CAST)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(EXT)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \qquad \textbf{(UNION)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : (\rho(L), \rho(\Phi))} \qquad \textbf{(REN)}$$

$$\frac{\Gamma, \mathsf{X} : \mathsf{T}_1 \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \mathsf{T}_1 \rightarrow \mathsf{T}_2} \qquad \textbf{(ABS)}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{E}_{\mathsf{f}} : \mathsf{T}_1 \rightarrow \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_{\mathsf{p}} : \mathsf{T}_{\mathsf{p}} \quad \mathsf{T}_{\mathsf{p}} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}} : \mathsf{T}_2}} \qquad \boxed{\textbf{(APP)}}$$

3.3 Semantics of Mei Core

The semantics of Mei Core is exactly the same as that of Mei Basic, since there is no new syntactic classes introduced. One difference is that a functor can be applied to more module expressions due to the new **(APP)** rule and the subtyping arguments. The evaluation of these functor applications still follows the substitution semantics as in Mei Basic.

Remark 3.3.1. Though we choose to keep the semantics of Mei Core the same as Mei Basic, it is possible to define a new rule for the functor application where the argument is of theory type, in which the argument theory is cast to the right type before the substitution.

$$\frac{}{(\texttt{functor}\ \mathsf{X} : (L_{\mathsf{T}}, \Phi_{\mathsf{T}}).\ \mathsf{E}_2)\ \mathsf{E_p} \longrightarrow \mathsf{E}_2[\mathsf{X} :=\ (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ \mathsf{E_p}]}$$

The motivation of this rule is illustrated in the following example. Let

$$F \equiv \texttt{functor}\ \mathsf{X} : \mathsf{Mult}.\ \texttt{functor}\ \mathsf{Y} : \mathsf{Mult}.\ \mathsf{X}{\oplus}\mathsf{Y}$$

be a functor. (F *Monoid Monoid*) should contain one sort *ele*, one operator $\circ$, but two copies of the constant e, because the sharing of *ele* and $\circ$ is specified by the argument type Mult, but the sharing of e is not. Therefore, e might be renamed automatically. Since we are not sure if the renaming is necessary at the time of the functor application, we have to propagate the casting information and postpone the renaming to the union operator to resolve it.

Note that, in contrast to the coercion semantics approach of subtyping, it is not necessary to cast functors when it is used to instantiate other functors. The reason is shown in the following artificial example. Assume that functors can be cast.

$$(((L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2}) \to \mathsf{T})\ \texttt{functor}\ \mathsf{X} : (L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1}).\ \mathsf{E})\ (L, \Phi, \Delta)$$

Which type should (L, Φ, Δ) be cast to, $(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})$ or $(L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2})$? The answer should be $(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})$, because $(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})$ specifies how (L, Φ, Δ) should be treated in the body of the functor. (Also note that $L_{\mathsf{T}_1} \subseteq L_{\mathsf{T}_2} \subseteq L$.) In other words, how to rename (L, Φ, Δ) to solve the name conflict depends on the behaviour of the functor, in particular the "real" argument type of the functor, not that expressed in the type of the functor. The type of the functor only specifies which theories can be used for instantiation. Since casting a functor does not change its behaviour, we should not cast functors for the sake of simplicity.

The reason that we do not choose it is that this approach rules out the possibility that, in some cases, we may want to force the result of (*F Monoid Monoid*) to have one copy of the constant e. We choose to put the onus on the user, i.e. the user can manually cast the argument theories into the intended shapes, which is more flexible.

Soundness of the type system. Similar to that of Mei Basic, the soundness of the type system of Mei Core is shown by the progress and preservation theorems as follows:

Theorem 3.3.2. *Let* E *be a well-typed module expression. Then either* $\mathsf{E} \in \mathit{NF}$ *or else there is an* E' *such that* $\mathsf{E} \longrightarrow \mathsf{E}'$.

Proof. By induction on a typing derivation of E. □

Theorem 3.3.3. *If* $\mathsf{E} : \ \mathsf{T}$ *and* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\mathsf{E}' : \ \mathsf{T}$.

Proof. By induction on a typing derivation of E. □

Normalization of well-typed module expressions. The proof of the normalization theorem, Theorem 3.3.9, follows exactly in the same fashion of that of Theorem 2.4.13, i.e. in two steps: (1) construct a set SN of expressions that are normalizable, and (2) show that every well-typed module expression is an element of SN.

The major differences between the normalization proofs in this section and the normalization proofs for Mei Basic are those difficulties introduced by the subtype relation. For instance, given a module expression E of type T, substituting strongly normalized expressions for the free variables in E respecting their types produces a module expression of a subtype of T, not necessarily T, as shown in Lemma 3.3.8.

Lemma 3.3.4. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\mathsf{E}\!\downarrow$ *iff* $\mathsf{E}'\!\downarrow$.

Definition 3.3.5. The set SN of module expressions is defined inductively as follows:

$$\frac{\mathsf{E} : \ (L, \Phi) \quad \mathsf{E}\!\downarrow}{\mathsf{E} \in \ \mathit{SN}_{(L,\Phi)}} \qquad \frac{\forall \mathsf{T_p} <: \mathsf{T}_1.\ \forall \mathsf{E_p} \in \mathit{SN}_{\mathsf{T_p}}.\ \mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2}}{\mathsf{E} \in \mathit{SN}_{\mathsf{T}_1 \to \mathsf{T}_2}}$$

Lemma 3.3.6. *If* $\mathsf{E} \in \mathit{SN}$, *then* $\mathsf{E}\!\downarrow$.

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L, \Phi)$. Directly follows Definition 2.4.9.

(2) $\mathsf{T} \equiv \mathsf{T}_1 \to \mathsf{T}_2$. Let $\mathsf{E_p} \in SN_{\mathsf{T}_1}$, clearly $\mathsf{T}_1 <: \mathsf{T}_1$. By Definition 3.3.5, $\mathsf{E}\ \mathsf{E_p} \in SN_{\mathsf{T}_2}$. By the induction hypothesis, $(\mathsf{E}\ \mathsf{E_p})\!\downarrow$, which implies $\mathsf{E}\!\downarrow$.

□

Lemma 3.3.7. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\mathsf{E} \in SN$ *iff* $\mathsf{E}' \in SN$.

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L, \Phi)$. Directly follows from Lemma 3.3.4 and Definition 3.3.5.

(2) $\mathsf{T} \equiv \mathsf{T}_1 \to \mathsf{T}_2$.

($\Rightarrow$) Let $\mathsf{T_p} <: \mathsf{T}_1$ and $\mathsf{E_p} \in SN_{\mathsf{T_p}}$ be an arbitrary module expression. $\mathsf{E}\ \mathsf{E_p} \in SN_{\mathsf{T}_2}$ by definition of SN. By the induction hypothesis, $\mathsf{E}'\ \mathsf{E_p} \in SN_{\mathsf{T}_2}$. Since the choice of $\mathsf{T_p}$ and $\mathsf{E_p}$ is arbitrary, the definition of SN gives the result.

($\Leftarrow$) Analogous to ($\Rightarrow$).

□

Lemma 3.3.8. *If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$, $\Gamma \equiv \mathsf{X}_1 : \mathsf{T}_1, \ldots, \mathsf{X_n} : \mathsf{T_n}$, $\mathsf{T}_{11} <: \mathsf{T}_1, \ldots, \mathsf{T_{nn}} <: \mathsf{T_n}$, *and* $\mathsf{E}_1 \in SN_{\mathsf{T}_{11}}, \ldots, \mathsf{E_n} \in SN_{\mathsf{T_{nn}}}$, *then* $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in SN_{\mathsf{T_{sub}}}$, *for some* $\mathsf{T_{sub}} <: \mathsf{T}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathsf{E} : \mathsf{T}$. Let $\Gamma \equiv \mathsf{X}_1 : \mathsf{T}_1, \ldots, \mathsf{X_n} : \mathsf{T_n}$ in the following proof.

ASSUMP. $\mathsf{E} \equiv \mathsf{X_i}$ and $\mathsf{T} \equiv \mathsf{T_i}$. Trivial.

CONST. $\mathsf{E} \equiv \mathsf{C}$. Assume $\sigma(\mathsf{C}) = \mathsf{E}'$. Since $\mathsf{C}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \equiv \mathsf{C}$, it is sufficient to show $\mathsf{C} \in SN_{\mathsf{T_{sub}}}$ for some $\mathsf{T_{sub}} <: \mathsf{T}$. Since $\vdash \mathsf{E}' : \mathsf{T}$ is in the premise, by the induction hypothesis, $\mathsf{E}' \in SN_{\mathsf{T_{sub}}}$, for some $\mathsf{T_{sub}} <: \mathsf{T}$. But $\mathsf{C} \longrightarrow \mathsf{E}'$, by Lemma 3.3.7, $\mathsf{C} \in SN_{\mathsf{T_{sub}}}$.

BASIC. $\mathsf{E} \equiv (L_\mathsf{T}, \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta)$. Since E is of theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}]\!\downarrow$. But $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \equiv \mathsf{E}$ is already in normal form.

CAST. $\mathsf{E} \equiv (L, \Phi)\ \mathsf{E}'$. It is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in SN_{(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})}$$

where $(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}}) \subseteq (L', \Phi')$. Assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{aligned} ((L,\Phi)\ \mathsf{E}')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] &= (L,\Phi)\ (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\ &\longrightarrow (L,\Phi)\ (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})) \\ &\longrightarrow (L,\Phi)\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \end{aligned}$$

EXT. $\mathsf{E} \equiv \mathsf{E}'$ `extended by` $(L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})$. It is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})}$$

where $(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}}) \subseteq (L', \Phi')$. Assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{aligned} & (\mathsf{E}'\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}))[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\ = \quad & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}])\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\ \longrightarrow \quad & ((L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\ \longrightarrow \quad & (L'_{\mathsf{sub}} \cup L_\mathsf{S},\ \Phi'_{\mathsf{sub}} \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))\ (L_{\mathsf{E}'} \uplus L_\mathsf{S},\ \Phi_{\mathsf{E}'} \uplus \Phi_\mathsf{S},\ \Delta_{\mathsf{E}'} \uplus \Delta_\mathsf{S}) \end{aligned}$$

Clearly $(L'_{\mathsf{sub}} \cup L_\mathsf{S},\ \Phi'_{\mathsf{sub}} \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S})) <: (L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))$.

UNION. $\mathsf{E} \equiv \mathsf{E}' \oplus \mathsf{E}''$. It is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\!\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$ and $\Gamma \vdash \mathsf{E}'' : (L'', \Phi'')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})}$$

where $(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}}) \subseteq (L', \Phi')$, and

$$\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{(L''_{\mathsf{sub}}, \Phi''_{\mathsf{sub}})}.$$

where $(L''_{\mathsf{sub}}, \Phi''_{\mathsf{sub}}) \subseteq (L'', \Phi'')$. Assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'})$ and $\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L''_{\mathsf{sub}}, \Phi''_{\mathsf{sub}})\ (L_{\mathsf{E}''},\ \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}''})$.

$$\begin{aligned} & (\mathsf{E}' \oplus \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\ = \quad & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \oplus (\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\ \longrightarrow \quad & (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}) \oplus (L''_{\mathsf{sub}}, \Phi''_{\mathsf{sub}})\ (L_{\mathsf{E}''},\ \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}''}) \\ \longrightarrow \quad & (L'_{\mathsf{sub}} \cup L''_{\mathsf{sub}},\ \Phi'_{\mathsf{sub}} \cup \Phi''_{\mathsf{sub}})\ (L_{\mathsf{E}'} \uplus L_{\mathsf{E}''},\ \Phi_{\mathsf{E}'} \uplus \Phi_{\mathsf{E}''},\ \Delta_{\mathsf{E}'} \uplus \Delta_{\mathsf{E}''}) \end{aligned}$$

Clearly, $(L'_{\mathsf{sub}} \cup L''_{\mathsf{sub}},\ \Phi'_{\mathsf{sub}} \cup \Phi''_{\mathsf{sub}}) <: (L' \cup L'',\ \Phi' \cup \Phi'')$.

REN. $\mathsf{E} \equiv \mathsf{E}'\ \mathtt{with}\ \rho$. It is sufficient to prove $\mathsf{E}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\downarrow$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in SN_{(L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})}.$$

where $(L''_{\mathsf{sub}}, \Phi''_{\mathsf{sub}}) \subseteq (L'', \Phi'')$. Assume $\mathsf{E}'[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \longrightarrow (L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rl} & (\mathsf{E}'\ \mathtt{with}\ \rho)[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \\ = & (\mathsf{E}'[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathtt{with}\ \rho \\ \longrightarrow & ((L'_{\mathsf{sub}}, \Phi'_{\mathsf{sub}})\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \mathtt{with}\ \rho \\ \longrightarrow & (\rho(L'_{\mathsf{sub}}),\ \rho(\Phi'_{\mathsf{sub}}))\ (\rho[L_{\mathsf{E}'}],\ \rho[\Phi_{\mathsf{E}'}],\ \rho[\Delta_{\mathsf{E}'}]) \end{array}$$

Clearly, $(\rho(L'_{\mathsf{sub}}),\ \rho(\Phi'_{\mathsf{sub}})) <: (\rho(L'),\ \rho(\Phi'))$.

ABS. $\mathsf{E} \equiv \mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}''$ and $\mathsf{T} \equiv \mathsf{T}' \to \mathsf{T}''$. In order to prove

$$(\mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ \in SN_{\mathsf{T}' \to \mathsf{T}''_{\mathsf{sub}}}$$

where $\mathsf{T}' \to \mathsf{T}''_{\mathsf{sub}} <: \mathsf{T}' \to \mathsf{T}''$ and hence $\mathsf{T}''_{\mathsf{sub}} <: \mathsf{T}''$, it is sufficient to prove

$$(\mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ \mathsf{E}'_{\mathsf{sub}} \in SN_{\mathsf{T}''_{\mathsf{sub}}}$$

for some $\mathsf{T}''_{\mathsf{sub}} <: \mathsf{T}''$, where $\mathsf{T}'_{\mathsf{sub}} <: \mathsf{T}'$ and $\mathsf{E}'_{\mathsf{sub}} : \mathsf{T}'_{\mathsf{sub}}$. Without loss of generality, we assume $\mathsf{X} \not\equiv \mathsf{X_i}$ for $1 \leq \mathsf{i} \leq \mathsf{n}$. We have

$$\begin{array}{rl} & ((\mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E}' \\ = & (\mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}''[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E}' \\ \longrightarrow & \mathsf{E}''[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ [\mathsf{X} := \mathsf{E}']. \end{array}$$

Since $\Gamma, \mathsf{X} : \mathsf{T}' \vdash \mathsf{E}'' : \mathsf{T}''$ is in the premise, by the induction hypothesis,

$$\mathsf{E}''[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ [\mathsf{X} := \mathsf{E}'_{\mathsf{sub}}] \in SN_{\mathsf{T}''_{\mathsf{sub}}}$$

for some $\mathsf{T}''_{\mathsf{sub}} <: \mathsf{T}''$. By lemma 3.3.7,

$$((\mathtt{functor}\ \mathsf{X} : \mathsf{T}'.\ \mathsf{E}'')[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E}'_{\mathsf{sub}} \in SN_{\mathsf{T}''}.$$

APP. $\mathsf{E} \equiv \mathsf{E}' \ \mathsf{E}''$. Assume $\Gamma \vdash \mathsf{E}' : \mathsf{T}'' \to \mathsf{T}$, $\Gamma \vdash \mathsf{E}'' : \mathsf{T}''_{\mathsf{sub}}$, and $\mathsf{T}''_{\mathsf{sub}} <: \mathsf{T}''$ are in the premise. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in \mathit{SN}_{\mathsf{T}''_{\mathsf{sup}} \to \mathsf{T}_{\mathsf{sub}}}$$

and

$$\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in \mathit{SN}_{\mathsf{T}''_{\mathsf{subsub}}},$$

where $\mathsf{T}''_{\mathsf{sup}} \to \mathsf{T}_{\mathsf{sub}} <: \mathsf{T}'' \to \mathsf{T}$ and $\mathsf{T}''_{\mathsf{subsub}} <: \mathsf{T}''_{\mathsf{sub}}$. From $\mathsf{T}''_{\mathsf{sup}} \to \mathsf{T}_{\mathsf{sub}} <: \mathsf{T}'' \to \mathsf{T}$, we have $\mathsf{T}'' <: \mathsf{T}''_{\mathsf{sup}}$ and $\mathsf{T}_{\mathsf{sub}} <: \mathsf{T}$, hence $\mathsf{T}''_{\mathsf{subsub}} <: \mathsf{T}''_{\mathsf{sup}}$ by rule **(TRAN)** of subtyping. (Note that Theorem 3.1.3 allows us to use rule **(TRAN)** when reasoning about the subtype relation.) By Definition 3.3.5,

$$\begin{aligned} & (\mathsf{E}' \ \mathsf{E}'')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\ = \ & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \ (\mathsf{E}''[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\ \in \ & \mathit{SN}_{\mathsf{T}_{\mathsf{sub}}}. \end{aligned}$$

□

Theorem 3.3.9. *If* $\vdash \mathsf{E} : \mathsf{T}$, *then* $E\downarrow$.

Proof. By Lemma 3.3.8, $\mathsf{E} \in \mathit{SN}_{\mathsf{T}_{\mathsf{sub}}}$ for some $\mathsf{T}_{\mathsf{sub}} <: \mathsf{T}$. By Lemma 3.3.6, $\mathsf{E}\downarrow$. □

3.4 Coercion

As discussed in §3.1, the subtyping mechanism allows us to regard an object of one type as an object of its supertype. However, it is limited: (1) Theories specified in different languages are not related to each other. For instance, the theory *Monoid* defined in §2.1.2 (not the theory *Monoid* defined in §2.1.3) is not of type Mult defined in §2.1.3, although they share similar structure. Only after renaming can we regard *Monoid* as of type Mult. (2) Two different axiomatizations of the same mathematical theory may not be related. For instance, let Nat be a theory of natural numbers with the zero element, the successor function, and corresponding axioms. Let NatNat be another theory of natural number with constants zero, one, ..., and functions plus, minus, times, and corresponding (possibly infinitely many) axioms. Though they should be equivalent, neither of them is a subtype of the other in Mei Core. (3) One type is regarded as a subtype of another only in one way since the subtype relation is based purely on syntax. However, in mathematical experience, some theories can be viewed as instances of other theories in more than one way. For instance, a theory of rings can be treated as a theory of monoids in two ways because there are two copies of monoids with respect to addition and multiplication residing in a ring structure.

We thus need a mechanism to relate theories and types similar to a fitting morphisms to overcome these limitations. However, we want to stay within the simple λ-calculus style system, in particular, keep the substitution semantics. This leads to our approach, *coercion*.

When we want to use a theory $T \in \mathsf{T_t}$ in the place where a theory of $\mathsf{T_s}$ is required and $\mathsf{T_t} <: \mathsf{T_s}$ is *not* derivable, we (1) first coerce T to $T' \in \mathsf{T_{tt}}$ such that $\mathsf{T_{tt}} <: \mathsf{T_s}$ is derivable, and then (2) use T' instead of T. The coercion functor is a functor of Mei Core of type $\mathsf{T_t} \to \mathsf{T_{tt}}$. In other words, instead of changing the parameter passing mechanism of functor application, we transfer and put the input theory in a shape that can be accepted by the functor. This is similar in spirit to the coercion semantics of the subtype relation, but more general. The coercion functor cannot be built automatically, since we need to justify why/how the objects in $\mathsf{T_t}$ can be treated as the objects in $\mathsf{T_s}$. The coercion functor can then be built from the *view* from $\mathsf{T_s}$ to $\mathsf{T_t}$ that justifies the treatment of an object of $\mathsf{T_t}$ as an object in $\mathsf{T_s}$. Syntactically, the simplest view is a *theory translation* that maps an expression of one theory (source) to an expression of another theory (target). However, in most cases, we want it to be a theory interpretation that preserves the validity of the sentences (refer to §3.4.1 for the definition of a theory interpretation). Building a view over functor types follows exactly the same way of defining the subtype relation over functor types.

The following example illustrates how to build a coercion functor from a given theory interpretation. The idea can be generalized over functor types straightforwardly.

Example 3.4.1. Let Monoid and Group be the theory types defined as follows:

$$\begin{array}{lll} \mathsf{Monoid} & \equiv & \texttt{language}\ \texttt{sort}\ mm \\ & & \qquad\qquad \texttt{const}\ e : mm \\ & & \qquad\qquad \texttt{func}\ \ \circ : mm^2 \to mm \\ & & \texttt{axioms}\ \ \forall x_1, x_2, x_3 : mm.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\ & & \qquad\qquad \forall x : mm.\ x \circ e = x \wedge e \circ x = x \end{array}$$

$$
\begin{aligned}
\mathsf{Group} \;\equiv\; & \texttt{language}\ \texttt{sort}\ gg\\
& \qquad\qquad \texttt{const}\ e : gg\\
& \qquad\qquad \texttt{func}\ \ \circ : gg^2 \rightarrow gg\\
& \qquad\qquad\qquad {}^{-1} : gg \rightarrow gg\\
& \texttt{axioms}\ \ \forall x_1, x_2, x_3 : gg.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3\\
& \qquad\qquad \forall x : gg.\ x \circ e = x \land e \circ x = x\\
& \qquad\qquad \forall x : gg.\ x \circ x^{-1} = e
\end{aligned}
$$

The trivial interpretation is the mapping $\{mm \mapsto gg, e \mapsto e, \circ \mapsto \circ\}$. The coercion functor that coerces a theory of type Group to that of Monoid can be defined as follows:

$$\texttt{functor}\ \mathsf{X} : \mathsf{Group}.\ \mathsf{X}\ \text{with}\ \{gg \mapsto mm, e \mapsto e, \circ \mapsto \circ,\ {}^{-1} \mapsto\ {}^{-1}\}$$

Although the example is not very interesting in practice, it clearly illustrates the idea of coercion.

3.4.1 Theory interpretation

Theory interpretation[3] is a standard notion in first-order logic, referring to a mapping of expressions of a theory T_1 (source theory) to expressions of a theory T_2 (target theory) and preserves the validity of sentences. It is used to transform theorems and problems from one context to another. In practice, most MMSs are based on some version of higher-order logic. Different notions of theory interpretation need to be defined for different versions of higher-order logic. For instance, if subtyping is admitted, a type in the source theory can be associated with a subtype in the target theory. Moreover, if partial functions are admitted, proof obligations other than those from axioms in the source theory are generated, such as the proof obligation concerning the definability of a function.

As an example, we present here a formalism of theory interpretation in terms of first-order logic. Theory interpretations for most versions of higher-order logics can be derived by lifting the first-order notion and adding specific mechanisms.

Definition 3.4.2. A *translation*, from a (source) theory T_1 to another (target) theory T_2, is a triple (T_1, T_2, ρ) where ρ is a symbol mapping consists of four functions ρ_s, ρ_c, ρ_f, ρ_p, such that:

[3] This subsection is adapted, almost literally, from [34].

1. ρ_s maps each type symbol in the language of T_1 to a type symbol in T_2 injectively.

2. ρ_c [ρ_f, ρ_p, respectively] maps each constant symbol [function symbol, predicate symbol] in the language of T_1 to a constant symbol [function symbol, predicate symbol] or a closed term [functional expression, predicate expression] in T_2 respecting its type (and arity).

Assume that we have the following term constructors:

$$\mathsf{t} ::= \mathsf{x} \mid \mathsf{c} \mid \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

where x is a variable, c is a constant, and f is an n-ary function symbol; and the following formula constructors:

$$\varphi ::= \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \mid \forall \mathsf{x}.\ \varphi \mid \exists \mathsf{x}.\ \varphi$$

where p is an n-ary predicate symbol.

Definition 3.4.3. An expression φ of T_1 is then translated to an expression, written $\Phi(\varphi)$, in T_2 via the translation $\Phi \equiv (T_1, T_2, \rho)$. It is inductively defined on the structure of terms and formulas:

1. $\Phi(\mathsf{x}) = \mathsf{x}$.

2. $\Phi(\mathsf{c}) = \rho(\mathsf{c})$.

3. $\Phi(\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n)) = \rho(\mathsf{f})(\Phi(\mathsf{t}_1), \ldots, \Phi(\mathsf{t}_n))$.

4. $\Phi(\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)) = \rho(\mathsf{p})(\Phi(\mathsf{t}_1), \ldots, \Phi(\mathsf{t}_n))$.

5. $\Phi(\varphi_1 \diamond \varphi_2) = \Phi(\varphi_1) \diamond \Phi(\varphi_2)$, where $\diamond$ is $\wedge, \vee, \Rightarrow$, or $\Leftrightarrow$.

6. $\Phi(\nabla \mathsf{x}.\ \varphi) = \nabla \mathsf{x}.\ \Phi(\varphi)$, where ∇ is $\forall$ or $\exists$.

Definition 3.4.4. A translation is an interpretation if $\Phi(\varphi)$ is a theorem in T_2 for each theorem φ of T_1.

Theorem 3.4.5. *A translation is an* interpretation *if each of the obligations, i.e. the translations of the axioms of T_1, is provable in T_2.*

3.4.2 Views and coercions

The syntax and rules of Mei extend those of Mei Core by adding a new syntactic class, *view*, and a new kind of module expressions, functor application with views.

Syntax and rules for views. The notion of a *view* is a generalization of both a theory interpretation (a view can be defined over functor types) and a subtype relation (a symbol mapping is involved in a view). A view from $\mathsf{T_s}$ to $\mathsf{T_t}$ shows explicitly how an object in $\mathsf{T_t}$ can be treated as an object in $\mathsf{T_s}$. The simplest views are the views between theory types, which are *theory translations* that map the expressions of one theory (the source) to the expressions of the other theory (the target). Syntactically, it is a triple $(\mathsf{T_s}, \mathsf{T_t}, \rho)$, in which $\mathsf{T_s}$ and $\mathsf{T_t}$ are the source and target theory types respectively and ρ is a *mapping* that maps symbols in $\mathsf{T_s}$ to those in $\mathsf{T_t}$. Views between functor types are defined in terms of the views between their source and target types inductively.

$$\texttt{VIEW} ::= (\texttt{TYPE, TYPE, MAPPINGS})$$

$$\texttt{MAPPINGS} ::= \texttt{MAPPING} \mid (\texttt{MAPPINGS, MAPPINGS})$$

view(V) is read as "V is a view object". Views can be derived by the following syntactic rules:

$$\frac{\mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L_{\mathsf{s}} \quad \mathbf{target}(\rho) \subseteq L_{\mathsf{t}}}{\mathbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho)} \qquad \textbf{(THY-VIEW)}$$

$$\frac{\mathbf{view}(\mathsf{T_{t_1}}, \mathsf{T_{s_1}}, \rho_1) \quad \mathbf{view}(\mathsf{T_{s_2}}, \mathsf{T_{t_2}}, \rho_2)}{\mathbf{view}(\mathsf{T_{s_1}} \to \mathsf{T_{s_2}}, \mathsf{T_{t_1}} \to \mathsf{T_{t_2}}, (\rho_1, \rho_2))} \qquad \textbf{(FUNC-VIEW)}$$

$$\frac{\mathbf{view}(\mathsf{T}_1, \mathsf{T}_2, \rho_1) \quad \mathbf{view}(\mathsf{T}_2, \mathsf{T}_3, \rho_2)}{\mathbf{view}(\mathsf{T}_1, \mathsf{T}_3, \rho_1 \circ \rho_2)} \qquad \textbf{(COMP-VIEW)}$$

Remark 3.4.6. Syntactically, a theory view is not necessarily a theory interpretation, i.e. it need not to be meaning preserving. However, a *good* theory view should be a theory interpretation, but this is not enforced by Mei. The underlying MMS system is responsible for guaranteeing that a view is indeed an interpretation, i.e. for providing the proofs for the obligations. In other words, the view rules are syntactically decidable.

Remark 3.4.7. It is interesting to compare our notion of a view with P. Wadler's notion [93]. Although both provide a way to treat an object of one type as an object of another type, there are a number of differences:

(1) Our views specify homomorphisms between module types, whereas Wadler's views specify isomorphisms between data types.

(2) In Wadler's views, there is a bijection between the viewing type and a subset of the viewed type. Reasoning over these two types can thus be mixed via a pair of functions representing the bijection. However, the purpose of our views is to match as many "reasonable" module expressions as possible with an argument type of a functor. It is thus naturally unidirectional.

(3) Our views are also defined for functor types, which is not found elsewhere.

(4) Our views for theory types can also be seen as a formalization of adapter patterns [42] from object-oriented design in our module system, where the mapping ρ plays the role of adapter methods within an adapter class.

Core, inner and outer extension of types. In §2.1.3 we defined the extension of a theory type. Now, we extend this idea to include functor types and distinguish three different kinds of extensions.

Definition 3.4.8. Let T be a module type. The core extension of T is defined inductively as follows:

(*a*) If $\mathsf{T} \equiv (L_{\mathsf{T}}, \Phi_{\mathsf{T}})$, then $\mathbf{Ext_{core}}(\mathsf{T}) = \{(L, \Phi, \Delta) \mid L_{\mathsf{T}} \subseteq L \text{ and } \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta))\}$.

(*b*) If $\mathsf{T} \equiv \mathsf{T}_1 \to \mathsf{T}_2$, then $\mathbf{Ext_{core}}(\mathsf{T}) = \{F \mid F \in \mathbf{Ext_{core}}(\mathsf{T}_1) \to \mathbf{Ext_{core}}(\mathsf{T}_2)\}$.

Definition 3.4.9. Let T be a module type. The inner extension of T is defined by reference to its core extension and subtype relations.

$$\mathbf{Ext_{inner}}(\mathsf{T}) = \{Obj \in \mathbf{Ext_{core}}(\mathsf{T_{sub}}) \mid \mathsf{T_{sub}} <: \mathsf{T}\}.$$

For a theory type, its core extension and inner extension coincide.

Definition 3.4.10. Let T be a module type. The outer extension of T is defined by reference to its core extension and views.

$$\mathbf{Ext_{outer}}(\mathsf{T}) = \{Obj \in \mathbf{Ext_{core}}(\mathsf{T_{sub}}) \mid \exists\, \rho.\ \mathbf{view}(\mathsf{T}, \mathsf{T_{sub}}, \rho)\}.$$

Remarks 3.4.11.

(*a*) The core extension of a type is handled by the module expression typing rules.

(*b*) The inner extension of a type is handled by the subtype rules.

Now we need a way to treat an object in the outer extension of a type as an object in either its inner or core extension. This is illustrated in Figure 3.2. We need a coercion functor that transfers objects of the outer extension to objects of the inner extension, which are equivalent in terms of being mutually viewable (i.e. mutually interpretable for theories). A functor should be automatically constructed from a given view.

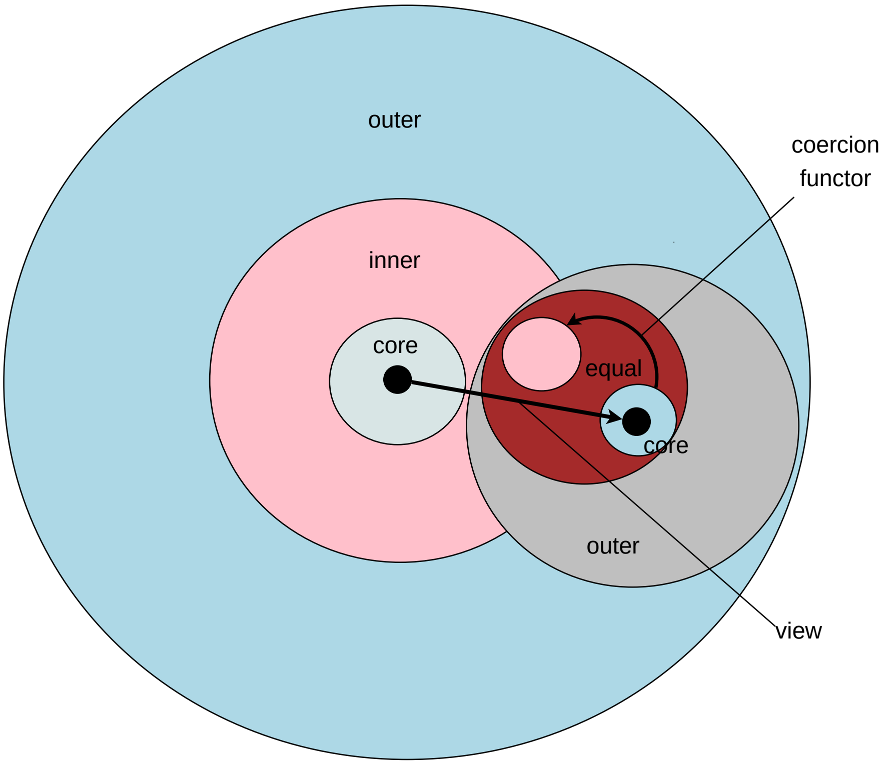

Figure 3.2: Core, inner, and outer extensions

Coercion semantics of views. The intention of both the views and the coercion functors is to provide a way to regard one theory (or functor) as another theory (or functor). It is then natural to define the semantics of a view as a coercion functor.

Intuitively, if $\mathsf{V} \equiv (\mathsf{T_s}, \mathsf{T_t}, \rho)$ is a view from $\mathsf{T_s}$ to $\mathsf{T_t}$, then its semantics, denoted by $[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T_t} \rightarrow \mathsf{T_{tt}}$, is a functor in Mei Core, called a coercion functor, that converts an object in $\mathsf{T_t}$ to an object in $\mathsf{T_{tt}}$, where $\mathsf{T_{tt}} <: \mathsf{T_s}$. For the sake of simplicity, at this moment, we assume that a theory view is one-to-one and maps a symbol in the source theory to a symbol (not an expression) in the target theory. We will discuss more general cases in §3.4.4.

(1) **Theory view.** $\mathsf{V} \equiv ((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho)$.

Let $\rho' = \mathsf{lift}(\rho^{-1}, L_{\mathsf{t}})$ be the function that adds to ρ^{-1} the homomorphic identity maps for the symbols in L_{t} but not in the range set of ρ. Then

$$[\![\mathsf{V}]\!]_{\mathsf{v}} : (L_{\mathsf{t}}, \Phi_{\mathsf{t}}) \rightarrow (L_{\mathsf{tt}}, \Phi_{\mathsf{tt}})$$

is the functor:

$$\texttt{functor}\ \mathsf{X} : (L_{\mathsf{t}}, \Phi_{\mathsf{t}}).\ (\mathsf{X}\ \texttt{extended by}\ (L_\emptyset,\ \Phi_\emptyset,\ \Delta_{\mathsf{obl}}))\ \texttt{with}\ \rho',$$

where $L_\emptyset$ is the empty language, $\Phi_\emptyset$ is the empty axiom set, and Δ_{obl} is the set of obligations generated by the view V.[4] The obligations are the images of axioms of the source theory of V via its mapping ρ, i.e. $\rho(\Phi_{\mathsf{s}})$. Clearly, $L_{\mathsf{s}} \subseteq L_{\mathsf{tt}}$, since ρ is one-to-one. $\Phi_{\mathsf{s}} \subseteq \Phi_{\mathsf{tt}}$, since the obligations are added. Thus $(L_{\mathsf{tt}}, \Phi_{\mathsf{tt}}) <: (L_{\mathsf{s}}, \Phi_{\mathsf{s}})$. $[\![\mathsf{V}]\!]_{\mathsf{v}}$ coerces a theory X of type $(L_{\mathsf{t}}, \Phi_{\mathsf{t}})$ to a theory of type $(L_{\mathsf{tt}}, \Phi_{\mathsf{tt}})$ by (1) adding the proof obligations to the theory showing that the target theory can be regarded as the source theory, and (2) translating the theory according to the view ρ. Note that it is not necessary to cast it to type $(L_{\mathsf{s}}, \Phi_{\mathsf{s}})$, since that can be handled by the subtype relation.

(2) **Functor view.** $\mathsf{V} \equiv (\mathsf{T_{s_1}} \rightarrow \mathsf{T_{s_2}}, \mathsf{T_{t_1}} \rightarrow \mathsf{T_{t_2}}, (\rho_1, \rho_2))$.

Let $c_1 = [\![\mathsf{T_{t_1}}, \mathsf{T_{s_1}}, \rho_1]\!]_{\mathsf{v}}$ be the coercion functor of type $\mathsf{T_{s_1}} \rightarrow \mathsf{T_{ss_1}}$ where $\mathsf{T_{ss_1}} <: \mathsf{T_{t_1}}$ and $c_2 = [\![\mathsf{T_{s_2}}, \mathsf{T_{t_2}}, \rho_2]\!]_{\mathsf{v}}$ be the coercion functor of type $\mathsf{T_{t_2}} \rightarrow \mathsf{T_{tt_2}}$ where $\mathsf{T_{tt_2}} <: \mathsf{T_{s_2}}$. Then

$$[\![\mathsf{V}]\!]_{\mathsf{v}} : (\mathsf{T_{t_1}} \rightarrow \mathsf{T_{t_2}}) \rightarrow (\mathsf{T_{s_1}} \rightarrow \mathsf{T_{tt_2}})$$

[4] In the case when the theory view is indeed a theory interpretation where each obligation is discharged by a proof, the coercion functor will produce a theory equivalent to the original theory. However, if the theory view cannot be proved to be a theory interpretation, the coerce functor will produce a theory which may not be equivalent to the original theory. This means that our approach guarantees a correct result only if the theory view is a theory interpretation. Otherwise it will give a syntactically correct but semantically wrong result.

is the functor:

$$\texttt{functor}\ \mathsf{F} : \mathsf{T}_{\mathsf{t}_1} \rightarrow \mathsf{T}_{\mathsf{t}_2}.\ \texttt{functor}\ \mathsf{X} : \mathsf{T}_{\mathsf{s}_1}.\ c_2\ (\mathsf{F}\ (c_1\ \mathsf{X}))$$

Notice that, as in the subtype relation of function types, the view component for the argument type is reversed (contravariant), while that for the result type has the same direction (covariant) as for the functor view. It follows the subtype relation since this is a generalized subtype relation. A functor in $\mathsf{T}_{\mathsf{t}_1} \rightarrow \mathsf{T}_{\mathsf{t}_2}$ is coerced to (or viewed as) a functor in $\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{tt}_2}$, which is an element of the inner extension of $\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{s}_2}$ by **(FUNC-SUB)**. In some sense, $\mathsf{T}_{\mathsf{t}_1} \rightarrow \mathsf{T}_{\mathsf{t}_2}$ is a "subtype" of $\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{s}_2}$, since it accepts an object of $\mathsf{T}_{\mathsf{s}_1}$ that can be coerced to an object of $\mathsf{T}_{\mathsf{t}_1}$ as input, and returns an object in $\mathsf{T}_{\mathsf{t}_2}$ that can be coerced to an object in $\mathsf{T}_{\mathsf{tt}_2}$, which in turn is an inner object of $\mathsf{T}_{\mathsf{s}_2}$.

(3) **View composition.** $\mathsf{V} \equiv (\mathsf{T}_1 \rightarrow \mathsf{T}_3, (\rho_1 \circ \rho_2))$, $\mathsf{V}_1 \equiv (\mathsf{T}_1 \rightarrow \mathsf{T}_2, \rho_1)$, and $\mathsf{V}_2 \equiv (\mathsf{T}_2 \rightarrow \mathsf{T}_3, \rho_2)$.

Let $[\![\mathsf{V}_1]\!]_{\mathsf{v}}$ be the coercion functor of type $\mathsf{T}_2 \rightarrow \mathsf{T}_{22}$ where $\mathsf{T}_{22} <: \mathsf{T}_1$ and $[\![\mathsf{V}_2]\!]_{\mathsf{v}}$ be the coercion functor of type $\mathsf{T}_3 \rightarrow \mathsf{T}_{33}$ where $\mathsf{T}_{33} <: \mathsf{T}_2$. Then

$$[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T}_3 \rightarrow \mathsf{T}_{22}$$

is the functor:

$$\texttt{functor}\ \mathsf{X} : \mathsf{T}_3.\ [\![\mathsf{V}_1]\!]_{\mathsf{v}}\ ([\![\mathsf{V}_2]\!]_{\mathsf{v}}\ \mathsf{X}).$$

Again $[\![\mathsf{V}]\!]_{\mathsf{v}}$ transfers an outer object of T_1 to an inner object, since $\mathsf{T}_{22} <: \mathsf{T}_1$.

Proposition 3.4.12. *If* V *is a view, then* $[\![\mathsf{V}]\!]_{\mathsf{v}}$ *is a functor in Mei Core.*

Proof. Straightforward induction on the definition of $[\![\ ]\!]_{\mathsf{v}}$. □

Proposition 3.4.13. *If* $\mathsf{V} \equiv (\mathsf{T}_{\mathsf{s}}, \mathsf{T}_{\mathsf{t}}, \rho)$ *is a view, then* $[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T}_{\mathsf{t}} \rightarrow \mathsf{T}_{\mathsf{tt}}$ *is a functor, where* $\mathsf{T}_{\mathsf{tt}} <: \mathsf{T}_{\mathsf{s}}$.

Proof. Straightforward induction on the definition of $[\![\ ]\!]_{\mathsf{v}}$. □

Algorithmic views. Similar to the subtyping rules in §3.1.1, the rules for views defined above are not syntax-directed, and thus cannot give a view checking algorithm directly even when the view is given in the conclusion. The problem is the rule **(COMP-VIEW)**. Since both T_1 and T_3 are metavariables, they can match any types. Also T_2 is in both premises of **(COMP-VIEW)** but not in the conclusion, and we thus have to *guess* a value for it, and check the premises and there is little chance of success. Similarly to the elimination of rule **(TRAN)** (Theorem 3.1.3), we might eliminate rule **(COMP-VIEW)** if we can prove the following theorem:

Theorem 3.4.14. *If we can derive* $\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}_\mathsf{t}, \rho)$*, we can derive it without using rule* **(COMP-VIEW)**.

Proof. The proof resembles that of Theorem 3.1.3. Let us consider cases of the last step of the derivation.

(1) Case **(THY-VIEW)** trivial.

(2) Case **(FUNC-VIEW)** $\mathsf{T}_\mathsf{s} \equiv \mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}$, $\mathsf{T}_\mathsf{t} \equiv \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}$, and $\rho \equiv (\rho_1, \rho_2)$.

$$\frac{\mathbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_1) \quad \mathbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_2)}{\mathbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_1, \rho_2))}$$

Since the sizes of both $\mathbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_1)$ and $\mathbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_2)$ are strictly smaller than that of $\mathbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_1, \rho_2))$, the latter can be derived without using rule **(COMP-VIEW)** by the induction hypothesis.

(3) Case **(COMP-VIEW)**

$$\frac{\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}, \rho_1) \quad \mathbf{view}(\mathsf{T}, \mathsf{T}_\mathsf{t}, \rho_2)}{\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}_\mathsf{t}, \rho_1 \circ \rho_2)}$$

By the induction hypothesis, we can assume that both $\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}, \rho_1)$ and $\mathbf{view}(\mathsf{T}, \mathsf{T}_\mathsf{t}, \rho_2)$ can be derived without using rule **(COMP-VIEW)**. There are thus two subcases for the derivation of $\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}, \rho_1)$ and $\mathbf{view}(\mathsf{T}, \mathsf{T}_\mathsf{t}, \rho_2)$ as follows:

(*i*) $\mathsf{T}_\mathsf{s} \equiv (L_\mathsf{s}, \Phi_\mathsf{s})$, $\mathsf{T}_\mathsf{t} \equiv (L_\mathsf{t}, \Phi_\mathsf{t})$, and $\mathsf{T} \equiv (L, \Phi)$.

$$\frac{\dfrac{\mathbf{map}(\rho_1) \quad \mathbf{source}(\rho_1) = L_\mathsf{s} \quad \ldots}{\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}, \rho_1)} \quad \dfrac{\mathbf{map}(\rho_2) \quad \mathbf{source}(\rho_2) = L \quad \ldots}{\mathbf{view}(\mathsf{T}, \mathsf{T}_\mathsf{t}, \rho_2)}}{\mathbf{view}(\mathsf{T}_\mathsf{s}, \mathsf{T}_\mathsf{t}, \rho_1 \circ \rho_2)}$$

can be replaced by

$$\frac{\textbf{map}(\rho_1 \circ \rho_2) \quad \textbf{source}(\rho_1 \circ \rho_2) = L_{\mathsf{s}} \quad \textbf{target}(\rho_1 \circ \rho_2) \subseteq L_{\mathsf{t}}}{\textbf{view}(\mathsf{T}_{\mathsf{s}}, \mathsf{T}_{\mathsf{t}}, \rho_1 \circ \rho_2)}.$$

Since the composition of two mappings is still a mapping, $\textbf{map}(\rho_1 \circ \rho_2)$ can be derived from $\textbf{map}(\rho_1)$ and $\textbf{map}(\rho_2)$.

(ii) $\mathsf{T}_{\mathsf{s}} \equiv \mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}$, $\mathsf{T}_{\mathsf{t}} \equiv \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}$, and $\mathsf{T} \equiv \mathsf{T}_1 \to \mathsf{T}_2$.

The derivation

$$\frac{\dfrac{\textbf{view}(\mathsf{T}_1, \mathsf{T}_{\mathsf{s}_1}, \rho_{\mathsf{s}_1}) \quad \textbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_2, \rho_{\mathsf{s}_2})}{\textbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_1 \to \mathsf{T}_2, (\rho_{\mathsf{s}_1}, \rho_{\mathsf{s}_2}))} \quad \dfrac{\textbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_1, \rho_{\mathsf{t}_1}) \quad \textbf{view}(\mathsf{T}_2, \mathsf{T}_{\mathsf{t}_2}, \rho_{\mathsf{t}_2})}{\textbf{view}(\mathsf{T}_1 \to \mathsf{T}_2, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_{\mathsf{t}_1}, \rho_{\mathsf{t}_2}))}}{\textbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_{\mathsf{t}_1} \circ \rho_{\mathsf{s}_1}, \rho_{\mathsf{s}_2} \circ \rho_{\mathsf{t}_2}))}$$

can be reorganized as the following derivation

$$\frac{\dfrac{\textbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_1, \rho_{\mathsf{t}_1}) \quad \textbf{view}(\mathsf{T}_1, \mathsf{T}_{\mathsf{s}_1}, \rho_{\mathsf{s}_1})}{\textbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_{\mathsf{t}_1} \circ \rho_{\mathsf{s}_1})} \quad \dfrac{\textbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_2, \rho_{\mathsf{s}_2}) \quad \textbf{view}(\mathsf{T}_2, \mathsf{T}_{\mathsf{t}_2}, \rho_{\mathsf{t}_2})}{\textbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_{\mathsf{s}_2} \circ \rho_{\mathsf{t}_2})}}{\textbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_{\mathsf{t}_1} \circ \rho_{\mathsf{s}_1}, \rho_{\mathsf{s}_2} \circ \rho_{\mathsf{t}_2}))}.$$

Again, since the sizes of both $\textbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_{\mathsf{t}_1} \circ \rho_{\mathsf{s}_1})$ and $\textbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_{\mathsf{s}_2} \circ \rho_{\mathsf{t}_2})$ are strictly smaller than that of $\textbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_{\mathsf{t}_1} \circ \rho_{\mathsf{s}_1}, \rho_{\mathsf{s}_2} \circ \rho_{\mathsf{t}_2}))$, the latter can be derived without using rule **(COMP-VIEW)** by the induction hypothesis,.

□

Theorem 3.4.14 shows that the algorithmic rules for views are both sound and complete with respect to the original rules with **(COMP-VIEW)**. Also notice that the algorithmic rules are syntax-directed and checking a view relation can always be reduced to checking smaller view relations. This justifies the following proposition:

Proposition 3.4.15. *For algorithmic view rules, view checking is decidable.*

Proof. Straightforward induction on the derivation of views, based on the fact that **is_view** is syntactically decidable. □

Note that "good view" checking is generally not decidable since it requires proving the obligations.

View constructions. We can see that the cost of the justification of a view is far more than that of a subtype relation. For instance, a symbol mapping is needed and it may be necessary to prove the mapping actually gives an interpretation when the module system is integrated with a particular MMS. It is thus worthwhile to save a view for later use, for example, name it and then put it in an environment.

In this case, the **(COMP-VIEW)** rule is useful in that it allows us to construct new views from existing views. The metavariables in the premise are not problematic, since the rule is used forwardly in the construction, not backwardly as in the derivation. We are not trying to find all possible compositions. Instead we are trying to build a new view by composing *existing* views. The construction rules are shown as follows:

$$\frac{\mathbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho) \qquad \rho' = \mathsf{lift}(\rho, L_{\mathsf{E}})}{\mathbf{view}((L_{\mathsf{s}} \cup L_{\mathsf{E}}, \Phi_{\mathsf{s}} \cup \Phi_{\mathsf{E}}), (L_{\mathsf{t}} \cup \rho'(L_{\mathsf{E}}), \Phi_{\mathsf{t}} \cup \rho'(\Phi_{\mathsf{E}})), \rho')} \quad \textbf{(EXT-VIEW)}$$

$$\frac{\mathbf{view}((L_{\mathsf{s}_1}, \Phi_{\mathsf{s}_1}), (L_{\mathsf{t}_1}, \Phi_{\mathsf{t}_1}), \rho_1) \quad \mathbf{view}((L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_2}), \rho_2) \quad \mathbf{consist}(\rho_1, \rho_2)}{\mathbf{view}((L_{\mathsf{s}_1} \cup L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_1} \cup \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_1} \cup L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_1} \cup \Phi_{\mathsf{t}_2}), \rho_1 \cup \rho_2)} \quad \textbf{(UNI-VIEW)}$$

$$\frac{\mathbf{view}((L_1, \Phi_1), (L_2, \Phi_2), \rho_1) \quad \mathbf{view}((L_2, \Phi_2), (L_3, \Phi_3), \rho_2)}{\mathbf{view}((L_1, \Phi_1), (L_3, \Phi_3), \rho_1 \circ \rho_2)} \quad \textbf{(COMP-VIEW)}$$

The function lift is a function that adds the homomorphic identity maps for the symbols declared in L_{E} to the given mapping. $\mathbf{consist}(\rho_1, \rho_2)$ means that ρ_1 and ρ_2 must be consistent with respect to $L_{\mathsf{s}_1} \cap L_{\mathsf{s}_2}$, i.e. for each symbol $\mathsf{x} \in L_{\mathsf{s}_1} \cap L_{\mathsf{s}_2}$, $\rho_1(\mathsf{x}) = \rho_2(\mathsf{x})$. The first rule asserts that, given a theory view, the lifted mapping gives a theory view between certain extensions of the source and target theories. The second rule asserts that, given two theory views, the "union" of them is also a theory view. The third rule is the composition rule based on the fact that the composition of two theory interpretations is an interpretation. $\circ$ represents the composition operator for binary relations. The soundness of these three rules is trivial.

These construction rules are not used in the view derivation; only the results of the construction are used. Before we start a view derivation, we first match it with the constructed views. If the matching succeeds, it is a view. In this sense, the constructed views serve the role of axioms in the logic. The soundness of the construction rules guarantees that the constructed views are views.

3.4.3 Extended module expressions

We can then extend the module expressions by a new form of functor application:

`EXPR EXPR` with view `VIEW`

The typing rule is defined by:

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T}_1 \to \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} : \mathsf{T_p} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)}{\Gamma \vdash (\mathsf{E_f}\ \mathsf{E_p} \text{ with view } (\mathsf{T}_1, \mathsf{T_p}, \rho)) : \mathsf{T}_2} \qquad \textbf{(APP-VIEW)}$$

3.4.4 Coercion functor for general theory views

In general, a view between two theory types is not necessarily one-to-one and it may map a symbol in the source theory to an expression in the target theory. However, we can always translate a general view to a restricted view. We illustrate this approach with the following examples:

Example 3.4.16. Let $\mathsf{V} \equiv ((L_\mathsf{s}, \Phi_\mathsf{s}), (L_\mathsf{t}, \Phi_\mathsf{t}), \rho)$ be a view. Assume ρ is not one-to-one such that $\rho(\mathsf{f_s}) = \mathsf{f_t}$ and $\rho(\mathsf{g_s}) = \mathsf{f_t}$, where $\mathsf{f_s}, \mathsf{g_s} \in L_\mathsf{s}$ are distinct. Let $\mathsf{g_t} \notin L_\mathsf{t}$ be a fresh symbol. We can define $[\![\mathsf{V}]\!]_\mathsf{v} : (L_\mathsf{t}, \Phi_\mathsf{t}) \to (L_\mathsf{tt}, \Phi_\mathsf{tt})$ as the functor

$$\texttt{functor}\ \mathsf{X} : (L_\mathsf{t}, \Phi_\mathsf{t}).\ (\mathsf{X}\ \texttt{extended by}\ (\{\mathsf{g_t}\},\ \{\varphi_\mathsf{t}\},\ \Delta_\mathsf{obl}))\ \texttt{with}\ \mathsf{lift}(\rho'^{-1}),$$

where φ_t is an axiom asserting that $\mathsf{f_t} = \mathsf{g_t}$ and ρ' is the mapping

$$\rho'(\mathsf{x}) = \begin{cases} \rho(\mathsf{x}) & \text{if } \mathsf{x} \not\equiv \mathsf{g_s} \\ \mathsf{g_t} & \text{otherwise,} \end{cases}$$

Essentially, a dummy symbol is added to the parameter theory to be used as the image symbol.

Example 3.4.17. Let $\mathsf{V} \equiv ((L_\mathsf{s}, \Phi_\mathsf{s}), (L_\mathsf{t}, \Phi_\mathsf{t}), \rho)$ be a view. Assume that $\rho(\mathsf{f_s}) = \mathsf{expr_t}$, where $\mathsf{expr_t}$ is an expression of L_t. Let $\mathsf{f_t} \notin L_\mathsf{t}$ be a fresh symbol. We can define $[\![\mathsf{V}]\!]_\mathsf{v} : (L_\mathsf{t}, \Phi_\mathsf{t}) \to (L_\mathsf{tt}, \Phi_\mathsf{tt})$ as the functor

$$\texttt{functor}\ \mathsf{X} : (L_\mathsf{t}, \Phi_\mathsf{t}).\ (\mathsf{X}\ \texttt{extended by}\ (\{\mathsf{f_t}\},\ \{\varphi_\mathsf{t}\},\ \Delta_\mathsf{obl}))\ \texttt{with}\ \mathsf{lift}(\rho'^{-1}),$$

where φ_t is an axiom asserting that $\mathsf{f_t} = \mathsf{expr_t}$ and ρ' is the mapping

$$\rho'(\mathsf{x}) = \begin{cases} \rho(\mathsf{x}) & \text{if } \mathsf{x} \not\equiv \mathsf{f_s} \\ \mathsf{f_t} & \text{otherwise.} \end{cases}$$

Again a dummy symbol is added to the parameter theory to be used as the image symbol.

The mechanisms shown in the above two examples can be combined to generate coercion functors automatically from the given views for theory types. It is then straightforward to define the coercion functors over functor types inductively based on those over theory types.

3.4.5 Functor instantiation

With the help of coercion functors, the instantiation of functors can accept more objects as input, as shown in Figure 3.3.

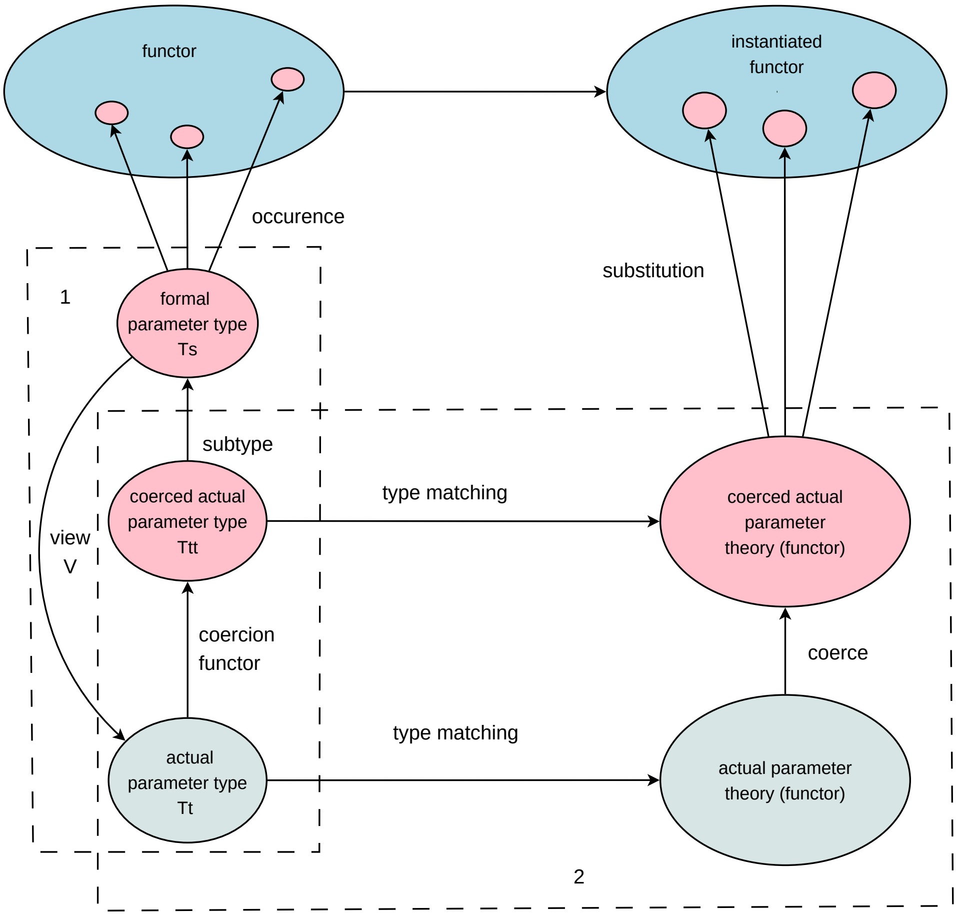

Figure 3.3: Functor Application with Coercion

Remark 3.4.18.

(1) By Proposition 3.4.13, if $\mathsf{V} \equiv (\mathsf{T_s}, \mathsf{T_t}, \rho)$ is a view, then $[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T_t} \to \mathsf{T_{tt}}$ is a functor where $\mathsf{T_{tt}} <: \mathsf{T_s}$. Thus rectangle 1 in Figure 3.3 commutes.

(2) Since V is a functor of type $\mathsf{T_t} \to \mathsf{T_{tt}}$, by the typing rule for functor application, rectangle 2 in Figure 3.3 commutes.

Note that, by using coercion functors, we keep our module system as simple as the simply typed λ-calculus, while supporting a very general mechanism for higher-order functors. The steps shown in Figure 3.3 are syntactically decidable.

The following example illustrates a functor application with views:

Example 3.4.19. Group and Set are two theory types and *GroupAct* is a functor representing a generic group action theory.

$$\begin{aligned}
\mathsf{Group} \;\equiv\; & \texttt{language sort } gg \\
& \qquad \texttt{const } e : gg \\
& \qquad \texttt{func } \circ : gg^2 \to gg \\
& \qquad\qquad {}^{-1} : gg \to gg \\
& \texttt{axioms } \forall x_1, x_2, x_3 : gg.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\
& \qquad \forall x : gg.\ x \circ e = x \wedge e \circ x = x \\
& \qquad \forall x : gg.\ x \circ x^{-1} = e = x^{-1} \circ x
\end{aligned}$$

$$\begin{aligned}
\mathsf{Set} \;\equiv\; & \texttt{language sort } ss \\
& \texttt{axioms } \ldots \\
\mathit{GroupAct} \;\equiv\; & \texttt{functor } \mathsf{X} :\ \mathsf{Group}.\ \texttt{functor } \mathsf{Y} :\ \mathsf{Set}.\ (\mathsf{X} \oplus \mathsf{Y})\ \texttt{extended by} \\
& \texttt{language func } \mathit{act} : gg \times ss \to ss. \\
& \texttt{axioms } \ \forall x : ss.\ \mathit{act}(e, x) = x \\
& \qquad \forall g, h : gg, x : ss.\ \mathit{act}(g \circ h, x) = \mathit{act}(g, \mathit{act}(h, x))
\end{aligned}$$

Let *Ring* be the ring theory defined in Example 2.1.13. We can partially instantiate *GroupAct* by *Ring* with a view as follows:

$$\mathit{GroupAct}\ \mathit{Ring}\ \texttt{with view}\ (\mathsf{Group}, \mathsf{Ring}, \{gg \mapsto rr,\ e \mapsto 0, \circ \mapsto +\})$$

where Ring is the type of *Ring*. The resulting functor is trivial. The corresponding coercion functor is

$$\texttt{functor } \mathsf{X} :\ \mathsf{Ring}.\ \mathsf{X}\ \texttt{with}\ \{rr \mapsto gg,\ 0 \mapsto e, + \mapsto \circ, 1 \mapsto 1, * \mapsto *\}$$

3.5 Syntax of Mei

The syntax and rules of Mei extend those of Mei Core by adding new syntactic classes: *view* and *mapping*, and a new module expression: functor application with views. This subsection presents the concrete syntax and rules of Mei as a whole. The parts that are different from Mei Core are boxed.

$$
\begin{array}{lll}
\texttt{EXPR} & ::= & \texttt{MOD-CONST} \\
 & \mid & \texttt{TYPE-SPEC THY-SPEC} \\
 & \mid & \texttt{TYPE-SPEC EXPR} \\
 & \mid & \texttt{EXPR extended by SPEC} \\
 & \mid & \texttt{EXPR } +y \texttt{ EXPR} \\
 & \mid & \texttt{EXPR with MAPPING} \\
 & \mid & \texttt{functor VAR : TYPE. EXPR} \\
 & \mid & \texttt{EXPR EXPR} \\
 & \mid & \boxed{\texttt{EXPR EXPR}\text{ with view }\texttt{VIEW}}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{TYPE} & ::= & \texttt{TYPE-CONST} \\
 & \mid & \texttt{TYPE-SPEC} \\
 & \mid & \texttt{TYPE} \to \texttt{TYPE}
\end{array}
$$

$$\texttt{THY-SPEC} ::= (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS})$$

$$\texttt{TYPE-SPEC} ::= (\texttt{LANG}, \texttt{AXIOMS})$$

$$\boxed{\texttt{VIEW} ::= (\texttt{TYPE, TYPE, MAPPINGS})}$$

$$\boxed{\texttt{MAPPINGS} ::= \texttt{MAPPING} \mid (\texttt{MAPPINGS, MAPPINGS})}$$

$$\texttt{MOD-CONST} ::= \texttt{IDENTIFIER}$$

$$\texttt{TYPE-CONST} ::= \texttt{IDENTIFIER}$$

$$\texttt{VAR} ::= \texttt{IDENTIFIER}$$

Rules for types.

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \textbf{closed}(L, \Phi)}{\textbf{type}(\mathsf{T})} \qquad \textbf{(THY-TYPE)}$$

$$\frac{\textbf{type}(\mathsf{T}_1) \quad \textbf{type}(\mathsf{T}_2)}{\textbf{type}(\mathsf{T}_1 \rightarrow \mathsf{T}_2)} \qquad \textbf{(FUNC-TYPE)}$$

Subtyping rules.

$$\frac{\mathsf{T} \equiv (L_1, \Phi_1) \quad \mathsf{T} \equiv (L_2, \Phi_2) \quad L_2 \subseteq L_1 \quad \Phi_2 \subseteq \Phi_1}{\mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(THY-SUB)}$$

$$\frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}} \qquad \textbf{(FUNC-SUB)}$$

View derivation rules.

$$\boxed{\frac{\textbf{map}(\rho) \quad \textbf{source}(\rho) = L_{\mathsf{s}} \quad \textbf{target}(\rho) \subseteq L_{\mathsf{t}}}{\textbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho)}} \qquad \boxed{\textbf{(THY-VIEW)}}$$

$$\boxed{\frac{\textbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_1) \quad \textbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_2)}{\textbf{view}(\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \rightarrow \mathsf{T}_{\mathsf{t}_2}, (\rho_1, \rho_2))}} \qquad \boxed{\textbf{(FUNC-VIEW)}}$$

View construction rules.

$$\boxed{\frac{\textbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho) \qquad \rho' = \mathsf{lift}(\rho)}{\textbf{view}((L_{\mathsf{s}} \cup L_{\mathsf{E}}, \Phi_{\mathsf{s}} \cup \Phi_{\mathsf{E}}), (L_{\mathsf{t}} \cup \rho'(L_{\mathsf{E}}), \Phi_{\mathsf{t}} \cup \rho'(\Phi_{\mathsf{E}})), \rho')}} \qquad \boxed{\textbf{(EXT-VIEW)}}$$

$$\boxed{\frac{\textbf{view}((L_{\mathsf{s}_1}, \Phi_{\mathsf{s}_1}), (L_{\mathsf{t}_1}, \Phi_{\mathsf{t}_1}), \rho_1) \quad \textbf{view}((L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_2}), \rho_2) \quad \textbf{consist}(\rho_1, \rho_2)}{\textbf{view}((L_{\mathsf{s}_1} \cup L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_1} \cup \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_1} \cup L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_1} \cup \Phi_{\mathsf{t}_2}), \rho_1 \cup \rho_2)}}$$

$$\textbf{(UNI-VIEW)}$$

$$\boxed{\frac{\textbf{view}((L_1, \Phi_1), (L_2, \Phi_2), \rho_1) \quad \textbf{view}((L_2, \Phi_2), (L_3, \Phi_3), \rho_2)}{\textbf{view}((L_1, \Phi_1), (L_3, \Phi_3), \rho_1 \circ \rho_2)}} \qquad \boxed{\textbf{(COMP-VIEW)}}$$

Rules for typing module expressions.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \qquad \textbf{(ASSUMP)}$$

$$\frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \qquad \textbf{(CONST)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) <: (L, \Phi)}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \qquad \textbf{(CAST)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(EXT)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \qquad \textbf{(UNION)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : (\rho(L), \rho(\Phi))} \qquad \textbf{(REN)}$$

$$\frac{\Gamma,\ \mathsf{X} : \mathsf{T}_1\ \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \mathsf{T}_1 \rightarrow \mathsf{T}_2} \qquad \textbf{(ABS)}$$

$$\frac{\Gamma \vdash \mathsf{E}_{\mathsf{f}} : \mathsf{T}_1 \rightarrow \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_{\mathsf{p}} : \mathsf{T}_{\mathsf{p}} \quad \mathsf{T}_{\mathsf{p}} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}} : \mathsf{T}_2} \qquad \textbf{(APP)}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{E}_{\mathsf{f}} : \mathsf{T}_1 \rightarrow \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_{\mathsf{p}} : \mathsf{T}_{\mathsf{p}} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T}_{\mathsf{p}}, \rho)}{\Gamma \vdash \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}\ \texttt{with view}\ (\mathsf{T}_1, \mathsf{T}_{\mathsf{p}}, \rho) : \mathsf{T}_2}} \qquad \boxed{\textbf{(APP-VIEW)}}$$

Theorem 3.5.1. *Type checking of Mei is decidable.*

Proof. Straightforward induction on the type derivation, since the typing rules are syntax-directed. For the case (APP-VIEW), by Proposition 3.4.15, $\mathbf{view}(\mathsf{T}_1, \mathsf{T}_{\mathsf{p}}, \rho)$ is decidable. □

3.6 Semantics of Mei

We define the semantics of Mei, $[\![\,]\!]$, in terms of module expressions of Mei Core in the sense that we translate every module expression of Mei to a module expression of Mei Core. In other words, we treat the new mechanisms of Mei as syntactic sugar.

VAR. $\mathsf{E} \equiv \mathsf{X}$:

$$[\![\mathsf{X}]\!] = \mathsf{X}$$

CONST. $\mathsf{E} \equiv \mathsf{C}$:

$$[\![\mathsf{C}]\!] = \mathsf{C}$$

BASIC. $\mathsf{E} \equiv (L,\ \Phi,\ \Delta)$:

$$[\![(L_{\mathsf{T}},\ \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)]\!] = (L_{\mathsf{T}},\ \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)$$

CAST. $\mathsf{E} \equiv \mathsf{T}\ \mathsf{E}'$:

$$[\![\mathsf{T}\ \mathsf{E}']\!] = \mathsf{T}\ [\![\mathsf{E}']\!]$$

EXT. $\mathsf{E} \equiv \mathsf{E}'$ `extended by` S:

$$[\![\mathsf{E}'\ \texttt{extended by}\ \mathsf{S}]\!] = [\![\mathsf{E}']\!]\ \texttt{extended by}\ \mathsf{S}$$

UNION. $\mathsf{E} \equiv \mathsf{E}_1 \oplus \mathsf{E}_2$:

$$[\![\mathsf{E}_1 \oplus \mathsf{E}_2]\!] = [\![\mathsf{E}_1]\!] \oplus [\![\mathsf{E}_1]\!]$$

REN. $\mathsf{E} \equiv \mathsf{E}'$ `with` ρ:

$$[\![\mathsf{E}'\ \texttt{with}\ \rho]\!] = [\![\mathsf{E}']\!]\ \texttt{with}\ \rho$$

ABS. $\mathsf{E} \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2$:

$$[\![\texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2]\!] = \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ [\![\mathsf{E}_2]\!]$$

APP. $\mathsf{E} \equiv \mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}$:

$$[\![\mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}]\!] = [\![\mathsf{E}_{\mathsf{f}}]\!]\ [\![\mathsf{E}_{\mathsf{p}}]\!]$$

APP-VIEW. $\mathsf{E} \equiv \mathsf{E_f}\ \mathsf{E_p}$ with view V:

$$[\![\mathsf{E_f}\ \mathsf{E_p} \text{ with view } \mathsf{V}]\!] \;= [\![\mathsf{E_f}]\!]\ ([\![\mathsf{V}]\!]_{\mathsf{v}}\ [\![\mathsf{E_p}]\!])$$

Lemma 3.6.1. *Let* $\mathsf{E} \in$ Mei *be a module expression. Then* $[\![\mathsf{E}]\!]$ *is a module expression in* Mei Core*.*

Proof. Straightforward induction on the definition of $[\![\,]\!]$. For case (APP-VIEW), use Proposition 3.4.12 to show that $[\![\mathsf{V}]\!]_{\mathsf{v}}$ is a functor in Mei Core. □

Lemma 3.6.2. *Let* $\mathsf{E} \in$ Mei *be a module expression. If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$*, then* $\Gamma \vdash [\![\mathsf{E}]\!] :\ \mathsf{T}$ *in* Mei Core*.*

Proof. Straightforward induction on the definition of $[\![\,]\!]$. Only the case (APP-VIEW) is interesting.

Let $\mathsf{E} \equiv \mathsf{E_f}\ \mathsf{E_p}$ with view V, where $\mathsf{V} \equiv (\mathsf{T}_1, \mathsf{T_p}, \rho)$. $\Gamma \vdash \mathsf{E} : \mathsf{T}$ is derived as follows:

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T}_1 \to \mathsf{T} \quad \Gamma \vdash \mathsf{E_p} : \mathsf{T_p} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} \text{ with view } (\mathsf{T}_1, \mathsf{T_p}, \rho) : \mathsf{T}}$$

By the induction hypothesis, $\Gamma \vdash [\![\mathsf{E_f}]\!] :\ \mathsf{T}_1 \to \mathsf{T}$ and $\Gamma \vdash [\![\mathsf{E_p}]\!] : \mathsf{T_p}$. By Proposition 3.4.15, $\mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)$ is decidable. By Proposition 3.4.13, $[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T_p} \to \mathsf{T_{pp}}$ and $\mathsf{T_{pp}} <: \mathsf{T}_1$. Then the following derivation finishes the proof.

$$\frac{\Gamma \vdash [\![\mathsf{E_f}]\!] :\ \mathsf{T}_1 \to \mathsf{T} \quad \dfrac{[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T_p} \to \mathsf{T_{pp}} \quad \Gamma \vdash [\![\mathsf{E_p}]\!] : \mathsf{T_p}}{\Gamma \vdash [\![\mathsf{V}]\!]_{\mathsf{v}}\ [\![\mathsf{E_p}]\!] :\ \mathsf{T_{pp}}} \quad \mathsf{T_{pp}} <: \mathsf{T}_1}{\Gamma \vdash [\![\mathsf{E_f}]\!]\ ([\![\mathsf{V}]\!]_{\mathsf{v}}\ [\![\mathsf{E_p}]\!]) :\ \mathsf{T}}$$

□

Theorem 3.6.3. *If* $\vdash \mathsf{E} :\ \mathsf{T}$*, then* $[\![\mathsf{E}]\!]{\downarrow}$*.*

Proof. By Lemma 3.6.1 and 3.6.2, $[\![\mathsf{E}]\!] : \mathsf{T}$ is a well-typed expression in Mei Core. The result follows directly from Theorem 3.3.9. □

3.7 Casting

In both Mei Basic and Mei Core, we have two casting rules, **(BASIC)** and **(CAST)**. **(BASIC)** helps the user to choose the principal type of a theory, while **(CAST)** lets the user cast a module expression of a theory type (subtype) to another theory type (supertype). **(CAST)** can be generalized for all module expressions including functor expressions:

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T_E} \quad \mathsf{T_E} <: \mathsf{T}}{\Gamma \vdash (\mathsf{T})\ \mathsf{E} : \mathsf{T}} \qquad \textbf{(CAST)}$$

This is so-called up-casting, which assigns a supertype of the type of E to it. The type checker can check if T is a supertype of $\mathsf{T_E}$ by the subtyping rules. The evaluator will simply ignore the type annotation and evaluate E as usual. Up-casting is a kind of abstraction, since it forgets some information about $\mathsf{T_E}$. For instance, we can cast the theory *Group* of type Group to the type Monoid.

Another kind of casting, down-casting, let the user assign a module expression to a module type that cannot be derived syntactically. For instance, if we have a theory *Group* of type Monoid, we may want to cast it back to type Group. This is not syntactically checkable since we need to evaluate *Group* to make sure that it is of type Group. The casting rule would be as follow:

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T_E}}{\Gamma \vdash (\mathsf{T})\ \mathsf{E} : \mathsf{T}} \qquad \textbf{(CAST}'\textbf{)}$$

The type checker simply trusts the user and keeps on type checking. Clearly, this will compromise the soundness of the type system, since the user might make a wrong casting, and then (1) the evaluation rules cannot preserve the type of the expression and (2) a well-typed expression might get stuck. To solve the first problem, we need to move some type checking work to the runtime system. For instance, we cannot simple ignore the casting information when evaluating a module expression, instead we need to check if its value is in fact of the new type.

$$\frac{\vdash \mathsf{N} : \mathsf{T}}{(\mathsf{T})\ \mathsf{N} \longrightarrow \mathsf{N}} \qquad \textbf{(CAST}'\textbf{)}$$

This will recover the type preservation property. Note that this runtime type checking is decidable and quick. The second problem can be handled by raising certain system exceptions.

We have also lost the normalization property in that some well-typed module expressions may not pass the runtime type checking and hence runtime exceptions are caught. However, we do not totally lose the normalization property, in the sense that, if the module expression passes the runtime type checking, it will be evaluated to some value. The good news is that we will not run into a divergent state with the help of the type system.

Remarks 3.7.1.

(1) In Mei Core, with the help of the subtype relation, it is not necessary to do up-casting. Therefore, the typing rule CAST is redundant. However, we keep it there to make it easy to extend Mei Core to support down-casting.

(2) In Mei, since the notion of view generalizes the subtype relations, the subtype relation can be expressed as a special view where the mapping is the identify on everything. We choose to keep the subtype relation because (1) we do not have to explicitly cast the actual parameter theory when building the coercion functors from views and (2) expressing subtype relations as views is not efficient since we need to build an identity functor and apply it.

Chapter 4

A module system with bounded universal types – DMei

4.1 Motivation

The original motivation of this chapter comes from the following example:

Example 4.1.1. Let us recall the Example 2.1.15 in §2.1.3. The theory *CommMonoid* is typed $\mathsf{CommMult}$ by the following derivation:

$$\frac{\Gamma \vdash \mathit{Comm} : \mathsf{Mult} \rightarrow \mathsf{CommMult} \quad \Gamma \vdash \mathit{Monoid} : \mathsf{Monoid} \quad \mathsf{Monoid} <: \mathsf{Mult}}{\Gamma \vdash \mathit{Comm\ Monoid} : \mathsf{CommMult}}$$

In the module system presented so far, the theory *Monoid* can be used as a theory of type Mult, either because *Monoid* is a theory of type Mult or because the type of *Monoid* is Monoid which is a subtype of the type Mult. For both considerations, the information that *Monoid* is also a theory of type Monoid gets lost. As a result, the instantiated theory *CommMonoid* is typed $\mathsf{CommMult}$ as shown in the type derivation. However, there are cases for which the user will want to use *CommMonoid* as a theory of $\mathsf{CommMonoid}$.

Basically there are two ways to solve this issue: (1) Let the user be able to add the lost information back to the type. (2) Try to keep the information during the type derivation. The solution following the first approach is the down-casting mechanism shown in §3.7. To get this information back to the type, we need to cast the theories (or the functors) down to the subtypes of their principal types.

To keep the information of parameter within the result type of the instantiated module expression, the simplest way is to use type variables. The result type of a functor type is defined in terms of the type variable, which represents the type of the parameter. The type of the functor is then a universal type over all possible parameter types. For instance, the type of *Comm* is $\texttt{forall}\ \mathsf{T_X}.\ \mathsf{T_X} \to (\mathsf{T_X}\ \texttt{extended by axioms}\ \forall\ x, y : ele.x \circ y = y \circ x)$. When *Monoid* is used to instantiate *Comm*, its principal type Monoid is used to substitute $\mathsf{T_X}$ in the type expression, giving type CommMonoid as expected. The problem is that the type variable in the type of *Comm* forgets the information that we want the parameter theory to be a theory of type Mult or that of a subtype of Mult, i.e. a theory with at least a binary multiplication operation. Thus we need a way to constrain the type variable. This leads to a bounded universal type, i.e. the parameter theory may not be an arbitrary type, but must, for example, be a subtype of Mult. The subtype relation here is necessary to help keep the type information [80].

4.1.1 Type variables and new typing rules

The intuition of the bounded universal type is that we can type *Comm* as $\texttt{forall}\ \mathsf{T_X} <: \mathsf{Mult}.\ \mathsf{T_X} \to (\mathsf{T_X}\ \texttt{extended by axioms}\ \forall\ x, y : ele.x \circ y = y \circ x)$. Following the approach of $\mathsf{F}_{<:}$ [80], a type system with bounded quantification, we need two abstract typing rules for function abstractions and type abstractions respectively.

$$\frac{\Gamma,\ \mathsf{X} : \mathsf{T}_1 \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \mathsf{T}_1 \to \mathsf{T}_2} \qquad \textbf{(ABS)}$$

$$\frac{\Gamma,\ \mathsf{T_X} <: \mathsf{T}_1 \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{forall}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{E}_2 : \texttt{forall}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2} \qquad \textbf{(TABS)}$$

However, a full type abstraction rule provides more than what we need. For example, the type $\texttt{forall}\ \mathsf{T_X} <: \mathsf{T}.\ \mathsf{T_X}$, which is the type of the polymorphic constants, is useless for our purpose. In fact, we only need type variables for polymorphic functors, so that the result type can be specified precisely with respect to the argument type. More importantly, in $\mathsf{F}_{<:}$, since type variables occur not only in the type expressions, but also in the module expressions, they can be used as the bound types in the type abstractions, e.g. $\texttt{forall}\ \mathsf{T_X} <: \mathsf{T}.\ \texttt{forall}\ \mathsf{T_Y} <: \mathsf{T_X}.\ \ldots$. This will result in the undecidability of the subtype relation for $\mathsf{F}_{<:}$. In [79], the subtyping rules were used to simulate a two-counter machine (a variant of Turing machines). The halting problem of the two-counter machine reduces to that of the subtype relation.

We propose to combine the two abstraction rules above as a single abstraction rule.

$$\frac{\Gamma, \mathsf{T_X} <: \mathsf{T}_1, \mathsf{X} : \mathsf{T_X} \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 :\ \texttt{forall}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T_X} \rightarrow \mathsf{T}_2} \quad \textbf{(ABS)}$$

Note that the type variables play only one role, as the polymorphic argument type of a functor type. This allows us to abbreviate $\texttt{forall}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T_X} \rightarrow \mathsf{T}_2$ as $\texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2$ without any ambiguity. **ABS** is then reformed as follows:

$$\frac{\Gamma, \mathsf{T_X} <: \mathsf{T}_1, \mathsf{X} : \mathsf{T_X} \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 :\ \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2} \quad \textbf{(ABS)}$$

This is similar to the dependent functor types in many other systems, e.g. OCaml [57, 58].

Remarks 4.1.2.

(1) The rule is similar to the abstraction rule in Mei Core, except that the type variable $\mathsf{T_X}$ might occur in the type expression T_2. Hence it can be replaced by a concrete type when the functor is instantiated.

(2) The type variables occur only in the type expression, not in the module expression. As a result, they cannot occur on the right hand side of the subtype relation.

(3) The type variables are tied with the module variables. If a module expression is closed, its type expression is also closed, i.e. it is without free type variables.

(4) The type variables are handled (generated) by the system. We assume that they are all distinct.

We thus only need rule(s) for functor applications.

$$\frac{\Gamma \vdash \mathsf{E_f} : \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \Gamma \vdash \mathsf{T_p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} :\ \mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]} \quad \textbf{(APP)}$$

The premises of the rule are almost identical to that of the functor application rule in Mei Core. In the conclusion of the rule, the concrete type of the actual parameter expression simultaneously replace all occurrences of $\mathsf{T_X}$ in T_2. This effectively does the functor type application inside the rule **APP**. Note that the formal definition of type substitutions is presented later in §4.2.

Remark 4.1.3. We are "eagerly" reducing type expressions. This is why we have $\mathsf{T}_2[\mathsf{T}_\mathsf{X} := \mathsf{T}_\mathsf{p}]$ instead of $(\texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2)\ \mathsf{T}_\mathsf{p}$ in **(APP)**. Type reduction is built into the rules.

Another issue is how to express the dependency between the parameter type and the result type, i.e. how to express T_2 in terms of T_X. Clearly, theory types and functor types are not enough. As in the above example, it is necessary to describe in the type of Comm the fact that any result theory is derived by adding the commutative axiom to the parameter theory. The types then contain the information about how theories (functors) are constructed, for example, by extension, union, renaming, and functor application.

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T}_\mathsf{E} \quad \Gamma \vdash \mathsf{T}_\mathsf{E} <: (L', \Phi') \quad \mathbf{closed}(L' \cup L, \Phi' \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L, \Phi, \Delta) : \mathsf{T}_\mathsf{E}\ \texttt{extended by}\ (L, \Phi \cup \mathbf{sen}(\Delta))}\ \textbf{(EXT-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_1 <: (L_1', \Phi_1') \quad \Gamma \vdash \mathsf{T}_2 <: (L_2', \Phi_2')}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2}\ \textbf{(UNI-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T} \quad \Gamma \vdash \mathsf{T} <: (L', \Phi') \quad \mathbf{map}(\rho)}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : \mathsf{T}\ \texttt{with}\ \rho}\qquad \textbf{(REN-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{E}_\mathsf{f} : \mathsf{T}_\mathsf{f} \quad \Gamma \vdash \mathsf{T}_\mathsf{f} <: \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_\mathsf{p} : \mathsf{T}_\mathsf{p} \quad \Gamma \vdash \mathsf{T}_\mathsf{p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E}_\mathsf{f}\ \mathsf{E}_\mathsf{p} : \mathsf{T}_\mathsf{f}\ \mathsf{T}_\mathsf{p}}$$

$$\textbf{(APP-VAR)}$$

Note that T_E in **EXT-VAR**, T_1 and T_2 in **UNI-VAR**, T in **REN-VAR**, T_f and possibly T_p in **APP-VAR** all contain type variables, otherwise rules **EXT**, **UNION**, **REN**, and **APP** will be applied.

Remark 4.1.4. Pierce's minimal type approach (see Chapter 28 in [80]) is not satisfactory for our purpose. In [80], suppose

$$\begin{aligned}
\mathsf{f} &= \lambda\mathsf{X} <: \mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}\}.\ \lambda\mathsf{y} : \mathsf{X}.\ \mathsf{y}\ 5;\\
\mathsf{f} &: \forall\mathsf{X} <: \mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}\}.\ \mathsf{X} \to \{\mathsf{a} : \mathsf{Nat}\}\\
\mathsf{g} &= \lambda\mathsf{x} : \mathsf{Nat}.\{\mathsf{a} = \mathsf{x}, \mathsf{b} = \mathsf{x}\};\\
\mathsf{g} &: \mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}, \mathsf{b} : \mathsf{Nat}\}
\end{aligned}$$

Then

$$\mathsf{f}\ (\mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}, \mathsf{b} : \mathsf{Nat}\})\ \mathsf{g} \ : \ \{\mathsf{a} : \mathsf{Nat}\}$$

We would prefer $\{\mathsf{a} : \mathsf{Nat}, \mathsf{b} : \mathsf{Nat}\}$ as the result type. Following our approach using their notation, f will be typed as follows:

$$\begin{aligned} \mathsf{f} &= \lambda \mathsf{X} <: \mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}\}.\ \lambda \mathsf{y} : \mathsf{X}.\ \mathsf{y}\ 5; \\ \mathsf{f} &: \forall \mathsf{X} <: \mathsf{Nat} \to \{\mathsf{a} : \mathsf{Nat}\}.\ \mathsf{X} \to (\mathsf{X}\ \mathsf{Nat}) \end{aligned}$$

Since the evaluation of $(\mathsf{X}\ \mathsf{Nat})$ is postponed until the type of g is available, the result type is $\{\mathsf{a} : \mathsf{Nat}, \mathsf{b} : \mathsf{Nat}\}$ as desired.

4.1.2 Subtyping rules

Naive subtyping rules. First, we need subtyping rules for theory types and functor types as in Mei Core. The most obvious definitions resemble those of Mei Basic.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_1 \subseteq L_2 \quad \Phi_1 \subseteq \Phi_2}{\vdash \mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(S-THY)}$$

$$\frac{\Gamma \vdash \mathsf{T}_{11} <: \mathsf{T}_1 \quad \Gamma \vdash \mathsf{T}_2[\mathsf{T}_\mathsf{X} := \mathsf{T}_1] <: \mathsf{T}_{22}[\mathsf{T}_\mathsf{X} := \mathsf{T}_{11}]}{\Gamma \vdash \mathtt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 <: \mathtt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_{11}.\ \mathsf{T}_{22}} \qquad \textbf{(S-FUNC)}$$

Note that **S-FUNC** ignores the construction information embedded in the result type by instantiating it with the parameter type. This gives the order (and the equality) between those functor types with different constructions, which is more than we need as shown in the following example:

Example 4.1.5. Let Monoid and Group be the theory type defined in Example 3.4.1.

We can define two functor types as follows:

$$\begin{array}{rcl}
\mathsf{GroupFromMonoid} & \equiv & \texttt{Functor}\ \mathsf{T_X} <: \mathsf{Monoid}. \\
& & (\mathsf{T_X}\ \texttt{with}\ \{mm \mapsto gg, e \mapsto e, \circ \mapsto \circ\})\ \texttt{extended by} \\
& & \texttt{language}\quad {}^{-1} : gg \to gg \\
& & \texttt{axioms}\ \ \forall x : gg.\ x \circ x^{-1} = e \\
\mathsf{MakeGroup} & \equiv & \texttt{Functor}\ \mathsf{T_X} <: \mathsf{Monoid}. \\
& & \texttt{language}\ \texttt{sort}\ gg \\
& & \qquad\qquad \texttt{const}\ e : gg \\
& & \qquad\qquad \texttt{opr}\ \ \circ : gg^2 \to gg \\
& & \qquad\qquad\qquad {}^{-1} : gg \to gg \\
& & \texttt{axioms}\ \ \forall x_1, x_2, x_3 : gg.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\
& & \qquad\quad\ \ \forall x : gg.\ x \circ e = x \wedge e \circ x = x \\
& & \qquad\quad\ \ \forall x : gg.\ x \circ x^{-1} = e
\end{array}$$

$\mathsf{GroupFromMonoid} <: \mathsf{MakeGroup}$ and $\mathsf{MakeGroup} <: \mathsf{GroupFromMonoid}$ by (S-FUNC). They are equal with respect to the subtype relation. However, let Ring be a theory type of rings. Then $\mathsf{GroupFromMonoid\ Ring} <: \mathsf{MakeGroup\ Ring}$ is derivable, but not $\mathsf{MakeGroup\ Ring} <: \mathsf{GroupFromMonoid\ Ring}$. Thus, the desired subtyping rules should admit $\mathsf{GroupFromMonoid} <: \mathsf{MakeGroup}$, but not $\mathsf{MakeGroup} <: \mathsf{GroupFromMonoid}$.

The key point is that, given

$$\Gamma \vdash \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T_1}.\ \mathsf{T_2} <: \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T_{11}}.\ \mathsf{T_{22}},$$

$\Gamma \vdash \mathsf{T_2}[\mathsf{T_X} := \mathsf{T_p}] <: \mathsf{T_{22}}[\mathsf{T_X} := \mathsf{T_p}]$ is not derivable (not monotonic), where $\mathsf{T_p} <: \mathsf{T_{11}}$. This motivates a new (S-FUNC) rule.

$$\frac{\Gamma \vdash \mathsf{T_{11}} <: \mathsf{T_1} \qquad \Gamma, \mathsf{T_X} <: \mathsf{T_{11}} \vdash \mathsf{T_2} <: \mathsf{T_{22}}}{\Gamma \vdash \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T_1}.\ \mathsf{T_2} <: \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T_{11}}.\ \mathsf{T_{22}}} \qquad \textbf{(S-FUNC)}$$

Clearly, monotonicity can be derived easily from the above rule. (Refer to Lemma 4.2.6 for the formal presentation.)

Second, we need the subtyping rules for the type variables, extension types, union types, renaming types, and application types. The following are the most obvious definitions:

$$\frac{\mathsf{T_X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T_X} <: \mathsf{T_X}} \qquad \textbf{(S-REF)}$$

$$\frac{\mathsf{T}_{\mathsf{X}} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_{\mathsf{X}} <: \mathsf{T}} \qquad \textbf{(S-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \texttt{extended by}\ \mathsf{S} <: \mathsf{T}_2 \ \texttt{extended by}\ \mathsf{S}} \qquad \textbf{(S-EXT)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: (L_2, \Phi_2) \quad \mathsf{S} \equiv (L_{\mathsf{S}}, \Phi_{\mathsf{S}})}{\Gamma \vdash \mathsf{T}_1 \ \texttt{extended by}\ \mathsf{S} <: (L_2 \cup L_{\mathsf{S}}, \Phi_2 \cup \Phi_{\mathsf{S}})} \qquad \textbf{(S-VAR-EXT)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_{11} \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: \mathsf{T}_{11} \ + \ \mathsf{T}_{22}} \qquad \textbf{(S-UNION)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION1)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: (L_{11}, \Phi_{11}) \quad \mathsf{T}_2 \equiv (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION2)}$$

$$\frac{\mathsf{T}_1 \equiv (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION3)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \texttt{with}\ \rho <: \mathsf{T}_2 \ \texttt{with}\ \rho} \qquad \textbf{(S-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: (L_2, \Phi_2)}{\Gamma \vdash \mathsf{T}_1 \ \texttt{with}\ \rho <: (\rho(L_2), \rho(\Phi_2))} \qquad \textbf{(S-VAR-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_2 \ \mathsf{T}_{\mathsf{p}_2}} \qquad \textbf{(S-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_2.\mathsf{T}_{22} \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}} <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_{\mathsf{p}} <: \mathsf{T}_{22}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_2]} \qquad \textbf{(S-VAR-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3} \qquad \textbf{(S-TRAN)}$$

Algorithmic subtyping rules. As in Mei Core, the transitivity rule **S-TRAN** is still problematic, since it is not algorithmic. In this case, we cannot simply eliminate the transitivity rule. For example, let us assume $\Gamma \vdash \mathsf{T_X} <: (L_1, \Phi_1)$ and $\Gamma \vdash (L_1, \Phi_1) <: (L_2, \Phi_2)$. In order to derive $\Gamma \vdash \mathsf{T_X} <: (L_2, \Phi_2)$, we have to use the transitivity rule. In system $\mathsf{F}_{<:}$ the transitivity rule is changed to a special form,

$$\frac{\Gamma \vdash \mathsf{T_X} <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T_X} <: \mathsf{T}_3}$$

where the left premise can be derived from (S-VAR). Our system is more complicated and thus the above rule is not applicable. For example, let us assume $\Gamma \vdash \mathsf{T_X} <: (L_1, \Phi_1)$, $\Gamma \vdash \mathsf{T_Y} <: (L_2, \Phi_2)$, and $\Gamma \vdash (L_1 \cup L_2, \Phi_1 \cup \Phi_2) <: (L, \Phi)$. The above transitivity rule is not enough to derive $\Gamma \vdash \mathsf{T_X} + \mathsf{T_Y} <: (L, \Phi)$. The left premise cannot be derived from (S-VAR) directly. It has to be derived from the rule (S-VAR-UNION1), which calculates the minimal upper bound of $\mathsf{T_X} + \mathsf{T_Y}$, in this case, $(L_1 \cup L_2, \Phi_1 \cup \Phi_2)$. In general, the left premise of our transitivity rule, must be derived from **S-VAR**, **S-VAR-EXT**, **S-VAR-UNION1**, **S-VAR-UNION2**,**S-VAR-UNION3**, **S-VAR-REN**, and **S-VAR-APP**. Effectively, this calculates the minimal upper bound of the left-hand-side of the conclusion based on the information from **S-VAR**, which can be compared with the right-hand-side. The main point is that the intermediate types (the minimal upper bound) used in the transitivity rule are *fixed* by the calculation, which eliminates the nondeterminism. To distinguish these calculation derivations, we use <:: for those subtype relations derived only by **S-VAR**, **S-VAR-EXT**, **S-VAR-UNION1**, **S-VAR-UNION2**, **S-VAR-UNION3**, **S-VAR-REN**, and **S-VAR-APP**. The subtyping rules are then reformulated as follow:.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_1 \subseteq L_2 \quad \Phi_1 \subseteq \Phi_2}{\vdash \mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(S-THY)}$$

$$\frac{\Gamma \vdash \mathsf{T}_{11} <: \mathsf{T}_1 \qquad \Gamma, \mathsf{T_X} <: \mathsf{T}_{11} \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 <: \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_{11}.\ \mathsf{T}_{22}} \qquad \textbf{(S-FUNC)}$$

$$\frac{\mathsf{T_X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T_X} <: \mathsf{T_X}} \qquad \textbf{(S-REF)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1\ \mathtt{extended\ by}\ \mathsf{S} <: \mathsf{T}_2\ \mathtt{extended\ by}\ \mathsf{S}} \qquad \textbf{(S-EXT)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_{11} \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: \mathsf{T}_{11} \ + \ \mathsf{T}_{22}} \qquad \textbf{(S-UNION)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \texttt{with}\ \rho <: \mathsf{T}_2 \ \texttt{with}\ \rho} \qquad \textbf{(S-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_2 \ \mathsf{T}_{\mathsf{p}_2}} \qquad \textbf{(S-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3} \qquad \textbf{(S-TRAN)}$$

$$\boxed{\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_\mathsf{X} <:: \mathsf{T}}} \qquad \boxed{\textbf{(S-VAR)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2) \quad \mathsf{S} \equiv (L_\mathsf{S}, \Phi_\mathsf{S})}{\Gamma \vdash \mathsf{T}_1 \ \texttt{extended by}\ \mathsf{S} <:: (L_2 \cup L_\mathsf{S}, \Phi_2 \cup \Phi_\mathsf{S})}} \qquad \boxed{\textbf{(S-VAR-EXT)}}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION1)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \mathsf{T}_2 \equiv (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION2)}$$

$$\frac{\mathsf{T}_1 \equiv (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION3)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2)}{\Gamma \vdash \mathsf{T}_1 \ \texttt{with}\ \rho <:: (\rho(L_2), \rho(\Phi_2))} \qquad \textbf{(S-VAR-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_2.\mathsf{T}_{22} \quad \Gamma \vdash \mathsf{T}_\mathsf{p} <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_\mathsf{p} <:: \mathsf{T}_{22}[\mathsf{T}_\mathsf{X} := \mathsf{T}_2]} \qquad \textbf{(S-VAR-APP)}$$

Now let us show that the (<:) rules define a *total* algorithm for the (<:) relation. This is shown by by assigning a complexity that is a natural number to each subtype relation.

Definition 4.1.6. We first define the complexity of a type T within a context Γ, written $\mathsf{compl}_{\Gamma}(\mathsf{T})$.

$$\begin{aligned}
&\mathsf{compl}_{\Gamma}(L, \Phi) = 1\\
&\mathsf{compl}_{\Gamma}(\mathsf{T_X}) = \mathsf{compl}_{\Gamma}(\mathsf{T}) \qquad \text{if } \mathsf{T_X} <: \mathsf{T} \in \Gamma\\
&\mathsf{compl}_{\Gamma}(\mathsf{T}\ \mathtt{extended\ by}\ \mathsf{S}) = \mathsf{compl}_{\Gamma}(\mathsf{T}) + 1\\
&\mathsf{compl}_{\Gamma}(\mathsf{T}_1\ +\ \mathsf{T}_2) = \mathsf{compl}_{\Gamma}(\mathsf{T}_1) + \mathsf{compl}_{\Gamma}(\mathsf{T}_2) + 1\\
&\mathsf{compl}_{\Gamma}(\mathsf{T}\ \mathtt{with}\ \rho) = \mathsf{compl}_{\Gamma}(\mathsf{T}_1) + 1\\
&\mathsf{compl}_{\Gamma}(\mathsf{T_f}\mathsf{T_p}) = \mathsf{compl}_{\Gamma}(\mathsf{T_f}) + \mathsf{compl}_{\Gamma}(\mathsf{T_p})\\
&\mathsf{compl}_{\Gamma}(\mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2) = \mathsf{compl}_{\Gamma, \mathsf{T_X} <: \mathsf{T}_1}(\mathsf{T}_2) + 1
\end{aligned}$$

The complexity of the subtype relations is defined based on that of the types.

$$\begin{aligned}
&\mathsf{compl}_{\Gamma}(\mathsf{T}_1 <: \mathsf{T}_2) = \mathsf{compl}_{\Gamma}(\mathsf{T}_1) + 2\\
&\mathsf{compl}_{\Gamma}(\mathsf{T}_1 <:: \mathsf{T}_2) = \mathsf{compl}_{\Gamma}(\mathsf{T}_1) + 1
\end{aligned}$$

Note that we increase the complexity by 2 for $<:$ to distinguish it from $<::$. This is important in the proof of the following theorem for the case corresponding to the rule **(S-TRAN)**.

Theorem 4.1.7. *The subtyping algorithm defined above is total.*

Proof. In any subtyping rule of $<:$ and $<::$, the complexity of the conclusion is strictly greater than that of any of the premises. □

Equivalence of naive and algorithmic subtyping rules. It is sufficient to prove the following theorem; the other direction is trivial.

Lemma 4.1.8.

(1) *If* $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$ *and* $\Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3$*, then* $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3$.

(2) *If* $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$ *and* $\Gamma \vdash \mathsf{T}_2 <:: \mathsf{T}_3$*, then* $\Gamma \vdash \mathsf{T}_1 <:: \mathsf{T}_3$.

Proof. We prove both statements simultaneously by induction on the sum of the sizes of the two derivations.

(1) Let the left derivation be

S-THY. Trivial.

S-FUNC. Directly from the induction hypothesis.

S-REF. The right derivation gives the result.

S-EXT.

$$\frac{\Gamma \vdash \mathsf{T}_1' <: \mathsf{T}_2'}{\Gamma \vdash \mathsf{T}_1' \texttt{ extended by } \mathsf{S} <: \mathsf{T}_2' \texttt{ extended by } \mathsf{S}}$$

Two subcases for the right derivation.

S-EXT.

$$\frac{\Gamma \vdash \mathsf{T}_2' <: \mathsf{T}_3'}{\Gamma \vdash \mathsf{T}_2' \texttt{ extended by } \mathsf{S} <: \mathsf{T}_3' \texttt{ extended by } \mathsf{S}}$$

By the induction hypothesis of (1), $\Gamma \vdash \mathsf{T}_1' <: \mathsf{T}_3'$ and the result follows directly from the rule (S-EXT).

S-TRAN.

$$\frac{\dfrac{\Gamma \vdash \mathsf{T}_2' <:: (L_2, \Phi_2) \quad \mathsf{S} \equiv (L_\mathsf{S}, \Phi_\mathsf{S})}{\Gamma \vdash \mathsf{T}_2' \texttt{ extended by } \mathsf{S} <:: (L_2 \cup L_\mathsf{S}, \Phi_2 \cup \Phi_\mathsf{S})} \quad \Gamma \vdash (L_2 \cup L_\mathsf{S}, L_2 \cup L_\mathsf{S}) <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_2' \texttt{ extended by } \mathsf{S} <: \mathsf{T}_3}$$

By the induction hypothesis of (2), $\Gamma \vdash \mathsf{T}_1' <:: (L_2, \Phi_2)$. Replacing T_2' in the above derivation by T_1' gives the result.

S-APP.

$$\frac{\Gamma \vdash \mathsf{T}_1' <: \mathsf{T}_2' \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}_1' \ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_2' \ \mathsf{T}_{\mathsf{p}_2}}$$

Two subcases for the right derivation.

S-APP.

$$\frac{\Gamma \vdash \mathsf{T}_2' <: \mathsf{T}_3' \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_2} <: \mathsf{T}_{\mathsf{p}_3}}{\Gamma \vdash \mathsf{T}_2' \ \mathsf{T}_{\mathsf{p}_2} <: \mathsf{T}_3' \ \mathsf{T}_{\mathsf{p}_3}}$$

The result follows from the induction hypotheses and an application of the rule (S-APP).

S-TRAN.

$$\frac{\dfrac{\Gamma \vdash \mathsf{T}_2' <:: \texttt{Functor } \mathsf{T}_\mathsf{X} <: \mathsf{T}_{22}.\mathsf{T}_{222} \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_2} <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}_2' \ \mathsf{T}_{\mathsf{p}_2} <:: \mathsf{T}_{222}[\mathsf{T}_\mathsf{X} := \mathsf{T}_{22}]} \quad \Gamma \vdash \mathsf{T}_{222}[\mathsf{T}_\mathsf{X} := \mathsf{T}_{22}] <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_2' \ \mathsf{T}_{\mathsf{p}_2} <: \mathsf{T}_3}$$

By the induction hypothesis of (1), $\Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{22}$ By the induction hypothesis of (2), $\Gamma \vdash \mathsf{T}'_1 <:: \texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_{22}.\mathsf{T}_{222}$. In the above derivation, replacing T'_2 and $\mathsf{T}_{\mathsf{p}_2}$ by T'_1 and $\mathsf{T}_{\mathsf{p}_1}$ respectively gives the result.

The proofs of cases (S-UNION) and (S-REN) are similar to that of (S-EXT).

(2) Let the left derivation be

S-THY. Trivial, since $\Gamma \vdash (L, \Phi) <:: \mathsf{T}_3$ is not derivable.

S-FUNC. Trivial, since $\Gamma \vdash (\texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_{22}.\mathsf{T}_{222}) <:: \mathsf{T}_3$ is not derivable.

S-REF. The right derivation gives the result.

S-EXT.

$$\frac{\Gamma \vdash \mathsf{T}'_1 <: \mathsf{T}'_2}{\Gamma \vdash \mathsf{T}'_1\ \texttt{extended by}\ \mathsf{S} <: \mathsf{T}'_2\ \texttt{extended by}\ \mathsf{S}}$$

The right derivation must be

$$\frac{\Gamma \vdash \mathsf{T}'_2 <:: (L_3, \Phi_3) \quad \mathsf{S} \equiv (L_{\mathsf{S}}, \Phi_{\mathsf{S}})}{\Gamma \vdash \mathsf{T}'_2\ \texttt{extended by}\ \mathsf{S} <:: (L_3 \cup L_{\mathsf{S}}, \Phi_3 \cup \Phi_{\mathsf{S}})}$$

where $\mathsf{T}_3 \equiv (L_3 \cup L_{\mathsf{S}}, \Phi_3 \cup \Phi_{\mathsf{S}})$. By the induction hypothesis of (2), $\mathsf{T}'_1 <:: (L_3, \Phi_3)$. Replacing T'_2 by T'_1 in the above derivation gives the result.

S-APP.

$$\frac{\Gamma \vdash \mathsf{T}'_1 <: \mathsf{T}'_2 \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}'_1\ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}'_2\ \mathsf{T}_{\mathsf{p}_2}}$$

The right derivation must be

$$\frac{\Gamma \vdash \mathsf{T}'_2 <:: \texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_{22}.\mathsf{T}_{222} \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_2} <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}'_2\ \mathsf{T}_{\mathsf{p}_2} <:: \mathsf{T}_{222}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{22}]}$$

where $\mathsf{T}_3 \equiv \mathsf{T}_{222}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{22}]$. By the induction hypothesis of (1), $\Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{22}$. By the induction hypothesis of (2), $\Gamma \vdash \mathsf{T}'_1 <:: \texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_{22}.\mathsf{T}_{222}$. In the above derivation, replacing T'_2 and $\mathsf{T}_{\mathsf{p}_2}$ by T'_1 and $\mathsf{T}_{\mathsf{p}_1}$ respectively gives the result.

The proofs for cases (S-UNION) and (S-REN) are similar to that for (S-EXT).

□

Theorem 4.1.9. *If* $\mathsf{T}_1 <: \mathsf{T}_2$ *is derivable from the naive rules, it is derivable from the algorithmic rules.*

Proof. By Lemma 4.1.8. □

4.2 Syntax of DMei Core

DMei Core is an extension of Mei Core that supports dependent functor types.

$$\begin{array}{lcl}
\texttt{EXPR} & ::= & \texttt{MOD-CONST} \\
& | & \texttt{TYPE-SPEC}\ \texttt{THY-SPEC} \\
& | & \texttt{TYPE-SPEC}\ \texttt{EXPR} \\
& | & \texttt{EXPR extended by SPEC} \\
& | & \texttt{EXPR} \oplus \texttt{EXPR} \\
& | & \texttt{EXPR with MAPPING} \\
& | & \texttt{functor MOD-VAR} : \texttt{TYPE}.\ \texttt{EXPR} \\
& | & \texttt{EXPR}\ \texttt{EXPR}
\end{array}$$

$$\begin{array}{lcl}
\texttt{TYPE} & ::= & \texttt{TYPE-CONST} \\
& | & \boxed{\texttt{TYPE-VAR}} \\
& | & \texttt{TYPE-SPEC} \\
& | & \boxed{\texttt{TYPE extended by TYPE-SPEC}} \\
& | & \boxed{\texttt{TYPE} + \texttt{TYPE}} \\
& | & \boxed{\texttt{TYPE with MAPPING}} \\
& | & \boxed{\texttt{Functor TYPE-VAR} <: \texttt{TYPE}.\ \texttt{TYPE}} \\
& | & \boxed{\texttt{TYPE}\ \texttt{TYPE}}
\end{array}$$

$$\texttt{THY-SPEC} \ ::= \ (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS})$$

$$\texttt{TYPE-SPEC} \ ::= \ (\texttt{LANG}, \texttt{AXIOMS})$$

`MOD-CONST` ::= `IDENTIFIER`

`TYPE-CONST` ::= `IDENTIFIER`

`MOD-VAR` ::= `IDENTIFIER`

`TYPE-VAR` ::= $\boxed{\texttt{IDENTIFIER}}$

Rules for types. **type**(T) asserts that T is a type. **closed**(L, Φ) asserts that **lang**$(\Phi) \subseteq L$.

$$\boxed{\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathbf{type}(\mathsf{T}_\mathsf{X})}} \qquad \boxed{\textbf{(VAR-TYPE)}}$$

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \mathbf{closed}(L, \Phi)}{\vdash \mathbf{type}(\mathsf{T})} \qquad \textbf{(THY-TYPE)}$$

$$\boxed{\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}) \quad \Gamma \vdash \mathsf{T} <: (L, \Phi) \quad \mathbf{closed}(L \cup L_\mathsf{S}, \Phi \cup \Phi_\mathsf{S})}{\Gamma \vdash \mathbf{type}(\mathsf{T}\ \texttt{extended by}\ (L_\mathsf{S}, \Phi_\mathsf{S}))}} \qquad \boxed{\textbf{(EXT-TYPE)}}$$

$$\boxed{\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_1) \quad \Gamma \vdash \mathbf{type}(\mathsf{T}_2)}{\Gamma \vdash \mathbf{type}(\mathsf{T}_1 + \mathsf{T}_2)}} \qquad \boxed{\textbf{(UNION-TYPE)}}$$

$$\boxed{\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}) \quad \Gamma \vdash \mathsf{T} <: (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathbf{type}(\mathsf{T}\ \texttt{with}\ \rho)}} \qquad \boxed{\textbf{(REN-TYPE)}}$$

$$\boxed{\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_1) \quad \Gamma,\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1 \vdash \mathbf{type}(\mathsf{T}_2)}{\Gamma \vdash \mathbf{type}(\texttt{Functor}\mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2)}} \qquad \boxed{\textbf{(FUNC-TYPE)}}$$

$$\boxed{\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_\mathsf{f}) \quad \Gamma \vdash \mathbf{type}(\mathsf{T}_\mathsf{p})}{\Gamma \vdash \mathbf{type}(\mathsf{T}_\mathsf{f}\ \mathsf{T}_\mathsf{p})}} \qquad \boxed{\textbf{(APP-TYPE)}}$$

Rules for subtyping.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_1 \subseteq L_2 \quad \Phi_1 \subseteq \Phi_2}{\vdash \mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(S-THY)}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_{11} <: \mathsf{T}_1 \qquad \Gamma, \mathsf{T}_\mathsf{X} <: \mathsf{T}_{11} \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 <: \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_{11}.\ \mathsf{T}_{22}}} \qquad \boxed{\textbf{(S-FUNC)}}$$

$$\boxed{\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_\mathsf{X} <: \mathsf{T}_\mathsf{X}}} \qquad \boxed{\textbf{(S-REF)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1\ \texttt{extended by}\ \mathsf{S} <: \mathsf{T}_2\ \texttt{extended by}\ \mathsf{S}}} \qquad \boxed{\textbf{(S-EXT)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_{11} \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}_1\ +\ \mathsf{T}_2 <: \mathsf{T}_{11}\ +\ \mathsf{T}_{22}}} \qquad \boxed{\textbf{(S-UNION)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1\ \texttt{with}\ \rho <: \mathsf{T}_2\ \texttt{with}\ \rho}} \qquad \boxed{\textbf{(S-REN)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}_1\ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_2\ \mathsf{T}_{\mathsf{p}_2}}} \qquad \boxed{\textbf{(S-APP)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3}} \qquad \boxed{\textbf{(S-TRAN)}}$$

$$\boxed{\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_\mathsf{X} <:: \mathsf{T}}} \qquad \boxed{\textbf{(S-VAR)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2) \quad \mathsf{S} \equiv (L_\mathsf{S}, \Phi_\mathsf{S})}{\Gamma \vdash \mathsf{T}_1\ \texttt{extended by}\ \mathsf{S} <:: (L_2 \cup L_\mathsf{S}, \Phi_2 \cup \Phi_\mathsf{S})}} \qquad \boxed{\textbf{(S-VAR-EXT)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})}} \qquad \boxed{\textbf{(S-VAR-UNION1)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \mathsf{T}_2 \equiv (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})}} \qquad \boxed{\textbf{(S-VAR-UNION2)}}$$

$$\boxed{\frac{\mathsf{T}_1 \equiv (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})}} \qquad \boxed{\textbf{(S-VAR-UNION3)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2)}{\Gamma \vdash \mathsf{T}_1 \ \mathtt{with} \ \rho <:: (\rho(L_2), \rho(\Phi_2))}} \qquad \boxed{\textbf{(S-VAR-REN)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{T}_1 <:: \mathtt{Functor} \ \mathsf{T}_\mathsf{X} <: \mathsf{T}_2.\mathsf{T}_{22} \quad \Gamma \vdash \mathsf{T}_\mathsf{p} <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_\mathsf{p} <:: \mathsf{T}_{22}[\mathsf{T}_\mathsf{X} := \mathsf{T}_2]}} \qquad \boxed{\textbf{(S-VAR-APP)}}$$

Type evaluation function and type substitution. From §4.1, we can see that we also need a substitution function for type variables. In addition, after the type variable is replaced by a type expression, we need to simplify the type expression, i.e. we need to evaluate the type expression to its normal form just as we do for the module expressions.

The *type evaluation function*, $[\![\,]\!]_{\mathsf{eval}}$, and the *type substitution function* for type expressions, $\mathsf{T}[\mathsf{T}_\mathsf{X} := \mathsf{T}_\mathsf{p}]$, are defined simultaneously as follows:

$$\begin{aligned}
[\![\mathsf{T}_\mathsf{X}]\!]_{\mathsf{eval}} &= \mathsf{T}_\mathsf{X} \\
[\![(L, \Phi)]\!]_{\mathsf{eval}} &= (L, \Phi) \\
[\![\mathsf{T} \ \mathtt{extended\ by} \ \mathsf{S}]\!]_{\mathsf{eval}} &= \begin{cases} (L_\mathsf{T} \cup L_\mathsf{S}, \Phi_\mathsf{T} \cup \Phi_\mathsf{S}) & \text{if } [\![\mathsf{T}]\!]_{\mathsf{eval}} \equiv (L_\mathsf{T}, \Phi_\mathsf{T}) \\ & \text{and } \mathsf{S} \equiv (L_\mathsf{S}, \Phi_\mathsf{S}) \\ [\![\mathsf{T}]\!]_{\mathsf{eval}} \ \mathtt{extended\ by} \ \mathsf{S} & \text{otherwise.} \end{cases}
\end{aligned}$$

$$[\![\mathsf{T}_1\ +\ \mathsf{T}_2]\!]_{\mathsf{eval}} = \begin{cases} (L_1 \cup L_2, \Phi_1 \cup \Phi_2) & \text{if } [\![\mathsf{T}_1]\!]_{\mathsf{eval}} \equiv (L_1, \Phi_1) \\ & \text{and } [\![\mathsf{T}_2]\!]_{\mathsf{eval}} \equiv (L_2, \Phi_2) \\ [\![\mathsf{T}_1]\!]_{\mathsf{eval}}\ +\ [\![\mathsf{T}_2]\!]_{\mathsf{eval}} & \text{otherwise.} \end{cases}$$

$$[\![\mathsf{T}\ \mathtt{with}\ \rho]\!]_{\mathsf{eval}} = \begin{cases} (\rho(L), \rho(\Phi)) & \text{if } [\![\mathsf{T}]\!]_{\mathsf{eval}} \equiv (L, \Phi) \\ [\![\mathsf{T}]\!]_{\mathsf{eval}}\ \mathtt{with}\ \rho & \text{otherwise.} \end{cases}$$

$$[\![\mathtt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1.\ \mathsf{T}_2]\!]_{\mathsf{eval}} = \mathtt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1.\ [\![\mathsf{T}_2]\!]_{\mathsf{eval}}$$

$$[\![\mathsf{T}\ \mathsf{T}_{\mathsf{p}}]\!]_{\mathsf{eval}} = \begin{cases} \mathsf{T}_2[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] & \text{if } [\![\mathsf{T}]\!]_{\mathsf{eval}} \equiv \mathtt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1.\mathsf{T}_2 \\ [\![\mathsf{T}]\!]_{\mathsf{eval}}\ [\![\mathsf{T}_{\mathsf{p}}]\!]_{\mathsf{eval}} & \text{otherwise.} \end{cases}$$

$$\mathsf{T}_{\mathsf{Y}}\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = \begin{cases} \mathsf{T}_{\mathsf{p}} & \text{if } \mathsf{T}_{\mathsf{X}} \equiv \mathsf{T}_{\mathsf{Y}} \\ \mathsf{T}_{\mathsf{Y}} & \text{otherwise.} \end{cases}$$

$$(L, \Phi)[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = (L, \Phi)$$

$$(\mathsf{T}\ \mathtt{extended\ by}\ \mathsf{S})\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = [\![(\mathsf{T}\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])\ \mathtt{extended\ by}\ \mathsf{S}]\!]_{\mathsf{eval}}$$

$$(\mathsf{T}_1\ +\ \mathsf{T}_2)\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = [\![(\mathsf{T}_1\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])\ +\ (\mathsf{T}_2\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])]\!]_{\mathsf{eval}}$$

$$(\mathsf{T}\ \mathtt{with}\ \rho)\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = [\![(\mathsf{T}\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])\mathtt{with}\ \rho]\!]_{\mathsf{eval}}$$

$$(\mathtt{Functor}\ \mathsf{T}_{\mathsf{Y}} <: \mathsf{T}_1.\ \mathsf{T}_2)\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = \begin{cases} [\![\mathtt{Functor}\ \mathsf{T}_{\mathsf{Y}} <: \mathsf{T}_1.\ \mathsf{T}_2\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}]]\!]_{\mathsf{eval}} \\ \quad \text{if } \mathsf{T}_{\mathsf{X}} \not\equiv \mathsf{T}_{\mathsf{Y}} \text{ and } \mathsf{T}_{\mathsf{Y}} \text{ is not free in } \mathsf{T}_{\mathsf{p}} \\ [\![\mathtt{Functor}\ \mathsf{T}_{\mathsf{Z}} <: \mathsf{T}_1.\ \mathsf{T}_2'\ [\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}]]\!]_{\mathsf{eval}} \\ \quad \text{if } \mathsf{T}_{\mathsf{X}} \not\equiv \mathsf{T}_{\mathsf{Y}} \text{ and } \mathsf{T}_{\mathsf{Y}} \text{ is free in } \mathsf{T}_{\mathsf{p}} \\ \mathtt{Functor}\ \mathsf{T}_{\mathsf{Y}} <: \mathsf{T}_1.\ \mathsf{T}_2 \qquad \text{if } \mathsf{T}_{\mathsf{X}} \equiv \mathsf{T}_{\mathsf{Y}}. \end{cases}$$

$$(\mathsf{T}\ \mathsf{T}_{\mathsf{pp}})[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}] = [\![(\mathsf{T}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])\ (\mathsf{T}_{\mathsf{pp}}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_{\mathsf{p}}])]\!]_{\mathsf{eval}}$$

where, in the second case[1] for functor types, $\mathsf{T}_2' \equiv \mathsf{T}_2[\mathsf{T}_{\mathsf{Y}} := \mathsf{T}_{\mathsf{Z}}]$ and T_{Z} is a fresh type variable.

[1] We rename the bound variable to avoid the variable capture problem. It is not a problem for the implementation since identifiers, which are all distinct, can be used to represent type variables.

Definition 4.2.1. Two module types are *α-equivalent*, written $\mathsf{T}_1 =_\alpha \mathsf{T}_2$, if they differ only in the names of the bound variables.

We will consider two module types to be equivalent if they are α-equivalent.

Lemma 4.2.2. *If* $\Gamma \equiv \mathsf{T}_{\mathsf{X}_1} <: \mathsf{T}_1, \ldots, \mathsf{T}_{\mathsf{X}_n} <: \mathsf{T}_n$ *and* $\Gamma \vdash \textbf{type}(\mathsf{T})$*, then* $[\![\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_1] \ldots [\mathsf{T}_{\mathsf{X}_n} := \mathsf{T}_n]]\!]_{\mathsf{eval}}$ *is either*

(1) *a theory type* (L, Φ)*, or*

(2) *a functor type* $\texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2$*.*

Proof. By induction on the rules for types. □

Corollary 4.2.3. *If* $\vdash \textbf{type}(\mathsf{T})$*, then* $[\![\mathsf{T}]\!]_{\mathsf{eval}}$ *is either*

(1) *a theory type* (L, Φ)*, or*

(2) *a functor type* $\texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2$*.*

Proof. By Lemma 4.2.2. □

Lemma 4.2.4.

(1) *If* $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$*, then* $\Gamma \vdash [\![\mathsf{T}_1]\!]_{\mathsf{eval}} <: [\![\mathsf{T}_2]\!]_{\mathsf{eval}}$*.*

(2) *If* $\Gamma \vdash \mathsf{T}_1 <:: \mathsf{T}_2$*, then* $\Gamma \vdash [\![\mathsf{T}_1]\!]_{\mathsf{eval}} <:: [\![\mathsf{T}_2]\!]_{\mathsf{eval}}$*.*

Proof. Simultaneous induction on the derivations of $<:$ and $<::$ respectively. □

Lemma 4.2.5 (Monotonicity of substitution w.r.t. body)**.** *Let* T_1, T_2*, and* T *be type expressions. If* $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$*, then* $\Gamma \vdash \mathsf{T}[\mathsf{T}_\mathsf{X} := \mathsf{T}_1] <: \mathsf{T}[\mathsf{T}_\mathsf{X} := \mathsf{T}_2]$.

Proof. By induction on the complexity of T. Only the cases for functor types and application types are interesting.

(1) $\mathsf{T} \equiv \texttt{Functor}\ \mathsf{T}_\mathsf{Y} <: \mathsf{T}_\mathsf{s}.\ \mathsf{T}_\mathsf{t}$. If $\mathsf{T}_\mathsf{X} \equiv \mathsf{T}_\mathsf{Y}$, it is trivial. Assume $\mathsf{T}_\mathsf{X} \not\equiv \mathsf{T}_\mathsf{Y}$,

$$\begin{array}{cll} & (\texttt{Functor}\ \mathsf{T}_\mathsf{Y} <: \mathsf{T}_\mathsf{s}.\ \mathsf{T}_\mathsf{t})[\mathsf{T}_\mathsf{X} := \mathsf{T}_1] & \\ \equiv & \texttt{Functor}\ \mathsf{T}_\mathsf{Y} <: \mathsf{T}_\mathsf{s}.\ (\mathsf{T}_\mathsf{t}[\mathsf{T}_\mathsf{X} := \mathsf{T}_1]) & \text{by definition} \\ <: & \texttt{Functor}\ \mathsf{T}_\mathsf{Y} <: \mathsf{T}_\mathsf{s}.\ (\mathsf{T}_\mathsf{t}[\mathsf{T}_\mathsf{X} := \mathsf{T}_2]) & \text{by i.h. and rule S-FUNC} \\ \equiv & (\texttt{Functor}\ \mathsf{T}_\mathsf{Y} <: \mathsf{T}_\mathsf{s}.\ \mathsf{T}_\mathsf{t})[\mathsf{T}_\mathsf{X} := \mathsf{T}_2] & \text{by definition} \end{array}$$

(2) $\mathsf{T} \equiv \mathsf{T_f}\ \mathsf{T_p}$.

$$\begin{array}{lll} & (\mathsf{T_f}\ \mathsf{T_p})[\mathsf{T_X} := \mathsf{T}_1] & \\ \equiv & [\![\mathsf{T_f}[\mathsf{T_X} := \mathsf{T}_1]\ \mathsf{T_p}[\mathsf{T_X} := \mathsf{T}_1]]\!]_{\mathsf{eval}} & \text{by definition} \\ <: & [\![\mathsf{T_f}[\mathsf{T_X} := \mathsf{T}_2]\ \mathsf{T_p}[\mathsf{T_X} := \mathsf{T}_2]]\!]_{\mathsf{eval}} & \text{by i.h., rule S-APP, Lemma 4.2.4} \\ \equiv & (\mathsf{T_f}\ \mathsf{T_p})[\mathsf{T_X} := \mathsf{T}_2] & \text{by definition} \end{array}$$

□

Lemma 4.2.6 (Monotonicity of substitution w.r.t. parameter)**.**

(1) *If* $\Gamma,\ \mathsf{T_X} <: \mathsf{T} \vdash \mathsf{T}_1 <: \mathsf{T}_2$*, then for all* $\mathsf{T_p}$ *such that* $\Gamma \vdash \mathsf{T_p} <: \mathsf{T}$*,* $\Gamma \vdash \mathsf{T}_1[\mathsf{T_X} := \mathsf{T_p}] <: \mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]$.

(2) *If* $\Gamma,\ \mathsf{T_X} <: \mathsf{T} \vdash \mathsf{T}_1 <:: \mathsf{T}_2$*, then for all* $\mathsf{T_p}$ *such that* $\Gamma \vdash \mathsf{T_p} <: \mathsf{T}$*,* $\Gamma \vdash \mathsf{T}_1[\mathsf{T_X} := \mathsf{T_p}] <:: \mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]$.

Proof. Simultaneous induction on the derivations of $<:$ and $<::$ respectively. □

Rules for typing module expressions.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \qquad \textbf{(ASSUMP)}$$

$$\frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \qquad \textbf{(CONST)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) <: (L, \Phi)}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \qquad \textbf{(CAST)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(EXT)}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E} : \mathsf{T_E} \quad \Gamma \vdash \mathsf{T_E} <: (L', \Phi') \quad \mathbf{closed}(L' \cup L, \Phi' \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \mathtt{extended\ by}\ (L, \Phi, \Delta) : \mathsf{T_E}\ \mathtt{extended\ by}\ (L, \Phi \cup \mathbf{sen}(\Delta))}}$$

$\boxed{\textbf{(EXT-VAR)}}$

$$\dfrac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \qquad \textbf{(UNION)}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_1 <: (L'_1, \Phi'_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2}} \ \boxed{\textbf{(UNI-VAR1)}}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{T}_2 <: (L'_2, \Phi'_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2}} \ \boxed{\textbf{(UNI-VAR2)}}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_1 <: (L'_1, \Phi'_1) \quad \Gamma \vdash \mathsf{T}_2 <: (L'_2, \Phi'_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2}}$$

$\boxed{\textbf{(UNI-VAR3)}}$

$$\dfrac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \mathtt{with}\ \rho : (\rho(L), \rho(\Phi))} \qquad \textbf{(REN)}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E} : \mathsf{T} \quad \Gamma \vdash \mathsf{T} <: (L', \Phi') \quad \mathbf{map}(\rho)}{\Gamma \vdash \mathsf{E}\ \mathtt{with}\ \rho : \mathsf{T}\ \mathtt{with}\ \rho}} \qquad \boxed{\textbf{(REN-VAR)}}$$

$$\boxed{\dfrac{\Gamma, \mathsf{T_X} <: \mathsf{T}_1, \mathsf{X} : \mathsf{T_X} \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2}} \qquad \boxed{\textbf{(ABS)}}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E_f} : \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \Gamma \vdash \mathsf{T_p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} :\ \mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]}} \qquad \boxed{\textbf{(APP)}}$$

$$\boxed{\dfrac{\Gamma \vdash \mathsf{E_f} : \mathsf{T_f} \quad \Gamma \vdash \mathsf{T_f} <: \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T_1}.\ \mathsf{T_2} \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \Gamma \vdash \mathsf{T_p} <: \mathsf{T_1}}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} : \mathsf{T_f}\ \mathsf{T_p}}}$$

$\boxed{\textbf{(APP-VAR)}}$

Remark 4.2.7. The evaluation order of the above rules does matter. For instance, we want to apply the rule **EXT** before applying the rule **EXT-VAR** in order to avoid further need of type reduction. In other words, we want to derive the *evaluated* type directly.

Theorem 4.2.8. *Type checking of DMei Core is decidable.*

Proof. By induction on the syntax of module expressions. Clearly the type checking rules defined above are syntax-directed. For each module expression, there is a rule either giving the type of the module expression or reducing it to type-checking a subexpression which is strictly smaller than the original module expression with possibly a subtype checking. The result follows directly from the fact that subtype checking is decidable as shown in Theorem 4.1.7. □

Corollary 4.2.9. *Type inferencing of DMei Core is decidable.*

4.3 Semantics of DMei Core

The semantics of DMei Core is exactly the same as that of Mei Core, since there are no new syntactical classes introduced. The only difference is that the result type of a functor application will keep the information contained in the parameter type. Therefore, the result module expression can be employed in more contexts. However the evaluation of these functor applications follows exactly the same substitution semantics as in Mei Core and Mei Basic. The proof of the normalization theorem, Theorem 4.3.6, follows exactly the same pattern as that of Theorems 2.4.13 and 3.3.9.

Normalization of the well-typed module expressions. The normalization proof for DMei Core resembles that for Mei Core, i.e. in two steps: (1) construct a set *SN* of expressions that are normalizable and (2) show that every well-typed module expression is an element of *SN*.

Due to the existence of type variables, the major change in the normalization proofs in this section from the normalization proofs for Mei Basic and Mei Core is to reduce type expressions via the type evaluation function and the type substitution function.

Lemma 4.3.1. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\mathsf{E}{\downarrow}$ *iff* $\mathsf{E}'{\downarrow}$.

Definition 4.3.2. The set of module expressions, *SN*, is defined inductively as follows:

$$\frac{\mathsf{E} :\ (L,\Phi) \quad \mathsf{E}{\downarrow}}{\mathsf{E} \in\ \mathit{SN}_{(L,\,\Phi)}} \qquad \frac{\forall \mathsf{T_p} <: \mathsf{T}_1.\ \forall \mathsf{E_p} \in \mathit{SN}_{\mathsf{T_p}}.\ \mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]}}{\mathsf{E} \in \mathit{SN}_{\mathtt{Functor}\ \mathsf{T_X}\ <:\ \mathsf{T}_1.\ \mathsf{T}_2}}$$

Lemma 4.3.3. *If* $\mathsf{E} \in \mathit{SN}$, *then* $\mathsf{E}{\downarrow}$.

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L,\Phi)$. Directly follows from Definition 2.4.9.

(2) $\mathsf{T} \equiv \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2$. Let $\mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_1}$, clearly $\mathsf{T}_1 <: \mathsf{T}_1$. By Definition 4.3.2, $\mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2[\mathsf{T_X}:=\mathsf{T}_1]}$. By the induction hypothesis, $(\mathsf{E}\ \mathsf{E_p}){\downarrow}$, which implies $\mathsf{E}{\downarrow}$.

□

Lemma 4.3.4. *If* $\mathsf{E} \longrightarrow \mathsf{E}'$, *then* $\mathsf{E} \in \mathit{SN}$ *iff* $\mathsf{E}' \in \mathit{SN}$.

Proof. By induction on T.

(1) $\mathsf{T} \equiv (L,\Phi)$. Directly follows Lemma 4.3.1 and Definition 4.3.2.

(2) $\mathsf{T} \equiv \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2$.

(⇒) Let $\mathsf{T_p} <: \mathsf{T}_1$ and $\mathsf{E_p} \in \mathit{SN}_{\mathsf{T_p}}$ be an arbitrary module expression. $\mathsf{E}\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2[\mathsf{T_X}:=\mathsf{T_p}]}$ by definition of *SN*. By the induction hypothesis, $\mathsf{E}'\ \mathsf{E_p} \in \mathit{SN}_{\mathsf{T}_2[\mathsf{T_X}:=\mathsf{T_p}]}$. Since the choice of $\mathsf{T_p}$ and $\mathsf{E_p}$ is arbitrary, definition of *SN* gives the result.

(⇐) Analogous to (⇒).

□

Lemma 4.3.5. *If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$, $\Gamma \equiv \mathsf{T}_{\mathsf{X}_1} <: \mathsf{T}_1, \mathsf{X}_1 : \mathsf{T}_{\mathsf{X}_1}, \ldots, \mathsf{T}_{\mathsf{X}_\mathsf{n}} <: \mathsf{T}_\mathsf{n}, \mathsf{X}_\mathsf{n} : \mathsf{T}_{\mathsf{X}_\mathsf{n}}$. *Let* $\vdash \mathsf{T}_{11} <: \mathsf{T}_1$, …, $\vdash \mathsf{T}_{\mathsf{nn}} <: \mathsf{T}_\mathsf{n}$ *and* $\mathsf{E}_1 \in \mathit{SN}_{\mathsf{T}_{11}}, \ldots, \mathsf{E}_\mathsf{n} \in \mathit{SN}_{\mathsf{T}_{\mathsf{nn}}}$. *Then* $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in \mathit{SN}_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}}:=\mathsf{T}_{\mathsf{nn}}]}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathsf{E} : \mathsf{T}$. Let $\Gamma \equiv \mathsf{X}_1 : \mathsf{T}_{\mathsf{X}_1}, \mathsf{T}_{\mathsf{X}_1} <: \mathsf{T}_1, \ldots, \mathsf{X}_\mathsf{n} : \mathsf{T}_{\mathsf{X}_\mathsf{n}}$ in the following proof.

ASSUMP. $\mathsf{E} \equiv \mathsf{X}_\mathsf{i}$ and $\mathsf{T} \equiv \mathsf{T}_\mathsf{i}$. Trivial.

CONST. $\mathsf{E} \equiv \mathsf{C}$. Assume $\sigma(\mathsf{C}) = \mathsf{E}'$. Since $\mathsf{C}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \equiv \mathsf{C}$, it is sufficient to show $\mathsf{C} \in \mathit{SN}_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}] \ldots [\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]}$. Since $\vdash \mathsf{E}' : \mathsf{T}$ is in the premise, $\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}] \ldots [\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}] \equiv \mathsf{T}$. By the induction hypothesis, $\mathsf{E}' \in \mathit{SN}_\mathsf{T}$. But $\mathsf{C} \longrightarrow \mathsf{E}'$, so by Lemma 4.3.4, $\mathsf{C} \in \mathit{SN}_\mathsf{T}$.

BASIC. $\mathsf{E} \equiv (L_\mathsf{T}, \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta)$. Note that $\mathsf{T} \equiv (L_\mathsf{T}, \Phi_\mathsf{T})$, hence, $\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}] \ldots [\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}] = \mathsf{T}$. Since E is of theory type, it is sufficient to prove $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]\downarrow$. But $\mathsf{E}[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \equiv \mathsf{E}$ already in normal form.

CAST. $\mathsf{E} \equiv (L, \Phi)\ \mathsf{E}'$. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in \mathit{SN}_{(L',\Phi')}.$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rcl}
((L,\Phi)\ \mathsf{E}')[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] & = & (L,\Phi)\ (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}]) \\
& \longrightarrow & (L,\Phi)\ ((L',\Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})) \\
& \longrightarrow & (L,\Phi)\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'}) \\
& \in & \mathit{SN}_{(L,\,\Phi)} \\
& = & \mathit{SN}_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}] \ldots [\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]}
\end{array}$$

EXT. $\mathsf{E} \equiv \mathsf{E}'$ `extended by` $(L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})$. There are two subcases corresponding to the rules **EXT** and **EXT-VAR**.

(*i*) **EXT**. Assume $\Gamma \vdash \mathsf{E}' : (L', \Phi')$ is in the premise. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1] \ldots [\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in \mathit{SN}_{(L',\Phi')}.$$

Let us assume $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L', \Phi')\ (L_{\mathsf{E}'}, \Phi_{\mathsf{E}'}, \Delta_{\mathsf{E}'})$.

$$\begin{array}{rl}
& (\mathsf{E}'\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}))[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\
= & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}])\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\
\longrightarrow & ((L', \Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\
\longrightarrow & (L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))\ (L_{\mathsf{E}'} \uplus L_\mathsf{S},\ \Phi_{\mathsf{E}'} \uplus \Phi_\mathsf{S},\ \Delta_{\mathsf{E}'} \uplus \Delta_\mathsf{S}) \\
\in & SN_{(L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))} \\
= & SN_{(L',\ \Phi')\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})} \\
= & SN_\mathsf{T} \\
= & SN_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]}
\end{array}$$

(ii) **EXT-VAR**. Assume $\Gamma \vdash \mathsf{E}' : \mathsf{T}'$ and $\Gamma \vdash \mathsf{T}' <: (L, \Phi)$ are in the premise. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \in SN_{\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]}.$$

By Lemmas 4.2.6, $\vdash \mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}] <: (L, \Phi)$. Hence $\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]$ is a theory type. Let us assume $\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}] \equiv (L', \Phi')$ and $\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \longrightarrow (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'})$.

$$\begin{array}{rl}
& (\mathsf{E}'\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}))[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}] \\
= & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X}_\mathsf{n} := \mathsf{E}_\mathsf{n}])\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\
\longrightarrow & ((L', \Phi')\ (L_{\mathsf{E}'},\ \Phi_{\mathsf{E}'},\ \Delta_{\mathsf{E}'}))\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}) \\
\longrightarrow & (L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))\ (L_{\mathsf{E}'} \uplus L_\mathsf{S},\ \Phi_{\mathsf{E}'} \uplus \Phi_\mathsf{S},\ \Delta_{\mathsf{E}'} \uplus \Delta_\mathsf{S}) \\
\in & SN_{(L' \cup L_\mathsf{S},\ \Phi' \cup \Phi_\mathsf{S} \cup \mathbf{sen}(\Delta_\mathsf{S}))} \\
= & SN_{(L',\ \Phi')\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})} \\
= & SN_{\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S})} \\
= & SN_{(\mathsf{T}'\ \texttt{extended by}\ (L_\mathsf{S},\ \Phi_\mathsf{S},\ \Delta_\mathsf{S}))[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]} \\
= & SN_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X}_\mathsf{n}} := \mathsf{T}_{\mathsf{nn}}]}
\end{array}$$

UNION. $\mathsf{E} \equiv \mathsf{E}' \oplus \mathsf{E}''$. Similar to the proof of theory extension. We need to consider four cases corresponding to the rules **UNION**, **UNI-VAR1**, **UNI-VAR2**, and **UNI-VAR3** respectively.

REN. $\mathsf{E} \equiv \mathsf{E}'$ `with` ρ. Similar to the proofs of theory extension and union. We need to consider two cases corresponding to the rules **REN** and **REN-VAR** respectively .

ABS. $\mathsf{E} \equiv \texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''}$ and $\mathsf{T} \equiv \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T'}.\ \mathsf{T''}$. Without loss of generality, we assume $\mathsf{X} \not\equiv \mathsf{X_i}$ for $1 \leq \mathsf{i} \leq \mathsf{n}$. In order to prove

$$\begin{aligned} & (\texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''})[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \\ \in\ & \mathcal{SN}_{(\texttt{Functor}\ \mathsf{T_X} <: \mathsf{T'}.\ \mathsf{T''})[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}]} \\ =\ & \mathcal{SN}_{\texttt{Functor}\ \mathsf{T_X} <: \mathsf{T'}.\ (\mathsf{T''}[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}])}, \end{aligned}$$

it is sufficient to prove

$$((\texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''})[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E_p} \in \mathcal{SN}_{\mathsf{T''}[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}][\mathsf{T_X}:=\mathsf{T_p}]}$$

for arbitrary $\mathsf{T_p} <: \mathsf{T'}$ and $\mathsf{E_p} : \mathsf{T_p}$. We have

$$\begin{aligned} & ((\texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''})[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E_p} \\ =\ & (\texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E_p} \\ \longrightarrow\ & \mathsf{E''}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ [\mathsf{X} := \mathsf{E_p}]. \end{aligned}$$

Since $\Gamma, \mathsf{T_X} <: \mathsf{T'}, \mathsf{X} : \mathsf{T_X} \vdash \mathsf{E''} : \mathsf{T''}$ is in the premise, by the induction hypothesis,

$$\mathsf{E''}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}]\ [\mathsf{X} := \mathsf{E_p}] \in \mathcal{SN}_{\mathsf{T''}[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}][\mathsf{T_X}:=\mathsf{T_p}]}.$$

By lemma 4.3.4,

$$((\texttt{functor}\ \mathsf{X} : \mathsf{T'}.\ \mathsf{E''})[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}])\ \mathsf{E_p} \in \mathcal{SN}_{\mathsf{T''}[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}][\mathsf{T_X}:=\mathsf{T_p}]}.$$

APP. $\mathsf{E} \equiv \mathsf{E'}\ \mathsf{E_p}$. There are two subcases corresponding to the rules **APP** and **APP-VAR**.

(*i*) **APP**. Assume $\Gamma \vdash \mathsf{E'} : \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T''}.\ \mathsf{T}$, $\Gamma \vdash \mathsf{E_p} : \mathsf{T_p}$, and $\mathsf{T_p} <: \mathsf{T''}$ are in the premises. By the induction hypothesis,

$$\mathsf{E'}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in \mathcal{SN}_{(\texttt{Functor}\ \mathsf{T_X} <: \mathsf{T''}.\ \mathsf{T})[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}]}$$

and

$$\mathsf{E_p}[\mathsf{X_1} := \mathsf{E_1}] \ldots [\mathsf{X_n} := \mathsf{E_n}] \in \mathcal{SN}_{\mathsf{T_p}[\mathsf{T_{X_1}}:=\mathsf{T_{11}}]\ldots[\mathsf{T_{X_n}}:=\mathsf{T_{nn}}]}.$$

Thus,

$$\begin{aligned}
& (\mathsf{E}'\ \mathsf{E_p})[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \\
= \ & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}])\ (\mathsf{E_p}[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]) \\
\in \ & \mathit{SN}_{\mathsf{T}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}][\mathsf{T_X}:=(\mathsf{T_p}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}])]} \quad \text{by Definition 4.3.2} \\
= \ & \mathit{SN}_{(\mathsf{T}[\mathsf{T_X}:=\mathsf{T_p}])[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}]}.
\end{aligned}$$

(*ii*) **APP-VAR**. Assume $\Gamma \vdash \mathsf{E}' : \mathsf{T}'$, $\Gamma \vdash \mathsf{T}' <: \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}''.\ \mathsf{T}$, $\Gamma \vdash \mathsf{E_p} : \mathsf{T_p}$, and $\Gamma \vdash \mathsf{T_p} <: \mathsf{T}''$ are in the premises. By the induction hypothesis,

$$\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \in \mathit{SN}_{\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}]}$$

and

$$\mathsf{E_p}[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \in \mathit{SN}_{\mathsf{T_p}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}]}.$$

By Lemma 4.2.6,

$$\begin{aligned}
& \vdash \mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}} := \mathsf{T_{nn}}] \\
<: \ & (\mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}''.\ \mathsf{T})[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}} := \mathsf{T_{nn}}] \\
= \ & \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}''.\ (\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}} := \mathsf{T_{nn}}]).
\end{aligned}$$

We thus can assume

$$\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}} := \mathsf{T_{nn}}] \equiv \mathtt{Functor}\ \mathsf{T_X} <: \mathsf{T}''_{\mathsf{sup}}.\ \mathsf{T_{sub}} \qquad (4.1)$$

where $\vdash \mathsf{T}'' <: \mathsf{T}''_{\mathsf{sup}}$ and $\mathsf{T_X} <: \mathsf{T}'' \vdash \mathsf{T_{sub}} <: (\mathsf{T}[\mathsf{T}_{\mathsf{X}_1} := \mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}} := \mathsf{T_{nn}}])$.

$$\begin{aligned}
& (\mathsf{E}'\ \mathsf{E_p})[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}] \\
= \ & (\mathsf{E}'[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}])\ (\mathsf{E_p}[\mathsf{X}_1 := \mathsf{E}_1]\ldots[\mathsf{X_n} := \mathsf{E_n}]) \\
\in \ & \mathit{SN}_{\mathsf{T_{sub}}[\mathsf{T_X}:=(\mathsf{T_p}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}])]} \quad \text{by Definition 4.3.2} \\
= \ & \mathit{SN}_{(\mathsf{T}'[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}])\ (\mathsf{T_p}[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}])} \\
& \qquad \text{by (4.1) and type evaluation} \\
= \ & \mathit{SN}_{(\mathsf{T}'\ \ \mathsf{T_p})[\mathsf{T}_{\mathsf{X}_1}:=\mathsf{T}_{11}]\ldots[\mathsf{T}_{\mathsf{X_n}}:=\mathsf{T_{nn}}]}.
\end{aligned}$$

□

Theorem 4.3.6. *If* $\vdash \mathsf{E} :\ \mathsf{T}$*,* $\mathsf{E}{\downarrow}$*.*

Proof. By Lemma 4.3.5, $\mathsf{E} \in \mathit{SN}_{\mathsf{T}}$. By Lemma 4.3.3, $\mathsf{E}{\downarrow}$. □

4.4 Integration with coercion

The integration of the coercion mechanism with dependent functor types is not trivial. The notion of theory view concerns only the relation between theories, not the way they are constructed. However the whole idea of dependent functor types is to embed the construction information in the functor types. Thus, a view from a dependent functor type to another must reflect the construction information. It is at least as complicated as the subtype relation, since the subtype relation is embedded in the view relation. We claim that, although to find a good definition of views between the dependent functor types is of theoretical interest, it is not of practical significance for two reasons: (1) it makes the module system too complicated, and (2) there is little chance for the user to use the views between dependent functor types. As a result, we make the design decision to integrate only the theory views with DMei Core to form DMei, which is nice and simple.

All mechanisms we need for DMei are already explained. We only need to assemble them to form DMei.

$$
\begin{array}{lcl}
\texttt{EXPR} & ::= & \texttt{MOD-CONST} \\
 & | & \texttt{TYPE-SPEC}\ \texttt{THY-SPEC} \\
 & | & \texttt{TYPE-SPEC}\ \texttt{EXPR} \\
 & | & \texttt{EXPR}\ \texttt{extended by}\ \texttt{SPEC} \\
 & | & \texttt{EXPR} \oplus \texttt{EXPR} \\
 & | & \texttt{EXPR}\ \texttt{with}\ \texttt{MAPPING} \\
 & | & \texttt{functor}\ \texttt{MOD-VAR} : \texttt{TYPE}.\ \texttt{EXPR} \\
 & | & \texttt{EXPR}\ \texttt{EXPR} \\
 & | & \boxed{\texttt{EXPR}\ \texttt{EXPR}\ \text{with view}\ \texttt{VIEW}} \\
 & & \\
\texttt{TYPE} & ::= & \texttt{TYPE-CONST} \\
 & | & \texttt{TYPE-VAR} \\
 & | & \texttt{TYPE-SPEC} \\
 & | & \texttt{TYPE}\ \texttt{extended by}\ \texttt{TYPE-SPEC} \\
 & | & \texttt{TYPE} + \texttt{TYPE} \\
 & | & \texttt{TYPE}\ \texttt{with}\ \texttt{MAPPING} \\
 & | & \texttt{Functor}\ \texttt{TYPE-VAR} <: \texttt{TYPE}.\ \texttt{TYPE-VAR} \to \texttt{TYPE} \\
 & | & \texttt{TYPE}\ \texttt{TYPE}
\end{array}
$$

$$\texttt{THY-SPEC} \;::=\; (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS})$$

$$\texttt{TYPE-SPEC} \;::=\; (\texttt{LANG}, \texttt{AXIOMS})$$

$$\boxed{\texttt{VIEW}} \;\; \boxed{::=} \;\; \boxed{\texttt{(TYPE, TYPE, MAPPING)}}$$

$$\texttt{MOD-CONST} \;::=\; \texttt{IDENTIFIER}$$

$$\texttt{TYPE-CONST} \;::=\; \texttt{IDENTIFIER}$$

$$\texttt{MOD-VAR} \;::=\; \texttt{IDENTIFIER}$$

$$\texttt{TYPE-VAR} \;::=\; \texttt{IDENTIFIER}$$

Rules for types. $\mathbf{type}(\mathsf{T})$ asserts that T is a type. $\mathbf{closed}(L, \Phi)$ asserts that $\mathbf{lang}(\Phi) \subseteq L$.

$$\frac{\mathsf{T}_{\mathsf{X}} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathbf{type}(\mathsf{T}_{\mathsf{X}})} \qquad \textbf{(VAR-TYPE)}$$

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \mathbf{closed}(L, \Phi)}{\vdash \mathbf{type}(\mathsf{T})} \qquad \textbf{(THY-TYPE)}$$

$$\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}) \quad \Gamma \vdash \mathsf{T} <: (L, \Phi) \quad \mathbf{closed}(L \cup L_{\mathsf{S}}, \Phi \cup \Phi_{\mathsf{S}})}{\Gamma \vdash \mathbf{type}(\mathsf{T} \;\texttt{extended by}\; (L_{\mathsf{S}}, \Phi_{\mathsf{S}}))} \qquad \textbf{(EXT-TYPE)}$$

$$\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_1) \quad \Gamma \vdash \mathbf{type}(\mathsf{T}_2)}{\Gamma \vdash \mathbf{type}(\mathsf{T}_1 + \mathsf{T}_2)} \qquad \textbf{(UNION-TYPE)}$$

$$\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}) \quad \Gamma \vdash \mathsf{T} <: (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathbf{type}(\mathsf{T} \;\texttt{with}\; \rho)} \qquad \textbf{(REN-TYPE)}$$

$$\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_1) \quad \Gamma,\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1 \vdash \mathbf{type}(\mathsf{T}_2)}{\Gamma \vdash \mathbf{type}(\texttt{Functor}\mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1.\ \mathsf{T}_2)} \qquad \textbf{(FUNC-TYPE)}$$

$$\frac{\Gamma \vdash \mathbf{type}(\mathsf{T}_{\mathsf{f}}) \quad \Gamma \vdash \mathbf{type}(\mathsf{T}_{\mathsf{p}})}{\Gamma \vdash \mathbf{type}(\mathsf{T}_{\mathsf{f}}\ \mathsf{T}_{\mathsf{p}})} \qquad \textbf{(APP-TYPE)}$$

Rules for subtyping.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_1 \subseteq L_2 \quad \Phi_1 \subseteq \Phi_2}{\vdash \mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(S-THY)}$$

$$\frac{\Gamma \vdash \mathsf{T}_{11} <: \mathsf{T}_1 \qquad \Gamma, \mathsf{T}_\mathsf{X} <: \mathsf{T}_{11} \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 <: \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_{11}.\ \mathsf{T}_{22}} \qquad \textbf{(S-FUNC)}$$

$$\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_\mathsf{X} <: \mathsf{T}_\mathsf{X}} \qquad \textbf{(S-REF)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1\ \texttt{extended by}\ \mathsf{S} <: \mathsf{T}_2\ \texttt{extended by}\ \mathsf{S}} \qquad \textbf{(S-EXT)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_{11} \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_{22}}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <: \mathsf{T}_{11} \ + \ \mathsf{T}_{22}} \qquad \textbf{(S-UNION)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1\ \texttt{with}\ \rho <: \mathsf{T}_2\ \texttt{with}\ \rho} \qquad \textbf{(S-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_{\mathsf{p}_2}}{\Gamma \vdash \mathsf{T}_1\ \mathsf{T}_{\mathsf{p}_1} <: \mathsf{T}_2\ \mathsf{T}_{\mathsf{p}_2}} \qquad \textbf{(S-APP)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_2 <: \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_3} \qquad \textbf{(S-TRAN)}$$

$$\frac{\mathsf{T}_\mathsf{X} <: \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{T}_\mathsf{X} <:: \mathsf{T}} \qquad \textbf{(S-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2) \quad \mathsf{S} \equiv (L_\mathsf{S}, \Phi_\mathsf{S})}{\Gamma \vdash \mathsf{T}_1\ \texttt{extended by}\ \mathsf{S} <:: (L_2 \cup L_\mathsf{S}, \Phi_2 \cup \Phi_\mathsf{S})} \qquad \textbf{(S-VAR-EXT)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION1)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_{11}, \Phi_{11}) \quad \mathsf{T}_2 \equiv (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION2)}$$

$$\frac{\mathsf{T}_1 \equiv (L_{11}, \Phi_{11}) \quad \Gamma \vdash \mathsf{T}_2 <:: (L_{22}, \Phi_{22})}{\Gamma \vdash \mathsf{T}_1 \ + \ \mathsf{T}_2 <:: (L_{11} \cup L_{22}, \Phi_{11} \cup \Phi_{22})} \qquad \textbf{(S-VAR-UNION3)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: (L_2, \Phi_2)}{\Gamma \vdash \mathsf{T}_1 \ \texttt{with} \ \rho <:: (\rho(L_2), \rho(\Phi_2))} \qquad \textbf{(S-VAR-REN)}$$

$$\frac{\Gamma \vdash \mathsf{T}_1 <:: \texttt{Functor} \ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_2.\mathsf{T}_{22} \quad \Gamma \vdash \mathsf{T}_{\mathtt{p}} <: \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \ \mathsf{T}_{\mathtt{p}} <:: \mathsf{T}_{22}[\mathsf{T}_{\mathsf{X}} := \mathsf{T}_2]} \qquad \textbf{(S-VAR-APP)}$$

View derivation rules.

$$\boxed{\frac{\mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L_{\mathsf{s}} \quad \mathbf{target}(\rho) \subseteq L_{\mathsf{t}}}{\mathbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho)}} \qquad \boxed{\textbf{(THY-VIEW)}}$$

View construction rules.

$$\boxed{\frac{\mathbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho) \qquad \rho' = \mathsf{lift}(\rho)}{\mathbf{view}((L_{\mathsf{s}} \cup L_{\mathsf{E}}, \Phi_{\mathsf{s}} \cup \Phi_{\mathsf{E}}), (L_{\mathsf{t}} \cup \rho'(L_{\mathsf{E}}), \Phi_{\mathsf{t}} \cup \rho'(\Phi_{\mathsf{E}})), \rho')}} \qquad \boxed{\textbf{(EXT-VIEW)}}$$

$$\boxed{\frac{\mathbf{view}((L_{\mathsf{s}_1}, \Phi_{\mathsf{s}_1}), (L_{\mathsf{t}_1}, \Phi_{\mathsf{t}_1}), \rho_1) \quad \mathbf{view}((L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_2}), \rho_2) \quad \mathbf{consist}(\rho_1, \rho_2)}{\mathbf{view}((L_{\mathsf{s}_1} \cup L_{\mathsf{s}_2}, \Phi_{\mathsf{s}_1} \cup \Phi_{\mathsf{s}_2}), (L_{\mathsf{t}_1} \cup L_{\mathsf{t}_2}, \Phi_{\mathsf{t}_1} \cup \Phi_{\mathsf{t}_2}), \rho_1 \cup \rho_2)}}$$

(UNI-VIEW)

$$\boxed{\frac{\mathbf{view}((L_1, \Phi_1), (L_2, \Phi_2), \rho_1) \quad \mathbf{view}((L_2, \Phi_2), (L_3, \Phi_3), \rho_2)}{\mathbf{view}((L_1, \Phi_1), (L_3, \Phi_3), \rho_1 \circ \rho_2)}} \qquad \boxed{\textbf{(COMP-VIEW)}}$$

Rules for typing module expressions.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \qquad \textbf{(ASSUMP)}$$

$$\frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \qquad \textbf{(CONST)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \qquad \textbf{(BASIC)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) <: (L, \Phi)}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \qquad \textbf{(CAST)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(EXT)}$$

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T}_{\mathsf{E}} \quad \Gamma \vdash \mathsf{T}_{\mathsf{E}} <: (L', \Phi') \quad \mathbf{closed}(L' \cup L, \Phi' \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L, \Phi, \Delta) : \mathsf{T}_{\mathsf{E}}\ \texttt{extended by}\ (L, \Phi \cup \mathbf{sen}(\Delta))}$$

(EXT-VAR)

$$\frac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \qquad \textbf{(UNION)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_1 <: (L_1', \Phi_1') \quad \mathsf{T}_2 \equiv (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2} \qquad \textbf{(UNI-VAR1)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{T}_2 <: (L_2', \Phi_2')}{\Gamma \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2} \qquad \textbf{(UNI-VAR2)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{T}_1 <: (L_1', \Phi_1') \quad \Gamma \vdash \mathsf{T}_2 <: (L_2', \Phi_2')}{\Gamma \vdash \mathsf{E}_1 \oplus\ \mathsf{E}_2 : \mathsf{T}_1 + \mathsf{T}_2}$$

(UNI-VAR3)

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : (\rho(L), \rho(\Phi))} \quad \textbf{(REN)}$$

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T} \quad \Gamma \vdash \mathsf{T} <: (L', \Phi') \quad \mathbf{map}(\rho)}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : \mathsf{T}\ \texttt{with}\ \rho} \quad \textbf{(REN-VAR)}$$

$$\frac{\Gamma, \mathsf{T_X} <: \mathsf{T}_1, \mathsf{X} : \mathsf{T_X} \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2 : \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2} \quad \textbf{(ABS)}$$

$$\frac{\Gamma \vdash \mathsf{E_f} : \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \Gamma \vdash \mathsf{T_p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} :\ \mathsf{T}_2[\mathsf{T_X} := \mathsf{T_p}]} \quad \textbf{(APP)}$$

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T_f} \quad \Gamma \vdash \mathsf{T_f} <: \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \Gamma \vdash \mathsf{T_p} <: \mathsf{T}_1}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p} : \mathsf{T_f}\ \mathsf{T_p}}$$

$$\textbf{(APP-VAR)}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{E_f} : \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p}\ \text{with view}\ (\mathsf{T}_1, \mathsf{T_p}, \rho) :\ \mathsf{T}_2[\mathsf{T_X} := \mathsf{lift}(\rho^{-1})(\mathsf{T_p})]}} \boxed{\textbf{(APP-VIEW)}}$$

$$\boxed{\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T_f} \quad \Gamma \vdash \mathsf{T_f} <: \texttt{Functor}\ \mathsf{T_X} <: \mathsf{T}_1.\ \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} :\ \mathsf{T_p} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p}\ \text{with view}\ (\mathsf{T}_1, \mathsf{T_p}, \rho) : \mathsf{T_f}\ (\mathsf{lift}(\rho^{-1})(\mathsf{T_p}))}}$$

$$\boxed{\textbf{(APP-VAR-VIEW)}}$$

The semantics of the view is defined as in §3.4.2. Consequently, the semantics of DMei is defined in terms of DMei Core in exactly the same way as Mei is defined in terms of Mei Core in §3.6. The following lemmas and theorem can thus be proved using the proofs of Lemmas 3.6.1 and 3.6.2 and Theorem 4.4.3 as guides respectively.

Lemma 4.4.1. *Let* $\mathsf{E} \in \text{DMei}$ *be a module expression,* $[\![\mathsf{E}]\!]$ *is a module expression in* DMei Core*.*

Lemma 4.4.2. *Let* $\mathsf{E} \in \text{DMei}$ *be a module expression. If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$*,* $\Gamma \vdash [\![\mathsf{E}]\!] :\ \mathsf{T}$ *in* DMei Core*.*

Theorem 4.4.3. *If* $\vdash \mathsf{E} :\ \mathsf{T}$*,* $[\![\mathsf{E}]\!]{\downarrow}$*.*

4.5 Tradeoff between Mei and DMei

As shown in §4.1, the advantage of DMei is to preserve the precise type information during functor application. The tradeoff is a necessarily more complicated type system, especially the subtype relation which is difficult to understand and implement. In addition, views are only definable for theory types, not functor types. While, in practice, the latter may not be a problem, the former is definitely not desirable.

As the designer, we prefer Mei for its simplicity. The lost type information may be added by allowing down-casting as indicated in §4.1. This postpones some type checking issues to the evaluation phase, which is not necessarily bad. In fact, in DMei some evaluation issues are moved to the type checking phase by expressing construction information in types, which, in our opinion, is just as bad as down-casting. In order to preserve the precise type information during functor application, we are forced to mix type checking with expression evaluation. The question is then in which phase we should put this mixed part. Our preference is to put it in the evaluation phase, which is in Mei. However, we should indicate that DMei has its own theoretical interest.

4.6 A simplified module system with dependent functor types – SDMei

4.6.1 Motivation

In §4.5, we argued that the subtype relation of DMei is too complicated for practical use. It is then desirable to simplify those rules. Looking at a functor type $\texttt{Functor}\ \mathsf{T}_{\mathsf{X}} <: \mathsf{T}_1.\ \mathsf{T}_2$, one observation is that T_1 has a role of upper bound. In most cases, the user is concerned with "what" it is, not "how" it is constructed. In other words, T_1 does not have to be a "dependent functor type". When we apply the above functor to some actual parameter with type T, we need to justify that $\mathsf{T} <: \mathsf{T}_1$. Thus, we only need subtype relations in the form of $\mathsf{T} <: \mathsf{T}_1$ where T is a (possibly) dependent type and T_1 is a "simple functor type" or theory type as in Mei.

4.6.2 Intuition

Note that, for a functor type $\texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2$, if there is no occurrence of T_X in T_2, it is not a dependent functor type. In other words, it is equivalent to the functor type $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ in Mei. Therefore we can abbreviate $\texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2$ as $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ if there is no occurrence of T_X in T_2.

Definition 4.6.1. A type T is called *simple* if it is a theory type $\mathsf{T} \equiv (L, \Phi)$ or it is functor type of the form $\mathsf{T} \equiv \mathsf{T}_1 \rightarrow \mathsf{T}_2$ where both T_1 and T_2 are simple.

Note that a simple type in DMei corresponds to a type in Mei.

In order to relate a DMei type to a simple type by a subtype relation, we need to reduce a DMei type to a simple type. The simplest way to do this is to replace all type variables in the DMei type with the bound type of the type variable. The *reduction function*, $[\![\,]\!]_{\mathsf{red}}$, is defined formally as follows. Let us assume that T is evaluated, and then by Corollary 4.2.3, we only need to consider two cases.

$$[\![\mathsf{T}]\!]_{\mathsf{red}} = \begin{cases} [\![\mathsf{T}_1]\!]_{\mathsf{red}} \rightarrow [\![\mathsf{T}_2\ [\mathsf{T}_\mathsf{X} := [\![\mathsf{T}_1]\!]_{\mathsf{red}}]]\!]_{\mathsf{red}} & \text{if } \mathsf{T} \equiv \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 \\ \mathsf{T} & \text{otherwise.} \end{cases}$$

Proposition 4.6.2. *If* $\vdash \mathsf{T}$ *is derivable in DMei Core,* $[\![\mathsf{T}]\!]_{\mathsf{red}}$ *is simple.*

Proof. By induction on $\mathsf{compl}(\mathsf{T})$, where

$$\mathsf{compl}(\mathsf{T}) = \begin{cases} \mathsf{compl}(\mathsf{T}_1) + \mathsf{compl}(\mathsf{T}_2) + 1 & \text{if } \mathsf{T} \equiv \texttt{Functor}\ \mathsf{T}_\mathsf{X} <: \mathsf{T}_1.\ \mathsf{T}_2 \\ 0 & \text{otherwise.} \end{cases}$$

i.e. the number of occurrences of dependent functor type abstractions in T. □

With the help of the reduction function, we can define the subtyping rule we want as follows. $\mathbf{simple}(\mathsf{T})$ asserts that a type T is simple.

$$\frac{\mathbf{simple}(\mathsf{T}_2) \quad [\![\mathsf{T}_1]\!]_{\mathsf{red}} <: \mathsf{T}_2}{\mathsf{T}_1 <: \mathsf{T}_2}$$

The other subtyping rules are those for simple types as in Mei.

We can justify the above rule by showing that, if a subtype relation can be derived by the new subtyping rules, it can also be derived by the subtyping rules of DMei. This is formally presented later in Theorem 4.6.4.

4.6.3 SDMei Core

Syntax of SDMei Core

SDMei Core is an simplified version of DMei Core. The abstract syntax is the same as DMei Core shown in 4.2.

Rules for types. Same as DMei Core shown in 4.2.

Rules for subtyping.

$$\frac{\mathsf{T}_1 \equiv (L_1, \Phi_1) \quad \mathsf{T}_2 \equiv (L_2, \Phi_2) \quad L_1 \subseteq L_2 \quad \Phi_1 \subseteq \Phi_2}{\vdash \mathsf{T}_1 <: \mathsf{T}_2} \qquad \textbf{(S-THY)}$$

$$\boxed{\frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2} \quad \textbf{simple}(\mathsf{T}_{\mathsf{s}_1}, \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{t}_2})}{\mathsf{T}_{\mathsf{s}_1} \rightarrow \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \rightarrow \mathsf{T}_{\mathsf{t}_2}}} \qquad \boxed{\textbf{(S-FUNC)}}$$

$$\boxed{\frac{\textbf{simple}(\mathsf{T}_2) \quad [\![\mathsf{T}_1]\!]_{\mathsf{red}} <: \mathsf{T}_2}{\mathsf{T}_1 <: \mathsf{T}_2}} \qquad \boxed{\textbf{(S-DPT)}}$$

Lemma 4.6.3. $\vdash \mathsf{T} <: [\![\mathsf{T}]\!]_{\mathsf{red}}$ *is derivable in DMei Core.*

Proof. This follows directly the Monotonicity Lemma 4.2.6. □

Theorem 4.6.4. *If* $\vdash \mathsf{T}_1 <: \mathsf{T}_2$ *is derivable in SDMei Core, it is also derivable in DMei Core.*

Proof. By induction on the subtyping rules defined above. For the case **S-DPT**, we use Lemma 4.6.3. □

Rules for typing module expressions. Same as for DMei Core shown in 4.2.

Semantics of SDMei Core

The semantics of SDMei Core is exactly the same as that of DMei Core, since there are no new syntactical classes introduced. The only difference is that the subtype relation is smaller than that of DMei Core. Therefore, there are fewer well-typed module expressions. This is not necessarily bad, as we argued in §4.6.1.

The proof of the normalization property follows directly from the fact that, if a module expression is well-typed in SDMei Core, then it is well-typed in DMei Core, as shown in the following theorem.

Theorem 4.6.5. *If* $\vdash \mathsf{E} : \ \mathsf{T}$*, then it is derivable in DMei Core.*

Proof. Since the only difference between SDMei Core and DMei Core is their subtyping rules, this follows directly from Theorem 4.6.4. □

Corollary 4.6.6. *If* $\vdash \mathsf{E} : \ \mathsf{T}$*,* $E{\downarrow}$*.*

4.6.4 Integration with coercion

Clearly, it is easy to extend SDMei Core by views over theory types as in DMei. However, since the subtype relation of SDMei Core is so simple, we would like to ask "Can we integrate views for functor types as well as for theory types with SDMei Core?" We leave this for future work.

Chapter 5

Comparison of Mei with other module systems

We show, in this chapter, that a very simple module system like Mei can be powerful. We show that many modular mechanisms supported by various systems can be implemented directly or indirectly in Mei. In particular, we investigate ML-family module systems, the modularity mechanisms for algebraic specification languages and MMSs (theorem provers and computer algebra systems), and an expressive language of signatures.

5.1 ML-family module systems

As we noted in §1.3 and §1.4, an ML-family module system that supports higher-order functors (not all dialects of ML supports higher-order functors) is similar to Mei Core, a subsystem of Mei. In other words, every mechanism useful for an MMS that is supported by an ML-family module system is also supported by Mei. For instance, higher-order functors and subtyping are supported. In addition, Mei supports views which are not supported in any ML-family module system.

There are features of some ML-family module systems that are not supported by Mei. For instance, generative functors, modules as first-class values, and the big theory approach. Some of them are not suitable for MMSs, such as the big theory approach. Others are are not necessary, as discussed in §1.1.1. Therefore they are not included in our design goals. In addition, ML-family modules also provide a namespace management mechanism, while in Mei renaming is used to solve name conflicts, as stated in Remark 2.1.14.

It is worthwhile mentioning Leroy's *modular module system* [58], which is also designed to be language independent as Mei. In fact, the implementation of Mei in Chapter 7 is inspired largely by [58].

5.2 Algebraic specification languages

While most specification languages support the extension, union, and renaming operations as in Mei, they usually have a different parameterized specification mechanism. The body of the parameterized specification is, in general, defined as an extension of the parameter specification(s). Therefore, only one copy of the parameter occurs in the body definition. Parameter passing is via fitting morphisms, another name for theory views. The semantics is defined by pushouts and only first-order parameterized specifications are supported, which can be simulated easily in Mei. Most languages discussed in this section follow this approach. Many specification languages support their own modular mechanisms other than those stated above. Some of them are not supported *directly* in Mei. However, most of them can be implemented in Mei in a different way.

It is also worth indicating that R. Jimémez and F. Orejas presented a framework in which the body of the parameterized theory is defined as an arbitrary expression over the parameter, including λ-abstraction and application [54]. As in Mei, it supports higher-order parameterized specifications with a fitting morphism style parameter passing mechanism. However, its β-reduction is not properly defined, making its semantics not operational, as fully discussed in 5.2.4.

5.2.1 Maude

Maude [19, 31, 32, 33] is a specification language based on a rewriting logic. Modules are the basic building blocks of Maude. A module consists of sorts, operators on these sorts, and equations (axioms) specifying how they interact. Effectively a module in Maude defines the initial algebra specified by its axioms. In other words, the carrier set only consists of the generated terms and these terms are distinct. "Theory" in

Maude is a notion similar to module. The only difference is that theories have loose semantics, i.e. there is no additional constraint other than the equations specified. While the distinction between initial and loose semantics is necessary for equational logic systems, it is not necessary for the higher-order logic systems, since induction principles can be employed to make the distinction. As in many other languages, the basic services for module hierarchies are module importing, summation importing, and module renaming. There are three different modes of importation: *protecting, extending, and including.* These modes put semantic constraints on the inclusion relation between submodules and supermodules, but they are the user's promises and are not checked and discharged by the system. "`protecting`" means that the imported module is not changed in anyway, intuitively no junk or confusion is added. Thus the importing module is a conservative extension of the imported module. "Junk" means that new terms are added to the carrier set. "Confusion" means that distinct terms in the imported module are identified in the importing module. "`extending`" means that junk is allowed, but not confusion. This allows new elements to be added, whereas the predefined data remains unchanged. "`including`" is the most general form with only a few requirements; for example "`protecting`" and "`extending`" importation are not destroyed down the hierarchy. Again, this mechanism is necessary for the equational logic systems, not higher-order logic systems.

The most powerful modular mechanism of Maude is parameterized modules. Parameterized theories and parameterized views are two novel parametric mechanisms of Maude.

Parameterized modules. A parameterized module takes a module as input and returns a module as output, similar to a first-order functor in Mei. For example,

```
(fmod SET(X::TRIV) is
  sorts Set(X) NeSet(X) .
  ...
endfm)
```

defines a module `SET` parameterized by a theory `TRIV`.

A theory is used to declare the interface of a parameterized module. This is the reason that a theory has loose semantics. For example, the interface theory `TRIV` can be defined as follows:

```
(fth TRIV is
  sort Elt .
endfth)
```

A view is a named theory interpretation from a (source) theory to a (target) module that specifies how the target module satisfies the source theory. It is used as the parameter passing mechanism for the instantiation of a parameterized module. Similarly as in Mei, whether a view is indeed an interpretation is not checked by Maude. For example,

```
(view Int from TRIV to INT is
   sort Elt to Int .
endv)
```

defines a view named `Int` from the generic theory `TRIV` to the concrete module `INT`. It can then be used to instantiate the parameterized module `SET(X::TRIV)` to obtain a concrete module for a set of integers as in: `SET(Int)`.

Parameterized views. While a module can be built by instantiating a parameterized module with a view, there are cases where users want to build a new parameterized module from existing parameterized modules. This is achieved in Maude as *parameterized views* and instantiations of parameterized modules with parameterized views. For instance, `LIST(X::TRIV)` and `SET(X::TRIV)` are two parameterized modules. Suppose we want to build a parameterized module `LIST-SET(X::TRIV)` from them. We can define a parameterized view named `Set` as follows:

```
(view Set(X::TRIV) from TRIV to SET(X) is
   sort Elt to Set(X) .
endv)
```

`LIST(Set)`, an instantiation of `List` with the view `Set`, is then a parameterized module of lists of sets of elements. It can be instantiated by modules like `NAT`. As shown in Figure 5.1, a structured module is used in the instantiation and the structural information of the parameter module is kept. In fact, this is a composition of two parameterized modules in the sense that an application of `SET` is fed to `LIST` with the view `Set`.

This can be simulated in Mei as follows. Let $\mathit{LIST} : \mathsf{TRIV} \rightarrow \mathsf{LIST}$ and $\mathit{SET} : \mathsf{TRIV} \rightarrow \mathsf{SET}$ be functors. Define

$$\mathit{LIST\text{-}SET} \equiv \texttt{functor}\ \mathsf{X} : \mathsf{TRIV}.\ \mathit{LIST}\ (\mathit{SET}\ \mathsf{X}\ \texttt{with view}\ \mathsf{V}),$$

where V is a view similar to the view Set in Maude. By a little abuse of notation, $\mathit{LIST\text{-}SET} = \mathit{LIST} \circ \mathit{SET}$, the composition of *List* and *Set*. Clearly *LIST-SET* functions in the same way as $\mathsf{LIST(Set)}$.

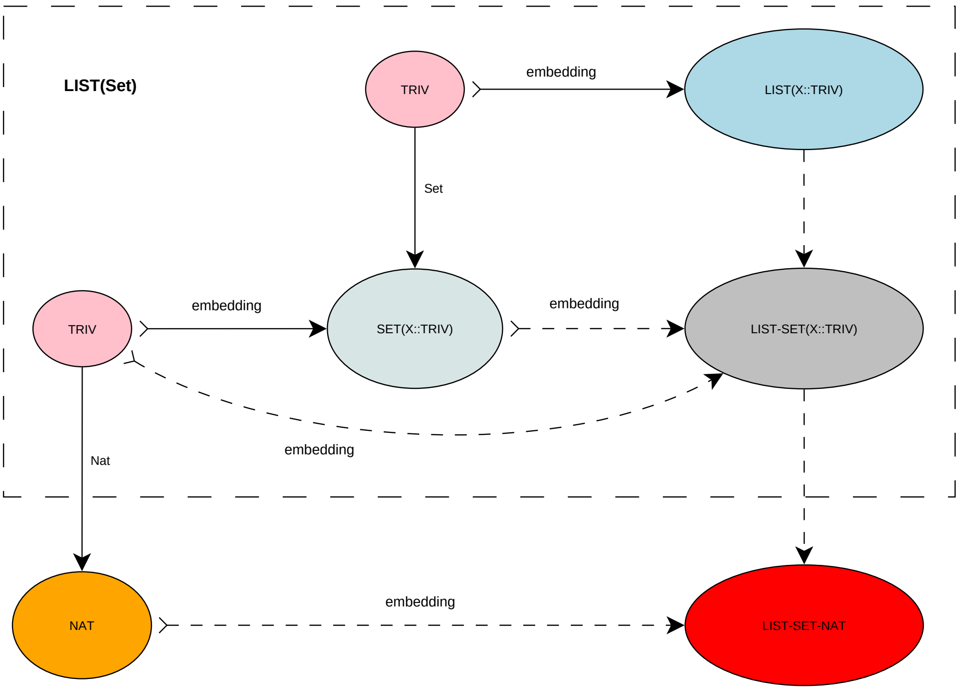

Figure 5.1: Instantiation by parameterized view

In addition, there is a notion of instantiation of parameterized views with views in Maude. For instance, Set(Nat), an instantiation of the parameterized view `Set` with the view `Nat`, is a view from TRIV to SET(Nat), a set of natural numbers. As seen in Figure 5.2, this is a composition of the view Set with a view from SET(X::TRIV) to SET(Nat), which is a view lifted from Nat. It can then be simulated in Mei by the view composition and view lifting mechanisms. In other word, an instantiation of parameterized views is indeed a view composition.

Parameterized theories. Maude also supports a notion of a parameterized theory, i.e. a theory that is parameterized by another (possibly parameterized) theory. For instance, we can define `LIST(X::SET(Y::TRIV))`, where `SET(Y::TRIV)` is a parameterized theory. Let `FINSET(Y::TRIV)` be a parameterized module of finite sets. Then

```
(view FinSet(X::TRIV) from SET(X) to FINSET(X) is
  sort Set(X) to FinSet(X) .
endv)
```

is a parameterized view named `FinSet`. `LIST(FinSet)` is then a parameterized mod-

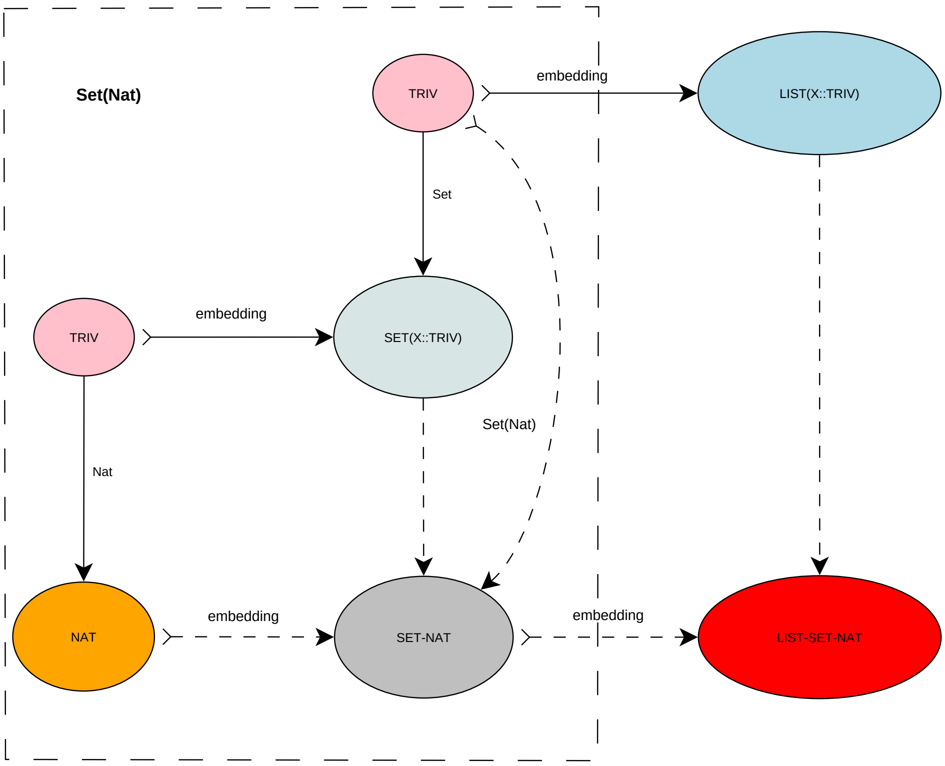

Figure 5.2: Parameterized view instantiation

ule of lists of finite sets of elements and it can be instantiated by modules like `NAT` as illustrated in Figure 5.3.

Note that `LIST` is *not* a higher-order parameterized module. Only modules can be used to instantiate it, not parameterized modules. The instantiation of `LIST` is enforced to be a parameterized module composition, since `FinSet` must be a parameterized view. In other words, a module parameterized by a parameterized theory can only be used to compose with other modules, and the parameterized theory specifies the class of modules with which it can be composed.

Parameterized theories correspond to a type $(\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1$ that does not exist in Mei. It specifies a class of expressions that have to be derived from functor applications. For instance, let $F :\ \mathsf{T}_1 \to \mathsf{T}_2$ and $X : \mathsf{T}_1$, then $(F\ X)$ is of type $(\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1$. $(F\ X)$ can then be used to instantiate a functor $G : ((\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1) \to T_3$. In Maude G is only used to compose with F as in the following context:

$$H = \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ G\ (F\ \mathsf{X}) = G \circ F$$

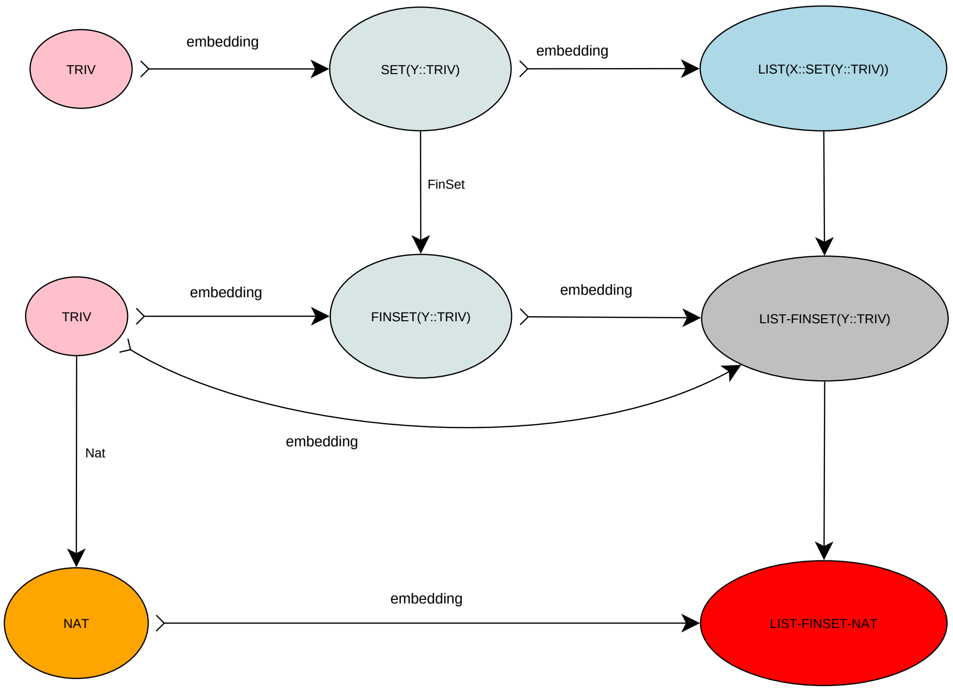

Figure 5.3: Instantiation of modules parameterized by parameterized theory

Parameterized theories also provide a simple sharing mechanism, e.g. there is one copy of SET in any instantiation of MOD(X::LIST(Z::SET),Y::STACK(Z::SET)), but two copies in one of MOD(X::LIST(Z1::SET),Y::STACK(Z2::SET)).

The new type is not necessary in Mei: (1) Declaring $G : \mathsf{T}_2 \to \mathsf{T}_3$ does not prevent us from composing G with F. There is no need to make the restriction that G can only be used to compose with $F : \mathsf{T}_1 \to \mathsf{T}_2$ for some fixed T_1. (2) Sharing is expressed by naming in Mei, i.e. the same name refers to the same object within a theory. (3) The new type complicates the type system. For example, H defined above will be typed $\mathsf{T}_1 \to ((((\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1) \to \mathsf{T}_3)\ ((\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1))$, which is hard to comprehend.

5.2.2 Specware

Specware [90] has a novel module system. It follows closely a category-theoretic approach. Specware supports mainly three specification structuring mechanisms: *translation*, *importation*, and a powerful *colimit* operation over *diagrams*. Translation and importation are the same as our renaming and extension respectively. The colimit operation over diagrams is the core modular mechanism of Specware. Although most algebraic specification languages use the notion of a colimit to define the semantics, very few of them directly support the colimit operation as in Specware.

A diagram is a directed multigraph where specifications are nodes and morphisms are edges. A colimit of a diagram is a specification constructed by first taking the disjoint union of the node specifications and then building the quotient from the equivalence relations induced by the morphisms. The composition of specifications is then constructed by building a diagram and calculating its colimit. For instance, a parameterized specification can be seen as a diagram consisting of two nodes, the actual parameter specification and the parameterized specification, and one edge, the morphism between them. The instantiation of a parameterized specification is a diagram obtained by expanding the diagram for the parameterized specification by a node for the actual parameter specification and an edge for the corresponding fitting morphism. The semantics of an instantiation is then defined as the colimit of the appropriate diagram.

The current version of Specware supports a notion of substitution, which is similar to parameterized specification. Instead of fixing the parameter, any subspecification in the importation hierarchy can be used as the formal parameter. The formal parameter actually used for instantiation is determined by the fitting morphism, i.e. its source specification. To make it more practical, it is necessary to build a more sophisticated mechanism of parameterized specifications on top of the colimit mechanism. In fact, the parameterized theories and parameterized views in Maude can be translated directly to diagram constructions, as shown in [33].

In one sense, the colimit operation is more general than the parametric mechanism in Mei, since the colimit is defined over arbitrary diagrams that may not be representable in Mei. In another sense, the parametric mechanism in Mei is more general, since it supports higher-order functors, which do not have a counterpart in Specware. In addition, functors in Mei may contain multiple copies of the parameter theory along different morphisms. The semantics of functor instantiations is then akin to the multiple pushouts in [71], which may not be expressed by the colimit operator. One particular drawback of Specware is that a solid category theory background is required in order to use it.

5.2.3 CASL

Casl [3, 4, 7, 48] is an algebraic specification language based on partial first-order logic. A basic specification consists of a signature Σ and a set of sentences (axioms or constraints) over Σ. A signature declares a set S of sorts, sets $TF_{w,s}$ and $PF_{w,s}$ of total function symbols and partial function symbols respectively, and a set P_w of predicate symbols, where w is a sequence of argument sorts and s is a result sort, called function and predicate profiles respectively. Signatures are related by a signature morphism: $(S, TF, PF, P) \to (S', TF', PF', P')$, which consists of a mapping from S to S', and, for each profile, a mapping between the corresponding sets of function and predicate symbols respectively [3].

Casl provides a number of mechanisms for structuring specifications, summarized as follows [48].

- Translation: *SP* `with` *SM*.
  This is a renaming mechanism and *SM* is a symbol mapping. The signature Σ given by *SP* and symbol mapping *SM* together determine a new signature Σ' and a morphism from Σ to Σ'.

- Reductions: *SP* `hide` *SL* and *SP* `reveal` *SM*.
  In the case of a hiding reduction, the signature Σ' determined by the signature Σ given by *SP* and the set of symbols listed by *SL* is the largest subsignature of Σ that does not contain any of the listed symbols. Note that hiding a sort entails hiding all the operations and predicate symbols whose profiles involve that sort.

  In the case of a revealing reduction, the signature Σ' determined by the signature Σ given by *SP* and the set of symbols mapped by *SM* is the smallest subsignature of Σ that contains all of the listed symbols. Note that revealing an operation or predicate symbol entails revealing the sorts involved in its profile.

- Union: SP_1 `and` ... `and` SP_n.
  The signature of the union is obtained by the ordinary union of the Σ_i (not their disjoint union). Thus all (non-localized) occurrences of a symbol in the SP_i are interpreted uniformly. If the same name is declared both as a total and as a partial operation with the same profile (in different signatures), the operation becomes total in the union.

- Extension: SP_1 `then` ... `then` SP_n.
  SP_1 determines an extension from the local environment to a complete signature

Σ_1. For $i = 2, ..., n$ each SP_i determines an extension from Σ_{i-1} to a complete signature Σ_i. The signature determined by the entire extension is then Σ_n.

- Local specification: `local` SP_1 `within` SP_2.
  This is equivalent to: $\{SP_1$ `then` $SP_2\}$ `hide` SY_1, ..., SY_n,
  where SY_1, ..., SY_n are all the symbols declared by SP_1 that are not already in the current local environment.The hiding must not affect symbols that are declared only in SP_2.

- Named specification with parameters:
  `spec` SN $[SP_1]$ ... $[SP_n]$ `given` SP''_1, ..., $SP''_m = SP$ `end`.
  This defines the name SN to refer to a *generic* specification with respect to a (possibly empty) list of parameters SP_1, ..., SP_n. SP''_1, ..., SP''_m is a (possibly empty) list of import specifications. The generic specification SP' is essentially the union of imported specifications extended by the union of parameter specifications, extended by the body SP:
  $\{SP''_1$ `and` ... `and` $SP''_m\}$ `then` $\{SP_1$ `and` ... `and` $SP_n\}$ `then` SP
  The generic specification can be instantiated as $SN[FA_1]...[FA_n]$, where FA_i is a *fitting argument*: SP'_i `fit` SM_i. Each FA_i determines a fitting morphism from SP_i to SP'_i. The fitting argument morphism has to be the identity on all symbols declared by SP''_1, ..., SP''_m. Lifting the fitting morphisms FA_1, ..., FA_n yields a morphism FM applicable to SP'. The instantiated specification $SN[FA_1]...[FA_n]$ is:
  $\{SP'$ `with` $FM\}$ `and` SP'_1 `and` ...`and` SP'_n.
  The instantiation is a push-out of the body and argument signatures. Parameter matching may also be achieved by (explicit) use of named views between the parameter and argument specifications.

- View: `view` VN $[SP_1]$ ... $[SP_n]$ `given` SP''_1, ..., SP''_m : SP `to` $SP' = SM$ `end`
  This defines, according to a symbol mapping SM, a morphism from SP to a parameterized specification which has a body SP', a list of parameters SP_1, ..., SP_n, and a set of imports SP''_1, ..., SP''_m. Therefore the same view can be instantiated with different fitting arguments, giving compositions of the morphism defined by the view with other fitting morphisms. Note that the source SP of the view is not in the scope of the view parameters SP_1, ..., SP_n, and view instantiation affects only the target of the generic view.

Most of these operations have their counterparts in Mei except for the specification reductions, local specifications, and parameterized views.

(1) Specification reductions hide auxiliary information, which has the same functionality of up-casting in Mei, i.e. casting a theory to its supertype. The supertype effectively shows the revealing part. Reductions can then be implemented in terms of up-casting.

(2) Since local specification can be implemented on top of reduction, they can be then implemented on top of upcasts.

(3) A view in Casl is a restricted version of a parameterized view in Maude, in that (a) the parameter specification occurs only in the target specification, (b) the parameter specification is not parameterized, and (c) the parameterized view has to be instantiated when it is referred. Hence they can be implemented by view compositions and view lifting in Mei, as shown in 5.2.1.

5.2.4 An algebraic framework for higher-order modules.

The module system [54] proposed by Jimémez and Orejas is the system closest to our system. The major goal of this system is to integrate the λ-calculus style higher-order modules and the flexible fitting morphism style parameter passing. In contrast to our coercion approach, they adopt the pushout (in fact multiple pushout) style semantics for the instantiation of the parameterized specifications, and generalize the idea to account for higher-order modules.

A category, *MSpec*, is introduced to denote parameterized specifications. The category *Spec* of basic specifications is a full subcategory of *MSpec*. The objects in *MSpec* are either objects in *Spec* or triples $\langle IMP, RES, F\rangle$. *IMP* and *RES* are both objects in *MSpec*. F is a morphism from *IMP* to *RES* showing how *IMP* is used to build *RES*. Note that an object in *MSpec* has two roles, as a specification and as a type of specifications. *IMP* and *RES* are roughly the argument type and the result type of a parameterized specification. Since $\langle IMP, RES, F\rangle$ is an object in *MSpec*, it can be used as a type in defining other objects in *MSpec*. The morphisms between objects in *Spec* are exactly morphisms in *Spec*. A morphism $h : \langle IMP, RES, F\rangle \rightarrow \langle IMP', RES', F'\rangle$ is a pair $\langle h_1, h_2\rangle$, where $h_1 : IMP' \rightarrow IMP$ and $h_2 : RES \rightarrow RES'$. This is very similar to our definition of a view. One more restriction on a higher-order morphism is that it must respect F and F', in the sense that RES' is built from IMP' (expressed by F') following exactly the same way that *RES* is built from *IMP* (expressed by F). A notion of instantiation is then defined over *MSpec*. Given MSP_{act} and $MSP = \langle MSP1, MSP2, F\rangle$ in *MSpec*, the result of the instantiation of *MSP* by MSP_{act} via h is a parameterized specification $MSP_{res} = MSP(MSP_{act})_h$ and a specification morphism $h' : MSP2 \rightarrow MSP_{res}$.

A class of specification expressions, including λ-abstraction, application, and morphism expressions, are then defined. A denotational semantics of the expressions is defined in terms of *MSpec*. In particular, the semantics of application is defined via the *instantiation* of parameterized specifications.

A notion of substitution is defined for the specification expressions, followed by a definition of β-reduction, giving an operational semantics for the specification expression. The substitution is more general in the sense that the component specification expressions, as well as the component morphism expressions involved, need to be handled properly. By systematically manipulating the substitution of both specification and morphism expressions, there is no need to coerce the parameter specification before the instantiation of the parameterized specifications. This is the major contribution of this work.

The operational semantics is proved to be correct with respect to the denotational semantics. However, it is not completely operational for two main reasons:

(1) The substitution of a specification expression is defined in terms of the substitution of its component expressions, both specification and morphism expressions. While the denotational semantics of the substitution of morphism expressions is defined in terms of *MSpec*, its operational semantics is not defined. In other words, the operational semantics of a specification expression is defined in terms of the operational semantics of its component specifications and the denotational semantics of its component morphisms. This makes the substitution of specification expressions not operational. As a result, β-reduction is not operational.

We believe that the operational semantics of a specification expression should be defined in terms of the operational semantics of its component specifications and the operational semantics of its component morphisms. As a result, the substitution function of a specification expression should return two values, a specification expression and a morphism expression, that show the relation between the expressions before and after the substitution, as in the definition of instantiations for *MSpec*. Therefore, we might need to define substitution functions for both specification expressions and morphism expressions mutually.

(2) Although it is declared that a specification expression of sort *Spec* can have com-

> ponent expressions in *MSpec*, e.g. applications of parameterized specifications, the substitution function of specification expressions is only defined for those specification expressions of sort *Spec* whose components are in *Spec*. The substitution function is defined in terms of an unspecified function *assign*, which is only defined over *Spec*. In other words, the substitution function will halt once a specification expression of sort *Spec* is encountered, even though it may have component expressions in *MSpec*.
>
> The *assign* function is not fully specified in [54]; therefore there is no way to justify its correctness. However, we can conclude that, since *assign* is not defined in terms of the substitutions of its component expressions, β-reduction is not operational. We believe that an explicit definition of the *assign* function, mutually defined with the substitution function, is needed in order to give an operational semantics, i.e. β-reduction.

Hence, the operational semantics does not provide an obvious guideline for the implementation. A complete operational semantics should define the *assign* function and the substitution functions for specification and morphism mutually, which would be much more complicated than the current definition. This added complexity is another drawback of this system, making it both hard to understand and hard to implement.

In addition, using a (parameterized) specification in *MSpec* as the argument type in defining another specification might possibly expose more information than we want. In case the parameter itself is a parameterized specification, the fitting morphism has to respect its argument and result specifications as well as the the morphism from the argument to the result. This extra restriction is not an accident. However, there are cases in which we do not want to show how the result specification is constructed from the argument specification in the type. Therefore all the parameterized specifications with the right argument specification and result specification, regardless of how they are constructed, can be used as the actual parameter matching that type. This case is not allowed in [54].

Nevertheless, we are in favour of our approach from the user's point, as shown in the following example. Assume that we are defining a module expression "`E extended by S`", where `E` is a complicated expression possibly containing functor applications. Following our approach, we can figure out the set of symbols we can use in defining `S` by the type of `E`. Following the approach in [54], we are forced to evaluate `E` in order to figure out the set of symbols. In the presence of higher-order functors, evaluation can be quite costly, while the type of `E` is much easier to compute. On the other hand, our approach can be seen as an instance of their approach where we fix the language of the pushout to be an extension of that of the result type, i.e. derived from that of the argument type instead of the parameter. However the structure of the parameter is kept during coercion.

5.3 Theorem provers

We will see in this section that the modular mechanisms of theorem provers are weak, making them the right application area for Mei. We will focus on the parametric modular mechanisms in theorem provers.

In addition to the modularity mechanisms supported by Mei, we also discuss the support of theory interpretation. Although a theory view in Mei is not necessary a theory interpretation, it is intended to represent a theory interpretation. When integrating Mei with a theorem prover, it is nicer when Mei can send a request to the theorem prover to verify that a theory view is indeed a theory interpretation. Thus it is important for a theorem prover to support a sophisticated theory interpretation mechanism. In addition to being used as a parameter passing mechanism, a theory interpretation helps a user to exploit a theory, i.e. prove more theorems, by reusing the theorems developed in the other theories (contexts). It is worthwhile discussing theory interpretations in both roles.

5.3.1 IMPS

IMPS is an interactive theorem prover designed to support mathematical reasoning [36, 37, 38]. Two of its key concepts are *little theories* and *theory interpretation*. Theories are at least as important as their component theorems in IMPS. The activities of IMPS can be seen as creating, developing, and linking theories. Full discussions of the little theories approach and theory interpretation can be found in [34, 38]. The definition of theory interpretation in section 3.4.1 is directly derived from IMPS.

An IMPS theory is constructed from a (possibly empty) set of subtheories, a language, and a set of axioms. The subtheory relation and theory interpretation are ways to relate theories. A theory hierarchy can then be constructed via the subtheory relation. For instance, a theory of monoids is a subtheory of a theory of groups. In Mei, this is expressed by the union and extension operations.

In IMPS, all theories are generic and defined over arbitrary types. They can be instantiated in a polymorphic manner via an explicitly defined theory interpretation.

This is similar to the parameterized theories in the algebraic specification languages and first-order functor in Mei, as shown in Figure 5.4.

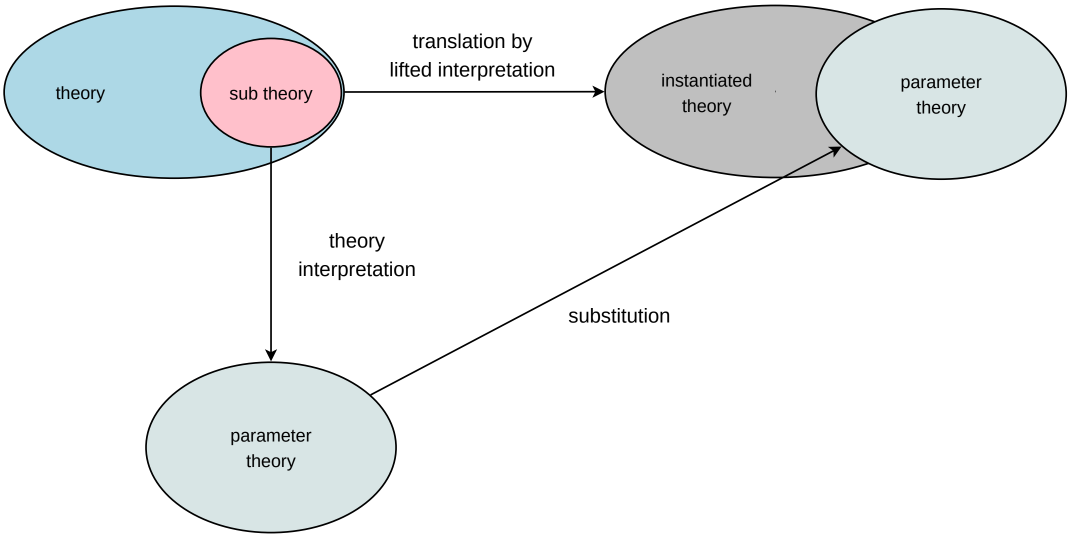

Figure 5.4: IMPS Parameterized Theory Application

The major differences are : (1) In IMPS, there is no separate notion of parameterized theory. The subtheory used as the interface is identified at the time of instantiation by building an interpretation from it to the actual parameter theory, i.e. the target theory of the interpretation as in Specware. (2) Only one occurrence of a subtheory can be replaced for each instantiation.

On the one hand, our parameterized theories in Mei are more general than generic theories in IMPS, since we support high-order functors. On the other hand, they are not as flexible, since the formal parameter theory is fixed, and hence can be instantiated for fewer actual parameters. We consider this inflexibility as an advantage of our approach. It gives more information about the intension of functors, and therefore, is easier to use.

Beside being used as the parameter passing mechanism, an explicit interpretation mechanism as in IMPS is very strong, flexible, and applicable in many circumstances. For instance, theorems proved in an abstract theory can be used in any concrete context via a theory interpretation. The advantages of this approach are: (i) properties of the abstract theory are proved once and for all; (ii) properties can be stated neatly in that context and information is isolated by the abstraction. A special use of this approach is an interpretation from an abstract theory to itself which supports symmetry and duality arguments in an automatic way. Rather than helping to build new theories from existing ones, this kind of use of the theory interpretations effectively links the theories and helps to exploit them further by using theorems developed under related contexts (theories). While our concerns is "*How to construct new modules*", IMPS is concerned with another issue of a module system, "*How to link existing modules*".

5.3.2 PVS

PVS stands for Prototype Verification System [72, 73, 74, 88]. It is an environment for specification and verification. A PVS specification is a collection of theories. A PVS theory consists of a theory name, a list of formal parameters, an export part, an assumption part, and a theory body. The theory body is the main part of the theory, consisting of top-level importations, axioms, and theorems. Exporting is a mechanism to hide some names declared in the current theory, and it can be simulated by up-casting in Mei as shown in §5.2.3. The assumption part gives constraints over the current theory, which have to hold for any instance of the theory. Internally, the assumptions are the same as axioms. Externally, they generate obligations which must be proved for each import of the theory [72, 74]. The importation mechanism is similar to that of IMPS and hence can be expressed by union and extension in Mei.

What we called "parameterized theories" are termed *theories as parameter* in PVS [73]. They are roughly first-order functors in Mei.

```
group_homomorphism[G1, G2: THEORY group]: THEORY
BEGIN
    x, y: VAR G1.G
    f: VAR [G1.G -> G2.G]
    homomorphism?(f): bool = FORALL x, y: f(x ∘ y) = f(x) ∘ f(y)
    ...
END group_homomorphism
```

Symbol names are qualified by the theory names, i.e. we will have (unlike the case in Mei) that G1.G $\neq$ G2.G. Effectively, two copies of the group theory are parameters of the generic theory group_homomorphism. Theory interpretations are used as parameter passing mechanisms. Two different theory interpretations can be provided to instantiate the theory group_homomorphism.

Parameterized theories and theory interpretations. PVS provides two notions related to theory interpretation: *parameterized theory* and explicit *theory interpretation* [73].

A parameterized theory in PVS is parameterized by generic types and constants specified in the formal parameter list. A generic type parameter is an uninterpreted type. Axioms are simulated by the assumptions over these formal parameters. This provides a way of presenting abstract theories. For example, an abstract theory of groups is defined as follows:

```
group[G: TYPE, ∘ :  [G, G -> G], e:  G, inv:  [G -> G]]: THEORY
BEGIN
  ASSUMING
  a, b, c:  VAR G
  associativity :  ASSUMPTION a ∘ (b ∘ c) = (a ∘ b) ∘ c
  ...
  ENDASSUMING
  leftcancellation :  THEOREM ...
  ...
END group
```

An instance of a group can be imported by providing all actual parameters of group theory corresponding to the formal parameters, like `group[int, +, 0, -]`. This actually supplies a translation from the abstract (source) theory to the concrete (target) theory. The assumptions are then checked to generate TCCs (type correctness conditions) which are essentially proof obligations. Once the TCCs are proved, an interpretation is constructed and all interpreted theorems of the abstract group theory are available in the (target) theory importing it.

The explicit interpretation mechanism resembles much of its theory parameterizations mechanism, with axioms replacing the corresponding assumptions. An explicit mapping is specified for uninterpreted types and constants of the source theory into the current theory. The interpreted theorems are considered proved and available for use if they are proved in the abstract theory. The group example above can then be reformalized as follows:

```
group:  THEORY
BEGIN
  G: TYPE
  ∘:  [G, G -> G]
  ...
```

```
    associativity :  AXIOM FORALL a, b, c:  a ∘ (b ∘ c) = (a ∘ b) ∘ c
    ...
  END group
```

The interpretation can then be used in the context when the group theory is imported.

```
    IMPORTING group{{G := int, ∘ := +, e := 0, inv := -}}
```

5.3.3 Isabelle

Isabelle [56, 70] is a generic theorem prover which supports user-defined logics. The primitive theory system of Isabelle provides only a theory importation mechanism, which can be expressed in Mei by the union and extension operations. Although L. Paulson [76] proposed and D. Aspinall [2] implemented an ML-family module system for Isabelle, it does not seem to be an official part of the current Isabelle distribution. In addition, Isabelle's meta-logic is a higher-order constructive logic with Π and Σ types in which dependent records can be modeled. Modules can then be represented as dependent records [22]. Since modules are first-class citizens, functors are then functions over dependent record types. This will resemble the ML-family module systems. However, no module system based on dependent records is implemented in the current Isabelle distribution.

Axiomatic type classes and locales. Isabelle provides two other notions: *axiomatic type classes* [96] and *locales* [6], for modular development of theories. Both are intended to resolve the tension between theory reuse and the concise expression of theorems, where the former desires minimal context but the latter asks for maximal context. This is exactly the purpose of theory interpretations.

An axiomatic type class in Isabelle is similar to a type class in *Haskell* [27], a functional programming language. A user can declare polymorphic constants over types in a particular type class and assert axioms about these constants. Consequently, theorems can be proved over these constants and types in this type class, and can be reused within any types in this particular type class. In order to use the theorems proved for a type class at a particular type, we need to prove that this type is an instance of the type class. A mapping needs to be constructed to connect the polymorphic constants declared in a type class (abstract theory) with a concrete expression over this particular type, which is essentially a translation from the abstract theory to the concrete theory. Then all axioms asserted for the type class need proofs with respect to the mapping in the context of the concrete type. This is exactly the same as proving the obligations generated by an interpretation. However, there is no syntactic mapping. That is, the fixed constants in the axiomatic class are the same as those used the special type, although their definitions are not the same in different concrete types. Similarly the theorems proved for the axiomatic class are syntactically identical over all types in that class, although their meanings differ. In particular, if some fixed constants in the axiomatic class are already defined in the special type for other usage, there is no easy way to resolve the name conflict. This significantly restricts the use of axiomatic classes. In addition, axiomatic type classes currently relate to one type only, so that parameterisation involving multiple types is not possible [77].

A locale is a device to isolate an environment for theorem development and theorem reuse. A locale represents a theory. Within a locale, constants of the theory are fixed and axioms are assumed. Consequently, theorems of the theory are expressed over the constants and proved with respect to the axioms. Outside the locale, primitive constants of the theory are parameters, and axioms are assumptions about the formal parameters. Every locale effectively defines a predicate over parameters, which encodes axioms. Then theorems of the locale can be exported to the outside world with the defined predicate embedded as an assumption. The exported theorems are generic, in that they can be applied to any concrete structure if we can show that the predicate is true over the actual parameters. The matching of formal parameters and actual parameters is indeed an interpretation (translation) and the predicate over actual parameters is the encoding of the generated proof obligations in terms of IMPS terminology. This is an advantage of locales over axiomatic type classes.

Locales can be imported, renamed, and merged. Effectively, locales are variants of theory interpretations similar to the parameterized theory and theory interpretation mechanism in PVS. Opening a locale in Isabelle is then the same as importing an instantiation of a parameterized theory or importing an abstract theory with an interpretation in PVS. However, the generic theorems exported by a locale can be used individually without opening the locale, since the predicates embedded in the theorems explicitly specify the language and axioms of the locale. Conceptually, locales can replace Isabelle theories since they possesses all functionalities of theories. Locale expressions provide a more flexible way for combining locales than theory constructions, in particular locales can be renamed and merged. This is similar to the theory operations in Mei. However, neither generic locales nor generic theories, in the sense of functors, is supported in Isabelle [6].

5.3.4 Coq

Coq [63] is a theorem prover based on the Curry-Howard correspondence. Stating a theorem is writing a well-formed type, and proving this theorem is finding a term of that type. Consequently, proof checking is reduced to type checking. Although Coq's logic is strong enough to formalize a module system as dependently typed records, J. Chrzkszcz [17, 18] implemented a stratified ML-family module system following Leroy's manifest type approach [57]. The basic module expressions include structure, signature, (higher-order) functor, and functor signature. Modules (structures) group together related concepts, and higher-order modules (functors) provide module abstraction. Structures, signatures, functors, and functor signatures correspond to theories, theory types, functors, and functor types in Mei respectively.

Coq supports an importing mechanism akin to that of PVS. Whereas the importing relation is transitive, the visibility is not. Considering the case that module `M` imports module `N` and module `N` in turns imports module `R`, then `N` and `R` are respectively visible in `M` and `N`, whereas `R` is not automatically visible in `M`. Using `Require Export R` in module `N` makes it visible in `M` [63].

The module system of Coq is similar to Mei Core, except for its support of translucent types. The major advantage of Mei over Coq's module system is the fitting morphism style parameter passing mechanism, which allows modules defined in different languages to instantiate a functor, as long as the modules share the same structure with the formal parameter of the functor.

5.3.5 Automath

Automath [10, 11] is a language for expressing entire mathematical theories in such a way that the their correctness can be verified by a computer system, e.g. a theorem prover. Same as in Coq, Automath employs the Curry-Howard correspondence. Thus stating a theorem is writing a well-formed type and proving this theorem is finding a term whose type corresponds to the theorem in question. The major modularity mechanisms of Automath are *books* and *telescopes*.

Mathematical knowledge is organized as Automath books[1]. An Automath book

[1] Most of this subsection about Automath is taken from [100].

consists of a sequence of lines. There are three kinds of lines:

(1) Context line.

$$a_1 : F_1, \ldots, a_n : F_n$$

A context line introduces a set of variables with certain types or satisfying certain assumptions. Note that dependencies may happen here, in the sense that F_i may employ $a_1, \ldots, a_{i-1}$.

(2) Definition line.

$$a_1 : F_1, \ldots, a_n : F_n \vdash C := E : F$$

This introduces either a constant C of type F with the definition E, or a theorem C where F is the sentence of C and E is one of its proofs.

(3) Primitive notion line.

$$a_1 : F_1, \ldots, a_n : F_n \vdash C := \texttt{prim} : F$$

This introduces either a primitive constant of type F or an axiom (i.e. a sentence with a primitive proof).

Essentially a book is a Mei theory. However contexts of different books are not related. Theorems in different books have to be proved from their own contexts even though they are essentially the same theorem. To deal with this problem, Automath uses the notion of *telescopes*. A telescope encodes a context as follows:

$$[a_1 : F_1] \ldots [a_n : F_n].$$

For example a telescope representing a theory of groups is:

$$[gg : Type][e : gg][\circ : gg^2 \to gg][\ ^{-1} : gg \to gg][p : \texttt{Axiom}_{\texttt{group}}],$$

where $\texttt{Axiom}_{\texttt{group}}$ is the conjunction of the axioms for a theory of groups. A telescope defines a "type". A sequence of values $(v_1, \ldots, v_n)$ *fits* a telescope if

$$v_1 : F_1, \ldots, \text{and } a_n : F_n.$$

As stated in [100], a telescope functions like a dependent record type [20]. The idea behind telescopes is very similar to the idea behind Isabelle's locales. Both are intended to solve the problem of theory reuse (which is closely related to generic mechanisms like Mei's functors). The predicate induced by a locale can be seen as a function whose argument type is a dependent record type which is indeed a telescope. Roughly speaking a book can be seen as a theory, and a telescope as a theory type in Mei.

5.4 Computer algebra systems

The modularity mechanisms of computer algebra systems are diverse. Some, like Mathematica, only support preliminary modularity mechanisms. Others support first-order parameterized modules under different terminologies. Since computation is emphasized rather than theorem proving, theory interpretation is not employed. Therefore, some kind of type matching is used for parameter passings.

5.4.1 Maple

Maple [67, 68] is a computer algebra system that combines a programming language with an interface that handles mathematical expressions in traditional mathematical notation. Maple is interactive and the programming language is interpreted and dynamically typed.

The major modular mechanism of is Maple modules [68]. The following is a simple Maple module:

```
module()
   export e1;
   local a, b;
   global message;
   a := 2;
   ...
end module
```

A Maple module encapsulates a group of reusable variables and Maple commands. Local variables are not visible outside the definition of the module in which they occur [68]. The difference between global variables and exported local variables is that exported local variables have to be accessed by using the :- member selection operator. A Maple module is very similar to a Maple procedure. While a Maple module has only one instance when it is loaded and exists until it is unloaded, a Maple procedure can be called multiple times creating multiple instances, since local variables of a Maple procedure can exist after the termination of the procedure.

Maple modules are used in various ways to model Pascal-style records, packages, objects, as well as generic programming.

- A Pascal-style record is simulated by a module containing only exported local variables. Syntactic sugar is provided so that the users can use Pascal-style record syntax.

- Maple modules can be use to create a Maple package which is a collection of procedures and other data that can be treated a whole. This is the typical use of Maple modules. Maple modules and Maple packages are distinguished. Maple packages can also be created by using tables which is not of our concern.

- Maple modules can be used to simulate objects with both state and behaviour. The state of an object is represented by the local and exported local variables. A constructor is simulated by wrapping a module with a procedure with certain parameters. This is also called a parameterized module in Maple.

  ```
  MakeComplex := proc(real, imag)
     module()
        description "a complex number";
        export re, im, abs, arg;
        re := () -> real;
        ...
     end module
  end proc:
  ```

  The value of the last statement within the procedure is returned as the value of a procedure call. For example, `MakeComplex (4,5)` returns a Maple module representing a complex number. By wrapping a Maple module with a procedure, we can have multiple instances of a Maple module with different states. This essentially simulates objects where the Maple module is a class and a procedure call of the constructor (the wrapping procedure) creates an object of that class.

- By supporting the notion of interfaces, Maple also supports generic programming. A Maple interface is a sequence of symbols. For example, a Maple interface for a ring can be written as

  ```
  '\type/Ring\' := ' 'module' (
     '+'::procedure,
     ...
     zero, one
  )':
  ```

A generic procedure can be defined as follows:

```
calRing := proc (R::Ring)
   ...
end proc:
```

A generic object constructor can be defined similarly.

```
newModule := proc (R::Ring)
   ...
   module()
      ...
   end module
end proc:
```

The Maple module defined within `newModule` can use symbols declared in the type `Ring` to define new mechanisms. When `newModule` is applied to a Maple module that is an instance of type `Ring`, a concrete instance of the Maple module defined within `newModule` will be created.

While Maple provides several interesting use of Maple modules, the last use, generic object constructors, is the most interesting one. This essentially simulates Mei's first-order functors with the help of Maple procedures. The major difference is that Maple's generic object constructors are generative functors that create incompatible instances of modules with states. Unlike Mei, Maple does not support module operators such as extension, union, and renaming.

5.4.2 Mathematica

Mathematica [43] is a computer algebra system and the major competitor of Maple. It provides an interface for creating and manipulating programmatic structures that includes graphics, mathematics, program code etc.

Although Mathematica is a powerful system, it has a simple module notion, *package*. A package is a collection of functions, some of which are *local* to the package. From the user point of view, a package is just a namespace for those exported symbols. When a package is loaded, all its exported functions can be access directly if the package is on the context path which is a set of package names.

5.4.3 Axiom

Axiom [53] is a computer algebra system that is especially useful for symbolic calculations, mathematical research, and the development of mathematical algorithms. It has a strongly typed high-level programming language for expressing abstract mathematical concepts. Axiom defines a strongly typed, mathematically correct type hierarchy, for mathematical objects (such as rings, fields, and polynomials) as well as data structures from computer science (e.g. lists, trees, and hash tables). "*The crucial strength of AXIOM lies in its excellent structural features and unlimited expandability—it is an open, modular system designed to support an ever growing number of facilities with minimal increase in structural complexity.*" [53] The major modular notions of Axiom are *packages*, *domains*, and *categories*.

The most important modular mechanisms are domains and categories. Axiom deals with many kinds of mathematical objects such as numbers and polynomials. It organizes these objects using domains. Every Axiom object belongs to a domain. A domain is an abstract data type whose data representations and behaviour implementations are hidden. Categories are types of domains. A domain can be asserted to be in a category if it implements all the exported operations declared in the category. In terms of our modular notions, a domain is roughly a theory, and a category is a theory type. The crucial difference between domains and theories is that users can refer to objects of other domains within the current domain since Axiom follows the big theory approach[2] as all other CASs. A typical definition of a domain is:

```
DomainForm : Exports == Implementation where

  [type declarations]

  Exports == [Category Assertions] with

    list of exported operations

  Implementation == [Add Domain] add

    [Rep := Representation]
```

[2] Axiom *could* follow the little theories approach, but in practice this does not seem to be the case.

list of function definitions for exported operations

There are several interesting features in this simple definition form:

(1) The *Category Assertions* list all the categories of which the domain is a member. Since categories are actually types of domains as we will investigate later, *Category Assertions* are type annotations. Note that a domain can be of more than one category.

(2) The *list of exported operations* declares a set of exported operations of the domain which forms an interface of the domain. This, in our system, is handled by the type of theories due to the dual roles of module types.

(3) The *Add Domain* can be seen as a domain on which the current domain is built. If an implementation for an exported operation is not provided, Axiom will goes to the add domain to find an implementation.

(4) The *list of function definitions for exported operations* provides the real implementations.

(5) The *Representation* is a really novel feature of Axiom. The representation is usually another domain which is the concrete representation of the current domain. Within the implementation part, this representation domain is treated as equal to the current domain. The implementation of operations of the current domain will be written in terms of the operations of the representation domain. We may view this as the adapter pattern in the object oriented design where the current domain builds an adaptor between the export interface of the current domain and the representation domain. In our module system, this can be simulated by two abstract types within one theory in which the equivalence of these two types is expressed axiomatically. However, only the type corresponding to the current domain is declared in the type of the theory whereas the type corresponding to the representation domain and the equivalence axiom are hidden. Although it can be done in other systems as we showed above, we need to emphasize that Axiom gives a neat way to present the idea of representations.

In addition, there are parameterized domains in Axiom. Axiom has a notion of *domain constructors* which can take input and produce new domains. A domain is then a domain constructor without any parameter. In this case, the *Category Assertions* assert that all domains created by the constructor are members of these categories regardless the value of the parameter. Domain constructors with parameters are similar to our first-order functors. They are more general than first-order functors in the sense that they can take any object as input, not necessarily a domain. For instance, a domain constructor can take an integer as input to build a domain of vector spaces with particular dimension. When a domain is required by a domain constructor, a category is used to specify the collection of domains that can be used as input.

Packages are just special domains without states. A package does not denote a class of objects but a collection of operations. Whereas a domain abstracts both data and behaviour, a package abstracts only behaviour. Syntactically, a package definition does not have *Category Assertions*, *Add Domain*, and *Representation*. Like domains, packages can have parameters whose type can be either a category or a domain.

Now let us look at categories closely. Roughly speaking, a category is a type representing a collection of domains. It specifies a set of operations exported by the domains of the category. Moreover it specifies a set of properties by axioms that must be satisfied by the domains of the category. This is similar to our theory types. However, axioms are not presented in any domain.

In addition, a category can be built by extending another category. This effectively builds a hierarchy of categories. For instance, a `SemiGroup` category can be built by extending a `Set` category. There are several interesting issues regarding this category hierarchy:

(1) There is a special category `Type` which is the root of the hierarchy.

(2) Two major branches in Axiom are the basic algebra hierarchy and the data structure hierarchy.

(3) The structure of a category specifies the construction of the category but not the construction of the domains in this category. When a category is listed in a *Category Assertions* part of a domain, only the flat content of the category matters not its position in the hierarchy.

(4) The hierarchy of categories effectively defines a subtype relation between categories. If a category B extends another category A, then B is a subtype of A. This is because that, if a domain is a member of a category B which extends a category A, it is a member of the category A. This is similar to the idea of subclass as subtype. It is safe here because that the category B inherits not only the operation prototypes of the category A but also the behaviour of the operations enforced by inheriting the axioms.

Like domains, categories that have parameters are called category constructors. Note that a category constructor only expresses how a category can be constructed from its parameter(s). In general, it says nothing about how a domain of a category is constructed from its parameter(s). For instance, it is perfectly fine to assert a domain constructor to be a member of a category without any parameter as long as every domain constructed by the domain constructor is a member of the category. Comparing Axiom's module system with Mei:

(1) Taking domains as theories and categories as theory types, Axiom supports union and extension. However, renaming is not supported.

(2) Axiom supports first-order functors in terms of domain constructors. However, higher-order functors are not available. Only a category or an instance of a category constructor, not a category constructor alone, can be used to specify the type of the parameter of a domain constructor.

(3) Axiom supports a subtyping over categories which is defined via the hierarchy of categories. It is simpler than Mei's subtyping relation.

5.4.4 Aldor

Aldor [81, 94] was originally the programming language of Axiom but is now used more in other settings. It combines imperative, functional, and object-oriented features. As in Axiom, the major modular mechanisms are domains[3] and categories. Categories are types of domains. The other two novel features of Aldor are: (1) both types and functions are first class values which can be constructed dynamically and manipulated in the same way as any other values and (2) dependent types. Since both categories and domains as well as types and functions are first class values, higher-order functors are supported naturally as normal functions over categories and function types over categories. Aldor has an elaborate type system to handle these features and others like unions and records uniformly. Investigating the whole type system would be a huge amount of work. Here we only discuss domains, categories, functions over domains, function types over categories, and dependent types[4].

[3] Since the packages are special domains, we will not discuss packages here.

[4] We are only interested in dependent function types in which the result type may depend on the argument type. We will not talk about dependencies between fields of records.

As presented in [81], the general form of a domain is :

$$\texttt{add}\{\texttt{Rep} \texttt{==>} T; x_1 : T_1 == t_1; \ldots ; x_n : T_n == t_n\}$$

where, for $i = 1 \ldots n$, x_i is an exported symbol, T_i is the type of it (possibly a category), and t_i is a term of T_i. Rep is the representation type and T is the abstract type corresponding the current domain. The type of it is a category:

$$\texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}$$

There is a subtype relation over categories:

$$\frac{m \geq n \quad p_1, \ldots, p_m \text{ permutes } 1, \ldots, m}{\texttt{with}\{x_1 : T_1; \ldots ; x_1 : T_n\} \sqsubseteq \texttt{with}\{x_{p_1} : T_{p_1}; \ldots ; x_{p_m} : T_{p_m}\}}.$$

However, the above subtype is not applicable to named categories. Named categories follow Axiom's approach. A category B is a subtype of a named category A only if B extends A. In other words type expressions are not evaluated. This is an important *explicit* design decision in Aldor.

Since categories are types of domains as other normal types, we can define functions over categories as follows:

$$(x : T) : T\ \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}\ \texttt{+->} x\ \texttt{add}\{x_1 : T_1 == t_1; \ldots ; x_n : T_n == t_n\}$$

of type

$$T \rightarrow T\ \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}.$$

where T is a category. Thus a function can be applied to any domain of category T or any domain of a subtype of category T. The application is of type T $\texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}$. Clearly, higher-order functions over categories can be defined straightforwardly as normal functions. With dependent function types, we can do better:

$$\begin{aligned}&(T_{\texttt{x}} : \texttt{Category}) : T_{\texttt{x}} \rightarrow T_{\texttt{x}}\ \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}\ \texttt{+->}\\ &\quad (x : T_{\texttt{x}}) : T_{\texttt{x}}\ \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}\ \texttt{+->} x\ \texttt{add}\{x_1 : T_1 == t_1; \ldots ; x_n : T_n == t_n\}\end{aligned}$$

of type

$$(T_{\texttt{x}} : \texttt{Category}) \rightarrow (T_{\texttt{x}} \rightarrow T_{\texttt{x}}\ \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}).$$

Here $T_{\mathtt{x}}$: `Category` means that $T_{\mathtt{x}}$ is a variable of type `Category`, i.e. a category variable. It can then be replaced by any category T. Applying this function above to any category T and then to a domain of type T will give a domain of type T `with`$\{x_1 : T_1; \ldots ; x_n : T_n\}$ which will change according to the actual type T. By $T_{\mathtt{x}}$: `Category`, we are quantifying over all categories. However, the category variable is not bounded by a particular category, i.e. the subtype relation is not used to bound universal quantification as in DMei. We thus compare the modular mechanism of Aldor with that of DMei as follows:

(1) Aldor supports dependent type over categories like DMei, but type variables may not be bounded.

(2) Aldor supports subtyping over categories similar to the subtyping over theory types in DMei. Also the subtyping in Aldor is extended to cover function types over categories in the same way as in DMei (which is not presented above).

(3) Functions and types are first class values in Aldor. This implies more flexible function definitions and type definitions which are not fully presented above.

(4) Aldor does not have a notion of view as in DMei. However, we believe, it is possible to incorporate the idea of views with Aldor's type system.

5.4.5 Focal

Focal [64] is a language designed for the development of certified computer algebra libraries. The computation part of a Focal code is compiled to an OCaml file. The specification part of a Focal code is is compiled to a Coq file and then checked by Coq. The basic building block of Focal is the notion of *species* which supports both inheritance and parameterization.

A species consists of three parts: (1) A carrier represented by the `rep` keyword which can be either abstract or bound with a concrete representation. The abstract type can be referred to as `self` within a species. `self` and its concrete representation are equivalent inside the species. This resembles the treatment of carrier type and its representation used in Axiom and Aldor. (2) A computational part consisting of a set of methods that are either declared or defined. Methods of a species can access both the abstract type and the concrete representation of the carrier. (3) A specification part consisting of properties (declared) and theorems (proved).

Apart from species, Focal has two important notions of interfaces and collections. An interface is a purely abstract species in which all methods are abstract. A collection is a completely defined species in which all the methods are implemented. Roughly speaking interfaces and collections corresponds to theory types and theories in Mei. Species sit between interfaces and collections. An interface specifies the abstract view of species and collections, i.e. the abstract carrier type and the prototypes of exported methods. Each species or collection implicitly defines its corresponding interface. Species serve as types of collections in the parameterized species introduced later in this section. We do not have a counterpart of species in Mei. A theory type plays two roles in Mei, as a type and as an interface.

Looking from another angle, interfaces, species, and collections are similar to interfaces, abstract classes, and classes of object oriented languages. A species can then inherit from one or several species as in class inheritance. While a species receives all the declaration and definition of methods from its parent, it can redefine some of the inherited methods but must keep the types. In this case, the proofs of theorems that depend on the implementations of the method (so-called def-dependency) must be transformed to properties and need to be reproved. This significantly differs from the inheritance of Axiom's category and the theory extension operation of Mei.

Remark 5.4.1. Since overwriting is allowed in the inheritance, a child species might not be seen as a subtype as its parent species. The behaviour of the child species might be different from the behaviour of its parent species.

Species can also be parameterized by both values of particular type, called entities in Focal, and collections in some species. As in algebraic specification languages, species are both types and objects. In the parameter list, a species is a type representing a class of collections that implement the species. When a parameterized species is instantiated, the instantiated species is an object. As in Axiom and Aldor, the order of the parameter list is important since it is essentially a dependent record type. The form

```
species modulo ( r is int_ring, n in r )...
```

defines a species that has two parameters, a collection `t` that implements the species `int_ring` and an entity `n` in this collection. Here the type of `n` depends on the value of `r`.

Although we can see interfaces and collections as Mei's theory types and theories, the central notion of Focal is species. Only species can be parameterized, and essentially a parameterized species is a first-order functor in Mei.

5.4.6 Magma

Magma [9] is a computer algebra system based on universal algebra and category theory. Any object definable in Magma is an element of some algebraic structure. Knowledge is organized as *magmas, categories, and varieties* in Magma. A many-sorted algebra A (or the carrier set of A) is called a magma; a category is a class of magmas satisfying a particular set of axioms and sharing a common representation; and a collection of categories whose magmas satisfy a common set of axioms forms a variety [12]. Instead of providing a general modularity mechanism like Mei, Magma provides a set of predefined *constructors* that allowing users to construct new magmas in a particular way. These constructors are closely related to certain algebra construction notions such as subalgebra[5] etc. We will briefly describe how to construct a magma in Magma. We will use many algebra notions without formally introducing their definitions. Readers can refer to [9] or [65] for the definitions.

The creation of a magma M in the Magma system requires, in principle, two steps:

(1) The definition of an appropriate *free magma* F.

(2) The construction of the desired magma M from F by applying a sequence of constructions.

Given a variety V (e.g., groups, rings), the free magma F in V with n generators[6] is the unique algebraic structure whose elements are all possible finite algebraic combinations of the n generators, which corresponds to a term algebra [12].

`FreeGroup` is a predefined Magma function that constructs free magmas. Then

```
FG3<a,b,c> := FreeGroup(3);
```

creates a free magma, `FG3`, generated by three generators, `a`, `b`, and `c`. Now we can construct a subalgebra from `FG3` as follows:

```
G2<x,y>, i := sub<FG3 | a, a*b]>;
```

`sub` is a predefined subalgebra constructor. It takes a free magma and a set of generators and returns two values, `G2` and `i`. `G2` is a magma that is a subalgebra of the given free magma `FG3` generated by $\{a, a * b\}$ (which can be referred to as $\{x, y\}$

[5] Informally a subalgebra of an algebra A is a subset of A that is closed under the algebra operations of A.

[6] Informally, $X \subseteq A$ is called the set of generators of an algebra A if the closure of X under the algebraic operations of A equals A. Note that Magma talks about *finitely generated* algebras.

now). `i` is a morphism from `G2` to `FG3`. Since the representation of `G2` is irrelevant to that of `FG3`, the morphism `i` is necessary to connect an element of `G2` to the same element in `FG3` with possibly different representations.

Remarks 5.4.2. (1) Magma provides five magma constructor, **sub**, **ncl**, **ideal**, **quo**, and **ext**, designating submagmas, normal closures (in the case of groups), ideals, quotients and extensions [12].

(2) Magma provides five built-in magmas, **IntegerRing**, **RationalField**, **RealField**, **Strings**, and **Booleans**.

(3) Magma provides shorthand constructors that allow users to construct magmas without explicitly constructing the free magma.

(4) It is not clear if users can define new categories representing, say, abstract data structures.

Magma's modularity mechanism follows a quite different approach form Mei. Roughly speaking we can view magmas as theories, categories and varieties as theory types. The magma constructors are then first-order functors. However, Magma provides a fixed number of such functors. The users can employ these functors but cannot add new functors. In other words, Magma provides a library of functors but hides from users the details of the module system that constructs functors. It is thus the implementors' (not the users') responsibility to provide a powerful library. Although we do not think it is in general a good idea to hide the module system from users, we believe Magma's five magma constructors are useful for many MMSs. It is thus an interesting research topic to implement these magma constructors on top of a MMS together with a module system, either Mei or DMei.

5.5 An expressive language of signatures

Dialects of ML usually have a rich module system. An ML-family module system itself is a small typed functional language, where structures and functors are objects and signatures are types. While modules can be manipulated in various ways, types (signatures) are described by enumerating their parts. (This is also true for Mei.) N. Ramsey, K. Fisher, and P. Govereau present a language that manipulates signatures, e.g. two signatures can be combined to form a new signature [83]. It is important to note that the operations over signatures are very different from DMei's module types such as extension and union. The language of signatures exposes no information about how a structure matches a signature. For instance, a signature that combines two signatures is the same as the *flat* signature that is the union of the two component signatures. A structure that implements the combination signature also implements the flat signature. However, in DMei a module expression of a union type must be built from two subexpressions of corresponding component types. Looking from another view, the language of signature can be seen as Mei on the module type level in the sense that the operations are over module types instead of module expressions. We briefly summarize the language of signatures as follows. Interested readers can refer to [83] for details.

(1) Adding, removing, rebinding, and moving components. `adding` which adds new components to a signature is similar to Mei's extension operation over theories. There is no counterpart of `removing` which removes components of a signature in Mei. Both `rebinding` and `moving` can be seen as a removing followed by an adding where `rebinding` is "in place".

(2) Revealing and sealing types and modules. `revealing` makes an abstract type manifest, i.e. binding with a concrete representation type. This can be seen in Mei's terminologies as extending a theory by the concrete representation type and an axiom specifying the equivalency of them. `sealing` makes a manifest type abstract. It is much more complicated than removing the concrete representation type and corresponding axioms. In fact, the concrete representation type might not even be hidden. Only the equivalence axiom is removed. There is no counterpart of `sealing` in Mei.

(3) Combining signatures. Roughly speaking, `andalso` which combines two signatures is similar to the union operation of Mei. It is more complicated because of the need to deal with the abstract and manifest types.

The motivation of the language of signatures is different from that of Mei. We might say that Mei is an expressive language of theories. Interestingly they share a number of mechanisms. It might be a good idea to build a language of theory types for Mei, but not for DMei (since theory types of DMei is already quite complicated).

Chapter 6

Possible extensions of Mei

In Chapter 5, we investigate some existing module systems. We show that many mechanisms supported by these systems can be simulated in Mei. However, it will be quite heavy work if users are forced to do the simulation by themselves. A better approach would be to define a set of new operations in terms of the existing operations. In this section, we will discuss some of these extensions.

6.1 Theory definition

In §2.2, we assumed that a theory definition expression is explicitly annotated by its intended type. If the user's intention is to expose all facts of a theory, it will be tedious to repeat them in this type. In that case, we can allow the user to specify a theory definition without explicit type casting. The default type casting information, i.e. the language of the theory definition together with the axioms and sentences of the theorems of the theory definition, is then added by the system automatically.

```
EXPR ::= ...
       |  THY-SPEC
```

The semantics of this new module expression is defined in terms of module expressions of Mei. The semantics of the other module expressions can then be defined recursively. Let use $[\![\,]\!]$ for the semantics functions as usual. Then

$$[\![(L, \Phi, \Delta)]\!] \quad = \quad (L, \Phi \cup \mathbf{sen}(\Delta))\ (L, \Phi, \Delta)$$

Note that the user has the freedom to choose the part of theory they want to expose by casting it to the intended type. By exposing all axioms and theorems, the theory may be used in a place where a theory axiomatized in a different way is required, as long as the type of the latter theory is a subtype of the type of the former theory, i.e. the axioms in the latter theory are theorems in the former.

6.2 Hiding and revealing

Theory hiding and revealing are two useful operations. The user may be able to hide (or reveal) a list of symbols defined in a theory definition. These are similar to the hiding and revealing operations in Casl [72].

```
EXPR ::= ...
       |  EXPR hide LANG
       |  EXPR reveal LANG
```

Again, we define the semantics of the above syntactic sugaring in terms of module expressions of Mei.

$$[\![(L,\Phi,\Delta)\ \texttt{hide}\ L_{\mathsf{h}}]\!] = (L_{\bar{\mathsf{h}}},\Phi_{\bar{\mathsf{h}}})\ (L,\Phi,\Delta) \tag{6.1}$$

$$[\![\mathsf{E}\ \texttt{hide}\ L_{\mathsf{h}}]\!] = (L_{\bar{\mathsf{h}}},\Phi_{\bar{\mathsf{h}}})\ \mathsf{E} \tag{6.2}$$

$$[\![(L,\Phi,\Delta)\ \texttt{reveal}\ L_{\mathsf{r}}]\!] = (L_{\mathsf{rr}},\Phi_{\mathsf{r}})\ (L,\Phi,\Delta) \tag{6.3}$$

$$[\![\mathsf{E}\ \texttt{reveal}\ L_{\mathsf{r}}]\!] = (L_{\mathsf{rr}},\Phi_{\mathsf{r}})\ \mathsf{E} \tag{6.4}$$

In (6.1), the simplest case is $L_{\bar{\mathsf{h}}} = L\backslash L_{\mathsf{h}}$. Then $\Phi_{\bar{\mathsf{h}}} = \{\varphi \in \Phi\cup\mathbf{sen}(\Delta) \mid \mathbf{lang}(\varphi) \subseteq L_{\bar{\mathsf{h}}}\}$, where $\mathbf{lang}(\varphi)$ refers to the set of symbols occur in φ. Intuitively, $\Phi_{\bar{\mathsf{h}}}$ is the largest subset of Φ whose members do not contain symbols listed in L_{h}. (6.2) is similar to (6.1) except that it works over expressions of a theory type instead of over expressions of theory definitions. Assume that the type of E is (L,Φ). $L_{\bar{\mathsf{h}}}$ is defined exactly as in (6.1). Then $\Phi_{\bar{\mathsf{h}}} = \{\varphi \in \Phi \mid \mathbf{lang}(\varphi) \subseteq L_{\bar{\mathsf{h}}}\}$.

Remark 6.2.1. In case L is many-sorted (typed), the definition of $L_{\bar{\mathsf{h}}}$ is more complicated. Intuitively, if we hide a sort symbol, all operator symbols concerning this sort symbol should be hidden. Assume $L = L^{\mathsf{s}} \cup L^{\mathsf{o}}$ and $L_{\mathsf{h}} = L^{\mathsf{s}}_{\mathsf{h}} \cup L^{\mathsf{o}}_{\mathsf{h}}$, where L^{s} and $L^{\mathsf{s}}_{\mathsf{h}}$ are sort symbols, and L^{o} and $L^{\mathsf{o}}_{\mathsf{h}}$ are operator symbols. Then $L_{\bar{\mathsf{h}}} = L^{\mathsf{s}}_{\bar{\mathsf{h}}} \cup L^{\mathsf{o}}_{\bar{\mathsf{h}}}$, where $L^{\mathsf{s}}_{\bar{\mathsf{h}}} = L^{\mathsf{s}}\backslash L^{\mathsf{s}}_{\mathsf{h}}$ and $L^{\mathsf{o}}_{\bar{\mathsf{h}}} = \{\mathsf{o} \in L^{\mathsf{o}} \mid \mathsf{o} \notin L^{\mathsf{o}}_{\mathsf{h}} \wedge \mathsf{sort}\,(\mathsf{o}) \subseteq L^{\mathsf{s}}_{\bar{\mathsf{h}}}\}$. $\mathsf{sort}\,(\mathsf{o})$ refers to the set of sort symbols occurs in o.

In (6.3), $L_{\overline{\mathsf{rr}}} = L_{\mathsf{r}}$. Then $\Phi_{\mathsf{r}} = \{\varphi \in \Phi \cup \mathbf{sen}(\Delta) \mid \mathbf{lang}(\varphi) \cap L_{\mathsf{rr}} \neq \emptyset\}$. Intuitively, Φ_{r} is the smallest subset of Φ whose members contain at least one symbol listed in L_{r}. (6.4) is similar to (6.3) except that it works over expressions of a theory type instead of theory definitions. Assume that the type of E is (L, Φ). $L_{\bar{\mathsf{r}}}$ is defined exactly as in (6.3). Then $\Phi_{\mathsf{r}} = \{\varphi \in \Phi \mid \mathbf{lang}(\varphi) \cap L_{\mathsf{rr}} \neq \emptyset\}$.

Remark 6.2.2. In case L is many-sorted (typed), if we reveal an operator symbol, all of its sort symbols should be revealed. Assume $L = L^{\mathsf{s}} \cup L^{\mathsf{o}}$ and $L_{\mathsf{r}} = L^{\mathsf{s}}_{\mathsf{r}} \cup L^{\mathsf{o}}_{\mathsf{r}}$. Then $L_{\mathsf{rr}} = L^{\mathsf{s}}_{\mathsf{rr}} \cup L^{\mathsf{o}}_{\mathsf{r}}$, where $L^{\mathsf{s}}_{\mathsf{rr}} = \{\mathsf{s} \in L^{\mathsf{s}} \mid \mathsf{s} \in L^{\mathsf{s}}_{\mathsf{r}} \vee \exists \mathsf{o}.\ \mathsf{s} \in \mathsf{sort}\,(\mathsf{o})\}$.

Remarks 6.2.3.

(1) It is very important to note that hiding a symbol is not the same as eliminating it. It just prevents references to it from outside the theory. In other words, the original theory should be a conservative extension of the revealed theory, i.e. every sentence involving only the revealed symbols which is true in the original theory, is also true in the revealed theory. For instance, we might hide the inverse operator and the axioms defining it in the theory *Group*. However, in the revealed theory, the theorem showing the existence of the inverse of every group element is still valid. In this case, the original theory is just a definitional extension of the result theory.

(2) It is also important to note that we cannot hide a particular axiom without hiding any symbol. There is no point to just hiding an axiom, since hiding an axiom does not cause its logical consequences (including the axiom itself) to be hidden. Even worse, it may make the theory look like a different theory. For instance, let *CommGroup* be a commutative group theory. Hiding the commutative axiom does not invalidate the theorems proved by using the commutative axiom. Thus, although the revealed theory looks like a group theory like *Group*, it will still represents a commutative group theory.

We also note that, in order to define the semantics of a new module expression, it is necessary to know the type of it subexpressions, which represents the original expression. It is then necessary to lift the typing rules of Mei to account for these new module expressions as follows:

$$\frac{\mathbf{closed}(L, \Phi, \Delta) \quad L_{\mathsf{h}} \subseteq L}{\Gamma \vdash (L, \Phi, \Delta)\ \texttt{hide}\ L_{\mathsf{h}} :\ (L_{\bar{\mathsf{h}}}, \Phi_{\bar{\mathsf{h}}})} \qquad \textbf{(HIDE1)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad L_{\mathsf{h}} \subseteq L}{\Gamma \vdash \mathsf{E}\ \texttt{hide}\ L_{\mathsf{h}} :\ (L_{\bar{\mathsf{h}}}, \Phi_{\bar{\mathsf{h}}})} \qquad \textbf{(HIDE2)}$$

$$\frac{\textbf{closed}(L, \Phi, \Delta) \quad L_{\mathsf{r}} \subseteq L}{\Gamma \vdash (L, \Phi, \Delta)\ \texttt{reveal}\ L_{\mathsf{r}} :\ (L_{\mathsf{rr}}, \Phi_{\mathsf{r}})} \qquad \textbf{(REVEAL1)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad L_{\mathsf{r}} \subseteq L}{\Gamma \vdash \mathsf{E}\ \texttt{reveal}\ L_{\mathsf{r}} :\ (L_{\mathsf{rr}}, \Phi_{\mathsf{r}})} \qquad \textbf{(REVEAL2)}$$

6.3 Local theories

Given the hiding operation, we can define a notion of a *local* theory, i.e. a theory only visible within another theory. The symbols declared in a local theory are only visible within the local theory and not within its surrounding theory. The axioms and theorems of the local theory are visible inside its surrounding theory but not outside its surrounding theory. Since local theories usually sit in a theory specification, we need a new kind of theory specification as follows:

$$\begin{array}{lll} \texttt{THY-SPEC} & ::= & \ldots \\ & \mid & \texttt{local}\ \texttt{EXPR}\ \texttt{within}\ (\texttt{LANG}, \texttt{AXIOMS}, \texttt{THMS}) \end{array}$$

Theory specification occurs only in theory definition and theory extension. We define the semantics of these two cases in terms of module expressions of Mei:

$$\begin{array}{rcl} [\![\texttt{local}\ \mathsf{E}\ \texttt{within}\ \mathsf{S}]\!] & = & (\mathsf{E}\ \texttt{extended by}\ \mathsf{S})\ \texttt{hide}\ L_{\mathsf{E}} \\ [\![\mathsf{E}_1\ \texttt{extended by}\ (\texttt{local}\ \mathsf{E}_2\ \texttt{within}\ \mathsf{S})]\!] & = & ((\mathsf{E}_1 \oplus \mathsf{E}_2)\ \texttt{extended by}\ \mathsf{S})\ \texttt{hide}\ L_{\mathsf{E}_2} \end{array}$$

where L_{E} $[L_{\mathsf{E}_2}]$ is the language of the type of E $[\mathsf{E}_2]$ and $\mathsf{S} \equiv (L, \Phi, \Delta)$.

As for hiding and revealing operations, in order to define the semantics of a local theory expression, it is necessary to know the type of it subexpressions. It is then necessary to lift the typing rules of Mei to account for the local theory expressions as follows:

$$\frac{\Gamma \vdash (\mathsf{E}\ \mathtt{extended\ by}\ \mathsf{S})\ \mathtt{hide}\ L_{\mathsf{E}} :\ (L, \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathtt{local}\ \mathsf{E}\ \mathtt{within}\ \mathsf{S} :\ (L, \Phi \cup \mathbf{sen}(\Delta))} \qquad \textbf{(LOCAL1)}$$

$$\frac{\Gamma \vdash ((\mathsf{E}_1 \oplus \mathsf{E}_2)\ \mathtt{extended\ by}\ \mathsf{S})\ \mathtt{hide}\ L_{\mathsf{E}_2} :\ (L_{\mathsf{E}_1} \cup L, \Phi_{\mathsf{E}_1} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}_1\ \mathtt{extended\ by}\ (\mathtt{local}\ \mathsf{E}_2\ \mathtt{within}\ \mathsf{S}) :\ (L_{\mathsf{E}_1} \cup L, \Phi_{\mathsf{E}_1} \cup \Phi \cup \mathbf{sen}(\Delta))} \ \textbf{(LOCAL2)}$$

where $\mathsf{S} \equiv (L, \Phi, \Delta)$, $\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}})$, $\Gamma \vdash \mathsf{E}_1 : (L_{\mathsf{E}_1}, \Phi_{\mathsf{E}_1})$, and $\Gamma \vdash \mathsf{E}_2 : (L_{\mathsf{E}_2}, \Phi_{\mathsf{E}_2})$.

Remark 6.3.1. One side condition for **(LOCAL2)** is that L_{E_1} and L_{E_2} must be disjoint, because it makes no sense to allow a local theory L_{E_2} to interfere with an outside theory L_{E_1}. A practical solution is to systematically add a unique tag to all symbols in L_{E_2}. This will not affect other parts of a module expression in which this local theory is a submodule expression, because all symbols in L_{E_2} are immediately hidden.

6.4 Functor composition

A sequence of function applications is called a function composition. Since functors are simply functions over theories, we can define a sequence of functor applications as a functor composition following the definition of function composition:

```
EXPR  ::=  ...
        |  EXPR ∘ EXPR
```

Again, we define the semantics of functor composition in terms of the module expressions of Mei. Let $\mathsf{E_f} :\ \mathsf{T_{f_1}} \rightarrow \mathsf{T_{f_2}}$ and $\mathsf{E_g} :\ \mathsf{T_{g_1}} \rightarrow \mathsf{T_{g_2}}$.

$$[\![\mathsf{E_f} \circ \mathsf{E_g}]\!] = \mathtt{functor}\ \mathsf{X} : \mathsf{T_{g_1}}.\ \mathsf{E_f}\ (\mathsf{E_g}\ \mathsf{X})$$

Note that a functor composition requires that the result type of $\mathsf{E_g}$ matches the parameter type of $\mathsf{E_f}$ by the subtype relation, i.e. $\mathsf{T_{g_2}} <: \mathsf{T_{f_1}}$.

Since the notion of view is a generalization of the subtype relation, it is then natural to use views instead of the subtype relation to connect the result type of $\mathsf{E_g}$ and the parameter type of $\mathsf{E_f}$. This gives us a more general form of functor compositions.

```
EXPR  ::=  ...
        |  EXPR ∘ EXPR with view VIEW
```

The semantics of the functor composition with view is defined as follows. Let $\mathsf{E_f} : \mathsf{T_{f_1}} \rightarrow \mathsf{T_{f_2}}$ and $\mathsf{E_g} : \mathsf{T_{g_1}} \rightarrow \mathsf{T_{g_2}}$.

$$[\![\mathsf{E_f} \circ \mathsf{E_g}\ \texttt{with view}\ (\mathsf{T_{f_1}}, \mathsf{T_{g_2}}, \rho)]\!] = \texttt{functor}\ \mathsf{X} : \mathsf{T_{g_1}}.\ \mathsf{E_f}\ (\mathsf{E_g}\ \mathsf{X})\ \texttt{with view}\ (\mathsf{T_{f_1}}, \mathsf{T_{g_2}}, \rho)$$

This is exactly the same as the so-called parameterized module instantiation with the parameterized view in Maude [19]. Our notion clearly indicates that this is a variant of functor composition, not functor application. We believe that Maude's notion is somewhat misleading. In addition, by explicitly representing functor composition, there is no need to introduce the notion of parameterized view, since the parameter is presented as the parameter of the second functor in the functor composition.

Note that we need the type information of $\mathsf{E_g}$ in order to define the semantics of a functor composition $\mathsf{E_f} \circ \mathsf{E_g}$. It is then necessary to lift the typing rules of Mei to account for functor compositions as follows:

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T_{f_1}} \rightarrow \mathsf{T_{f_2}} \quad \Gamma \vdash \mathsf{E_g} : \mathsf{T_{g_1}} \rightarrow \mathsf{T_{g_2}} \quad \mathsf{T_{g_2}} <: \mathsf{T_{f_1}}}{\Gamma \vdash \mathsf{E_f} \circ \mathsf{E_g} : \mathsf{T_{g1}} \rightarrow \mathsf{T_{f_2}}} \qquad \textbf{(FCOMP)}$$

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T_{f_1}} \rightarrow \mathsf{T_{f_2}} \quad \Gamma \vdash \mathsf{E_g} : \mathsf{T_{g_1}} \rightarrow \mathsf{T_{g_2}} \quad \mathbf{view}(\mathsf{T_{f_1}}, \mathsf{T_{g_2}}, \rho)}{\Gamma \vdash \mathsf{E_f} \circ \mathsf{E_g}\ \texttt{with view}\ (\mathsf{T_{f_1}}, \mathsf{T_{g_2}}, \rho) : \mathsf{T_{g_1}} \rightarrow \mathsf{T_{f_2}}} \qquad \textbf{(FCOMP-VIEW)}$$

6.5 View lifting, view union, and view composition

In §3.4.2, we give three rules to construct new views from existing views. It would be nice if we could provide some notation to express these view constructions as follows:

```
VIEW ::= ...
       |  lift VIEW by THY-SPEC
       |  VIEW ⊕ VIEW
       |  VIEW ∘ VIEW
```

They are called *view lifting*, *view union*, and *view composition* respectively. The semantics of these new view objects is defined in terms of the views in Mei:

$$
\begin{array}{l}
[\![\texttt{lift}\ ((L_{\mathsf{s}},\Phi_{\mathsf{s}}),(L_{\mathsf{t}},\Phi_{\mathsf{t}}),\rho)\ \texttt{by}\ (L_{\mathsf{E}},\Phi_{\mathsf{E}})]\!] \\
\quad = \quad ((L_{\mathsf{s}}\cup L_{\mathsf{E}},\Phi_{\mathsf{s}}\cup\Phi_{\mathsf{E}}),(L_{\mathsf{t}}\cup\rho'(L_{\mathsf{E}}),\Phi_{\mathsf{t}}\cup\rho'(\Phi_{\mathsf{E}})),\rho') \\
[\![\texttt{lift}\ \mathsf{V}\ \texttt{by}\ \mathsf{S}]\!] \\
\quad = \quad [\![\texttt{lift}\ [\![\mathsf{V}]\!]\ \texttt{by}\ \mathsf{S}]\!] \\
[\![((L_{\mathsf{s}_1},\Phi_{\mathsf{s}_1}),(L_{\mathsf{t}_1},\Phi_{\mathsf{t}_1}),\rho_1)\ \oplus\ ((L_{\mathsf{s}_2},\Phi_{\mathsf{s}_2}),(L_{\mathsf{t}_2},\Phi_{\mathsf{t}_2}),\rho_2)]\!] \\
\quad = \quad ((L_{\mathsf{s}_1}\cup L_{\mathsf{s}_2},\Phi_{\mathsf{s}_1}\cup\Phi_{\mathsf{s}_2}),(L_{\mathsf{t}_1}\cup L_{\mathsf{t}_2},\Phi_{\mathsf{t}_1}\cup\Phi_{\mathsf{t}_2}),\rho_1\cup\rho_2) \\
[\![\mathsf{V}_1\ \oplus\ \mathsf{V}_2]\!] \\
\quad = \quad [\![[\![\mathsf{V}_1]\!]\ \oplus [\![\mathsf{V}_2]\!]]\!] \\
[\![((L_1,\Phi_1),(L_2,\Phi_2),\rho_1)\ \circ\ ((L_2,\Phi_2),(L_3,\Phi_3),\rho_2)]\!] \\
\quad = \quad ((L_1,\Phi_1),(L_3,\Phi_3),\rho_1\circ\rho_2) \\
[\![\mathsf{V}_1\ \circ\ \mathsf{V}_2]\!] \\
\quad = \quad [\![[\![\mathsf{V}_1]\!]\ \circ\ [\![\mathsf{V}_2]\!]]\!]
\end{array}
$$

The semantics of these new view objects are justified directly by the three view construction rules in §3.4.2. It is possible to reformulate these rules as follows, so that the view checking can be done before the translation.

$$
\frac{\mathbf{view}((L_{\mathsf{s}},\Phi_{\mathsf{s}}),(L_{\mathsf{t}},\Phi_{\mathsf{t}}),\rho) \qquad \rho'=\mathsf{lift}(\rho,L_{\mathsf{E}})}{\mathbf{view}(\texttt{lift}\ ((L_{\mathsf{s}},\Phi_{\mathsf{s}}),(L_{\mathsf{t}},\Phi_{\mathsf{t}}),\rho)\ \texttt{by}\ (L_{\mathsf{E}},\Phi_{\mathsf{E}}))} \qquad \textbf{(EXT-VIEW)}
$$

$$
\frac{\mathbf{view}((L_{\mathsf{s}_1},\Phi_{\mathsf{s}_1}),(L_{\mathsf{t}_1},\Phi_{\mathsf{t}_1}),\rho_1) \quad \mathbf{view}((L_{\mathsf{s}_2},\Phi_{\mathsf{s}_2}),(L_{\mathsf{t}_2},\Phi_{\mathsf{t}_2}),\rho_2) \quad \mathbf{consist}(\rho_1,\rho_2)}{\mathbf{view}(((L_{\mathsf{s}_1},\Phi_{\mathsf{s}_1}),(L_{\mathsf{t}_1},\Phi_{\mathsf{t}_1}),\rho_1)\ \oplus\ ((L_{\mathsf{s}_2},\Phi_{\mathsf{s}_2}),(L_{\mathsf{t}_2},\Phi_{\mathsf{t}_2}),\rho_2))}
$$

$$\textbf{(UNI-VIEW)}$$

$$
\frac{\mathbf{view}((L_1,\Phi_1),(L_2,\Phi_2),\rho_1) \quad \mathbf{view}((L_2,\Phi_2),(L_3,\Phi_3),\rho_2)}{\mathbf{view}(((L_1,\Phi_1),(L_2,\Phi_2),\rho_1)\ \circ\ ((L_2,\Phi_2),(L_3,\Phi_3),\rho_2))} \qquad \textbf{(COMP-VIEW)}
$$

As discussed in §5.2.1, the instantiation of a parameterized view can be easily derived from view lifting and view composition. Also our notion is closer to standard practice, in which an operation of this kind would be understood as a view composition rather than as a view instantiation.

Chapter 7

A structural implementation of Mei

In this section, we present an implementation of Mei. The implementation is written in OCaml, extensively using the modular mechanisms of OCaml. The implementation is parameterized by the interfaces of an underlying language and a notion of theory interpretation. The separation between Mei and an underlying MMS shows the flexibility of Mei. The implementation is both a test of Mei and an argument in favour of Mei, since the modular mechanisms used in the implementation are also supported in Mei with some minor differences. The code of the implementation can be found in the appendix of [99].

7.1 Structure of the implementation

Figure 7.1 shows the structure of our implementation. The modules connected by solid arrow lines are the implementation of Mei. A solid arrow line shows that the target OCaml module is built on top of the source module. Another module, `Fol` (stands for first-order logic), is an OCaml structure satisfying the signature `MMS_SYN`. They are used together, shown by the dotted arrow lines, to build the top-level testing environment consisting of `Fol_MeiCore`, `Fol_MeiCore_Eval`, and `Fol_Mei`.

`MMS_SYN` is a signature representing an MMS. `MEICORE` is a signature represent-

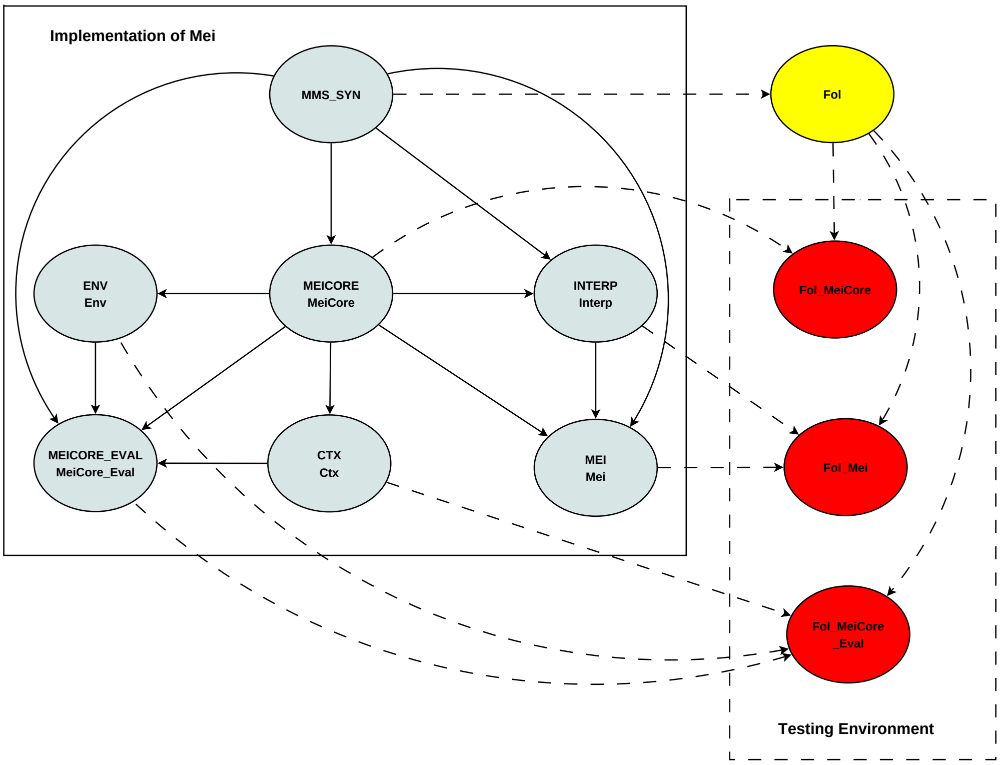

Figure 7.1: The structure of the implementation

ing Mei Core, which is built on top of MMS_SYN. MeiCore is a functor whose formal parameter is MMS_SYN which implements MEICORE. ENV and CTX represent the environment and context of Mei Core, whose implementations are functors taking MEICORE as a formal parameter. MEICORE_EVAL represents the type checker and evaluation function, which is built on top of MMS_SYN, MEICORE, ENV, CTX. MeiCore_Eval is a functor implementing MEICORE_EVAL. Similarly, Mei implements MEI, which is built on top of MEICORE and represents Mei. The whole implementation can be seen as a big "functor" parameterized by the signature MMS_SYN. Applying this big "functor" to an OCaml module implementing a particular logic, say Fol, we get Mei over a first-order logic. We can then do testing within the derived OCaml modules. The whole structure follows Leroy's approach in [58].

7.2 Identifiers

The idea of using identifiers instead of names as the reference of theories and functors is borrowed from [58]. The purpose of using identifiers is to implement the static binding between names and module expressions. A name used in a module expression always refers to the object bound to it at the time when the module expression is defined, not the one bound to it when we evaluate the module expression. In other words, redefining a name will not affect the meanings of those module expressions which are defined before the redefining of the name.

An identifier is a pair of a name and a stamp, where the stamp uniquely identifies the module expression it refers to. The module names used in Mei are replaced by identifiers in the abstract syntax trees representing the module expressions of Mei. This can be during the parsing or as a separate pass before type checking. For example, assume that a theory T is bound with a name N when we define a module expression E. An identifier, say *id*, is generated and bound to T. In the representation of E, N is replaced by *id* which is bound to T. Now redefine N such that it binds with another theory, say T'. Then a new identifier, say id', is generated and bound to the theory T'. Clearly, if we define a new module expression E′ using the name N, it will refer to T' since N will be replaced by id' which is bound to T'. However, if we evaluate E now, *id* is found in the representation of E. Hence T will be used in the evaluation. This is necessary as otherwise the redefinition of N might invalidate the type argument of E established before the redefinition of N.

Remarks 7.2.1.

(1) A theory definition (or a functor abstraction) and a module name are like a value and a variable in a functional language. They are statically bound as shown above. However, it is very natural to develop a theory in several stages at different times. It is thus important to distinguish these steps with the redefinition of a module name since it is desirable to reflect the changes in this case.

For example, taking Example 2.1.13, a theory of monoids is bound with a name `Monoid` which is referred to by a module expression representing a theory of rings. Later, we may explore the theory of monoids by proving new theorems. In this case, the newly proved theorems are included in the result of the evaluation of the module expression representing the theory of rings. This is not a redefinition of the name `Monoid` where the name *Monoid* is bound with a totally different theory (or even a module expression representing a functor). Thus we support dynamic binding for theory exploration and static binding for module name redefinition.

(2) By using identifiers instead of names as the internal representation, we do not need to worry about the "variable capture" problem since all identifiers are distinct by their stamps.

The syntax of identifiers is defined by the following signature:

```
{}
module type IDENT = sig
  type t
  val create: string -> t
  val name: t -> string
  val equal: t -> t -> bool
end
```

`create` returns a fresh identifier with the given name; `name` returns the name of the given identifier; `equal` checks the equality of two identifiers. The following is a simple implementation `Ident` of `IDENT`. The stamp of an identifier is an integer. The stamp is increased by one at each `create` operation, giving a fresh identifier. The equality of identifiers depends on the equality of stamps.

```
module Ident:IDENT = struct
  type t = {name: string; stamp: int}
  let currstamp = ref 0
  let create s = currstamp := !currstamp + 1;
            {name = s; stamp = !currstamp}
  let name id = id.name
  let equal id1 id2 = (id1.stamp = id2.stamp)
end
```

7.3 Abstract syntax for MMS

The abstract syntax tree representing the underlying MMS system should implement the following signature:

```
module type MMS_SYN = sig
  type language
  type mapping
  type sentence
  type proof
  val is_language: language -> bool
  val is_sentence: language -> sentence -> bool
  val is_mapping_of: language -> mapping -> bool
  val is_proof: language -> sentence list -> sentence*proof -> bool
  val empty_lang: language
  val merge_lang: language -> language -> language
  val is_sublang: language -> language -> bool
  val lift: language -> mapping -> mapping
  val inv: mapping -> mapping
  val find_ren_ext: language*language -> language*language -> mapping
  val find_ren_union: language*language -> language*language
              -> mapping*mapping
  val language_translate: language -> mapping -> language
  val sentence_translate: sentence -> mapping -> sentence
  val proof_translate: proof -> mapping -> proof
end
```

`language` represents symbols: sort symbols and operation symbols in equational logic; sort symbols, constant symbols, function symbols, and predicate symbols in first-order logic; or atomic types and constants in simple type theory. The type `sentence` represents sentences: equations in equational logic; closed formulas in first-order logic; or closed expressions of type boolean in simple type theory. The type `proof` represents derivations of a sentence from a given list of sentences. The type `mapping` represents symbol mappings between theories.

`is_language`, `is_sentence`, `is_mapping_of`, and `is_proof` are the justifications of well-formedness of the four syntactic classes respectively. Type checking of an MMS is encoded in these four functions. For instance, in simple type theory, `is_language` can justify the well-formedness of type expressions and bindings of a type expression with a constant and `is_sentence` is the type checker of sentences with respect to type expressions and type bindings of constants. Other functions are auxiliary functions that are employed in module type checking and module expression evaluation. For instance, `empty_lang` returns an empty language and `merge_lang` returns the union of two given languages.

Remark 7.3.1. In fact, the signature MMS_SYNTAX can even be used to represent the abstract syntax of programming languages. language represents the primitive types, sentence represents the type expressions, and proof represents the implementations. is_sentence is the justification of the well-formedness of the type expressions and is_proof is the type checker of the proof with respect to the corresponding type expression.

7.4 Abstract syntax for Mei Core

The following signature, MEICORE, represents the abstract syntax tree of the module expressions and module types of Mei. MEICORE is built on top of an MMS, represented by an OCaml structure Mms implementing the signature MMS_SYN.

```
module type MEICORE = sig
  module Mms:MMS_SYN
  type thy_type = {
     lang_type: Mms.language;
     axioms_type: (Ident.t * Mms.sentence) list}
  type mod_type =
     Base of thy_type
   | Arrow of mod_type * mod_type
  type spec = {
     lang_spec:Mms.language;
     axioms_spec:(Ident.t * Mms.sentence) list;
     thms_spec:(Ident.t * Mms.sentence * Mms.proof) list}
  type mod_expr =
     Ident of Ident.t
   | Theory of spec * thy_type
   | Cast of mod_expr * thy_type
   | Extension of mod_expr * spec
   | Rename of mod_expr * Mms.mapping
   | Union of mod_expr * mod_expr
   | Functor of Ident.t * mod_type * mod_expr
   | Apply of mod_expr * mod_expr
  val is_subtype: mod_type -> mod_type -> bool
  val is_thytype: thy_type -> bool
  val is_thy: spec -> bool
  val subst: mod_expr -> (Ident.t * mod_expr) -> mod_expr
  val is_thytype_of: spec -> thy_type -> bool
  val merge_thytype: thy_type -> thy_type -> thy_type
  val merge_spec: spec -> spec -> spec
  val amal_ext: spec * thy_type -> spec -> spec * thy_type
  val amal_union: (spec*thy_type)-> (spec*thy_type)-> (spec*thy_type)
  val thy_type_translate: thy_type -> Mms.mapping -> thy_type
  val spec_translate: spec -> Mms.mapping -> spec
end
```

mod_expr represents module expressions which evaluate to either theories or functors. Theories built from scratch are represented by type spec. mod_type denotes either theory types or functor types. Theory types are represented by thy_type.

Functions declared in MEICORE are auxiliary functions for module type checking and module expression evaluation. For instance, is_subtype implements the subtyping rules, is_thytype justifies the closedness of theory types, and is_thy justifies the closedness of theory expressions etc. The most important function is subst, which implements the substitution function defined in 2.4.1.

The implementation of MEICORE is a functor MeiCore that takes any module satisfying the signature MMS_SYNTX and returns a module satisfying the signature MEICORE.

```
module MeiCore(TheMms: MMS_SYN) =
struct
  module Mms = TheMms
  (*type definition is the same as in MEICORE*)
  (*most functions are omitted*)
  let rec subst me (id, me_sub) =
     match me with
     Ident id’ -> if Ident.equal id id’ then me_sub else me
    | Theory (thy,tp) -> me
    | Cast(me’, tp) ->
         let me’’ = subst me’ (id, me_sub) in Cast(me’’,tp)
    | Extension(me’, spec) ->
         let me’’ = subst me’ (id, me_sub) in Extension(me’’, spec)
```

```
   | Rename(me', rho) ->
        let me'' = subst me' (id, me_sub) in Rename(me'',rho)
   | Union(me1,me2) ->
        let me1' = subst me1 (id, me_sub) in
        let me2' = subst me2 (id, me_sub) in
        Union(me1',me2')
   | Functor(id', tp, me') ->
        if Ident.equal id id' then me
        else let me'' = subst me' (id, me_sub) in
        Functor(id', tp, me'')
   | Apply(me_f, me_p) ->
        let me_f' = subst me_f (id, me_sub) in
        let me_p' = subst me_p (id, me_sub) in
        Apply(me_f',me_p')
end
```

7.5 Environment and context

`ENV` is a data structure recording the current bindings of identifiers with module expressions or module types, represented by `mod_expr` and `mod_type`.

```
module type ENV =
sig
  type mod_expr
  type mod_type
  type content
  type t = (Ident.t * content) list
  val empty: t
  val add_expr: Ident.t -> mod_expr -> t -> t
  val find_expr: Ident.t -> t -> mod_expr
  val mem_expr: Ident.t -> t -> bool
  val add_type: Ident.t -> mod_type -> t -> t
  val find_type: Ident.t -> t -> mod_type
  val mem_type: Ident.t -> t -> bool
end
```

The environment is applicative in the sense that the two add functions leave the original environment unchanged and returns a new environment with additional new entries. Given an environment, identifiers are the abbreviations of those defined module expressions and module types.

The implementation of ENV is a functor Env parameterized by a structure MEICORE.

```
module Env (TheMod:MEICORE) =
struct
  type mod_expr = TheMod.mod_expr
  type mod_type = TheMod.mod_type
  type content = Mod of mod_expr | Type of mod_type
  type t = (Ident.t * content) list
  (*function definitions omitted*)
end
```

The abstract types mod_expr and mod_type are bound to the concrete types TheMod.mod_expr and TheMod.mod_type by the type equivalence constraints.

CTX is similar to ENV except that it is a set of bindings of identifiers with module types. It represents the module type binding derived from the functor definition, which is used for type checking. The definitions of CTX and Ctx are not presented here.

7.6 Type checking and evaluation of Mei Core

We are now ready to define the signature for the type checking and evaluation functions for Mei Core. It is built on top of four modules, Mms represents the underlying MMS, Mod represents module expressions of Mei Core, Env represents an environment, and Ctx represents a context.

```
module type MEICORE_EVAL = sig
  module Mms:MMS_SYN
  module Mod:MEICORE
  module Env:ENV with type mod_expr = Mod.mod_expr and
                      type mod_type = Mod.mod_type
  module Ctx:CTX with type mod_type = Mod.mod_type
  val type_of: Mod.mod_expr -> Ctx.t -> Env.t -> Mod.mod_type
  val eval: Mod.mod_expr -> Env.t -> Mod.mod_expr
end
```

type_of infers the module type of the given module expression under the given environment and context. eval type checks and evaluates the given module expressions under the given environment.

The implementation is an OCaml functor parameterized by an MMS, Mei Core, an environment, and a context. The last three are functors. Hence, MeiCore_Eval is a higher-order functor. Note that module Mms = TheMms constrains the equality between the submodule Mms in MEICORE and the argument module TheMms.

```
module MeiCore_Eval
  (TheMms:MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) -> MEICORE with
    module Mms = TheMms)
  (TheEnvFunc: functor (TheMod:MEICORE) -> ENV with
    type mod_expr =  TheMod.mod_expr and type mod_type = TheMod.mod_type)
  (TheCtxFunc: functor (TheMod:MEICORE) -> CTX with
    type mod_type = TheMod.mod_type)
= struct
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)
  module Env = TheEnvFunc(Mod)
  module Ctx = TheCtxFunc(Mod)
  (*auxiliary functions are omitted*)
  let rec type_of te ctx env =
     match te with
     Mod.Ident id -> if Ctx.mem id ctx then Ctx.find id ctx
        else (if Env.mem_expr id env
        then type_of (Env.find_expr id env) ctx env
        else raise (MEICORE_EVAL "Module identity not defined."))
   | Mod.Theory (thy, thytype) ->
        if ((Mod.is_thy thy) && (Mod.is_thytype thytype)
        && Mod.is_thytype_of thy thytype) then Mod.Base thytype
        else raise(MEICORE_EVAL "Theory definition not closed.")
   | Mod.Cast(te’, thytype) -> if Mod.is_thytype thytype then
```

```
        (match type_of te' ctx env with
          Mod.Base thytype' ->
            if (Mod.is_subtype (Mod.Base thytype') (Mod.Base thytype))
            then Mod.Base thytype
            else raise (MEICORE_EVAL "Invalid casting.")
        | _ -> raise (MEICORE_EVAL "Functors cannot be casted."))
        else raise (MEICORE_EVAL "Theory type not closed.")
   | Mod.Extension(te',spec) -> (match type_of te' ctx env with
        Mod.Base thytype ->
          let thy_of_thytype = {Mod.lang_spec = thytype.Mod.lang_type;
             Mod.axioms_spec = thytype.Mod.axioms_type;
             Mod.thms_spec = []} in
          if (Mod.is_thy (Mod.merge_spec thy_of_thytype spec))
          then let get_axiom (id, phi, pf) = (id, phi) in
             let thytype' = {Mod.lang_type = spec.Mod.lang_spec;
               Mod.axioms_type = union spec.Mod.axioms_spec
                 (List.map get_axiom spec.Mod.thms_spec)} in
             Mod.Base (Mod.merge_thytype thytype thytype')
          else raise (MEICORE_EVAL "Extended theory not closed")
        |_ -> raise (MEICORE_EVAL "Functor cannot be extended."))
   | Mod.Rename (te', rho) -> (match type_of te' ctx env with
         Mod.Base thytype ->
            if Mms.is_mapping_of thytype.Mod.lang_type rho
            then Mod.Base (Mod.thy_type_translate thytype rho)
            else raise (MEICORE_EVAL "Bad renaming.")
        |_ -> raise (MEICORE_EVAL "Functors cannot be renamed."))
   | Mod.Union(te1,te2) ->
        (match (type_of te1 ctx env, type_of te2 ctx env) with
        (Mod.Base thytype1, Mod.Base thytype2) ->
            Mod.Base (Mod.merge_thytype thytype1 thytype2)
        | _ -> raise (MEICORE_EVAL "Functors cannot be merged."))
   | Mod.Functor(id, tp, te') -> let ctx' = Ctx.add id tp ctx in
        Mod.Arrow (tp, type_of te' ctx' env)
   | Mod.Apply(te_f, te_p) -> (match type_of te_f ctx env with
          Mod.Arrow(tp_1,tp_2) ->
            if Mod.is_subtype (type_of te_p ctx env) tp_1
          then tp_2
          else raise (MEICORE_EVAL "Parameter type mismatch.")
      | _ -> raise (MEICORE_EVAL "Functor required."))

let rec eval te env =
   match te with
   Mod.Ident id -> if Env.mem_expr id env
      then let te' = Env.find_expr id env in eval te' env
      else Mod.Ident id
 | Mod.Theory (thy,tp) -> te
 | Mod.Cast(te', tp) -> let te_eval = eval te' env in
      (match  te_eval with
        Mod.Theory (spec,typtype) -> Mod.Theory (spec,tp)
      | _ -> Mod.Cast(te_eval, tp))
 | Mod.Extension(te',spec) -> let te_eval = eval te' env in
      (match te_eval with
        Mod.Theory (spec_eval, tp) -> let (new_spec, new_tp) =
          Mod.amal_ext (spec_eval,tp) spec
          in Mod.Theory (new_spec, new_tp)
      | _ -> Mod.Extension(te_eval, spec))
 | Mod.Rename(te', rho) -> let te_eval = eval te' env in
      (match te_eval with
        Mod.Theory (spec, thytype) -> Mod.Theory
          (Mod.spec_translate spec rho,
              Mod.thy_type_translate thytype rho)
      | _ -> Mod.Rename(te_eval, rho))
 | Mod.Union(te1,te2) -> let te1_eval = eval te1 env in
      let te2_eval = eval te2 env in
      (match (te1_eval, te2_eval) with
        (Mod.Theory (spec1,tp1), Mod.Theory (spec2,tp2)) ->
          let (new_spec, new_tp) =
              Mod.amal_union (spec1,tp1) (spec2,tp2)
          in Mod.Theory (new_spec, new_tp)
      | (_, _) -> Mod.Union(te1_eval, te2_eval))
 | Mod.Functor(id, tp, te') -> (match te' with
       Mod.Apply(te'', Mod.Ident id') -> if Ident.equal id id'
```

```
         then eval te'' env else Mod.Functor(id, tp, te')
       | _ -> Mod.Functor(id, tp, te'))
   | Mod.Apply(te_f, te_p) -> (match eval te_f env with
         Mod.Functor(id, tp, te') ->
            eval (Mod.subst te' (id, te_p)) env
       | _ -> raise (MEICORE_EVAL "Funtor required."))
end
```

`type_of` is defined over the abstract syntax tree of module expressions and implements the typing rules defined in §3.2. For instance, a functor is well-typed if the body of the functor is well-typed with respect to the context enriched by the binding of the argument identifier and the argument type. `eval` is defined over the abstract syntax tree of module expressions and implements the evaluation rules defined in §2.4.2. For instance, a functor application is evaluated based on the substitution function defined in `MEICORE`.

7.7 Theory translation

An implementation of following signature represents the abstract syntax tree of theory translations. `TRAN` is built on top of a module of `MMS_SYN` and that of `MEICORE`, since it needs information from both the MMS and the module system.

```
module type TRAN = sig
  module Mms: MMS_SYN
  module Mod: MEICORE
  val thm_from_obl: Mod.thy_type -> Mod.thy_type ->
    Mms.mapping -> (Ident.t * Mms.sentence * Mms.proof) list
end
```

`thm_from_obl` takes a theory translation (the source type, the target type, and the mapping) and generates a list of obligations. As indicated in Remark 3.4.6, although our intention is that the implementation of `TRAN` should be an interpretation, it is not forced by our implementation. A proper implementation of `TRAN` has to employ the MMS and its proof system to check if a translation is an interpretation. This is not part of our work.

The following is a toy functor `Tran` only for testing:

```
module Tran
  (TheMms:MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) -> MEICORE with
      module Mms = TheMms) = struct
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)
  let thm_from_obl thytype1 thytype rho = []
end
```

7.8 Abstract syntax for Mei

A module representing the abstract syntax tree of module expressions in Mei should satisfy the following signature. It is built on top of an MMS, Mei Core, and a theory interpretation. The last one is used to build a representation of views from which coercion functors are constructed.

```
module type MEI =
sig
  module Mms: MMS_SYN
  module Mod: MEICORE
  module Tran: TRAN
  type mappings = Single of Mms.mapping | Pair of mappings * mappings
  type view = Mod.mod_type * Mod.mod_type * mappings
  type mod_expr = Ident of Ident.t
   | Theory of Mod.spec * Mod.thy_type
   | Cast of mod_expr * Mod.thy_type
   | Extension of mod_expr * Mod.spec
   | Rename of mod_expr * Mms.mapping
   | Union of mod_expr * mod_expr
   | Functor of Ident.t * Mod.mod_type * mod_expr
   | Apply of mod_expr * mod_expr
   | V_Apply of mod_expr * mod_expr * view
  val coerce: mod_expr -> Mod.mod_expr
end
```

`view` represents view as defined in §3.4.2. `mod_expr` represents module expressions of Mei. In particular, a module expression can be a functor application with a view.

`coerce` implements the semantics of Mei defined in §3.6, i.e. translates module expressions of Mei into module expressions of Mei Core.

The functor `Mei` is defined straightforwardly.

```
module Mei
  (TheMms: MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) -> MEICORE
        with module Mms = TheMms)
  (TheTranFunc: functor (TheMms:MMS_SYN)
    -> functor (TheModFunc: functor (TheMms:MMS_SYN)
        -> MEICORE with module Mms = TheMms)
    -> TRAN with module Mms = TheMms and
        module Mod = TheModFunc(TheMms))
= struct
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)
  module Tran = TheTranFunc(TheMms)(TheModFunc)
  (*type definition is the same as in MEI*)
  let rec functor_of_view v = match v with
      (Mod.Base thytype1, Mod.Base thytype2, Single rho) ->
        let rho' = Mod.Mms.lift thytype2.Mod.lang_type
                (Mod.Mms.inv rho) in
        let del = Tran.thm_from_obl thytype1 thytype2 rho in
        let idX = Ident.create("X") in
        let spec = {Mod.lang_spec = Mms.empty_lang;
                Mod.axioms_spec = []; Mod.thms_spec = del} in
        let me1 = Mod.Extension(Mod.Ident idX, spec) in
        let me2 = Mod.Rename(me1, rho') in
        let me3 = Mod.Cast(me2, thytype1) in
        Mod.Functor (idX, Mod.Base thytype2, me3)
    | (Mod.Arrow (type1, type2), Mod.Arrow (type1', type2'),
      Pair (rhos1, rhos2)) ->
        let c1 = functor_of_view (type1', type1,rhos1) in
        let c2 = functor_of_view (type2, type2',rhos2) in
        let idX = Ident.create("X") in
        let idF = Ident.create("F") in
        let me1 = Mod.Apply (c1, Mod.Ident idX) in
        let me2 = Mod.Apply (Mod.Ident idF, me1) in
        let me3 = Mod.Apply (c2, me2) in
        let me4 = Mod.Functor (idX, type1, me3) in
        Mod.Functor (idF, Mod.Arrow (type1', type2'), me4)
    | _ -> raise (MEI "Bad view")
  let rec coerce me = match me with
      Ident id -> Mod.Ident id
    | Theory (spec, tp) -> Mod.Theory (spec,tp)
    | Cast (me', btyp) -> Mod.Cast (coerce me', btyp)
    | Extension (me', spec) -> Mod.Extension (coerce me', spec)
    | Rename (me', rho) -> Mod.Rename (coerce me', rho)
    | Union (me1, me2) -> Mod.Union (coerce me1, coerce me2)
    | Functor (id, tp, me') -> Mod.Functor (id, tp, coerce me')
    | Apply (me_f, me_p) -> Mod.Apply (coerce me_f, coerce me_p)
    | V_Apply (me_f, me_p, v) ->
     Mod.Apply (coerce me_f, Mod.Apply (functor_of_view v,coerce me_p))
end
```

Note that `coerce` employs an auxiliary function, `functor_of_view`, to calculate a coercion functor from a given view. This implements the coercion semantics of views defined in §3.4.2.

7.9 An application over a system of first-order logic

Now all the modules for Mei have been defined. Once we define an implementation of `MMS_SYN`, we can build a module system on top of it. We first define a module `Name`. We separate it from the definition of Mei so that it may be redefined later to account for a more general implementation. In our implementation of `Name`, `t = string`.

```
module type NAME = sig
  type t
  val create: string -> t
  val generate_name: string -> t
  val name: t -> string
  val equal: t -> t -> bool
end
```

Given an implementation of NAME, the following implementation of MMS_SYN defines an abstract syntax tree of a first-order logic.

```
module Fol =
struct
  type sort = Name.t
  type const = Name.t * sort
  type func = Name.t * sort list * sort
  type pred = Name.t * sort list
  type variable = Ident.t * sort
  type term =
     Var of variable
   | Const of const
   | App of func * term list
  type formula =
     Eq of term * term
   | PredApp of pred * term list
   | And of formula * formula
   | Or of formula * formula
   | Imply of formula * formula
   | Nega of formula
   | Forall of variable * formula
   | Exist of variable * formula
  type language = {sorts: sort list; consts: const list;
    funcs: func list; preds: pred list}
  type mapping = {sm: (sort * sort) list; cm: (const * const) list;
    fm: (func * func) list; pm: (pred * pred) list}
  type sentence = formula
  type proof
  (*functions are ommitted*)
end
```

As shown in §7.4, MeiCore is a functor that takes a module of type MMS_SYN as a parameter. It defines the data types representing module types and module expressions based on the abstract types declared in MMS_SYN, e.g. language, mapping, sentence, and proof. By applying MeiCore to Fol, in which the abstract types declared in MMS_SYN are defined concretely, we derive a real implementation of Mei Core over a first-order logic system represented by Fol as follows:

```
module Fol_MeiCore = MeiCore(Fol)
```

A concrete module type checker and a concrete module expression evaluator are built by applying functor MeiCore_Eval to Fol (representing the syntax of a first-order logic), MeiCore (representing module types and module expressions), and Env and Ctx (representing the environment and context). First, we substitute Fol, MeiCore, Env, and Ctx for the formal parameters TheMms, TheModFunc, TheEnvFunc, and TheCtxFunc respectively. Then, the concrete representation of Mei Core over a first-order logic, Mod in MeiCore, is constructed by applying MeiCore to Fol. Then the environment and context are built by applying Env and Ctx to the derived concrete representation of Mei Core. Finally, the module type checker and module expression evaluator of Mei Core are built on top of the derived concrete representation of Mei Core, the environment, and the context as shown in §7.6.

```
module Fol_MeiCore_Eval =MeiCore_Eval(Fol)(MeiCore)(Env)(Ctx)
```

Similarly, an implementation of Mei is derived by applying functor Mei to Fol, MeiCore, and Tran (representing a toy interpretation). Again, first a concrete implementation of Mei Core of a first-order logic is constructed by applying MeiCore to Fol. The representation of Mei's module type and module expressions as well as the coercion functions are then defined.

```
module Fol_Mei = Mei(Fol)(MeiCore)(Tran)
```

Note that the construction of these three modules is independent, in the sense that the order of the construction is irrelative.

The three modules Fol_MeiCore, Fol_MeiCore_Eval, and Fol_Mei form the top-level testing environment. A module expression of Mei is represented by the type mod_expr in Fol_Mei. It can be translated to a module expression of Mei Core represented by the type mod_expr in Fol_MeiCore via the function coerce in Fol_Mei. The translated module expression is then type checked and evaluated by the function type_of and eval respectively in Fol_MeiCore_Eval. This finishes the implementation illustrated in Figure 7.1.

Chapter 8

A module system for multi-logic MMSs

Although Mei is logic independent, it is assumed to be built on top of an MMS with a particular logic. It is natural to ask if Mei can be applied on top of a system that supports multiple logics, e.g. a logical framework [78].

In this chapter, we briefly present a skeleton of a module system for a logical framework like multi-logic system.

(1) First, there is a logical framework supporting a module system, e.g. Mei, where the logical framework's logic is a single logic. We will call this logic a *meta-logic* since it is used to formalizing custom-designed logics.

(2) One more layer should be built on top of the logical framework's theories consisting custom-designed *logics* and *theories* of a particular logic. (They will be called logics and theories respectively in this chapter.) They are represented internally as a logical framework's theories, which are called *meta-theories* in this chapter. In particular, all theories within one logic will be represented as meta-theories which share a common sub-meta-theory representing the logic as shown in Figure 8.1.

(3) Although all meta-theories can be manipulated as in Mei, logics should be constructed by explicit definition, not derived from other logics. This makes sense because only a few logics need to be formalized within the logical framework.

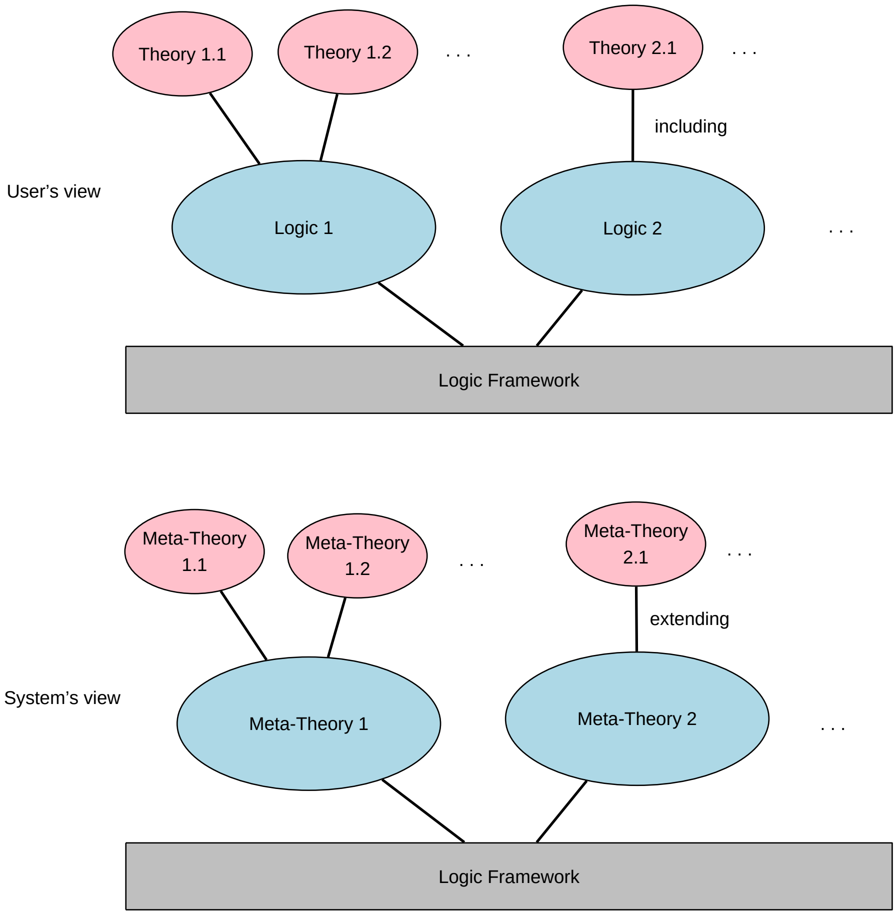

Figure 8.1: Logics and theories as meta-theories

(4) A network of logic interpretations in terms of meta-theory interpretations should be constructed at the time when a logic is added to the logical framework. One issue is that a logic interpretation may be very hard to build even though it does exist. Generally speaking, this requires a formalization of one logic within another logic.

(5) A theory is also represented as a meta-theory which has a particular sub-meta-theory, called the logic of the theory. It is then easy to identify if two theories are in the same logic or not. Theory operations such as extension, union, renaming, functor application are only applicable to theories in the same logic.

(6) Theory interpretations can be defined between two theories from the same logic or

two different logics. Ideally, the user only needs to define a translation between two theories without considering their logics. The translation between logics should be attached automatically to form a meta-theory translation, which should be checked to see if it is an interpretation. This is easy if both theories are in one logic. (Mechanisms should be provided to avoid rechecking the identity logic part.) It is not obvious if this can be done if the two theories are in different logics. This last issue is itself an interesting research topic.

(*a*) A theory specific symbol (non-logical symbol) in the source logic might not have its counterpart in any theory of the target logic. For instance, assume that the target theory has the logical symbols negation, conjunction, disjunction, and implication, whereas the source logic has only the logical symbols such as negation, conjunction, and disjunction. The implication symbol will be defined as a non-logic symbol in some particular theory in the source logic in terms of negation and conjunction. Clearly, there is no non-logical symbol for implication in any theory in the target logic.

(*b*) There are cases that, although there is no logic interpretation (or the logic interpretation is very hard to build) from a logic, say L_1, to another logic, say L_2, There is a logic interpretation from a sublogic of L_1, say L_1', in which a number of theories can be formalized easily, to L_2. It is hard for the system to identify such a sublogic where the logic interpretation exists. It is possible to separate L_1' from L_1 and build logic interpretations from L_1' to both L_1 and L_2. This later approach will necessarily complicate the logic network and it is better to be hidden from the user in the sense that a service requirement to a theory in L_1 using only L_1' will be redirected to L_1' under the screen as shown in Figure 8.2.

(7) A language should be defined to manipulate logics and theories as well as logic interpretations and theory interpretations. The semantics of this language should be defined in terms of the module expressions over meta-theories, which are only presented in the abstract syntactic tree level to avoid exposure to the user.

The above discussion is based on logical frameworks such as LF [78] or Isabelle [70]. The logic interpretation can then be easily expressed as a meta-theory interpretation over the meta-logic. In other words, the theory interpretations between theories from different logics are reduced to those between meta-theories from the same meta-logic. A further issue is how the ideas above fit in a framework where no single meta-logic

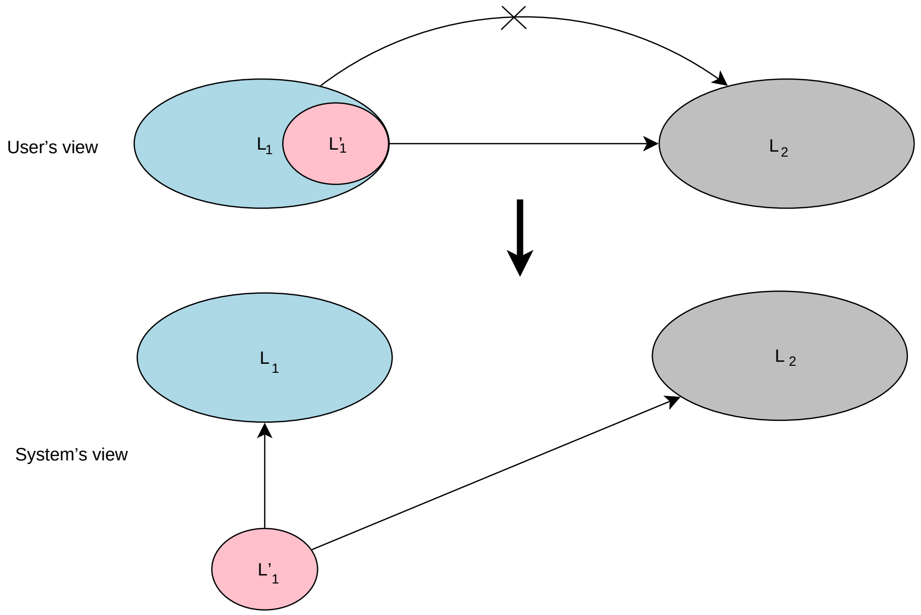

Figure 8.2: Sub-logic issue

is presented, e.g. FFMM [39, 40], a Formal Framework for Managing Mathematics. This may require a different formalization of Mei, in which we need a formalism of logics as well as theories. This is left for future work.

Chapter 9

Conclusion and future work

In this thesis we have presented several module systems, among which Mei is the most important, designed for MMSs. Mei integrates most of the modular mechanisms from both ML-family module systems and algebraic specification languages. In particular Mei supports higher-order functors with a fitting morphism like parameter passing mechanism. This provides a higher level of abstraction and module reusability. We showed Mei's power by comparing it with some current module systems. Moreover, Mei is relatively simple because of the use of views and coercion functors which separate the concerns of parameter passing from functor application. The analogy between Mei and type λ-calculus has a definite beauty.

There are two basic ways to use Mei:

(1) Build Mei as a module layer on top of an MMS, as we suggest in Chapter 7. The module system Mei is conceptually implemented as a "functor" with respect to an abstract signature `MMS_SYN`. Then we can formalize the underlying MMS as a structure implementing the signature `MMS_SYN`. The module layer is then built automatically by applying the functor to the structure. The implementation in Chapter 7 is an experiment. For practical use, we probably need a more sophisticated implementation to integrate Mei with the reasoning mechanisms of the underlying MMS such as theory interpretations and proofs. The implementation should also implement the features we present in Chapter 6.

(2) In addition to using Mei as a whole module system, some ideas of Mei can be incorporated in existing module systems. In particular, the idea of views and

> their coercion semantics can be easily integrated with an ML-family module system. This can be seen from the relation between Mei and Mei Core. Views can be defined for theories if theory interpretation is supported. It can be easily extended for functors if there is some notion of a type of a functor when higher-order functors are supported. The coercion semantics can be defined as in Mei. This is essentially because the views and their coercion semantics are orthogonal to the other mechanisms provided by Mei Core.

Future work includes: (1) a real implementation of Mei for an MMS, (2) a simplified version of DMei as discussed in §4.6, (3) a module system supporting multi-logic as discussed in §8, and (4) a proof or refutation of Conjecture 2.4.18.

Appendix A

Source code of the implementation

```
(* Title:          mei.ml
   Author:         Jian Xu
   Affiliation:    Department of Computing and Software
                   McMaster University
   Date:           May 2007
*)

module type IDENT = sig
  type t

  val create: string -> t
  val name: t -> string
  val equal: t -> t -> bool
end

module Ident:IDENT = struct
  type t = {name: string; stamp: int}

  let currstamp = ref 0
  let create s = currstamp := !currstamp + 1;
              {name = s; stamp = !currstamp}
  let name id = id.name
  let equal id1 id2 = (id1.stamp = id2.stamp)
end

module type MMS_SYN = sig
  type language
  type mapping
  type sentence
  type proof

  val is_language: language -> bool
  val is_sentence: language -> sentence -> bool
  val is_mapping_of: language -> mapping -> bool
  val is_proof: language -> sentence list -> sentence*proof -> bool
  val empty_lang: language
  val merge_lang: language -> language -> language
  val is_sublang: language -> language -> bool
  val lift: language -> mapping -> mapping
  val inv: mapping -> mapping
  val find_ren_ext: language*language -> language*language -> mapping
  val find_ren_union: language*language -> language*language
                  -> mapping*mapping
  val language_translate: language -> mapping -> language
  val sentence_translate: sentence -> mapping -> sentence
  val proof_translate: proof -> mapping -> proof

  (*pretty printer*)
  val print_lang: language -> unit
  val print_mapping: mapping ->unit
  val print_sentence: sentence -> unit
  val print_proof: proof -> unit
end

module type MEICORE = sig
  module Mms:MMS_SYN
  type thy_type = {
    lang_type: Mms.language;
    axioms_type: (Ident.t * Mms.sentence) list}
  type mod_type =
    Base of thy_type
   |Arrow of mod_type * mod_type
  type spec = {
    lang_spec:Mms.language;
    axioms_spec:(Ident.t * Mms.sentence) list;
    thms_spec:(Ident.t * Mms.sentence * Mms.proof) list}
  type mod_expr =
     Ident of Ident.t
   | Theory of spec * thy_type
   | Cast of mod_expr * thy_type
   | Extension of mod_expr * spec
   | Rename of mod_expr * Mms.mapping
   | Union of mod_expr * mod_expr
   | Functor of Ident.t * mod_type * mod_expr
   | Apply of mod_expr * mod_expr

  val is_subtype: mod_type -> mod_type -> bool
  val is_thytype: thy_type -> bool
  val is_thy: spec -> bool
  val subst: mod_expr -> (Ident.t * mod_expr) -> mod_expr
  val is_thytype_of: spec -> thy_type -> bool
  val merge_thytype: thy_type -> thy_type -> thy_type
  val merge_spec: spec -> spec -> spec
  val amal_ext: spec * thy_type -> spec -> spec * thy_type
  val amal_union: (spec*thy_type) -> (spec*thy_type) -> (spec*thy_type)
  val thy_type_translate: thy_type -> Mms.mapping -> thy_type
  val spec_translate: spec -> Mms.mapping -> spec
  val print_thy_type: thy_type -> unit
  val print_type: mod_type -> unit
  val print_spec: spec -> unit
  val print_mod_expr: mod_expr -> unit
end

module MeiCore(TheMms: MMS_SYN) = struct
module Mms = TheMms

(*type definition is the same as in MEICORE*)
type thy_type = {
   lang_type: Mms.language;
   axioms_type: (Ident.t * Mms.sentence) list}
type mod_type =
   Base of thy_type
 | Arrow of mod_type * mod_type
type spec = {
   lang_spec: Mms.language;
   axioms_spec: (Ident.t * Mms.sentence) list;
   thms_spec: (Ident.t * Mms.sentence * Mms.proof) list}
type mod_expr =
   Ident of Ident.t
 | Theory of spec * thy_type
 | Cast of mod_expr * thy_type
 | Extension of mod_expr * spec
 | Rename of mod_expr * Mms.mapping
 | Union of mod_expr * mod_expr
 | Functor of Ident.t * mod_type * mod_expr
 | Apply of mod_expr * mod_expr

(*auxiliary functions*)
let subset l1 l2 = let p x = List.mem x l2 in List.for_all p l1
let union l1 l2 = let p x = not (List.mem x l2)
                  in (List.filter p l1) @ l2
let remove x l = let p y = not (x = y) in List.filter p l
let empty l = match l with [] -> true | _ -> false

(*functions for types*)
let rec is_subtype modtype modtype' =
  match (modtype, modtype') with
    (Base thytype, Base thytype') ->
      Mms.is_sublang thytype'.lang_type thytype'.lang_type
      && subset thytype'.axioms_type thytype.axioms_type
  | (Arrow (modtype1, modtype2), Arrow (modtype1', modtype2')) ->
     is_subtype modtype1' modtype1 && is_subtype modtype2 modtype2'
  | _ -> false
let is_thytype thytype =
  let (_, axioms) = List.split thytype.axioms_type in
  Mms.is_language thytype.lang_type &&
  List.for_all (Mms.is_sentence thytype.lang_type) axioms
let merge_thytype thytype1 thytype2 =
  {lang_type = Mms.merge_lang thytype1.lang_type thytype2.lang_type;
  axioms_type = union thytype1.axioms_type thytype2.axioms_type}
let thy_type_translate thytype rho =
  let new_lang = Mms.language_translate thytype.lang_type rho in
  let translate (id, phi) = (id, Mms.sentence_translate phi rho) in
  let new_axioms = List.map translate thytype.axioms_type in
  {lang_type = new_lang; axioms_type = new_axioms}

(*functions for modules*)
let is_thy spec =
  let (_, axioms) = List.split spec.axioms_spec in
  let is_thm (_, phi, pf) =
      Mms.is_proof spec.lang_spec axioms (phi, pf) in
      Mms.is_language spec.lang_spec &&
      List.for_all (Mms.is_sentence spec.lang_spec) axioms &&
      List.for_all is_thm spec.thms_spec
let is_thytype_of spec thytype =
  let sen_of (n,thm,_) = (n,thm) in
  let thms = List.map sen_of spec.thms_spec in
  is_subtype (Base {lang_type = spec.lang_spec;
      axioms_type = spec.axioms_spec @ thms}) (Base thytype)
let type_of_thy thy =
  Base {lang_type = thy.lang_spec;axioms_type = thy.axioms_spec}
let merge_spec spec1 spec2 =
  {lang_spec = Mms.merge_lang spec1.lang_spec spec2.lang_spec;
  axioms_spec = union spec1.axioms_spec spec2.axioms_spec;
  thms_spec = union spec1.thms_spec spec2.thms_spec}
let spec_translate spec rho =
    let new_lang = Mms.language_translate spec.lang_spec rho in
    let translate (id, phi) = (id, Mms.sentence_translate phi rho) in
    let new_axioms = List.map translate spec.axioms_spec in
    let translate' (id, phi, pf) =
        (id, Mms.sentence_translate phi rho,
        Mms.proof_translate pf rho) in
    let new_thms = List.map translate' spec.thms_spec in
    {lang_spec = new_lang; axioms_spec = new_axioms;
    thms_spec = new_thms}
  let amal_union (spec1, thytype1) (spec2, thytype2) =
    let (rho1, rho2) =
        Mms.find_ren_union (spec1.lang_spec, thytype1.lang_type)
        (spec2.lang_spec, thytype2.lang_type) in
    let (spec1', spec2') =
        (spec_translate spec1 rho1, spec_translate spec2 rho2) in
    (merge_spec spec1' spec2', merge_thytype thytype1 thytype2)
  let amal_ext (spec1, thytype1) spec2 =
    let sen_of (n,thm,_) = (n,thm) in
    let thms = List.map sen_of spec2.thms_spec in
    let thytype2 = {lang_type = spec2.lang_spec;
        axioms_type = union spec2.axioms_spec thms} in
    let rho1 = Mms.find_ren_ext (spec1.lang_spec, thytype1.lang_type)
        (spec2.lang_spec, thytype2.lang_type) in
    let spec1' = spec_translate spec1 rho1 in
    (merge_spec spec1' spec2, merge_thytype thytype1 thytype2)

  (*substitution function*)
  let rec subst me (id, me_sub) =
    match me with
     Ident id' -> if Ident.equal id id' then me_sub else me
   | Theory (thy,tp) -> me
   | Cast(me', tp) ->
       let me'' = subst me' (id, me_sub) in Cast(me'',tp)
   | Extension(me', spec) ->
       let me'' = subst me' (id, me_sub) in Extension(me'', spec)
   | Rename(me', rho) ->
```

```
      let me'' = subst me' (id, me_sub) in Rename(me'',rho)
   | Union(me1,me2) ->
      let me1' = subst me1 (id, me_sub) in
      let me2' = subst me2 (id, me_sub) in
      Union(me1',me2')
   | Functor(id', tp, me') ->
      if Ident.equal id id' then me
      else let me'' = subst me' (id, me_sub) in
      Functor(id', tp, me'')
   | Apply(me_f, me_p) ->
      let me_f' = subst me_f (id, me_sub) in
      let me_p' = subst me_p (id, me_sub) in
      Apply(me_f',me_p')

(*pretty printer*)
let print_thy_type thytype =
  print_string "{"; Mms.print_lang thytype.lang_type;
  print_string "\n"; print_string "Axioms:"; print_newline();
  let print_axiom (id, phi) =
    (print_string "\t";print_string (Ident.name id);print_string ":";
    Mms.print_sentence phi; print_newline() ) in
  List.iter print_axiom thytype.axioms_type;
  print_string "}"; print_newline()
let rec print_type tp =
    match tp with
    Base thytype -> print_thy_type thytype
  | Arrow (tp1, tp2) -> print_string "{"; print_type tp1;
     print_string "}"; print_newline(); print_string " -> ";
     print_newline(); print_string "{"; print_type tp2;
     print_string "}"
let print_spec spec =
  print_string "{"; Mms.print_lang spec.lang_spec; print_newline();
  print_string "Axioms:"; print_newline();
  let print_axiom (id, phi) = (print_string "\t";
    print_string (Ident.name id);print_string ": ";
    Mms.print_sentence phi; print_newline() ) in
    List.iter print_axiom spec.axioms_spec; print_newline();
    print_string "Theorems:";
    let print_theorem (id, phi, pf) = (print_string "\t";
      print_string (Ident.name id);print_string ": ";
      Mms.print_sentence phi; Mms.print_proof pf; print_newline()) in
    List.iter print_theorem spec.thms_spec; print_string "}";
    print_newline()
  let rec print_mod_expr me =
      match me with
      Ident id -> print_string (Ident.name id)
    | Theory (spec, thytype) -> print_spec spec; print_newline()
    | Cast (me',tp) -> print_string "(";
       print_thy_type tp; print_string ") "; print_string " ";
       print_mod_expr me'; print_newline()
    | Extension (me', spec) -> print_mod_expr me';
       print_string " extended by "; print_newline();
       print_spec spec; print_newline()
    | Rename (me', mp) -> print_mod_expr me';
       print_string " "; Mms.print_mapping mp; print_newline()
    | Union (me1, me2) -> print_mod_expr me1;  print_string " + ";
       print_mod_expr me2; print_newline()
    | Functor (id, tp, me') -> print_string "Functor ";
       print_string (Ident.name id); print_string ":";
       print_type tp; print_string ". ";
       print_mod_expr me'; print_newline()
    | Apply (me1, me2) -> print_string "("; print_mod_expr me1;
       print_string " @ "; print_mod_expr me2; print_string ")";
       print_newline()
end

module type ENV = sig
  type mod_expr
  type mod_type
  type content
  type t = (Ident.t * content) list
  val empty: t
  val add_expr: Ident.t -> mod_expr -> t -> t
  val find_expr: Ident.t -> t -> mod_expr
  val mem_expr: Ident.t -> t -> bool
  val add_type: Ident.t -> mod_type -> t -> t
  val find_type: Ident.t -> t -> mod_type
  val mem_type: Ident.t -> t -> bool
end

module Env (TheMod:MEICORE) = struct
  type mod_expr = TheMod.mod_expr
  type mod_type = TheMod.mod_type
  type content = Mod of mod_expr | Type of mod_type
  type t = (Ident.t * content) list

  let empty = []
  let add_expr id modu env = (id, Mod modu)::env
  let add_type id tp env = (id, Type tp)::env
  let rec find id env = match env with
      [] -> raise Not_found
    | (id', cont) :: tl ->
        if Ident.equal id id' then cont else find id tl
  let find_expr id env =
    match find id env with
      Mod modu -> modu
    | _ -> raise Not_found
  let find_type id env =
    match find id env with
      Type tp -> tp
    | _ -> raise Not_found
  let rec mem_expr id env = match env with
      [] -> false
    | (id', Mod modu) :: tl ->
        if Ident.equal id id' then true else mem_expr id tl
    | (id', Type tp) :: tl -> mem_expr id tl
  let rec mem_type id env = match env with
```

```
      [] -> false
    | (id’, Mod modu) :: tl -> mem_expr id tl
    | (id’, Type tp) :: tl ->
        if Ident.equal id id’ then true else mem_expr id tl
end

module type CTX = sig
  type mod_type
  type t = (Ident.t * mod_type) list

  val empty: t
  val add: Ident.t -> mod_type -> t -> t
  val find: Ident.t -> t -> mod_type
  val mem: Ident.t -> t -> bool
end

module Ctx (TheMod:MEICORE) = struct
  type mod_type = TheMod.mod_type
  type t = (Ident.t * mod_type) list

  let empty = []
  let add id modtype ctx = (id, modtype)::ctx
  let rec find id ctx = match ctx with
      [] -> raise Not_found
    | (id’, modtype) :: tl ->
        if Ident.equal id id’ then modtype else find id tl
  let rec mem id ctx = match ctx with
      [] -> false
    | (id’, modtype) :: tl ->
        if Ident.equal id id’ then true else mem id tl
end

module type MEICORE_EVAL = sig
  module Mms:MMS_SYN
  module Mod:MEICORE
  module Env:ENV with type mod_expr = Mod.mod_expr and
```

```
                    type mod_type = Mod.mod_type
  module Ctx:CTX with type mod_type = Mod.mod_type

  val type_of: Mod.mod_expr -> Ctx.t -> Env.t -> Mod.mod_type
  val eval: Mod.mod_expr -> Env.t -> Mod.mod_expr
  val print_type: Mod.mod_type -> unit
  val print_mod_expr: Mod.mod_expr -> unit
end

module MeiCore_Eval
  (TheMms:MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) ->
    MEICORE with module Mms = TheMms)
  (TheEnvFunc: functor (TheMod:MEICORE) ->
    ENV with type mod_expr =  TheMod.mod_expr and
    type mod_type = TheMod.mod_type)
  (TheCtxFunc: functor (TheMod:MEICORE) ->
    CTX with type mod_type = TheMod.mod_type)
= struct
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)
  module Env = TheEnvFunc(Mod)
  module Ctx = TheCtxFunc(Mod)
  exception MEICORE_EVAL of string

  (*auxiliary functions*)
  let subset l1 l2 = let p x = List.mem x l2 in List.for_all p l1
  let union l1 l2 = let p x = not (List.mem x l2) in
    (List.filter p l1) @ l2
  let remove x l = let p y = not (x = y) in List.filter p l
  let empty l = match l with [] -> true | _ -> false

  (*type checking function*)
  let rec type_of te ctx env =
     match te with
     Mod.Ident id -> if Ctx.mem id ctx then Ctx.find id ctx
```

```
      else (if Env.mem_expr id env
      then type_of (Env.find_expr id env) ctx env
      else raise (MEICORE_EVAL "Module identity not defined."))
  | Mod.Theory (thy, thytype) ->
      if ((Mod.is_thy thy) && (Mod.is_thytype thytype)
       && Mod.is_thytype_of thy thytype)
      then Mod.Base thytype
      else raise(MEICORE_EVAL "Theory definition not closed.")
  | Mod.Cast(te', thytype) -> if Mod.is_thytype thytype then
      (match type_of te' ctx env with
        Mod.Base thytype' ->
          if (Mod.is_subtype (Mod.Base thytype') (Mod.Base thytype))
          then Mod.Base thytype
          else raise (MEICORE_EVAL "Invalid casting.")
      | _ -> raise (MEICORE_EVAL "Funtors cannot be casted."))
       else raise (MEICORE_EVAL "Theory type definition not closed.")
  | Mod.Extension(te',spec) -> (match type_of te' ctx env with
        Mod.Base thytype ->
          let thy_of_thytype = {Mod.lang_spec = thytype.Mod.lang_type;
            Mod.axioms_spec = thytype.Mod.axioms_type;
            Mod.thms_spec = []} in
          if (Mod.is_thy (Mod.merge_spec thy_of_thytype spec))
          then let get_axiom (id, phi, pf) = (id, phi) in
            let thytype' = {Mod.lang_type = spec.Mod.lang_spec;
              Mod.axioms_type = union spec.Mod.axioms_spec
              (List.map get_axiom spec.Mod.thms_spec)} in
            Mod.Base (Mod.merge_thytype thytype thytype')
          else raise (MEICORE_EVAL "The result theory is not closed")
      | _ -> raise (MEICORE_EVAL "Functor cannot be extended."))
  | Mod.Rename (te', rho) -> (match type_of te' ctx env with
        Mod.Base thytype ->
          if Mms.is_mapping_of thytype.Mod.lang_type rho
          then Mod.Base (Mod.thy_type_translate thytype rho)
          else raise (MEICORE_EVAL "Bad renaming.")
      | _ -> raise (MEICORE_EVAL "Functors cannot be renamed."))
  | Mod.Union(te1,te2) ->
```

```
      (match (type_of te1 ctx env, type_of te2 ctx env) with
        (Mod.Base thytype1, Mod.Base thytype2) ->
           Mod.Base (Mod.merge_thytype thytype1 thytype2)
      | _ -> raise (MEICORE_EVAL "Functors cannot be merged."))
  | Mod.Functor(id, tp, te') -> let ctx' = Ctx.add id tp ctx in
       Mod.Arrow (tp, type_of te' ctx' env)
  | Mod.Apply(te_f, te_p) -> (match type_of te_f ctx env with
        Mod.Arrow(tp_1,tp_2) ->
          if Mod.is_subtype (type_of te_p ctx env) tp_1
          then tp_2
          else raise (MEICORE_EVAL
            "Type mismatched in functor application.")
      | _ -> raise (MEICORE_EVAL
        "Functor required for functor application."))

let rec eval te env =
  match te with
   Mod.Ident id -> if Env.mem_expr id env
     then let te' = Env.find_expr id env in eval te' env
     else Mod.Ident id
  | Mod.Theory (thy,tp) -> te
  | Mod.Cast(te', tp) -> let te_eval = eval te' env in
       (match  te_eval with
        Mod.Theory (spec,typtype) -> Mod.Theory (spec,tp)
      | _ -> Mod.Cast(te_eval, tp))
  | Mod.Extension(te',spec) -> let te_eval = eval te' env in
       (match te_eval with
         Mod.Theory (spec_eval, tp) -> let (new_spec, new_tp) =
           Mod.amal_ext (spec_eval,tp) spec in
           Mod.Theory (new_spec, new_tp)
      | _ -> Mod.Extension(te_eval, spec))
  | Mod.Rename(te', rho) -> let te_eval = eval te' env in
       (match te_eval with
         Mod.Theory (spec, thytype) -> Mod.Theory
           (Mod.spec_translate spec rho,
               Mod.thy_type_translate thytype rho)
      | _ -> Mod.Rename(te_eval, rho))
  | Mod.Union(te1,te2) -> let te1_eval = eval te1 env in
      let te2_eval = eval te2 env in
       (match (te1_eval, te2_eval) with
         (Mod.Theory (spec1,tp1), Mod.Theory (spec2,tp2)) ->
           let (new_spec, new_tp) =
             Mod.amal_union (spec1,tp1) (spec2,tp2)
           in Mod.Theory (new_spec, new_tp)
      | (_, _) -> Mod.Union(te1_eval, te2_eval))
  | Mod.Functor(id, tp, te') -> (match te' with
         Mod.Apply(te'', Mod.Ident id') -> if Ident.equal id id'
         then eval te'' env else Mod.Functor(id, tp, te')
      | _ -> Mod.Functor(id, tp, te'))
  | Mod.Apply(te_f, te_p) -> (match eval te_f env with
         Mod.Functor(id, tp, te') ->
           eval (Mod.subst te' (id, te_p)) env
      | _ -> raise (MEICORE_EVAL
        "Funtor required for functor application."))

  (*pretty printer*)
  let print_type = Mod.print_type
  let print_mod_expr = Mod.print_mod_expr
end

module type TRAN = sig
  module Mms: MMS_SYN
  module Mod: MEICORE

  val thm_from_obl: Mod.thy_type -> Mod.thy_type ->
    Mms.mapping -> (Ident.t * Mms.sentence * Mms.proof) list
end

module Tran
  (TheMms:MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) ->
    MEICORE with module Mms = TheMms)
```

```
= struct
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)

  let thm_from_obl thytype1 thytype rho = []
end

module type MEI = sig
  module Mms:MMS_SYN
  module Mod: MEICORE
  module Tran: TRAN

  type mappings = Single of Mms.mapping | Pair of mappings * mappings
  type view = Mod.mod_type * Mod.mod_type * mappings
  type mod_expr =
     Ident of Ident.t
   | Theory of Mod.spec * Mod.thy_type
   | Cast of mod_expr * Mod.thy_type
   | Extension of mod_expr * Mod.spec
   | Rename of mod_expr * Mms.mapping
   | Union of mod_expr * mod_expr
   | Functor of Ident.t * Mod.mod_type * mod_expr
   | Apply of mod_expr * mod_expr
   | V_Apply of mod_expr * mod_expr * view

  val coerce: mod_expr -> Mod.mod_expr
  val print_mod_expr: mod_expr -> unit
end

module Mei
  (TheMms: MMS_SYN)
  (TheModFunc: functor (TheMms:MMS_SYN) ->
    MEICORE with module Mms = TheMms)
  (TheTranFunc: functor (TheMms:MMS_SYN)
    -> functor (TheModFunc: functor (TheMms:MMS_SYN) ->
      MEICORE with module Mms = TheMms)
    -> TRAN with module Mms = TheMms and
      module Mod = TheModFunc(TheMms))
= struct
  exception MEI of string
  module Mms = TheMms
  module Mod = TheModFunc(TheMms)
  module Tran = TheTranFunc(TheMms)(TheModFunc)

  (*type definition is the same as in MEI*)
  type mappings = Single of Mod.Mms.mapping
    | Pair of mappings * mappings
  type view = Mod.mod_type * Mod.mod_type * mappings
  type mod_expr =
      Ident of Ident.t
    | Theory of Mod.spec * Mod.thy_type
    | Cast of mod_expr * Mod.thy_type
    | Extension of mod_expr * Mod.spec
    | Rename of mod_expr * Mms.mapping
    | Union of mod_expr * mod_expr
    | Functor of Ident.t * Mod.mod_type * mod_expr
    | Apply of mod_expr * mod_expr
    | V_Apply of mod_expr * mod_expr * view

  let rec functor_of_view v = match v with
      (Mod.Base thytype1, Mod.Base thytype2, Single rho) ->
        let rho' = Mod.Mms.lift thytype2.Mod.lang_type
          (Mod.Mms.inv rho) in
        let del = Tran.thm_from_obl thytype1 thytype2 rho in
        let idX = Ident.create("X") in
        let spec = {Mod.lang_spec = Mms.empty_lang;
          Mod.axioms_spec = []; Mod.thms_spec = del} in
        let me1 = Mod.Extension(Mod.Ident idX, spec) in
        let me2 = Mod.Rename(me1, rho') in
        let me3 = Mod.Cast(me2, thytype1) in
        Mod.Functor (idX, Mod.Base thytype2, me3)
    | (Mod.Arrow (type1, type2), Mod.Arrow (type1', type2'),
    Pair (rhos1, rhos2)) ->
      let c1 = functor_of_view (type1’, type1,rhos1) in
      let c2 = functor_of_view (type2, type2’,rhos2) in
      let idX = Ident.create("X") in
      let idF = Ident.create("F") in
      let me1 = Mod.Apply (c1, Mod.Ident idX) in
      let me2 = Mod.Apply (Mod.Ident idF, me1) in
      let me3 = Mod.Apply (c2, me2) in
      let me4 = Mod.Functor (idX, type1, me3) in
      Mod.Functor (idF, Mod.Arrow (type1’, type2’), me4)
  | _ -> raise (MEI "Bad view")

let rec coerce me = match me with
    Ident id -> Mod.Ident id
  | Theory (spec, tp) -> Mod.Theory (spec,tp)
  | Cast (me’, btyp) -> Mod.Cast (coerce me’, btyp)
  | Extension (me’, spec) -> Mod.Extension (coerce me’, spec)
  | Rename (me’, rho) -> Mod.Rename (coerce me’, rho)
  | Union (me1, me2) -> Mod.Union (coerce me1, coerce me2)
  | Functor (id, tp, me’) -> Mod.Functor (id, tp, coerce me’)
  | Apply (me_f, me_p) -> Mod.Apply (coerce me_f, coerce me_p)
  | V_Apply (me_f, me_p, v) ->
      Mod.Apply (coerce me_f,
        Mod.Apply (functor_of_view v, coerce me_p))

(*pretty printer*)
let rec print_mappings rhos = match rhos with
    Single rho -> Mms.print_mapping rho
  | Pair (rhos1, rhos2) -> print_string "(" ;
      print_mappings rhos1; print_string ", ";
      print_mappings rhos2;print_string ")"
let print_view (tp1, tp2, mps) = (print_string "{";
  Mod.print_type tp2; print_string "}"; print_newline();
  print_string " as "; print_newline();
  print_string "{"; Mod.print_type tp1; print_string "}";
  print_string " by "; print_mappings mps)
  let rec print_mod_expr me =
      match me with
      Ident id -> print_string (Ident.name id)
    | Theory (spec, thytype) -> Mod.print_spec spec; print_newline()
    | Cast (me’,tp) -> Mod.print_thy_type tp; print_string " ";
        print_mod_expr me’; print_newline()
    | Extension (me’, spec) -> print_mod_expr me’;
      print_string " extended by "; print_newline();
      Mod.print_spec spec; print_newline()
    | Rename (me’, mp) -> print_mod_expr me’;  print_string " ";
        Mms.print_mapping mp; print_newline()
    | Union (me1, me2) -> print_mod_expr me1;  print_string " + ";
        print_mod_expr me2; print_newline()
    | Functor (id, tp, me’) -> print_string "Functor ";
        print_string (Ident.name id); print_string ":";
        Mod.print_type tp; print_string ". ";
        print_mod_expr me’; print_newline()
    | Apply (me1, me2) -> print_string "("; print_mod_expr me1;
        print_string " @ "; print_mod_expr me2;
        print_string ")"; print_newline()
    | V_Apply (me_f, me_p, v) -> print_string "("; print_mod_expr me_f;
        print_string " @ ";
        print_mod_expr me_p; print_string " with "; print_newline();
        print_view v; print_string "()"; print_newline()
end

module type NAME = sig
  type t

  val create: string -> t
  val generate_name: string -> t
  val name: t -> string
  val equal: t -> t -> bool
end

module Name : NAME = struct
```

```
  type t = string

  let newnum = ref 0
  let create s = s
  let generate_name s = newnum := !newnum + 1; s^(string_of_int !newnum)
  let name n = n
  let equal n1 n2 = (n1 = n2)
end

module Fol = struct
  exception FIRST_ORDER of string
  type sort = Name.t
  type const = Name.t * sort
  type func = Name.t * sort list * sort
  type pred = Name.t * sort list
  type variable = Ident.t * sort
  type term =
      Var of variable
    | Const of const
    | App of func * term list
  type formula =
      Eq of term * term
    | PredApp of pred * term list
    | And of formula * formula
    | Or of formula * formula
    | Imply of formula * formula
    | Nega of formula
    | Forall of variable * formula
    | Exist of variable * formula
  type language = {sorts: sort list; consts: const list;
    funcs: func list; preds: pred list}
  type mapping = {sm: (sort * sort) list; cm: (const * const) list;
    fm: (func * func) list; pm: (pred * pred) list}
  type sentence = formula
  type proof
```

```
(*auxiliary functions*)
let subset l1 l2 = let p x = List.mem x l2 in List.for_all p l1
let union l1 l2 = let p x = not (List.mem x l2)
  in (List.filter p l1) @ l2
let inter l1 l2 = let p x = (List.mem x l2) in (List.filter p l1)
let diff l1 l2 = let p x = not (List.mem x l2) in (List.filter p l1)
let remove x l = let p y = not (x = y) in List.filter p l
let empty l = match l with [] -> true | _ -> false

(*functions for language*)
let empty_lang = {sorts = []; consts = []; funcs = []; preds = []}
let is_lang sorts consts functions predicates =
  let is_const (name, s) = List.mem s sorts in
  let is_func (name, sl, s) = List.mem s sorts && subset sl sorts in
  let is_pred (name, sl) = subset sl sorts in
  List.for_all is_const consts && List.for_all is_func functions
      && List.for_all is_pred predicates
let is_language lang =
  is_lang lang.sorts lang.consts lang.funcs lang.preds
let is_sublang lang_1 lang_2 = is_language lang_1
  && is_language lang_2
  && (subset lang_1.sorts lang_2.sorts)
  && (subset lang_1.consts lang_2.consts)
  && (subset lang_1.funcs lang_2.funcs)
  && (subset lang_1.preds lang_2.preds)
let eq_lang lang_1 lang_2 = (is_sublang lang_1 lang_2)
  && (is_sublang lang_2 lang_1)
let rec merge_lang lang1 lang2 =
  let new_sorts = union lang1.sorts lang2.sorts in
  let new_consts = union lang1.consts lang2.consts in
  let new_funcs = union lang1.funcs lang2.funcs in
  let new_preds = union lang1.preds lang2.preds in
  {sorts = new_sorts; consts = new_consts;
    funcs = new_funcs; preds = new_preds}
let language_translate lang rho =
  let image pair_list x = List.assoc x pair_list in {
    sorts = List.map (image rho.sm) lang.sorts;
    consts = List.map (image rho.cm) lang.consts;
    funcs = List.map (image rho.fm) lang.funcs;
    preds = List.map (image rho.pm) lang.preds}

  (*functions for mapping*)
  let sub_mapping rho1 rho2 = subset rho1.sm rho2.sm
    && subset rho1.cm rho2.cm && subset rho1.fm rho2.fm
    && subset rho1.pm rho2.pm
  let eq_mapping rho1 rho2 = sub_mapping rho1 rho2 && sub_mapping rho2 rho1
  let is_mapping rho =
    let sort_compatible s1 s2 = let s1' = List.assoc s1 rho.sm in
        Name.equal s1' s2 in
    let const_compatible ((name1, s1), (name2, s2)) =
      sort_compatible s1 s2 in
    let func_compatible ((name1,sl1,s1), (name2,sl2,s2)) =
      List.for_all2 sort_compatible sl1 sl2
        && sort_compatible s1 s2 in
    let pred_compatible ((name1,sl1), (name2,sl2)) =
        List.for_all2 sort_compatible sl1 sl2 in
    List.for_all const_compatible rho.cm
      && List.for_all func_compatible rho.fm
      && List.for_all pred_compatible rho.pm
  let source_of rho =
    let fst pair_list = let (sl,_) = List.split pair_list in sl
    in {sorts = fst rho.sm; consts = fst rho.cm;
      funcs = fst rho.fm; preds = fst rho.pm}
  let is_mapping_of lang rho = is_mapping rho
    && eq_lang lang (source_of rho)
  let rec lift lang rho =
    let find_diff list pair_list =
      let (sl,_) = List.split pair_list in
      let p x = not (List.mem x sl) in
      List.filter p list in
    let s_diff = find_diff lang.sorts rho.sm in
    let c_diff = find_diff lang.consts rho.cm in
    let f_diff = find_diff lang.funcs rho.fm in
    let p_diff = find_diff lang.preds rho.pm in
    let sm_diff = List.combine s_diff s_diff in
    let new_sm = rho.sm @ sm_diff in
    let find_image sort = List.assoc sort new_sm in
    let cm_diff =
      let make_const_pair (name, sort) =
        ((name,sort),(name, find_image sort)) in
      List.map make_const_pair c_diff in
    let fm_diff =
      let make_func_pair (name, sorts, sort) =
        ((name,sorts,sort),
          (name, List.map find_image sorts,find_image sort)) in
      List.map make_func_pair f_diff in
    let pm_diff =
      let make_pred_pair (name, sorts) =
        ((name,sorts),(name, List.map find_image sorts)) in
      List.map make_pred_pair p_diff in
    let new_cm = rho.cm @ cm_diff in
    let new_fm = rho.fm @ fm_diff in
    let new_pm = rho.pm @ pm_diff in
    {sm = new_sm; cm = new_cm; fm = new_fm; pm = new_pm}
  let inv rho = let rev (a,b) = (b,a) in
    {sm = List.map rev rho.sm;cm = List.map rev rho.cm;
    fm = List.map rev rho.fm;pm = List.map rev rho.pm}
  let find_ren_sort (sort_diff1, sort_comm1) (sort_diff2, sort_comm2)=
    let sort_candidate =
      diff (inter sort_diff1 sort_diff2)
        (inter sort_comm1 sort_comm2) in
    let make_pair s = (s, Name.generate_name (Name.name s)) in
    (List.map make_pair sort_candidate,
      List.map make_pair sort_candidate)
  let find_ren_const (const_diff1, const_comm1, rho_sm1)
    (const_diff2, const_comm2, rho_sm2) =
    let find_image s rho_sm = if List.mem_assoc s rho_sm
      then List.assoc s rho_sm else s in
    let change_const_sort rho_sm (n, s) = (n,find_image s rho_sm) in
    let new_const_diff1 =
      List.map (change_const_sort rho_sm1) const_diff1 in
    let new_const_diff2 =
      List.map (change_const_sort rho_sm2) const_diff2 in
    let const_candidate =
      diff (inter new_const_diff1 new_const_diff2)
        (inter const_comm1 const_comm2) in
    let make_pair (n, s) =
      ((n,s), (Name.generate_name (Name.name n), s)) in
    (List.map make_pair const_candidate,
      List.map make_pair const_candidate)
  let find_ren_func (func_diff1, func_comm1, rho_sm1)
    (func_diff2, func_comm2, rho_sm2) =
    let find_image s rho_sm = if List.mem_assoc s rho_sm
      then List.assoc s rho_sm else s in
    let change_func_sort rho_sm (n, sl, s) =
      (n, List.map (function s -> find_image s rho_sm)
        sl,find_image s rho_sm) in
    let new_func_diff1 =
      List.map (change_func_sort rho_sm1) func_diff1 in
    let new_func_diff2 =
      List.map (change_func_sort rho_sm2) func_diff2 in
    let func_candidate =
      diff (inter new_func_diff1 new_func_diff2)
        (inter func_comm1 func_comm2) in
    let make_pair (n, sl, s) =
      ((n,sl,s), (Name.generate_name (Name.name n), sl, s)) in
    (List.map make_pair func_candidate,
      List.map make_pair func_candidate)
  let find_ren_pred (pred_diff1, pred_comm1, rho_sm1)
    (pred_diff2, pred_comm2, rho_sm2) =
    let find_image s rho_sm = if List.mem_assoc s rho_sm
      then List.assoc s rho_sm else s in
    let change_pred_sort rho_sm (n, sl) =
      (n,List.map (function s -> find_image s rho_sm) sl) in
    let new_pred_diff1 =
      List.map (change_pred_sort rho_sm1) pred_diff1 in
    let new_pred_diff2 =
      List.map (change_pred_sort rho_sm2) pred_diff2 in
    let pred_candidate =
      diff (inter new_pred_diff1 new_pred_diff2)
        (inter pred_comm1 pred_comm2) in
    let make_pair (n, sl) =
      ((n,sl), (Name.generate_name (Name.name n), sl)) in
    (List.map make_pair pred_candidate,
      List.map make_pair pred_candidate)
  let find_ren_ext (lang_diff1, lang_comm1) (lang_diff2, lang_comm2)=
    let (rho_sm1, rho_sm2) =
      find_ren_sort (lang_diff1.sorts, lang_comm1.sorts)
        (lang_diff2.sorts, lang_comm2.sorts) in
    let (rho_cm1, rho_cm2) =
      find_ren_const (lang_diff1.consts, lang_comm1.consts, rho_sm1)
        (lang_diff2.consts, lang_comm2.consts, rho_sm2) in
    let (rho_fm1, rho_fm2) =
      find_ren_func (lang_diff1.funcs, lang_comm1.funcs, rho_sm1)
        (lang_diff2.funcs, lang_comm2.funcs, rho_sm2) in
    let (rho_pm1, rho_pm2) =
      find_ren_pred (lang_diff1.preds, lang_comm1.preds, rho_sm1)
        (lang_diff2.preds, lang_comm2.preds, rho_sm2) in
    lift lang_diff1 {sm=rho_sm1;cm=rho_cm1;fm=rho_fm1;pm=rho_pm1}
  let find_ren_union (lang_diff1,lang_comm1) (lang_diff2,lang_comm2) =
    let (rho_sm1, rho_sm2) =
      find_ren_sort (lang_diff1.sorts, lang_comm1.sorts)
        (lang_diff2.sorts, lang_comm2.sorts) in
    let (rho_cm1, rho_cm2) =
      find_ren_const (lang_diff1.consts, lang_comm1.consts, rho_sm1)
        (lang_diff2.consts, lang_comm2.consts, rho_sm2) in
    let (rho_fm1, rho_fm2) =
      find_ren_func (lang_diff1.funcs, lang_comm1.funcs, rho_sm1)
        (lang_diff2.funcs, lang_comm2.funcs, rho_sm2) in
    let (rho_pm1, rho_pm2) =
      find_ren_pred (lang_diff1.preds, lang_comm1.preds, rho_sm1)
        (lang_diff2.preds, lang_comm2.preds, rho_sm2) in
    (lift lang_diff1 {sm=rho_sm1;cm=rho_cm1;fm=rho_fm1;pm=rho_pm1},
      lift lang_diff2 {sm=rho_sm2;cm=rho_cm2;fm=rho_fm2;pm=rho_pm2})

  (*functions for sentence*)
  let rec is_term_of_lang lang t = match t with
      Var v -> true
    | Const c -> List.mem c lang.consts
    | App (f, t_list) -> List.mem f lang.funcs
      && List.for_all (is_term_of_lang lang) t_list
  let rec sort_of_term t = match t with
      Var (id, s) -> s
    | Const (name, s) -> s
    | App ((name, sl, s), t_list) ->
        if List.for_all2 sort_match sl t_list then s
        else raise (FIRST_ORDER
          "Arguments mismatched in function application.")
  and sort_match s t = Name.equal s (sort_of_term t)
  let is_sort_safe t = match t with
      Var v -> true
    | Const c -> true
    | App ((name, sl, s), t_list) ->
      List.for_all2 sort_match sl t_list
  let is_addmitted_term lang t =
    (is_term_of_lang lang t) && (is_sort_safe t)
  let rec var_of_term t = match t with
      Var v -> [v]
    | Const c -> []
    | App (f, t_list) -> let v_list = List.map var_of_term t_list in
        List.fold_left union [] v_list
  let rec term_translate t rho = match t with
      Var (id, s) -> Var (id, List.assoc s rho.sm)
    | Const c -> Const (List.assoc c rho.cm)
    | App (f, t_list) ->
        let t’_list =
          List.map (function tm -> term_translate tm rho) t_list in
        let f' = List.assoc f rho.fm in
        App (f', t'_list)
  let rec is_formula_of_lang lang phi = match phi with
      Eq (t,t') -> is_addmitted_term lang t
        && is_addmitted_term lang t'
        && Name.equal (sort_of_term t) (sort_of_term t')
    | PredApp ((name, sl), t_list) -> List.mem (name,sl) lang.preds
        && List.for_all (is_addmitted_term lang) t_list
        && List.for_all2 sort_match sl t_list
    | And (phi', phi'') -> is_formula_of_lang lang phi'
        && is_formula_of_lang lang phi''
    | Or (phi', phi'') -> is_formula_of_lang lang phi'
        && is_formula_of_lang lang phi''
    | Imply (phi', phi'') -> is_formula_of_lang lang phi'
        && is_formula_of_lang lang phi''
    | Nega phi' -> is_formula_of_lang lang phi'
    | Forall (x, phi') -> is_formula_of_lang lang phi'
    | Exist (x, phi') -> is_formula_of_lang lang phi'
  let rec var_of_formula phi = match phi with
      Eq (t,t') -> let var_set = var_of_term t in
        let var_set' = var_of_term t' in union var_set var_set'
    | PredApp (p, t_list) ->
        let var_set = List.map var_of_term t_list in
        List.fold_left union [] var_set
    | And (phi', phi'') -> let var_set' = var_of_formula phi' in
        let var_set'' = var_of_formula phi'' in
        union var_set' var_set''
    | Or (phi', phi'') -> let var_set' = var_of_formula phi' in
        let var_set'' = var_of_formula phi'' in
        union var_set' var_set''
    | Imply (phi', phi'') -> let var_set' = var_of_formula phi' in
        let var_set'' = var_of_formula phi'' in
        union var_set' var_set''
    | Nega phi' ->  var_of_formula phi'
    | Forall (x, phi') -> let var_set' = var_of_formula phi' in
```

```
      remove x var_set'
    | Exist (x, phi') ->  let var_set' = var_of_formula phi' in
      remove x var_set'
let is_closed_formula lang phi =
  (is_formula_of_lang lang phi) && empty (var_of_formula phi)
let is_sentence = is_closed_formula
let rec formula_translate phi rho = match phi with
    Eq (t,t') -> let t_new = term_translate t rho in
      let t'_new = term_translate t' rho in Eq (t_new, t'_new)
  | PredApp (p, t_list) ->
      let t'_list =
        List.map (function t'' -> term_translate t'' rho) t_list in
      let p' = List.assoc p rho.pm in PredApp (p', t'_list)
  | And (phi', phi'') ->
      let phi'_new = formula_translate phi' rho in
      let phi''_new = formula_translate phi'' rho in
      And (phi'_new, phi''_new)
  | Or (phi', phi'') ->
      let phi'_new = formula_translate phi' rho in
      let phi''_new = formula_translate phi'' rho in
      Or (phi'_new, phi''_new)
  | Imply (phi', phi'') ->
      let phi'_new = formula_translate phi' rho in
      let phi''_new = formula_translate phi'' rho in
      Imply (phi'_new, phi''_new)
  | Nega phi' -> Nega (formula_translate phi' rho)
  | Forall ((id, s), phi') -> let s' = List.assoc s rho.sm in
      Forall ((id,s'), formula_translate phi' rho)
  | Exist ((id,s), phi') -> let s' = List.assoc s rho.sm in
      Exist ((id,s'), formula_translate phi' rho)
let sentence_translate = formula_translate
let is_proof lang axioms (phi, pf) = true
let proof_translate pf rho = pf

(*pretty printer*)
let print_prod_sorts sorts =
  let rec print_help ss =
    match ss with
    [] -> ()
    | hd :: tl -> print_string (" * " ^ (Name.name hd));
        print_help tl
  in
  match sorts with
    [] -> ()
  | hd :: tl -> print_string (Name.name hd); print_help tl
let print_lang lang =
  print_string "Language: "; print_newline();
  print_string("    ");print_string "Sorts:"; print_newline();
    let print_sort s =
      (print_string ("        " ^ (Name.name s));
      print_newline()) in
  List.iter print_sort lang.sorts;
  print_string("    ");print_string "Constants:"; print_newline();
    let print_const (name,s) =
      (print_string ("        " ^ (Name.name name) ^ "
        : " ^ (Name.name s));
      print_newline()) in
  List.iter print_const lang.consts;
  print_string("    ");print_string "Functions:"; print_newline();
    let print_func (name, sl, s) =
      (print_string ("        " ^ (Name.name name) ^ " : ");
      print_prod_sorts sl; print_string (" -> " ^ (Name.name s));
       print_newline())
    in
  List.iter print_func lang.funcs;
  print_string("    ");print_string "Predicates:"; print_newline();
    let print_pred (name, sl) =
      (print_string ("        " ^ (Name.name name) ^ " : ");
      print_prod_sorts sl; print_newline())
    in
  List.iter print_pred lang.preds
let print_mapping rho =
    print_string "[";
      let print_sm (s1, s2) = (print_string (Name.name s1);
        print_string " |-> "; print_string (Name.name s2);
        print_string ",") in
    List.iter print_sm rho.sm;
      let print_const (name,s) = (print_string (Name.name name);
        print_string ": "; print_string (Name.name s)) in
      let print_cm (c1, c2) = print_const c1; print_string " |-> ";
        print_const c2; print_string "," in
    List.iter print_cm rho.cm;
      let print_func (name,sl,s) = (print_string (Name.name name);
        print_string ": "; print_prod_sorts sl;
        print_string " -> "; print_string (Name.name s))in
      let print_fm (f1, f2) = print_func f1; print_string " |-> ";
        print_func f2; print_string "," in
    List.iter print_fm rho.fm;
      let print_pred (name,sl) = (print_string (Name.name name);
        print_string ": "; print_prod_sorts sl) in
      let print_pm (p1, p2) = print_pred p1; print_string " |-> ";
        print_pred p2; print_string "," in
    List.iter print_pm rho.pm;
    print_string "]"
  let rec print_term t = match t with
       Var(v_id, _) -> print_string (Ident.name v_id)
     | Const(c_name, _) -> print_string (Name.name c_name)
     | App (f, ts) -> let (f_name, _, _) = f in
         print_string (Name.name f_name); print_string " ";
         print_string "("; print_terms ts; print_string ")"
  and print_terms ts = match ts with
       [] -> print_string ""
     | t :: tail -> print_term t;
     if List.length tail = 0 then ()
     else print_string ", "; print_terms tail
  let print_formula phi =
    let open_paren prec op_prec =
      if prec > op_prec then print_string "(" in
  let close_paren prec op_prec =
    if prec > op_prec then print_string ")" in
  let rec print prec phii =  (* prec is the current precedence *)
     match phii with
      Eq (t1, t2) -> print_term t1; print_string " = ";
        print_term t2
    | PredApp (pred, t_list) -> let (p_name, _) = pred in
        print_string (Name.name p_name); print_string " ";
        print_string "("; print_terms t_list; print_string ")"
    | And (phi1, phi2) ->
        open_paren prec 1;
        print 1 phi1; print_string " && "; print 1 phi2;
        close_paren prec 1
    | Or (phi1, phi2) ->
        open_paren prec 2;
        print 2 phi1; print_string " || "; print 2 phi2;
        close_paren prec 2
    | Imply (phi1, phi2) ->
        open_paren prec 0;
        print 0 phi1; print_string " --> "; print 0 phi2;
        close_paren prec 0
    | Nega phi1 ->
        open_paren prec 3;
        print_string " ! "; print 3 phi1;
        close_paren prec 3
    | Forall (v, phi1) ->
        let (id,_) = v in
        print_string " forall "; print_string (Ident.name id);
          print_string ". "; print 0 phi1
    | Exist (v, phi1) ->
        let (id,_) = v in
        print_string " exist "; print_string (Ident.name id);
        print_string ". "; print 0 phi1
  in
  print 0 phi
let print_sentence = print_formula
```

```
  let print_proof pf = print_string "pseudo-proof"
end

module Fol_MeiCore = MeiCore(Fol) module Fol_MeiCore_Eval =

MeiCore_Eval(Fol)(MeiCore)(Env)(Ctx) module Fol_Mei =

Mei(Fol)(MeiCore)(Tran)
```

Bibliography

[1] *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, 1827, 2000.

[2] D. R. Aspinall, "Isabelle modules — A new theory mechanism for Isabelle," Master's thesis, University of Cambridge, 1991.

[3] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki, "CASL: the common algebraic specification language," *Theor. Comput. Sci.*, vol. 286, no. 2, pp. 153–196, 2002.

[4] S. Autexier, D. Hutter, H. Mantel, and A. Schairer, "Towards an evolutionary formal software-development using CASL," in *WADT'99* [1], pp. 73–88.

[5] A. Bailey, "Coercion synthesis in computer implementations of type-theoretic frameworks," in *TYPES '96: Selected papers from the International Workshop on Types for Proofs and Programs*, (London, UK), pp. 9–27, Springer-Verlag, 1998.

[6] C. Ballarin, "Locales and locale expressions in Isabelle/Isar.," in *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers* (S. Berardi, M. Coppo, and F. Damiani, eds.), vol. 3085 of *Lecture Notes in Computer Science*, pp. 34–50, Springer, 2004.

[7] M. Bidoit, D. Sannella, and A. Tarlecki, "Architectural specifications in CASL," in *AMAST '98: Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, (London, UK), pp. 341–357, Springer-Verlag, 1999.

[8] T. Borzyszkowski, “Higher-order logic and theorem proving for structured specifications,” in *WADT’99* [1], pp. 401–418.

[9] W. Bosma, J. J. Cannon, and C. Playoust, “The magma algebra system i: The user language,” *J. Symb. Comput.*, vol. 24, no. 3/4, pp. 235–265, 1997.

[10] N. Bruijn in *Symposium on Automatic Demonstration* (J. P. Seldin and J. R. Hindley, eds.), Lecture Notes in Mathematics, pp. 29–61, Springer-Verlag, 1970.

[11] N. Bruijn, “A survey of the project AUTOMATH,” in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (J. P. Seldin and J. R. Hindley, eds.), pp. 579–606, Academic Press, 1980. Reprinted in Selected Papers in Automath, ed. R.P. Nederpelt and H. Geuvers (North-Holland, 1994), 141-161.

[12] J. Cannon and C. Playoust, *Algebraic programming with Magma I: An introduction to the Magma langage.* Secaucus, NJ, USA: Springer-Verlag NewYork, Inc., 2006.

[13] J. Carette, W. Farmer, and J. Wajs, “Trustable communication between mathematics systems,” in *Calculemus 2003 (11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Rome,Italy, September 2003* (T. Hardin and R. Rioboo, eds.), pp. 58–68, 2003.

[14] J. Carette, “Gaussian elimination: a case study in efficient genericity with metaocaml,” *Sci. Comput. Program.*, vol. 62, no. 1, pp. 3–24, 2006.

[15] J. Carette and O. Kiselyov, “Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code,” 2005.

[16] G. Castagna, “Covariance and contravariance: Conflict without a cause,” *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 3, pp. 431–447, May 1995.

[17] J. Chrzaszcz, *Modules in Type Theory with Generative Definitions.* PhD thesis, Warsaw Univerity and University of Paris-Sud, Jan 2004.

[18] J. Chrzkszcz, “Modules in Coq are and will be correct,” in *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers* (S. Berardi, M. Coppo, and F. Damiani, eds.), vol. 3085 of *LNCS*, pp. 130–146, Springer, 2004.

[19] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude manual*, March 2004. Version 2.1. Available from http://maude.cs.uiuc.edu/.

[20] R. L. Constable, "Recent results in type theory and their relationship to automath," in *Thirty Five years of Automath, Applied Logic Series* (F. Kamareddine, ed.), pp. 1–11, Kluwer Academic Publishers, 2003.

[21] T. Coquand, R. Pollack, and M. Takeyama, "Modules as dependently typed records." citeseer.ist.psu.edu/coquand02modules.html.

[22] T. Coquand, R. Pollack, and M. Takeyama, "A Logical Framework with Dependently Typed Records," *Fundamenta Informaticae*, vol. 65, no. 1–2, pp. 112–134, 2005.

[23] J. Courant, "An applicative module calculus," in *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, (London, UK), pp. 622–636, Springer-Verlag, 1997.

[24] J. Courant, "A module calculus for pure type systems," in *TLCA '97: Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, (London, UK), pp. 112–128, Springer-Verlag, 1997.

[25] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, pp. 238–252, 1977.

[26] R. David and K. Nour, "A short proof of the strong normalization of classical natural deduction with disjunction," *J. Symbolic Logic*, vol. 68, pp. 1277–1288, 2003.

[27] A. J. T. Davie, *An introduction to functional programming systems using Haskell*. New York, NY, USA: Cambridge University Press, 1992.

[28] T. Dimitrakos and T. Maibaum, "On a generalized modularization theorem," *Information Processing Letters*, vol. 74, no. 1–2, pp. 65–71, Apr. 2000.

[29] K. Donnelly and H. Xi, "A formalization of strong normalization for simply-typed lambda-calculus and system F," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 5, pp. 109–125, 2007.

[30] D. Dreyer, K. Crary, and R. Harper, "A type system for higher-order modules," in *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 236–249, ACM Press, 2003.

[31] F. Durán and J. Meseguer, "Parameterized theories and views in Full Maude 2.0," *Electr. Notes Theor. Comput. Sci.*, vol. 36, 2000.

[32] F. Durán and J. Meseguer, "Structured theories and institutions," *Theor. Comput. Sci.*, vol. 309, no. 1, pp. 357–380, 2003.

[33] F. Durán and J. Meseguer, "Maude's module algebra," *Sci. Comp. Program.*, vol. 66, no. 2, 2007.

[34] W. M. Farmer, "Theory interpretation in simple type theory," in *Higher-Order Algebra, Logic, and Term Rewriting* (J. H. et al., ed.), vol. 816 of *Lecture Notes in Computer Science*, pp. 96–123, Springer-Verlag, 1994.

[35] W. M. Farmer, "STMM: A set theory for mechanized mathematics," *Journal of Automated Reasoning*, vol. 26, pp. 269–289, 2001.

[36] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "IMPS: An Interactive Mathematical Proof System," *Journal of Automated Reasoning*, vol. 11, pp. 213–248, 1993.

[37] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "The IMPS user's manual," Tech. Rep. M-93B138, The MITRE Corporation, 1993.

[38] W. M. Farmer, J. Guttman, and F. J. Thayer, "Little theories," in *Automated Deduction—CADE-11* (D. Kapur., ed.), vol. 607 of *Lecture Notes in Computer Science*, pp. 567–581, Springer-Verlag, 1992.

[39] W. M. Farmer and M. v. Mohrenschildt, "A microkernel for a mechanized mathematics system," tech. rep., McMaster University, 2002.

[40] W. M. Farmer and M. V. Mohrenschildt, "An overview of a formal framework for managing mathematics," *Annals of Mathematics and Artificial Intelligence*, vol. 38, no. 1-3, pp. 165–191, 2003.

[41] S. Fechter, "An object-oriented model for the certified computer algebra library." Paper presented at FMOODS 2002 PhD workshop, Mar. 2002.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[43] R. J. Gaylord, S. N. Kamin, and P. R. Wellin, *An introduction to programming with Mathematica.* Cambridge, UK: Cambridge University Press, 2004.

[44] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg, "A constructive algebraic hierarchy in Coq," *J. Symb. Comput.*, vol. 34, no. 4, pp. 271–286, 2002.

[45] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, April 1989.

[46] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," in *Applications of Algebraic Specification using OBJ* (J. Goguen, ed.), Cambridge, 1993.

[47] R. Harper and M. Lillibridge, "A type-theoretic approach to higher-order modules with sharing," in *21st symposium Principles of Programming Languages*, pp. 123–137, ACM Press, 1994.

[48] A. Haxthausen and et.al., *CASL — The common algebraic specification language — Summary*, March 2001. Version 1.0.1. Available from http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/.

[49] A. E. Haxthausen and F. Nickl, "Pushouts of order-sorted algebraic specifications," in *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, (London, UK), pp. 132–147, Springer-Verlag, 1996.

[50] J. J. Hickey, "Nuprl-light: An implementation framework for higher-order logics," in *Conference on Automated Deduction*, pp. 395–399, 1997.

[51] D. Hutter, "Management of change in structured verification.," in *Automated Software Engineering*, pp. 23–, 2000.

[52] D. Hutter and M. Kohlhase, "Managing structural information by higher-order colored unification," *Journal of Automated Reasoning*, vol. 25, no. 2, pp. 123–164, 2000.

[53] R. D. Jenks and R. Sutor, *AXIOM: the scientific computation system.* New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[54] R. M. Jiménez and F. Orejas, "An algebraic framework for higher-order modules," in *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, (London, UK), pp. 1778–1797, Springer-Verlag, 1999.

[55] S. Kahrs, D. Sannella, and A. Tarlecki, "The definition of extended ML: a gentle introduction," *Theor. Comput. Sci.*, vol. 173, no. 2, pp. 445–484, 1997.

[56] F. Kammuller, "Modular reasoning in isabelle." PhD thesis, University of Cambridge, 1999. http://citeseer.ist.psu.edu/article/kammuller99modular.html.

[57] X. Leroy, "Manifest types, modules, and separate compilation," in *21st symposium Principles of Programming Languages*, pp. 109–122, ACM Press, 1994.

[58] X. Leroy, "A modular module system," *Journal of Functional Programming*, vol. 10, no. 3, pp. 269–303, 2000.

[59] X. Leroy, *The Objective Caml system*, May 2007. Release 3.10. Available from http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[60] Z. Luo, "Coercive subtyping," *Journal of Logic and Computation*, vol. 9, no. 1, 1999.

[61] Z. Luo and R. Pollack, "The LEGO proof development system: A user's manual," Tech. Rep. ECS-LFCS-92-211, University of Edinburgh, May 1992.

[62] D. B. MacQueen, "Using dependent types to express modular structure," in *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (New York, NY, USA), pp. 277–286, ACM Press, 1986.

[63] The Coq Development Team, *The Coq Proof Assistant Reference Manual.* LogiCal Project, 2004. Version 8.0. Available from http://coq.inria.fr/doc/main.html.

[64] The FoC Development Team, *FoC Reference Manual*, July 2003. Version 0.0. Available from http://www-spi.lip6.fr/ foc.

[65] K. Meinke and J. V. Tucker, "Universal algebra," in *Handbook of logic in computer science (vol. 1): background: mathematical structures*, pp. 189–368, New York, NY, USA: Oxford University Press, Inc., 1992.

[66] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML.* Cambridge, MA, USA: MIT Press, 1997.

[67] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, and P. DeMarco, *Maple 10 Introductory Programming Guide.* Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario, Canada, 2005. 388 pages.

[68] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco, *Maple 10 Advanced Programming Guide.* Waterloo Maple Inc., 2005.

[69] T. Mossakowski, "Colimits of order-sorted specifications," in *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, (London, UK), pp. 316–332, Springer-Verlag, 1997.

[70] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[71] F. Orejas, "Structuring and modularity," in *Algebraic Foundations of System Specification* (E. Astesiano, E. Kreowski, and B. Brückner, eds.), pp. 159–200, Springer-Verlag, 1999.

[72] S. Owre, J. M. Rushby, , and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (D. Kapur, ed.), vol. 607 of *Lecture Notes in Artificial Intelligence*, (Saratoga, NY), pp. 748–752, Springer-Verlag, jun 1992.

[73] S. Owre and N. Shankar, "Theory interpretations in PVS," Tech. Rep. SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[74] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.

[75] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15(12), pp. 1053–1058, 1972.

[76] L. C. Paulson, “Theories as ML structures, signatures, and functors.” University of Cambridge, January 1991.

[77] L. C. Paulson, “Organizing numerical theories using axiomatic type classes.,” *J. Autom. Reasoning*, vol. 33, no. 1, pp. 29–49, 2004.

[78] F. Pfenning and C. Schuermann, “Twelf user’s guide,” Tech. Rep. CMU-CS-98-173, Carnegie Mellon University, September 1998.

[79] B. C. Pierce, “Bounded quantification is undecidable,” *Information and Computation*, vol. 112, no. 1, July 1994.

[80] B. C. Pierce, *Types and Programming Languages.* The MIT Press, 2002.

[81] E. Poll and S. Thompson, “The Type System of Aldor,” Tech. Rep. 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK, July 1999.

[82] N. Ramsey, “ML module mania: A type-safe, separately compiled, extensible interpreter.,” *Electr. Notes Theor. Comput. Sci.*, vol. 148, no. 2, pp. 181–209, 2006.

[83] N. Ramsey, K. Fisher, and P. Govereau, “An expressive language of signatures,” in *ICFP ’05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, (New York, NY, USA), pp. 27–40, ACM Press, 2005.

[84] C. V. Russo, “Types for modules,” 1998. PhD thesis, Univ. of Edinburgh. http://citeseer.ist.psu.edu/russo98types.html.

[85] C. V. Russo, “Non-dependent types for standard ML modules,” in *Principles and Practice of Declarative Programming*, pp. 80–97, 1999.

[86] A. Saibi, “Typing algorithm in type theory with inheritance.,” in *POPL*, pp. 292–301, 1997.

[87] D. Sannella and A. Tarlecki, "A kernel specification formalism with higher — order parameterisation," in *Recent Trends in Data Type Specification. 7th Workshop on Specification of Abstract Data Types*, pp. 274–296, Springer LNCS 534, 1990.

[88] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.

[89] M. Shields and S. P. Jones, "First class modules for Haskell," in *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*, pp. 28–40, Jan. 2002.

[90] Y. V. Srinivas and R. Jullig, "Specware: Formal support for composing software," in *Mathematics of Program Construction*, pp. 399–422, 1995.

[91] M. Sulzmann, M. Odersky, and M. Wehr, "Type inference with constrained types," in *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

[92] P. A. S. Veloso and T. S. E. Maibaum, "On the modularization theorem for logical specifications.," *Inf. Process. Lett.*, vol. 53, no. 5, pp. 287–293, 1995.

[93] P. Wadler, "Views: a way for pattern matching to cohabit with data abstraction," in *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (New York, NY, USA), pp. 307–313, ACM Press, 1987.

[94] S. Watt, *Aldor User Guide*. Aldor.org, 2nd ed., 2002.

[95] J. B. Wells, "Typability and type checking in system f are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, no. 1-3, pp. 111–156, 1999.

[96] M. Wenzel, "Using axiomatic type classes in Isabelle," 2004. Technische Universität München.

[97] M. Wirsing, "Structured algebraic specifications: A kernel language," *Theoretical Computer Science*, vol. 42, pp. 123–249, 1986.

[98] J. Xu, "Mei — A module system for mechanized mathematics systems," in *Programming Languages for Mechanized Mathematics Workshop*, (Hagenberg, Austria).

[99] J. Xu, "Mei — A module system for mechanized mathematics systems," Tech. Rep. CAS-07-01-WF, McMaster University, Hamilton, Canada, 2007.

[100] J. Zucker, "Formalization of classical mathematics in Automath," in *Colloque International de Logique*, pp. 135–145, Paris: CNRS, 1977. Reprinted in Selected Papers in Automath, ed. R.P. Nederpelt and H. Geuvers (North-Holland, 1994), 127-139.