UNIVERSIDAD DE MÁLAGA
Escuela Técnica Superior de Ingeniería Informática

A Reflective Module Algebra with Applications to the Maude Language

Francisco Durán

A Doctoral Dissertation Directed by
Dr. José Meseguer

June 3, 1999

A Eva y a mi madre

D. José Meseguer Guaita, Investigador Principal en el Computer Science Laboratory del SRI International en Menlo Park, California (U.S.A.), y D. Ernesto Pimentel Sánchez, Profesor Titular de Universidad del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga

Certifican

Que D. Francisco J. Durán Muñoz, Licenciado en Informática por la Universidad de Málaga, ha realizado en el Computer Science Laboratory del SRI International, bajo la dirección de D. José Meseguer Guaita, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Un Álgebra de Módulos Reflexiva con Aplicaciones al Lenguaje Maude

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, 25 de Junio de 1999

Fdo.: José Meseguer Guaita
Investigador Principal
Computer Science Laboratory
SRI International
Menlo Park, California (U.S.A.)

Málaga, 25 de Junio de 1999

Fdo.: Ernesto Pimentel Sánchez
Profesor Titular de Universidad
Dpto. de Lenguajes y
Ciencias de la Computación
Universidad de Málaga

Contents

Preface

Introduction

It has long been recognized that large specifications are unmanageable unless they are built in a structured fashion from smaller specifications using specification-building operations. Modularity and module composition are central notions for specification languages and declarative programming languages. Although Parnas [92, 93] is the author of the possibly earliest work on software modules, Burstall and Goguen were the first to study the semantics of modular specifications and their composition operations in their language Clear [14]. They proposed the idea of "putting theories together" by composing them through operations having a clean and logic-independent categorical semantics [15]. Continuing in this line of work, Burstall and Goguen captured the minimal requirements that a logic must meet to be a reasonable specification framework and introduced the notion of *institution* [52, 53].

Categorical techniques have allowed since then the study of specification-building operations with independence from any specific formalism by different authors, giving rise to a large body of research, e.g., [98, 38, 2, 43, 33].

Algebraic specification is now a mature field of Computer Science because of its mathematical foundations. After Clear, the theory of algebraic specification has been implemented in many computing systems, such as OBJ [58, 47, 60], ACT ONE [44], ASL [99, 111], ASF [3], PLUSS [7], LPG [6], Larch [61], CASL [31], etc., and has become an important technique in software engineering methodologies. See [50, 8] for a comprehensive list of references.

The language OBJ [58, 47, 60] embodies many of the categorical module composition operations of Clear and has played a major rôle in the development of algebraic specification, and, more generally, of declarative programming. OBJ has been extended in different ways, namely towards object-oriented programming in FOOPS [55, 102], theorem proving in 2OBJ [57], and logic programming in Eqlog [54, 37]. Other languages that have appeared more recently, like Maude [21, 23, 27] and CafeOBJ [16], can also be included in the OBJ family of languages.

In all of these languages, the module operations are a central issue. Module operations belong to the *metalevel* of the logic in question, that is, they can be seen as functions taking modules and returning other modules as results. This is in fact the idea behind the module expressions in OBJ3 [60], in the module algebra by Bergstra, Heering, and Klint [4], and in subsequent research. In general, any specification language determines a set of constructs which can be used to build specifications. In them, the user has a fixed repertoire of operations available, and cannot express them, extend them, or reason about them *within the logic*. This means that although module operations, and in particular parameterization, provide certain "higher-order" capabilities [51], such capabilities remain limited in their applicability and provide only a fixed repertoire of higher-order operations.

This situation can be changed quite dramatically by introducing the notion of a *reflective logic* [28, 19]. Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic that can be faithfully interpreted in itself, giving access to its metatheory. In particular, rewriting logic [76, 83, 65] is reflective [29, 19].

A reflective logic like rewriting logic opens up many completely new possibilities for module composition. In a reflective logic, module operations are definable within the logic itself, providing higher-order capabilities by defining operations that transform, combine, and/or manipulate modules. Based on these ideas, it is possible to extend the *module algebra* of the system, and it is also possible to create within the logic a formal environment for it with tools for formal analysis, transformation, and theorem proving. Moreover, since the module operations are defined in the logic, this opens up the possibility of metareasoning about such operations within the logic. Furthermore, if the logic is efficiently executable then the formal specification of module composition operations can itself be used as a reasonable implementation.

If, in addition, the reflective logic has good properties as a logical framework [72], then a logic $\mathcal{L}$ can be represented in such a framework logic $\mathcal{F}$ by giving any *representation map* of the form

$$\Phi : \mathcal{L} \longrightarrow \mathcal{F}.$$

We can then define a module algebra for the logic $\mathcal{L}$ in the framework logic $\mathcal{F}$ by appropriately defining operations on modules of $\mathcal{L}$. If we have an efficient implementation for the reflective framework logic $\mathcal{F}$, as it is the case for rewriting logic using a system such as Maude, it becomes then possible not only to implement the logic $\mathcal{L}$ using the representation map Φ, but also to endow such an implementation of $\mathcal{L}$ with a module algebra for it defined in $\mathcal{F}$. Furthermore, we can even think about a *generic module algebra* defined in the framework logic $\mathcal{F}$ that can provide for any given logic $\mathcal{L}$ the corresponding module algebra as an instantiation.

Maude [21] is a reflective language based on rewriting logic [76] that essentially contains the OBJ3 language as an equational sublanguage. Rewriting logic extends equational logic and has very good properties as a logical framework, in which many other logics and many semantic formalisms can be naturally represented [71, 83]. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective, making the metatheory of rewriting logic accessible to the user in a clear and principled way.

Summary of the Thesis

Before defining a generic module algebra of very wide applicability we need to define logic-independent notions of module operations. As already mentioned, there is already a large body of work in this direction, originating in the work on Clear and institutions, and realized in languages such as those in the Clear/OBJ tradition.

We are particularly interested in a module algebra framework in which theories with *loose* semantics and modules with *initial* semantics coexist, giving rise to a uniform setting. We like to view a module or theory as a more general form of a structured theory, possibly with initiality constraints [96, 15, 100, 53, 45] on parts of its structure. In this setting, we want to be able to support parameterized theories and also parameterized views, allowing in this way the structuring of these entities as well. This will improve not only the flexibility of the module algebra in the definition of theories and views, but also the reusability of the different components. Reusability of views is something to which not much attention has been paid so far. We think that view reusability can be greatly increased by applying to them the same ideas developed for theories. We are therefore interested not only in the definition of parameterized or structured views, but in views going from structured theories to structured theories.

Traditionally, the semantics of module operations is given in terms of *flattened* modules, assuming that each module operation results in an equivalent flat module. We are interested in a more general notion of module operation. We propose a module algebra of module operations taking structured theories as arguments and giving structured theories as results. Some initial results in this direction are given in Chapter 2, introducing the notion of structured theories in a categorical way and giving some useful results for them.

As a first step towards a generic module algebra in a framework logic such as rewriting logic, we have defined an extensible module algebra for rewriting logic itself, and have realized that design using the reflective capabilities of the rewriting logic language Maude. We describe in Chapter 3 the language design of Full Maude, which extends Maude[1] with object-oriented notation, and with a number of module operations, namely module hierarchies, module renaming, parameterization of modules, and transformation of object-oriented modules into system modules.

As we shall see in the description of Full Maude, we are not dealing only with a set of module operations. We are dealing with a complete language, including an execution environment for such a language. The fact that Full Maude has Core Maude as a "sublanguage" may simplify the representation map between them, but the methodology can be used for many other languages and logics. Once we have an appropriate representation map, we can then define an execution environment for any other logic or language in the same way.

We explain in Chapter 4 the executable formal specification of Full Maude, which is in fact its implementation in the Maude 1.00 system[2]. We do not give the full specification. Instead, we focus on the more interesting parts but still allowing the reader the possibility of understanding the overall picture. For example, we explain in some detail how the transformation from object-oriented modules to system modules, which was originally proposed by Meseguer in [77], is integrated with the overall module algebra; we explain also in some detail the renaming of modules, and the instantiation of parameterized modules.

We should emphasize the extensibility of the module algebra with which Maude has been endowed in this way. We present the specification of Full Maude in sufficient detail so that the different extensions of Full Maude become easily understandable. Note that by 'extensions' we not only mean those adding new module transformation operations in the style of the transformation from object-oriented modules to system modules, but also new module combining and/or manipulating operations. We illustrate some of the possibilities for extensions of this kind with several examples:

- a module expression for union of modules, that is, the classical sum or union `+` operation taking two modules and returning the result of uniting them;

- as a simple, but interesting, example showing the new possibilities for defining module expressions, we present a module expression `TUPLE(`N`)` such that, given a nonzero natural number N, it generates a parameterized functional module specifying tuples of the corresponding size.

But there are many more possibilities. For example, one important area for defining new module operations is viewing *formal tools* as appropriate module operations that analyze and transform specifications. We present in Chapter 5

[1] Full Maude is part of Maude 1.00. Core Maude is the sublanguage supported by the kernel on which Full Maude was developed. Core Maude and Full Maude can be seen as two facets of the same language. In the future they will be fully integrated.

[2] Maude 1.00 is available free of charge from its web site at `maude.csl.sri.com`. The complete specification of Full Maude is available as part of the distribution package.

a substantial example of this kind, namely, a Church-Rosser checker that analyzes order-sorted equational specifications and generates a collection of proof obligations that must be satisfied to guarantee the Church-Rosser property.

Acknowledgments

José Meseguer deserves my warmest gratitude as a teacher, as a thesis supervisor, and specially as a colleague and friend. I thank him for introducing me to theoretical computer science in a practical way, and for his continuous support and help. His presence can be felt everywhere in this thesis. Indeed, this is the result of joint work, and as such should be recognized. We have been coauthors of several papers used in the preparation of this thesis, and we will be coauthors of some other papers using material prepared for the thesis that will be published in the near future.

Special thanks go to my fellow members of the Maude team Manuel Clavel, Steven Eker, Pat Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada for their great job designing and building the Maude language and system, and for many interesting discussions and ideas.

My warmest thanks to Narciso Martí-Oliet, for his continuous support and encouragement, for many interesting discussions, and for his patience and perseverance on my early attempts to understand category theory. Thanks for so many good and sharp comments and suggestions to many different manuscripts. Thanks also for your comments on this thesis.

Thanks to Peter Mosses for many interesting discussions on very different topics and for comments on this thesis. I have benefited very much from his ideas and suggestions, and from his experience with Action Semantics and with CASL. Thanks to Kokichi Futatsugi, Tom Maibaum, and Carolyn Talcott for very interesting discussions. Thanks to Kokichi Futatsugi, Ataru Nakagawa, Toshimi Sawada, Masaki Ishiguro, and Masayuki Fujita for their efforts to succesfully integrate the Church-Rosser checker presented in Chapter 5 within the Cafe environment. Thanks to Hélène Kirchner and the other members of the ELAN group for interesting discussions and exchanges of ideas.

Thanks to Christiano Braga, Manuel Clavel, Grit Denker, Dinesh, Alexander Knapp, Peter Ölveczky, Hassen Saïdi, Pierangela Samarati, Marco Schorlemmer, Mark-Oliver Stehr, and many others, for sharing their experience as international fellows or visitors at SRI's Computer Science Laboratory, and for many interesting discussions. Thanks to Jeff van Baalen, Christiano Braga, Manuel Clavel, Grit Denker, Alexander Knapp, Uri Nodelman, Peter Ölveczky, Dilia Rodríguez, and Mark-Oliver Stehr for being the early users of Maude and for suffering it, and/or for developing very interesting applications in Maude.

Thanks to César Muñoz for his comments to a draft of Chapter 5, and to Christiano Braga for his comments to Chapter 4.

Thanks to the people at Málaga, specially to Ernesto Pimentel and José María Troya, for believing in me and for being always willing to help me. Thanks also to Fernando Orejas for his help as tutor and for introducing me to José Meseguer.

Since it is so hard to make a list with all relatives, friends, and persons with whom one shares his everyday experiences and fears, and with which one is in debt, I have tried to avoid any personal references, but I have to include here Eva. She has not only been my wife in the distance for these three years, she has also been my friend, my support, my secretary, my interface with the rest of the world, and many other things, and therefore, I do not only want to

acknowledge how important she has been for this thesis, I want to acknowledge how important she is in my life. I owe her everything.

Partial support from the following research contracts and grants is gratefully acknowledged: support from a grant of Ministerio de Educación y Ciencia of the Spanish Goverment, support from DARPA through Rome Laboratories Contract F30602-97-C-0312, from DARPA and NASA through Contract NAS2-98073, from Office of Naval Research Contract N00014-96-C-0114, and from National Science Foundation Grant CCR-9633363.

Chapter 1

Institutions, Rewriting Logic, and the Maude Language

This chapter summarizes background material on the theory of institutions, rewriting logic, and the Maude language that will be needed in subsequent chapters. Most of the material has been adapted from already published sources. Specifically, Section 1.1 combines material from Goguen and Burstall's [53], Meseguer's [75], and Diaconescu, Goguen, and Stefaneas's [38]; Section 1.2 borrows material from Meseguer's [77] and from [21]; and most of Section 1.3 was taken from [21].

1.1 The Theory of Institutions

A general axiomatic theory of logics should adequately cover all the key ingredients of a logic. These include: a *signature*, a notion of *entailment* of a sentence from a set of axioms, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. A flexible axiomatic notion of a *proof calculus*, in which proofs of entailments, not just the entailments themselves, are first class citizens should also be included. The theory of *general logics* [75, 73, 18] is a study of these different ingredients of a logic and their interrelations, and combines them to obtain also a general notion of declarative language based on a given logic.

For our present purposes it will be the notions of signature, sentence for a signature, model, and satisfaction of a sentence by a model that play a crucial role. We use for this purpose the axiomatization of a logic's model theory proposed by Goguen and Burstall [52, 53], namely the notion of *institution*. In this section we introduce some of the basic concepts and results on institutions [53, 38, 75].

The theory of institutions allows us to discuss the relationship between theories and models without committing ourselves to a particular logical system. A key ingredient in a logical system is that of signature, and the possibility of translating sentences and models across signature morphisms. An institution consists of a category of signatures such that, associated with each signature, there are sentences, models, and a relationship of satisfaction which is stable under translation by signature morphisms.

Definition 1 [52] *An* institution $\mathcal{I}$ *is a 4-tuple* $(\mathbf{Sign}_{\mathcal{I}}, \mathrm{sen}_{\mathcal{I}}, \mathrm{Mod}_{\mathcal{I}}, \models)$ *such that*

- $\mathbf{Sign}_{\mathcal{I}}$ *is a category whose objects are called* signatures*,*
- $\mathrm{sen}_{\mathcal{I}} \colon \mathbf{Sign}_{\mathcal{I}} \longrightarrow \mathbf{Set}$ *is a functor associating to each signature* Σ *a set of* Σ-sentences*,*
- $\mathrm{Mod}_{\mathcal{I}} : \mathbf{Sign}_{\mathcal{I}} \longrightarrow \mathbf{Cat}^{\mathrm{op}}$ *is a functor that gives for each signature* Σ *a category whose objects are called* Σ-models*, and*
- $\models$ *is a function associating to each* $\Sigma \in |\mathbf{Sign}_{\mathcal{I}}|$ *a binary relation* $\models_\Sigma \subseteq |\mathrm{Mod}_{\mathcal{I}}(\Sigma)| \times \mathrm{sen}_{\mathcal{I}}(\Sigma)$ *called* satisfaction *in such a way that the following property holds for any* $M' \in |\mathrm{Mod}_{\mathcal{I}}(\Sigma')|$, $H : \Sigma \to \Sigma'$, $\varphi \in \mathrm{sen}_{\mathcal{I}}(\Sigma)$*:*

$$M' \models_{\Sigma'} \mathrm{sen}_{\mathcal{I}}(H)(\varphi) \quad\Longleftrightarrow\quad \mathrm{Mod}_{\mathcal{I}}(H)(M') \models_\Sigma \varphi.$$

The satisfaction condition just requires that, for any syntax translation between two signatures, a model of the second signature satisfies a translated sentence if and only if the translation of this model satisfies the original sentence. Note that $\mathrm{Mod}_{\mathcal{I}}$ is a contravariant functor, that is, translations of models go backwards.

Given a signature Σ, a presentation of a theory is given by a set Γ of Σ-sentences. We can therefore denote a theory presentation as a pair (Σ, Γ).

Given a presentation (Σ, Γ), we define the category $\mathrm{Mod}_{\mathcal{I}}(\Sigma, \Gamma)$ as the full subcategory of $\mathrm{Mod}_{\mathcal{I}}(\Sigma)$ determined by those models $M \in |\mathrm{Mod}_{\mathcal{I}}(\Sigma)|$ that satisfy all the sentences in Γ, i.e., $M \models_{\Sigma} \varphi$ for each $\varphi \in \Gamma$.

We can extend the satisfaction relation to sets of sentences as follows.

$$M \models_{\Sigma} \Gamma \text{ iff } M \models_{\Sigma} \varphi \text{ for each } \varphi \in \Gamma.$$

Then, the relation between sets of sentences and sentences given by

$$\Gamma \models_{\Sigma} \varphi \text{ iff } M \models_{\Sigma} \varphi \text{ for each } M \in |\mathrm{Mod}_{\mathcal{I}}(\Sigma, \Gamma)|$$

allows us to define any institution as an entailment system in the sense of [75]. For any signature Σ, the *closure* of a set Γ of Σ-sentences is $\Gamma^{\bullet} = \{\varphi \mid \Gamma \models_{\Sigma} \varphi\}$. The Σ-theory presented by (Σ, Γ) is then given by $(\Sigma, \Gamma^{\bullet})$.

Given presentations of theories (Σ, Γ) and (Σ', Γ'), a theory morphism $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\varphi \in \Gamma$ then $\mathrm{sen}_{\mathcal{I}}(H)(\varphi) \in \Gamma'^{\bullet}$, that is, for all $\varphi \in \Gamma$, $\Gamma' \models_{\Sigma'} \mathrm{sen}_{\mathcal{I}}(H)(\varphi)$. A theory morphism $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is called *axiom-preserving* if it satisfies the condition $\mathrm{sen}_{\mathcal{I}}(H)(\Gamma) \subseteq \Gamma'$.

Definition 2 *Given an institution $\mathcal{I}$, its category $\mathbf{Th}_{\mathcal{I}}$ of* theories *has as objects presentations of theories (Σ, Γ) and as arrows theory morphisms*[1].

For any institution $\mathcal{I}$, the model functor $\mathrm{Mod}_{\mathcal{I}}$ extends to a functor $\mathrm{Mod}_{\mathcal{I}}\colon \mathbf{Th}_{\mathcal{I}} \to \mathbf{Cat}^{op}$, by mapping a theory (Σ, Γ) to the full subcategory $\mathrm{Mod}_{\mathcal{I}}(\Sigma, \Gamma)$ of $\mathrm{Mod}_{\mathcal{I}}(\Sigma)$.

We denote by $\mathrm{sign}_{\mathcal{I}} : \mathbf{Th}_{\mathcal{I}} \to \mathbf{Sign}_{\mathcal{I}}$ the forgetful functor sending each theory to its underlying signature.

Liberality is an important property expressing the possibility of free constructions generalizing the principle of "initial algebra semantics."

Definition 3 [53] *An institution is* liberal *if all the forgetful functors* $\mathrm{Mod}_{\mathcal{I}}(H)\colon \mathrm{Mod}_{\mathcal{I}}(\Sigma', \Gamma') \to \mathrm{Mod}_{\mathcal{I}}(\Sigma, \Gamma)$ *induced by theory morphisms $H\colon (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ have left adjoints.*

Another property expressing the possibility of "putting theories together" by colimits is the exactness of an institution. This is an important model-theoretic property of many logical systems, stating that colimits are preserved by the model functor.

Definition 4 [75, 38] *An institution $\mathcal{I}$ is* exact *if its category of signatures $\mathbf{Sign}_{\mathcal{I}}$ is cocomplete and the model functor* $\mathrm{Mod}_{\mathcal{I}}\colon \mathbf{Sign}_{\mathcal{I}} \to \mathbf{Cat}^{\mathrm{op}}$ *preserves colimits, and is* semiexact *if $\mathbf{Sign}_{\mathcal{I}}$ has pushouts and $\mathrm{Mod}_{\mathcal{I}}$ preserves pushouts.*

1.2 Rewriting Logic

Rewriting logic [76, 83, 65] is parameterized by its underlying equational logic, which can be unsorted, many-sorted, order-sorted, or the recently developed

[1] Note that in the above definition the objects of $\mathbf{Th}_{\mathcal{I}}$ are *presentations* of theories. We follow here the terminology of Meseguer's general logics [75], instead of Goguen and Burstall's original definition [53]. In what follows, when we talk about a theory (Σ, Γ) we shall mean a theory presentation.

membership equational logic [78, 11, 12, 82]. To simplify the exposition, we treat here the *unsorted* case, following the presentation in [77].

A set Σ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-algebra is then a set A together with an assignment of a function $f_A : A^n \longrightarrow A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by T_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(X)$ the Σ-algebra of Σ-terms with variables in a set X. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E; in the same way, $T_{\Sigma,E}(X)$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Let $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given terms $t, u_1, \ldots, u_n$ in $T_\Sigma(\{x_1, \ldots, x_n\})$, $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation, we denote a sequence of objects $a_1, \ldots, a_n$ by $\vec{a}$; with this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\vec{u}/\vec{x})$.

Rewriting will operate on equivalence classes of terms modulo E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing an associativity axiom, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [34] and can be used to implement rewriting modulo many equational theories of interest.

Given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are Σ-terms possibly involving some variables. A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of labeled sequents $r : [t] \longrightarrow [t']$ called *rewrite rules*[2]. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\vec{x})] \longrightarrow [t'(\vec{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent) $\mathcal{R}$-rewrite*, and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction*:

1. **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}.$$

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

[2]We follow the presentation in [77], where to simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas presented here have been extended to conditional rules in [76] with very general rules of the form

$$r : [t] \longrightarrow [t'] \text{ if } [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

This increases considerably the expressive power of rewrite theories.

3. **Replacement**. For each rule $r : [t(\vec{x})] \longrightarrow [t'(\vec{x})]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\vec{w}/\vec{x})] \longrightarrow [t'(\vec{w'}/\vec{x})]}.$$

4. **Transitivity**.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

1.2.1 The Meaning of Rewriting Logic

We understand a logic as a method of correct reasoning about some class of entities. For equational logic, the entities in question are sets, functions between them, and the relation of identity between elements. For rewriting logic, the entities in question are *concurrent systems* having *states*, and evolving by means of *transitions*. The *signature* of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The *rewrite rules* in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Clearly, concurrent systems should be the *models* giving a semantic interpretation to rewriting logic, in the same way that algebras are the models giving a semantic interpretation to equational logic. A precise account of the model theory of rewriting logic consistent with the above system-oriented interpretation, is developed in full detail for the general conditional case in [76].

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system.

The computational and the logical viewpoints under which rewriting logic can be interpreted can be summarized in the following diagram of correspondences:

State	$\leftrightarrow$	*Term*	$\leftrightarrow$	*Proposition*
Transition	$\leftrightarrow$	*Rewriting*	$\leftrightarrow$	*Deduction*
Distributed Structure	$\leftrightarrow$	*Algebraic Structure*	$\leftrightarrow$	*Propositional Structure*

The last row of equivalences is actually quite important. Roughly speaking, it expresses the fact that a state can be transformed in a concurrent way only if it is nonatomic, that is, if it is *composed* out of smaller state components that can be changed independently. In rewriting logic this composition of a concurrent state is formalized by the *operations* of the signature Σ of the rewrite theory $\mathcal{R}$ that axiomatizes the system. From the logical point of view such operations can naturally be regarded as user-definable *propositional connectives* stating the particular structure that a given state has. A subtle additional point about the last row of equivalences is that the algebraic structure of a system also involves *equations*. Such equations describe the system's global state as a *concurrent data structure*; they can have a dramatic impact on the amount of concurrency available in a system.

Note that it follows from this discussion that rewriting logic is primarily a logic *of* change—in which the deduction directly corresponds to the change—as opposed to a logic to talk *about* change in a more indirect and global manner such as the different variants of modal and temporal logic.

1.2.2 Reflection in Rewriting Logic

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself.

Rewriting logic is reflective [29, 19] in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle\overline{\mathcal{R}}, \overline{t}\rangle$, in such a way that we have the following equivalence

$$(\dagger)\ \ \mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \longrightarrow \langle\overline{\mathcal{R}}, \overline{t'}\rangle.$$

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{R}}, \overline{t}\rangle \rightarrow \langle\overline{\mathcal{R}}, \overline{t'}\rangle \Leftrightarrow \mathcal{U} \vdash \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t}\rangle}\rangle \rightarrow \langle\overline{\mathcal{U}}, \overline{\langle\overline{\mathcal{R}}, \overline{t'}\rangle}\rangle \ldots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible—so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

To achieve a systematic descent into equivalent rewriting computations at lower levels, the key idea is to exploit the equivalence (†). A detailed proof of this equivalence has been given for the case of unconditional and unsorted theories [19]. The extension to the case of interest for Maude—namely to conditional rewrite theories with membership equational logic [82, 12] as the underlying equational logic—although nontrivial, is essentially unproblematic. We therefore assume a universal theory $\mathcal{U}$ for this more general class of finitely presented rewrite theories.

1.3 The Maude Language

Maude [21, 23, 27] is a high-level reflective language and high-performance system supporting both membership equational logic [82, 11] and rewriting logic [76, 83, 65] specification and programming for a wide range of applications. Maude has been influenced in important ways by the OBJ3 language [60], which can be regarded as an equational logic sublanguage.

As mentioned in Section 1.2, rewriting logic is a logic of concurrent change that can naturally deal with state and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency [76, 80, 83, 13]. Since the computational and logical interpretations of rewriting logic are like two sides of the same coin, the same reasons making it a good semantic framework at the computational level make it also a good *logical framework* at the logical level, that is, a *metalogic* in which many other logics can be naturally represented and implemented [74, 71, 67, 68, 9, 97, 17, 30, 19, 24].

Moreover, object-oriented design languages, architectural description languages, and languages for distributed components also have a natural semantics in rewriting logic [112, 86, 104, 88, 89]. Consequently, some of the most interesting applications of Maude are *metalanguage* applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

A key distinguishing feature of Maude from the other languages based on rewriting logic, namely ELAN [67, 10] and CafeOBJ [48, 36], is its systematic and efficient use of *reflection* [20, 22]—exploiting the fact that rewriting logic is reflective [29, 19]—a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

The largest Maude application developed so far is Full Maude (see Chapters 3 and 4). Two other substantial applications are an inductive theorem prover [24] and a Church-Rosser checker (see Chapter 5) for equational theories, that are part of a formal environment for Maude and for the CafeOBJ language. In addition, several language interpreters and strategy languages, a supercompiler, several object-oriented specifications—including cryptographic protocols and network applications—and a variety of executable translations mapping logics, architectural description languages, and models of computation into the rewriting logic reflective framework have been developed by different authors (see references in [79, 83, 65]).

In Maude there are two kinds of *modules*, namely functional and system modules. We explain the main features of these two kinds of modules and their semantics in what follows. See [21] for full details of the Maude language.

1.3.1 Membership Equational Logic and Functional Modules

Functional modules define data types and functions on them by means of equational theories whose equations are Church-Rosser and terminating. A mathematical model of the data and the functions is provided by the *initial algebra* defined by the theory, whose elements consist of equivalence classes of ground terms modulo the axioms. Evaluation of any expression to its reduced form using the equations as rewrite rules assigns to each equivalence class a unique canonical representative. Therefore, in a more concrete way we can equivalently think of the initial algebra as consisting of those canonical representatives, that is, of the values to which the functional expressions evaluate by algebraic simplification using the equations.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called *membership equational logic* [82, 12]. In addition to supporting sorts, subsort relations, and overloading of function symbols, functional modules also support *membership axioms*, a generalization of sort constraints [84] in which a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and of unconditional membership tests is satisfied. Such membership axioms can be used to define partial functions, that become defined when their arguments satisfy certain equational and membership conditions.

We can illustrate these ideas, as well as Maude's support for mixfix user-definable syntax, with a module `PATH` that forms paths over a graph. Consider the graph in Figure 1.1. This graph describes an automaton whose *states* are the nodes of the graph, and whose *transitions* are the labeled edges. A *behavior*

$$\begin{tikzcd} n_3 \arrow[r, "c"] & n_4 \arrow[d, "d"] & \\ n_1 \arrow[u, "b"] \arrow[r, shift left, "a"] & n_2 \arrow[l, shift left, "f"] \arrow[r, "e"] & n_5 \end{tikzcd}$$

Figure 1.1: An Automaton.

of the automaton is a *path* in the graph, that is, a concatenation of transitions such that the target state of one transition becomes the source state of the next transition. Of course, not all random concatenations of edges are legal paths, that is, not all strings of edges are behaviors of the automaton. The `PATH` module below axiomatizes the automaton and characterizes in a computable way its paths by means of a path concatenation operation, a length function, and source and target functions, together with appropriate axioms in membership equational logic.

```
fmod PATH is
  protecting MACHINE-INT .

  sorts Edge Path Path? Node .
  subsorts Edge < Path < Path? .

  ops n1 n2 n3 n4 n5 : -> Node .
  ops a b c d e f : -> Edge .
  op _;_ : Path? Path? -> Path? [assoc] .
  ops source target : Path -> Node .
  op length : Path -> MachineInt .

  var E : Edge .
  var P : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(E ; P) = source(E) if E ; P : Path .
  ceq target(P ; E) = target(E) if P ; E : Path .
  eq length(E) = 1 .
  ceq length(E ; P) = 1 + length(P) if E ; P : Path .

  eq source(a) = n1 .
  eq target(a) = n2 .
  eq source(b) = n1 .
  eq target(b) = n3 .
  eq source(c) = n3 .
  eq target(c) = n4 .
  eq source(d) = n4 .
  eq target(d) = n2 .
  eq source(e) = n2 .
  eq target(e) = n5 .
  eq source(f) = n2 .
  eq target(f) = n1 .
endfm
```

The Semantics of Functional Modules

Maude is a declarative language based on rewriting logic. But rewriting logic has its underlying equational logic as a parameter. There are, for example, unsorted, many-sorted, and order-sorted versions of rewriting logic, each containing the previous version as a special case. As mentioned above, the underlying equational logic chosen for Maude is *membership equational logic* [82, 12].

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, (K, Σ) a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a K-kinded set of *sorts*.

An Ω-*algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ \ t = t' \ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

$$(\forall X)\ \ t : s \ \ \mathit{if}\ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, etc. [82].

In Maude, *functional modules* are equational theories in membership equational logic satisfying the additional requirement of being Church-Rosser, sort-decreasing, and terminating. The semantics of a functional module is the *initial algebra* specified by its theory. All this is entirely similar to the semantics of "objects" (that correspond to modules in our sense) in OBJ [60]. Indeed, since membership equational logic conservatively extends order-sorted equational logic, Maude's functional modules extend OBJ modules.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. There is no need to declare kinds explicitly. The convenience of order-sorted notation is retained as syntactic sugar. Thus, an operator declaration

```
op push : Nat Stack -> NeStack .
```

is understood as syntactic sugar for the membership axiom

$$(\forall x, y)\ \ \mathit{push}(x, y) : \mathit{NeStack}\ \ \mathit{if}\ \ x : \mathit{Nat} \wedge y : \mathit{Stack}.$$

Similarly, a subsort declaration `NeStack < Stack` corresponds to the membership axiom

$$(\forall x)\ \ x : \mathit{Stack}\ \ \mathit{if}\ \ x : \mathit{NeStack}.$$

Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. Therefore, the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort-decreasing [12]. This guarantees that all terms in an equivalence class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. For a module satisfying such conditions any reduction strategy will reach a normal form. All that we say has to be understood for equational rewriting *modulo* particular axioms. Maude supports rewriting modulo associativity, commutativity, (left-, right-, or two-sided) identity, idempotence, and most combinations of these equational theories.

In membership equational logic the Church-Rosser property of terminating and sort-decreasing equations is indeed equivalent to the confluence of their critical pairs [12]. Furthermore, both equality and membership of a term in a sort are then *decidable* properties [12]. That is, the equality and membership predicates are *computable functions*. We can then use the metatheorem of Bergstra and Tucker [5] to conclude that such predicates are themselves specifiable by Church-Rosser and terminating equations as Boolean-valued functions. This has the pleasant consequence of allowing us to include inequalities $t \neq t'$ and negations of sort tests $not(t : s)$ in conditions of equations and of membership axioms, since such seemingly negative predicates can also be axiomatized *inside the logic* in a positive way, provided that we have a subspecification *not* involving inequalities or negations of (not necessarily free) constructors in which to do it, and that the specification is indeed Church-Rosser, terminating, and sort decreasing.

Let us denote membership equational logic by *MEqtl* and its associated rewriting logic by *MRWLogic*. Regarding an equational theory as a rewrite theory whose set of rules is empty defines a conservative map of logics [71]

$$\mathit{MEqtl} \longrightarrow \mathit{MRWLogic}.$$

This is the way in which Maude's functional modules are regarded as a special case of its more general system modules.

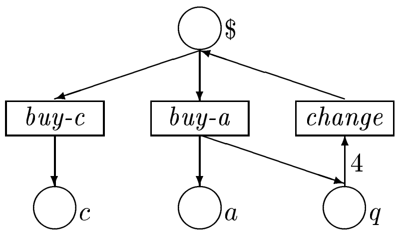
1.3.2 Rewriting Logic and System Modules

The type of rewriting typical of functional modules terminates with a single value as its outcome. In such modules, each step of rewriting is a step of *replacement of equals by equals*, until we find the equivalent, fully evaluated value. In general, however, a set of rewrite rules need not be terminating, and need not be Church-Rosser. That is, not only can we have infinite chains of rewriting, but we may also have highly divergent rewriting paths, that could never cross each other by further rewriting.

The essential idea of rewriting logic [76] is that the *semantics* of rewriting can be drastically changed in a very fruitful way. We no longer interpret a term t as a functional expression, but as a *state* of a system; we no longer interpret a rewrite rule $t \longrightarrow t'$ as an equality, but as a *local state transition*, stating that if a portion of a system's state exhibits the pattern described by t, then that portion of the system can change to the corresponding instance of t'. Furthermore, such a local state change can take place independently from, and therefore concurrently with, any other nonoverlapping local state changes.

The most general Maude modules are *system modules*. They specify the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ [76]. This initial model is a transition system whose states are equivalence classes $[t]$ of ground terms modulo the equations E in $\mathcal{R}$, and whose transitions are proofs $\alpha : [t] \longrightarrow [t']$ in rewriting logic—that is, concurrent computations in the system so described. Such proofs are equated modulo a natural notion of proof equivalence that computationally corresponds to the "true concurrency" of the computations (for a detailed construction of $\mathcal{T}_{\mathcal{R}}$ see [76]).

To illustrate the syntax of system modules we give as an example a simple concurrent system, namely a vending machine, as a Petri net. Petri nets have a straightforward rewriting logic semantics as initial models of their associated rewrite theories [76]; therefore, this example illustrates a general method to give executable formal specifications in Maude to Petri nets. Our Petri net represents a vending machine to buy cakes and apples; a cake costs a dollar and an apple three quarters. Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar. We can represent graphically such a machine in the conventional way as follows.



The so-called *places* of this net are cakes, apples, quarters, and dollars, denoted in the picture by circles labeled, respectively, by c, a, q, and \$. In each of these places several *tokens* can be placed. We can therefore think of the places as *slots* of our machine, in which units of a certain kind appear or disappear. Tokens in one or several of these places can then be consumed by *transitions*, denoted by rectangular boxes labeled by the transition's name, with incoming and outgoing arcs indicating which tokens are consumed and produced by the transition; we can think of such transitions as the *buttons* of our vending machine. Such transitions consume tokens from the places in their incoming arcs and produce new tokens in the places of the outgoing arcs. If several tokens must be either consumed or produced in a place, then the corresponding arc indicates the exact number. For example, the *change* transition requires four quarters to produce a dollar. The vending machine is *concurrent* because, provided enough tokens are available, we can *simultaneously push* several buttons and then can simultaneously get the combined results.

The distributed states of the machine, namely the collections of tokens available in the different places, are called *markings*. They can be naturally regarded as a *multiset* of places. Using juxtaposition notation, we can for example regard the state with one dollar and four quarters as the multiset $\$\ q\ q\ q\ q$. Meseguer and Montanari [85] observed that we can then view the Petri net as an ordinary graph, in which the transitions are the edges, and the nodes are multisets of places. Therefore, as a graph, this net has the following arcs:

$$\begin{array}{ll} \mathit{buy\text{-}c}: & \$ \longrightarrow c \\ \mathit{buy\text{-}a}: & \$ \longrightarrow a\ q \\ \mathit{change}: & q\ q\ q\ q \longrightarrow \$ \end{array}$$

The expression of this Petri net in rewriting logic is now obvious. We can view each of the labeled arcs of the Petri net as a rewrite rule in a rewrite theory having a binary *associative* and *commutative* operator __ (multiset union), so that rewriting happens modulo such axioms, that is, it is multiset rewriting. We can gather together the "places" $\$, q, a, c$ into a sort `Place` and view the states of the net, that is, the markings, as elements of a supersort `Marking` containing `Place` and endowed with a multiset union operator with empty syntax. The corresponding Maude module then becomes

```
mod PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op __ : Marking Marking -> Marking [assoc comm] .
  ops $ q a c : -> Place .

  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a  q .
  rl [change] : q  q  q  q => $ .
endm
```

The rewrite theory (Ω, E, L, R) corresponding to a system module has a signature Ω given by the sorts, subsort relations, and operator declarations, a set E of equations, that is assumed to be decomposed as a union $E = A \cup E'$, with A a set of axioms to rewrite modulo among those supported by Maude, and E' a set of Church-Rosser and terminating equations modulo A.

Using reflection the rewriting inference process can be controlled with great flexibility in Maude by means of *strategies* [30, 19] that can be defined by rewrite rules at the metalevel. However, the Maude interpreter provides a *default strategy* for executing expressions in system modules. The default strategy applies the rules in a top-down fair way (see [21]).

Semantics of System Modules

As already pointed out, the logic of Maude is the membership logic variant of rewriting logic *MRWLogic*. A *system module* is then a rewrite theory $\mathcal{R}$ in such a logic, and its semantics is the initial model $\mathcal{T}_{\mathcal{R}}$ [76], that is, the algebra of all rewriting computations for ground terms in the theory. From a systems perspective this model describes all the concurrent behaviors that the system so axiomatized can exhibit. From that perspective, a term t denotes a state of the system, and a proof term $\alpha : t \longrightarrow t'$ denotes a possibly concurrent computation.

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What we instead require for theories in system modules is that:

- The equations are divided into a set A of axioms, for which matching algorithms exist in the Maude implementation[3], and a set E' of equations that are Church-Rosser, terminating, and sort decreasing *modulo* A; that is, the equational part must be equivalent to a functional module.

- The rules R in the module are *coherent* [109] (or at least what might be called "weakly coherent" [77, 108]) with the equations E' modulo A. This means that appropriate critical pairs exist between rules and equations,

[3]Maude's rewrite engine has an extensible design, so that matching algorithms for new theories can be added and can be combined with existing ones [46]. In Maude 1.00, matching modulo associativity, commutativity, (left-, right-, or two-sided) identity, and idempotence, and most combinations of these attributes are supported.

allowing us to intermix rewriting with rules and rewriting with equations without loosing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E' \cup A$ with just a matching algorithm for A. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E' before applying any rule in R. This is precisely the strategy adopted by the Maude interpreter.

Since the state of the system specified by a system module is axiomatized as an abstract data type by the equations E' modulo A, and the rules in R are local rules for changing such a state, in practice the lefthand sides of rules in R only involve constructor patterns, so that coherence is a natural byproduct of good specification practice. Besides, using the completion methods in [109], one can check coherence, and one can try to make a set of rules coherent when they are not so.

1.3.3 Module Hierarchies

Specifications and code should be structured in *modules* of relatively small size to facilitate understandability of large systems, increase reusability of components, and localize the effects of system changes. In Maude, the fullest support of these goals is achieved in Full Maude, which has a rich and extensible *module algebra* supporting, in particular, parameterized programming techniques in the OBJ3 style [60]. However, Core Maude provides already some useful basic support for modularity by allowing the definition of module *hierarchies*, that is, acyclic graphs of module importations. Mathematically, we can think of such hierarchies as partial orders of *theory inclusions*, that is, the theory of the importing module contains the theories of its submodules as subtheories.

Recall that a *rewrite theory* is a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic. As already mentioned in Section 1.3.1, a system module is a rewrite theory with *initial semantics*. Note that we can use the inclusion of membership equational logic into rewriting logic to view a functional module specifying an equational theory (Ω, E) as a degenerate case of a rewrite theory, namely the rewrite theory $(\Omega, E, \emptyset, \emptyset)$. In fact the initial algebra of (Ω, E) and the initial model of $(\Omega, E, \emptyset, \emptyset)$ coincide [76]. Therefore, in essence we can view all modules as rewrite theories.

The most general form of module inclusion is provided by the `including` keyword. The `protecting` keyword is a more restricted form of inclusion, in the sense that it makes a *semantic assertion* about the relationship between the initial models of the two theories. Let $\mathcal{R} = (\Omega, E, L, R)$ be the rewrite theory specified by a system module, and let $\mathcal{R}' = (\Omega', E', L', R')$ be the theory of a supermodule, so that we have a theory inclusion $\mathcal{R} \subseteq \mathcal{R}'$. Then, we can view each model $\mathcal{M}'$ of $\mathcal{R}'$ as a model $\mathcal{M}'|_{\mathcal{R}}$ of $\mathcal{R}$, simply by disregarding the extra sorts, operations, equations, membership axioms, and rules in $\mathcal{R}' - \mathcal{R}$. Since, as explained in Section 1.3.2, the rewrite theories $\mathcal{R}$ and $\mathcal{R}'$ have respective initial models $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}'}$, by initiality of $\mathcal{T}_{\mathcal{R}}$ we always have a unique $\mathcal{R}$-homomorphism

$$h : \mathcal{T}_{\mathcal{R}} \longrightarrow \mathcal{T}_{\mathcal{R}'}|_{\mathcal{R}}.$$

The `protecting` importation asserts that for each sort s in the signature Ω of $\mathcal{R}'$ the function h_s is an isomorphism of categories[4]. Intuitively, this means that

[4] In the models of a rewrite theory the sorts are interpreted as categories, thus the require-

the initial model of the supermodule does not add any "junk" or any "confusion" to the initial model of the submodule.

Of course, the `protecting` assertion cannot be checked by Maude at run-time. It requires inductive theorem proving. Using the proof techniques in [11] together with an inductive theorem prover for membership equational logic such as the one described in [24] and a Church-Rosser checker such as the one described in Chapter 5, this can be done for functional modules; and it seems natural to expect that these techniques and tools will extend to similar ones for rewrite theories.

By contrast, the `including` assertion does not make such requirements on h. It does, however, make *some* requirements. Namely, if the subtheory $\mathcal{R}$ does itself contain a proper subtheory $\mathcal{R}_0$ that it imports in `protecting` mode, then the inclusion $\mathcal{R}_0 \subseteq \mathcal{R}'$ does still have to be protecting. If we do not want it to be, we have to say so by explicitly listing the module defining $\mathcal{R}_0$ in the list of modules imported in `including` mode.

1.3.4 Core Maude Predefined Modules

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. There are modules `BOOL`, `MACHINE-INT`, `QID`, `QID-LIST`, `META-LEVEL`, and `LOOP-MODE`. Predefined modules are explained in detail in [21], and their entire library is given in Appendix B.

The module `BOOL` defines the Boolean values and some predefined operators, like `_==_` and `_=/=_`, for equality and inequality, respectively, `if_then_else_fi`, and some of the usual Boolean predicates, like conjunction, disjunction, etc. The module `MACHINE-INT` provides sorts `MachineInt` and `NzMachineInt` for built-in integers and nonzero integers, respectively, and usual arithmetic operations on them. Built-in quoted identifiers in the LISP style are provided by the sort `Qid` in the `QID` module. This module also includes some built-in operations on quoted identifiers, as concatenation (`conc`) or indexing of a quoted identifier by a machine integer (`index`). In the rest of this section we explain in some detail the `META-LEVEL` module and the `LOOP-MODE` module.

The Module `META-LEVEL`

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. The functional module `META-LEVEL` efficiently implements key functionality of the universal theory $\mathcal{U}$. For example, it provides the following functionality.

- Maude terms are reified as elements of a data type `Term` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a descent function `meta-reduce`;

ment; for functional theories the requirement becomes that each such h_s is bijective. Note that the expected condition would have been to require h to be an *$\mathcal{R}$-isomorphism*. However, due to the presence of error elements at the kind level, the isomorphism condition would be too strong, since in general, when enlarging a signature, there will be new error terms that cannot be proved equal to old ones. See [11] for a detailed discussion of, and proof techniques for, protecting extensions in membership equational logic.

- the process of rewriting a term in a system module using Maude's default interpreter is reified by a descent function `meta-rewrite`; and
- parsing and pretty printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

`META-LEVEL` imports the module `QID-LIST` (lists of quoted identifiers) from the standard library of predefined modules, which contains in turn the modules `QID`, `MACHINE-INT`, and `BOOL`. We first introduce the syntax used in `META-LEVEL` for representing terms; then we explain how modules are represented; and finally we discuss the different built-in functions, namely, the descent functions, and the parsing and sort functions.

Representing Terms

Terms are reified as elements of the data type `Term` of terms, with the following signature[5]:

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op error* : -> Term .
```

The first declaration, making `Qid` a subsort of `Term`, is used to represent variables by the corresponding quoted identifiers. Thus, the variable `N` is represented by `'N`. The operator `{_}_` is used for representing constants essentially as pairs, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant `0` in the module `NAT` below is represented as `{'0}'Nat`. The operator `_[_]` corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted `_,_`. For example, the term `s s 0 + s 0` of sort `Nat` in module `NAT` is metarepresented as

```
'_+_['s_['s_[{'0}'Nat]], 's_[{'0}'Nat]].
```

Since terms in the module `META-LEVEL` can be metarepresented just as terms in any other module, as mentioned when discussing the universal theory, the representation of terms can be iterated. For example, the meta-metarepresentation $\overline{\overline{\overline{\texttt{s 0}}}}$ of the term `s 0` in `NAT` is the term

```
'_[_][{''s}'Qid, '{_}_[{''0}'Qid, {''Nat}'Qid]].
```

The last declaration for the data type of terms is a constant `error*` to be used as an error element.

Representing Modules

Functional and system modules are metarepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations,

[5]The module `META-LEVEL` also includes constructors to represent sort tests and lazy sort tests that we do not present here.

membership axioms, and rules are now metarepresented as we have already explained; (2) in the metarepresentation of modules we follow a *fixed order* in introducing the different kinds of declarations for sorts, subsort relations, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator `_;_`.

Besides sorts `Term` and `Module`, the predefined module `META-LEVEL` includes other sorts whose terms represent items appearing in a module or auxiliary data. There are sorts `Sort` (for sort identifiers[6]), `Attr` (for operator attributes) and `Assignment` (assignments of terms to variables in substitutions); sorts for the declaration of submodule inclusions (`Import`), sorts (`SortDecl`), subsort relations (`SubsortDecl`), operations (`OpDecl`), variables (`VarDecl`), equations (`Equation`), rules (`Rule`), and membership axioms (`MembAx`); sorts for sets[7] or lists of some of such declarations (`ImportList`, `SubsortDeclSet`, `OpDeclSet`, `VarDeclSet`, `EquationSet`, `RuleSet`, `MembAxSet`); and some other sorts, such as `QidList` or `Substitution`.

The syntax for the top-level operators representing functional and system modules is as follows

```
sorts FModule Module .
subsort FModule < Module .
op fmod_is_______endfm : Qid ImportList SortDecl SubsortDeclSet
     OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule .
op mod_is________endm : Qid ImportList SortDecl SubsortDeclSet
     OpDeclSet VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

Sort declarations are represented using a constructor

```
op sorts_. : QidSet -> SortDecl .
```

where `QidSet` is the sort of sets of quoted identifiers:

```
subsort Qid < QidSet .
op none : -> QidSet .
op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .
```

Subsort declarations are represented with the syntax:

```
subsort SubsortDecl < SubsortDeclSet .
op _<_ : Qid Qid -> SubsortDecl .
op none : -> SubsortDeclSet .
op __ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
     [assoc comm id: none] .
```

`META-LEVEL` also includes similar syntax declarations for representing the other elements that can appear in a module. Without going into all the details, let us for example focus in the representation of operator declarations. We have already mentioned the sorts `OpDecl` and `OpDeclSet`, which provide declarations of operators and sets of declarations of operators, respectively. Each operator declaration is given by a term of sort `OpDecl` using a constructor

```
op op_:_->_[_]. : Qid QidList Qid AttrSet -> OpDecl .
```

[6]User defined sorts are of sort `Qid`. The sort `Sort` is declared as a supersort of `Qid`.

[7]Although they are called "sets", actually they are "multisets" because there is missing the idempotence attribute or equation.

The sort AttrSet is declared as a supersort of Attr, and with a union operator __. The sort Attr is used for representing operator attributes, including attributes that are equational axioms used to rewrite modulo. For example, the commutativity attribute is represented by the constant

```
op comm : -> Attr .
```

Thus, the declaration of a commutative natural number addition operator can be represented by the term

```
op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
```

Note that names of sorts and operators (and also those of variables) are represented in META-LEVEL as quoted identifiers. For example, a "mixfix" operator like _+_ is represented by the corresponding quoted identifier '_+_, with underbar characters indicating the argument positions.

If there are no operator declarations in a particular module, then the corresponding argument will be none. Otherwise, the set is given by the associative constructor __, which is declared also with identity element none.

```
op none : -> OpDeclSet .
op __ : OpDeclSet OpDeclSet -> OpDeclSet [assoc comm id: none] .
```

To motivate the general syntax for representing modules, we illustrate it with a simple example—namely, a module NAT for natural numbers with zero and successor and with a commutative addition operator.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
endfm
```

The representation $\overline{\texttt{NAT}}$ of NAT in META-LEVEL is the following term of sort FModule:

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
endfm
```

Since NAT has no list of imported submodules and no membership axioms, those fields are filled, respectively, by the constants nil, of sort ImportList, and `none`, of sort `MembAxSet`. Similarly, since the zero and successor operators have no attributes, they have the empty set of attributes `none`.

The module `META-LEVEL` is included in Appendix B in its entirety. It extends in a natural way the fragment illustrated in the above example and should, for the most part, be self-explanatory by comparison with the Maude syntax, which is mirrored quite closely.

Note that—just as in the case of terms—terms of sort `Module` can be metarepresented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in [24], where modules are metarepresented as terms in the inference rules for induction, but they have to be meta-metarepresented as terms of sort `Term` when used in strategies that control the application of the inductive inference rules.

Descent Functions

The descent functions [20, 21] provide useful and efficient ways of reducing metalevel computations to object-level ones. We explain here only two of the built-in descent functions in `META-LEVEL`, namely `meta-reduce` and `meta-rewrite`.

The function `meta-reduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module. It has syntax[8]

```
op meta-reduce : Module Term -> Term [special (...)] .
```

When the second argument is the representation $\overline{t}$ of a term t in $\mathcal{R}$, the function `meta-reduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, e.g.,

```
Maude> red meta-reduce(NAT, s 0 + 0) .
result Term: s 0
```

Note that in order to simplify the presentation we use the meta-notations $\overline{t}$ and $\overline{Id}$ for t a term and Id the name of a module. Such a metarepresentation has to be explicitly given, that is, considering `NAT` as the module given above, the previous example must be written as follows.

```
Maude> red meta-reduce(
             fmod 'NAT is
                nil
                sorts 'Zero ; 'Nat .
                subsort 'Zero < 'Nat .
                op '0 : nil -> 'Zero [none] .
                op 's_ : 'Nat -> 'Nat [none] .
                op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
                var 'N : 'Nat .
                var 'M : 'Nat .
                none
                eq '_+_[{'0}'Nat, 'N] = 'N .
                eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
             endfm,
             '_+_['s_[{'0}'Nat], {'0}'Nat]) .
result Term: 's_[{'0}'Zero]
```

[8]The `special` attribute states that the operator is treated as a *built-in* operator, so it is associated to appropriate C++ code by "hooks" as stated next to the `special` attribute. To lighten the exposition, we omit the details about such hooks in special operators writing `[special (...)]` instead.

We shall see in Section 3.4 how we can extend the Maude system using its metaprogramming capabilities to be able to refer to the metarepresentation of modules previously entered in the system. We discuss in Section 1.3.6 some alternatives to do it in Core Maude.

The function `meta-rewrite` has syntax

```
op meta-rewrite : Module Term MachineInt -> Term [special (...)] .
```

It is entirely analogous to `meta-reduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a positive machine integer n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter, which applies the rules in a fair, top-down fashion. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end.

Parsing, Pretty Printing, and Sort Functions

Besides the descent functions already discussed, `META-LEVEL` provides several other functions that naturally belong to the universal theory and could have been equationally axiomatized in such a theory. However, for efficiency reasons they are provided as built-in functions. These functions allow parsing and pretty printing a term in a module at the metalevel, and performing efficiently a number of useful operations on the sorts declared in a module's signature.

The function `meta-parse` has syntax

```
op meta-parse : Module QidList -> Term [special (...)] .
```

It takes as arguments the representation of a module and the representation of a list of tokens as a list of quoted identifiers. It returns the metarepresentation of the parsed term of that list of tokens for the signature of the module, which is assumed to be unambiguous. If such a parsed term does not exist, the error constant `error*` is returned instead. For example, given the module `NAT` above and the input `'s '0 '+ '0`, we get the following result.

```
Maude> red meta-parse(NAT, ('s '0 '+ '0)) .
result Term: '_+_['s_[{'0}'Zero], {'0}'Zero]
```

The function `meta-pretty-print` has syntax

```
op meta-pretty-print : Module Term -> QidList [special (...)] .
```

It takes as arguments the representation of a module T and the representation of a term t. It returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by T. In the event of an error an empty list is returned. Thus, given the module `NAT`, we have, e.g.,

```
Maude> red meta-pretty-print(NAT,
             '_+_['_+_['s_[{'0}'Zero], {'0}'Zero], 's_[{'0}'Zero]]) .
result QidList: '`( 's '0 '+ '0 '`) '+ 's '0
```

The module `META-LEVEL` also provides some commonly needed functions on the poset of sorts of a module in a built-in way. For example, the function `leastSort` takes as arguments the representations of a module and a term and computes the (representation of the) least sort of that term in the module. Its syntax is as follows.

```
op leastSort : Module Term -> Sort [special (...)] .
```

Since it can return an error sort not defined by the user its result sort is `Sort`. For example, we can compute the least sort of the term $\overline{\texttt{N + s M}}$ in the module `NAT` as follows.

```
Maude> red least-sort(NAT, N + s M) .
result Qid: Nat
```

The `LOOP-MODE` Module

Using object-oriented concepts, we can specify in Maude a general input/output facility provided by the `LOOP-MODE` module shown below, that extends the module `QID-LIST` into a generic read-eval-print loop.

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System [special (...)] .
endm
```

The operator `[_,_,_]` can be seen as a persistent object with an input channel (the first argument), an output channel (the third argument), and a state (given by its second argument). This read-eval-print loop that `LOOP-MODE` provides is a simple mechanism developed for the release 1.00 of Maude that may not be maintained in future versions. Maude will be endowed with general built-in support for objects in the future; this will make possible more general and flexible solutions for dealing with input/output and persistent objects.

Besides having input and output channels, terms of sort `System` give us the possibility of maintaining a persistent state in their second component. This state has been declared in a completely generic way. In fact, the sort `State` in `LOOP-MODE` does not have any constructor. This gives complete flexibility for defining the terms we want to have for representing the persistent state of the loop in each particular application. In this way we can use this input/output facility not only for extensions of Maude like Full Maude, but also for other uses of Maude as a *metalanguage*, where the object language being implemented may be completely different from Maude. For each such language or tool the nature of the state of the system may be completely different. We can tailor the `State` sort to any such application by importing `LOOP-MODE` in a module in which we define the structure of the persistent state and the rewrite rules for changing that state and interacting with the loop.

We can illustrate the basic ideas with a toy example, namely a system in which the loop is used to echo each input list twice. In this case there is no need to maintain any state, so we can just declare a constant `null` to represent the empty state.

```
mod DUPLICATE is
  including LOOP-MODE .
  op null : -> State .
  vars Input Output : QidList .
  crl [duplicate] :
     [Input, null, Output]
     => [nil, null, Output Input Input]
        if Input =/= nil .
endm
```

Once this module has been entered, we must first initialize the loop by setting its initial state using the `loop` command. That is, we must give to the `loop` command the term of sort `State` that we desire as initial state. For this example, we can start a loop with empty input and output channels by typing

```
Maude> loop [nil, null, nil] .
```

Since in the current release only one input channel is supported (the current terminal), the way to distinguish the input passed to the loop from the input to the Maude system—modules or commands—is by enclosing them in parentheses. When something is written in the Maude prompt enclosed in parentheses it is converted into a list of quoted identifiers. This is done by first breaking the input stream into a sequence of tokens—that is, into a sequence of Maude identifiers—and then converting each of these tokens into a quoted identifier by putting a quote in front of it, and appending the results into a list of quoted identifiers, which is then placed in the first slot of the loop object. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop is printed on the terminal after applying the inverse process of "unquoting" each of the tokens in the list. We can think of the input and output events as *implicit rewrites* that transfer—in a slightly modified, quoted or unquoted form—the input and output data between two objects, namely the loop object and the "user" or "terminal" object.

Once the loop has been initialized we can input any data by writing it after the prompt enclosed in parentheses. For example, we can write

```
Maude> (a s d)
```

and then we get the following output:

```
a s d a s d
```

A somewhat more interesting example is a loop that echoes the input, but storing the last ten tokens, that is, it prints one token only if the number of tokens stored in the state is bigger than ten. We now really need a persistent state, albeit a simple one. We can represent the state as a pair consisting of a list of quoted identifiers—the tokens seen so far—and a counter measuring the length of such a list.

```
mod DUPLICATE-TEN is
  including LOOP-MODE .
  protecting MACHINE-INT .
  op <_;_> : QidList MachineInt -> State .
  op init : -> System .
  vars Input StoredInput Output : QidList .
  var QI : Qid .
  var Counter : MachineInt .
  rl [init] : init => [nil, < nil ; 0 >, nil] .
  rl [in] :
     [QI Input, < StoredInput ; Counter >, Output]
     => [Input, < StoredInput QI ; Counter + 1 >, Output] .
  crl [out] :
     [Input, < QI StoredInput ; Counter >, Output]
     => [Input, < StoredInput ; Counter - 1 >, Output QI]
        if Counter > 10 .
endm
```

A possible interaction with the system can then be given as follows:

```
Maude> loop init .
Maude> (a b)
Maude> (c d e f g h i)
Maude> (j k l)
a b
```

The above examples are toy examples to illustrate the basic features of the module LOOP-MODE. However, the most interesting applications of this module are *metalanguage* applications, in which Maude is used to define the syntax, parse, execute, and pretty print the execution results of a given object language or tool. In such applications, most of the hard work is done by the META-LEVEL module, but handling the input/output and maintaining the persistent state of the object language interpreter or tool is done by LOOP-MODE.

1.3.5 Metalanguage Uses of Maude

The metalanguage uses of Maude are a natural consequence of the good properties of rewriting logic as a logical and semantic framework. Indeed, one of the key goals of rewriting logic from its beginning has been to provide a *semantic framework* in which many models of computation—particularly concurrent and distributed ones—and languages can be naturally represented. Because of the intrinsic duality between logic and computation that rewriting logic supports, the very same reasons making rewriting logic a suitable semantic framework, make it also an attractive *logical framework* [71] to represent many different logics.

What is common to all these logical and semantic framework applications is that the models of computation, logics, or languages are represented in rewriting logic by mappings of the form

$$(\dagger)\ \ \Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

The representations are typically very simple and natural. They map theories or modules in $\mathcal{L}$ to rewrite theories.

By being reflective, a rewriting logic language such as Maude supports representation maps of the form (†), so that it becomes a *metalanguage* in which a very wide variety of programming, specification, and design languages, and of computational and logical systems can be both *semantically defined*, and *implemented* in it.

As already explained in Section 1.3.4, Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective and provide efficient support of reflective computation by means of the META-LEVEL module.

Indeed, using META-LEVEL we can both make the above representation map Φ executable, and we can execute the resulting rewrite theory representing a theory or module in $\mathcal{L}$, thus getting an implementation of $\mathcal{L}$ in Maude. Specifically, we can reify a representation map Φ of the form (†) by defining an abstract data type $\mathtt{Module}_{\mathcal{L}}$ representing modules in the logic or language $\mathcal{L}$. Since in META-LEVEL we also have a data type Module whose terms represent rewrite theories, we can then *internalize* the representation map Φ as an equationally defined function

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}.$$

In fact, thanks to the general metaresult of Bergstra and Tucker [5], any computable representation map Φ can be specified in this way by a finite number of Church-Rosser and terminating equations.

Having this representation map defined in Maude, we can then execute in Maude the rewrite theory $\overline{\Phi}(M)$ associated to a theory or module M in $\mathcal{L}$. This has been done, for example, for linear logic in [74, 19], and for structured Maude modules in Full Maude as explained in Chapters 3 and 4. But it could also be done for a very wide range of other languages and logics using the same method.

By defining the data type $\texttt{Module}_{\mathcal{L}}$ in an extension of `META-LEVEL` we can indeed define the syntax of $\mathcal{L}$ within Maude. However, to provide a satisfactory execution environment for $\mathcal{L}$ in Maude, we also have to support input/output and a persistent state for interacting with the interpreter for $\mathcal{L}$ that we want to define. That is, we want to be able to enter module definitions, execute commands, and get results of executions. The `LOOP-MODE` module introduced in Section 1.3.4 makes this possible. As a consequence, an environment for $\mathcal{L}$ in Maude will typically be realized by a module containing both `META-LEVEL` and `LOOP-MODE` as submodules.

1.3.6 Tokens, Bubbles, and Metaparsing

In order to generate in Maude an *environment* for a language $\mathcal{L}$, including the case of a language with user-definable syntax, the first thing we need to do is to define the syntax for $\mathcal{L}$-modules. This can be done by defining a data type $\texttt{Grammar}_{\mathcal{L}}$ for $\mathcal{L}$-modules, and with auxiliary declarations for commands and other constructs. Maude provides great flexibility to do this thanks to its mixfix front-end and to the use of bubbles[9] (any nonempty string of Maude identifiers). The intuition behind bubbles is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available. The Maude parser (MSCP [94]) provides a basis for this flexible syntax definition, and an efficient treatment of what might be called *syntactic reflection*. MSCP is implemented using SCP as its formal kernel [95].

The idea is that for a language that allows modules with user-definable syntax—as it is the case for Maude—it is natural to see its syntax as a combined syntax, at two different levels: what we may call the *top level* syntax of the language, and the user-definable syntax introduced in each module. The data type `Bubble` allows us to reflect this duality of levels in the syntax definition. Similar ideas have been exploited using ASF+SDF [107, 3, 62, 69].

To illustrate this concept, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:

```
fmod NAT3 is
   sort Nat3 .
   op s_ : Nat3 -> Nat3 .
   op 0 : -> Nat3 .
   eq s s s 0 = 0 .
endfm
```

Notice that the strings of characters inside the boxes are not part of the top level syntax of Maude. In fact, they can only be parsed with the grammar associated to the signature of the module `NAT3`. In this sense, we say that the syntax for Maude modules is a combination of two levels of syntax. The term `s s s 0`, for example, has to be parsed in the grammar associated to the signature of `NAT3`. The definition of the syntax of Maude in Maude must reflect this duality of syntax levels.

[9]In the release 1.00 of Maude bubbles can only be used in modules passed to the function `meta-parse` as arguments. The way of using and defining bubbles will change in future releases.

So far, we have talked about bubbles in a generic way. In fact, there can be many different kinds of bubbles. In Maude we can define different types of bubbles as built-in data types by parameterizing their definition. Thus, for example, a bubble of length one, which we call a *token*, can be defined as follows.

```
sort Token .
op token : Qid -> Token
       [special (id-hook Bubble        (1 1)
                 op-hook qidBaseSymbol (<Qids> : -> Qid))] .
```

Any name can be used to define a bubble sort. It is the special attribute

```
                 id-hook Bubble        (1 1)
```

in its constructor declaration that makes the sort `Token` a bubble sort. The second argument of the `id-hook` special attribute indicates the minimum and maximum length of such bubbles as strings of identifiers. Therefore, `Token` has only bubbles of size 1. To specify a bubble of any length we would use the pair of values 1 and -1. The operator used in the declaration of the bubble, in this case the operator `token`, is a bubble constructor that represents tokens in terms of their quoted form. For example, the token $\boxed{\texttt{abc123}}$ is represented as `token('abc123)`.

We can define bubbles of any length, that is, nonempty sequences of Maude identifiers, with the following declarations.

```
sort Bubble .
op bubble : QidList -> Bubble
       [special (id-hook Bubble        (1 -1 ( ))
                 op-hook qidListSymbol
                         (__ : QidList QidList -> QidList)
                 op-hook qidBaseSymbol (<Qids> : -> Qid))] .
```

In this case, the system will represent the bubble as a list of quoted identifiers under the constructor `bubble`. For example, the bubble $\boxed{\texttt{ab cd ef}}$ is represented as `bubble('ab 'cd 'ef)`. Note the form of the `id-hook` special attribute now. The symbols '`(`' and '`)`' in it indicate that parentheses have to be balanced inside a bubble. We could use any other pair of symbols instead of parentheses in the same way.

Different types of bubbles can be defined using the `id-hook` special attribute `Exclude`, which takes as parameter a list of identifiers to be excluded from the given bubble, that is, the bubble being defined cannot contain such identifiers.

Suppose that we give the following declarations to specify the syntax of a list of operator declarations in a module:

```
sorts Decl DeclList .
subsort Decl DeclList .
op op_: ->_. : Token Token -> Decl .
op op_:_->_. : Token Bubble Token -> Decl .
op __ : DeclList DeclList -> DeclList [assoc] .
```

With these declarations, bubbles could be bigger than expected. In particular, the following parse would be possible.

op $\boxed{\texttt{s\_}}$: $\boxed{\texttt{Nat3 -> Nat3 . op 0 :}}$ -> $\boxed{\texttt{Nat3}}$.

We can use the `id-hook` special attribute `Exclude` to avoid this situation. We can declare the sort `NeTokenList` with constructor `neTokenList` as a list of identifiers, of any length greater than one, excluding[10] the identifier '`.`' with the following declarations.

```
sort NeTokenList .
op neTokenList : QidList -> NeTokenList
       [special (id-hook Bubble        (1 -1 ( ))
                 op-hook qidListSymbol
                         (__ : QidList QidList -> QidList)
                 op-hook qidBaseSymbol (<Qids> : -> Qid)
                 id-hook Exclude (.))] .
```

We are now ready to give the signature to parse modules such as `NAT3` above. The following module `MINI-MAUDE-SYNTAX` uses the above definitions of sorts `Token`, `Bubble` and `NeTokenList` to define the syntax of a sublanguage of Maude, namely, many sorted, unconditional, functional modules, in which the declarations of sorts, variables and operators have to be done one at a time, and in which no attributes are supported for operators.

```
fmod MINI-MAUDE-SYNTAX is
  including QID-LIST .
  sorts Bubble Token NeTokenList
        PreModule PreCommand
        Decl DeclList .
  subsort Decl < DeclList .

  op token : Qid -> Token
         [special (id-hook Bubble        (1 1)
                   op-hook qidBaseSymbol (<Qids> : -> Qid))] .
  op bubble : QidList -> Bubble
         [special (id-hook Bubble        (1 -1 ( ))
                   op-hook qidListSymbol
                           (__ : QidList QidList -> QidList)
                   op-hook qidBaseSymbol (<Qids> : -> Qid))] .
  op neTokenList : QidList -> NeTokenList
         [special (id-hook Bubble        (1 -1)
                   op-hook qidListSymbol
                           (__ : QidList QidList -> QidList)
                   op-hook qidBaseSymbol (<Qids> : -> Qid)
                   id-hook Exclude (.))] .

  *** sort declaration
  op sort_. : Token -> Decl .
  *** operator declaration
  op op_: ->_. : Token Token -> Decl .
  op op_:_->_. : Token NeTokenList Token -> Decl .
  *** variable declaration
  op var_:_. : Token Token -> Decl .
  *** equation declaration
  op eq_=_. : Bubble Bubble -> Decl .
  *** functional module
  op fmod_is_endfm : Token DeclList -> PreModule .
  op __ : DeclList DeclList -> DeclList [assoc] .
endfm
```

[10]In general, to exclude identifiers `I1`, `I2`, ... , `Ik`, we use the syntax `Exclude (I1 I2...Ik)`.

Notice how we explicitly declare operators that correspond to the top level syntax of Maude, and how we represent as terms of sort `Bubble` those pieces of the module—namely, terms in equations—that can only be parsed with the user-defined syntax.

Then, the functional module `NAT3` above can be parsed as a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`. The name of this sort reflects the fact that not all terms of sort `PreModule` do actually represent Maude modules. In particular, for a term of sort `PreModule` to represent a Maude module all the bubbles must be correctly parsed as terms in the module's user-defined syntax.

When calling the function `meta-parse` with the metarepresentation of the module `MINI-MAUDE-SYNTAX` and the previous module transformed into a list of quoted identifiers[11], that is,

```
Maude> red meta-parse(
               MINI-MAUDE-SYNTAX,
               'fmod 'NAT3 'is
                   'sort 'Nat3 '.
                   'op 's_ ': 'Nat3 '-> 'Nat3 '.
                   'op '0 ': '-> 'Nat3 '.
                   'eq 's 's 's '0 '= '0 '.
               'endfm) .
```

we get the following metaterm as a result.

```
result Term:
   'fmod_is_endfm[
       'NAT3 ,
       '__['sort_.[ 'Nat3 ],
           '__['op_:_->_.[ 's_ , 'Nat3 , 'Nat3 ],
               '__['op_:`->_.[ '0 , 'Nat3 ],
                   'eq_=_.[ 's 's 's '0 , '0 ]]]]]
```

Of course, Maude does not return these boxes. Instead, the system returns the bubbles using their constructor form as specified in their corresponding declarations. For example, the bubbles `'Nat3` and `'s 's 's '0` are represented, respectively, as `token('Nat3)` and `bubble('s 's 's '0)`. Maude returns them metarepresented. The result given by Maude is therefore the following:

```
result Term:
   'fmod_is_endfm[
      'token[{''NAT3}'Qid],
      '__['sort_.['token[{''Nat3}'Qid]],
          '__['op_:_->_.['token[{''s_}'Qid],
                         'neTokenList[{''Nat3}'Qid],
                         'token[{''Nat3}'Qid]],
              '__['op_:`->_.['token[{''0}'Qid],
                             'token[{''Nat3}'Qid]],
                  'eq_=_.['bubble['__[{''s}'Qid, {''s}'Qid,
                                      {''s}'Qid, {''0}'Qid]],
                          'bubble[{''0}'Qid]]]]]]
```

This result is a metaterm of sort `Term`. To convert this term into a term of sort `FModule` is now straightforward. As already mentioned, we first have to extract from the term the module's signature. For this, we can use an equationally defined function

[11]This representation of the input as a list of quoted identifiers is given automatically by the read-eval-print loop supported by the built-in module `LOOP-MODE`.

```
op extractSignature : Term -> FModule? .
```

Notice that `extractSignature` is a partial function that is not well defined for metaterms of sort `Term` that do not metarepresent terms of sort `PreModule` in `MINI-MAUDE-SYNTAX`. Therefore, the result is in general an element of an error supersort `FModule?` of `FModule`. Once we have the signature of the module—expressed as a functional module with no equations and no membership axioms—we can then build terms of sort `FModule` in the same way with the equationally defined function `solveBubbles`, that recursively replaces each bubble in an equation by the result of calling `meta-parse` with the already extracted signature and with the quoted identifier form of the bubble.

```
op solveBubbles : Term FModule -> FModule? .
```

The partial function `processPreModuleTerm` takes a term and, if it has the appropriate form—that is, if it is a term metarepresenting a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`, and, furthermore, the `solveBubbles` function succeeds in parsing the bubbles in equations as terms—then it returns a term of sort `FModule`.

```
op processPreModuleTerm : Term -> FModule? .
eq processPreModuleTerm(T)
  = solveBubbles(T, extractSignature(T)) .
```

We have then the following reduction.

```
Maude> red processPreModuleTerm(
              meta-parse(
                 MINI-MAUDE-SYNTAX,
                 'fmod 'NAT3 'is
                     'sort 'Nat3 '.
                     'op 's_ ': 'Nat3 '-> 'Nat3 '.
                     'op '0 ': '-> 'Nat3 '.
                     'eq 's 's 's '0 '= '0 '.
                 'endfm)) .

Result FModule :
   fmod 'NAT3 is
      nil
      sorts 'Nat3 .
      none
      op '0 : nil -> 'Nat3 [none] .
      op 's_ : 'Nat3 -> 'Nat3 [none] .
      none
      none
      eq 's_['s_['s_[{'0}'Nat3]]] = {'0}'Nat3 .
   endfm
```

Extending `META-LEVEL`

From the examples in the previous section one can imagine that giving the metarepresentation of modules when calling functions taking them as arguments can become quite inconvenient. There are two very simple ways of avoiding it:

- One possibility is to declare in a module extending `META-LEVEL` a constant of sort `FModule` or `Module`, and then define it by an equation identifying such a constant with the metarepresentation of the flattened version of the module we are interested in using. However, the module may be large, and this makes this solution unsatisfactory.

- It is possible to refer in the metarepresentation of a module to built-in modules and to modules previously entered in Core Maude's database. This makes possible a second alternative, namely, entering the module of interest in Core Maude, and then referring to it in the metarepresentation of another module. Note that in this way we can use the module entered at the object level in calls to built-in functions, but we do not have access to its metarepresentation, that is, we cannot manipulate it as data. For example, we can define modules such as the following.

```
fmod FOO is
  sort Foo .
  op foo : -> Foo .
endfm

fmod BAR is
  including META-LEVEL .
  op M : -> FModule .
  eq M = (fmod 'M is
             including 'FOO .
             sorts none .
             none
             none
             none
             none
             none
          endfm) .
endfm
```

Note that, if desired, we could have added additional declarations to the metarepresentation of the module `M`. In the context of the `BAR` module, we can now refer to the metarepresentation of module `M`, which contains all the declarations in the module `FOO`, in any command.

Bubble sorts are not supported at the object level in the version 1.00 of Core Maude. The `special` attributes linking the constructors for the bubble sorts to the built-in ones are only supported at the metalevel, that is, the declarations of the constructor operators for bubble sorts have to be given in the metarepresentation of a module. But this does not mean that we have to give the metarepresentation of modules including them. We can write a module at the object level including all the declarations, except for those containing `special` attributes for bubble sorts. Such declarations involving `special` attributes can then be included in the metarepresentation of a module that includes the quoted name for the rest of the module of interest, which can be defined at the object level. We can illustrate these ideas by applying them to the module `MINI-MAUDE-SYNTAX` above, which can be made available at the metalevel by redefining at the object level the fragment not containing the `special` attributes, and then importing that fragment—adding the metarepresentation of the operations involving the `special` attributes—within a module term at the metalevel in the module `PARSING`.

```
fmod MINI-MAUDE-SYNTAX is
  including QID-LIST .
  sorts Bubble Token NeTokenList
        PreModule PreCommand Decl DeclList .
  subsort Decl < DeclList .
```

```
  *** sort declaration
  op sort_. : Token -> Decl .

  *** operator declaration
  op op_: ->_. : Token Token -> Decl .
  op op_:_->_. : Token NeTokenList Token -> Decl .

  *** variable declaration
  op var_:_. : Token Token -> Decl .

  *** equation declaration
  op eq_=_. : Bubble Bubble -> Decl .

  *** functional module
  op fmod_is_endfm : Token DeclList -> PreModule .
  op __ : DeclList DeclList -> DeclList [assoc] .
endfm

fmod PARSING is
  including META-LEVEL .
  op META-MINI-MAUDE-SYNTAX : -> FModule .
  eq META-MINI-MAUDE-SYNTAX
    = (fmod 'META-MINI-MAUDE-SYNTAX is
          including 'MINI-MAUDE-SYNTAX .
          sorts none .
          none
          op 'token : 'Qid -> 'Token
                [special(
                   (id-hook('Bubble, '1 '1)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'neTokenList : 'QidList -> 'NeTokenList
                [special(
                   (id-hook('Bubble, '1 '-1 '`( '`))
                    op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'bubble : 'QidList -> 'Bubble
                [special(
                   (id-hook('Bubble, '1 '-1 '`( '`))
                    op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                    op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          none
          none
          none
       endfm) .
endfm
```

Chapter 2

Categorical Constructions on Structured Theories

2.1 Structured Theories and their Semantics

Since the early seventies the algebraic specification of abstract data types became a central issue in computer science [59, 106, 44, 8, 50]. One of the important insights of Clear [15] was that large specifications should be structured to be more easily understandable and to localize the effect of changes. Burstall and Goguen proposed the idea of "putting theories together" by composing them through operations having a clean and logic-independent categorical semantics [15, 52, 53]. Categorical techniques have allowed since then the study of specification-building operations with independence from any specific formalism by different authors, e.g., [98, 38, 2, 91].

When considering theory-building operations in the Clear/OBJ tradition [15, 60], a natural next step is to regard the notion of a *structured theory* as a first-class object to which such operations are applied. Ideas of this kind have been used from the earliest work, leading, for example, to notions such as based theories [15], theories with initiality or other constraints [96, 15, 100, 53, 45], etc.

The system for program synthesis SPECWARE [103] adopts a different approach. In it, specification diagrams are defined by explicitly giving specifications and specification morphisms, describing systems in a way close to the "box-and-arrow" methodologies. Refinements of the diagrams are then given by families of morphisms [63]. Based on these ideas, Dimitrakos proposed in [40] a way of parameterizing specifications by diagrams of specifications, and of inducing an instantiation by means of a family of parallel instantiating morphisms whose sources are the components of the parameter diagram.

In this work, we generalize these notions to structured theories, and structured theories parameterized by structured theories. Instantiation of such structured theories is then defined by pushouts in the category of structured theories and structured theory morphisms.

We want to make explicit a very common and natural notion of structured theory, namely *theory hierarchies*. As we shall see in more detail in the language design and formal specification of the Full Maude module algebra (see Chapters 3 and 4), other richer notions such as parameterized module instantiation, renaming, and more general module expressions can be *normalized* to their resulting module hierarchies by applying the appropriate semantic equations for those richer constructors. The idea of providing a normal form for module expressions was first proposed by Bergstra *et al.* in their seminal work [4].

Having structured theories as first-class objects has in fact been the basic decision in the design of the Full Maude module algebra. That is, a general module expression is always normalized to a resulting module hierarchy. The evaluation of module expressions is then performed on structured theories, giving structured theories as their results. For execution purposes, such a hierarchy can be *flattened* to a single module, which is then compiled into the Core Maude rewrite engine. However, thanks to Core Maude's support for module hierarchies, in a future version of Maude this flattening step may not be necessary.

The notion of a hierarchy of theories can be formalized categorically as a functor $D : P \to \mathcal{J}$ from a poset P to the category $\mathcal{J}$ of theories and theory inclusions in a given institution $\boldsymbol{\mathcal{I}}$, mapping each $a \leq b$ in P to a theory inclusion $D(a) \hookrightarrow D(b)$ in $\mathcal{J}$. However, we present our development for a more general notion of structured theory, a notion in which the signature of a structured theory is formalized as a functor $D : I \to \mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ with I any small category and $\mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ the category of signatures and signature morphisms in a given institution $\boldsymbol{\mathcal{I}}$, that is, as a *diagram* from the "diagram scheme" category I to $\mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$.

Sentences for such a structured signature D are then pairs (i, φ) with $i \in I$, and $\varphi \in \mathrm{sen}_{\mathcal{I}}(D(i))$.

Therefore, a general categorical setting for studying categorical constructions on structured theories is to consider the category of diagrams $Dg(\mathcal{C})$ of a given category $\mathcal{C}$. Further requirements on the diagrams such as the diagram scheme category being a poset, the maps in the hierarchy being inclusions, and so on, can then be viewed as special cases of the more general framework of diagrams. We show in Section 2.2 that the category $Dg(\mathcal{C})$ is cocomplete for $\mathcal{C}$ any cocomplete category. In Section 2.3 we show that given a functor $U : \mathcal{C} \to \mathcal{A}$ which preserves colimits, and with $\mathcal{C}$ and $\mathcal{A}$ cocomplete categories, then the functor $Dg(U) \colon Dg(\mathcal{C}) \to Dg(\mathcal{A})$ also preserves colimits.

In Section 2.4 we define the institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$ of structured theories on any institution $\boldsymbol{\mathcal{I}}$, and present some useful cocompleteness results for its category of signatures and for the category of theories on it. We also show that, if the institution $\boldsymbol{\mathcal{I}}$ has a cocomplete category of signatures and is liberal and exact, then $\boldsymbol{\mathcal{S}(\mathcal{I})}$ is liberal. This is important in the generalization of this setting to deal with initiality constraints. We make use of a general construction by Goguen and Burstall in [53] that associate to an institution $\boldsymbol{\mathcal{I}}$ the institution $\boldsymbol{\mathcal{C}(\mathcal{I})}$ that extends $\boldsymbol{\mathcal{I}}$ with "freeness constraints," to construct the institution $\boldsymbol{\mathcal{S}(\mathcal{C}(\mathcal{I}))}$ of structured theories with freeness constraints, thus adding freeness constraints to our setting.

We finish the chapter illustrating with several examples in Section 2.5 the categorical constructions that we have defined.

2.2 Colimits in Diagram Categories

The categorical concept of colimit has been very important in the algebraic specification community for "putting theories together" since it was first proposed for that purpose by Burstall and Goguen [14].

Similarly, "putting structured theories together" will also be done by colimit constructions. Therefore, the issue of whether the category $Dg(\mathcal{C})$ of diagrams over a category $\mathcal{C}$ has colimits becomes important, since for $\mathcal{C} = \mathbf{Sign}_{\mathcal{I}}$ this will specialize to colimits of structured signatures, which can then be used to define colimits of structured theories. We show in this section that, if $\mathcal{C}$ is cocomplete, then $Dg(\mathcal{C})$ is also cocomplete.

We start by introducing some definitions and notational conventions. Left Kan extensions [70, 101] will be a fundamental notion in the proof of Theorem 11.

Definition 5 *Let $\mathcal{C}$ be a category and let $D \colon \mathcal{J} \to \mathcal{C}$ be a diagram in $\mathcal{C}$. A* cone *α for D in $\mathcal{C}$ consists of an object c in $\mathcal{C}$ and a family of morphisms $\alpha_i \colon D(i) \to c$ (one for each object i in $\mathcal{J}$) such that the following diagram commutes in $\mathcal{C}$ for each arrow $f \colon i \to j$ in $\mathcal{J}$.*

$$\begin{tikzcd} & c & \\ D(i) \arrow[ur, "\alpha_i"] \arrow[rr, "D(f)"'] & & D(j) \arrow[ul, "\alpha_j"'] \end{tikzcd}$$

We use the notation $\alpha \colon D \dot{\to} c$ for cones.

Given cones $\alpha : D \dot{\to} c$ and $\beta : D \dot{\to} c'$ for a diagram $D : \mathcal{J} \to \mathcal{C}$, a cone morphism *$\omega \colon \alpha \to \beta$ is a morphism $\omega \colon c \to c'$ in $\mathcal{C}$ such that for each object i*

in $\mathcal{J}$ the following diagram commutes.

$$\begin{tikzcd} c \arrow[rr, "\omega"] & & c' \\ & D(i) \arrow[ul, "\alpha_i"] \arrow[ur, "\beta_i"'] & \end{tikzcd}$$

A colimit *of D in $\mathcal{C}$ is an initial object in the category of cones over D in $\mathcal{C}$.*

Definition 6 *A category $\mathcal{C}$ is called* cocomplete *if all diagrams in $\mathcal{C}$ have colimits in $\mathcal{C}$, that is, if each functor $F: \mathcal{J} \to \mathcal{C}$ to $\mathcal{C}$ from a small category $\mathcal{J}$ has a colimit.*

Given a cocomplete category $\mathcal{C}$ and a small category $\mathcal{A}$, the category of functors from $\mathcal{A}$ to $\mathcal{C}$, which we denote by $\mathcal{C}^{\mathcal{A}}$, is also a cocomplete category [70]. Furthermore, in $\mathcal{C}^{\mathcal{A}}$ all colimits can be constructed pointwise.

Recall that, given categories $\mathcal{B}$ and $\mathcal{C}$, the category of functors $\mathcal{C}^{\mathcal{B}}$ has as objects functors $T: \mathcal{B} \to \mathcal{C}$, and as morphisms natural transformations between two such functors. Given another category $\mathcal{D}$, a functor $F: \mathcal{B} \to \mathcal{D}$ induces a functor $\mathcal{C}^F : \mathcal{C}^{\mathcal{D}} \to \mathcal{C}^{\mathcal{B}}$ by the rule $T \mapsto T \cdot F$ for objects, and $\varrho \mapsto \varrho \cdot F$ for arrows, that is, "compose with F on the right." The functor F also induces a functor $F^{\mathcal{C}}: \mathcal{B}^{\mathcal{C}} \to \mathcal{D}^{\mathcal{C}}$ by the rule $T \mapsto F \cdot T$, $\varrho \mapsto F \cdot \varrho$, that is, "compose with F on the left."

Theorem 7 (Left Kan extensions [70, 101]) *Let $\mathcal{B}$ be a small category, let $F: \mathcal{B} \to \mathcal{D}$ be a functor, and let $\mathcal{C}$ be a cocomplete category. Then, the functor $\tilde{F} = \mathcal{C}^F : \mathcal{C}^{\mathcal{D}} \to \mathcal{C}^{\mathcal{B}}$ has a left adjoint $\tilde{V}_F : \mathcal{C}^{\mathcal{B}} \to \mathcal{C}^{\mathcal{D}}$, called* left Kan extension *along F.*

$$\begin{tikzcd} \mathcal{C}^{\mathcal{B}} \arrow[r, shift right, "\tilde{V}_F"'] \arrow[r, phantom, "\top"] & \mathcal{C}^{\mathcal{D}} \arrow[l, shift right, "\tilde{F}"'] \end{tikzcd}$$

Recall also that, given a functor $F: \mathcal{B} \to \mathcal{D}$, for each $a \in |\mathcal{D}|$, the *comma category* $F{\downarrow}a$ has as objects pairs (x, f), where $x \in |\mathcal{B}|$ and $f: F(x) \to a$ in $\mathcal{D}$, and as morphisms from (x, f) to (x', f') triples (f, f', w) with $w: x \to x'$ in $\mathcal{B}$ such that $f' \cdot F(w) = f$. The projection functor $\Pi_a: F{\downarrow}a \to \mathcal{B}$ is then given by the rules $(x, f) \mapsto x$ and $(f, f', w) \mapsto w$.

Theorem 8 (Left Kan extension as a pointwise colimit [70, 101]) *Given functors $F: \mathcal{B} \to \mathcal{D}$ and $G: \mathcal{B} \to \mathcal{C}$, with $\mathcal{B}$ small and $\mathcal{C}$ cocomplete, the left Kan extension of G along F, $\tilde{V}_F(G): \mathcal{D} \to \mathcal{C}$, is constructed pointwise on the objects $a \in \mathcal{D}$ by the formula*

$$\tilde{V}_F(G)(a) = \text{colim } G \cdot \Pi_a = \text{colim } (F{\downarrow}a \xrightarrow{\Pi_a} \mathcal{B} \xrightarrow{G} \mathcal{C}).$$

On morphisms, $\tilde{V}_F(G)$ assigns to each $g: a \to a'$ in $\mathcal{D}$ the unique arrow induced by g

$$\tilde{V}_F(G)(g)\colon \text{colim } G \cdot \Pi_a \to \text{colim } G \cdot \Pi_{a'},$$

commuting with the colimiting cones.

The situation can be depicted as follows.

$$\begin{tikzcd} \mathcal{B} \arrow[rr, "F"] \arrow[dr, "G"'] & & \mathcal{D} \arrow[dl, dashed, "\tilde{V}_F(G)"] \\ & \mathcal{C} & \end{tikzcd}$$

Note that, given functors $F: \mathcal{B} \to \mathcal{D}$ and $G, G' : \mathcal{B} \to \mathcal{C}$, with $\mathcal{B}$ small and $\mathcal{C}$ cocomplete, the left Kan extension of a natural transformation $\alpha: G \to G'$, $\tilde{V}_F(\alpha): \tilde{V}_F(G) \to \tilde{V}_F(G')$, is defined as the function assigning to each object d of $\mathcal{D}$ an arrow $\tilde{V}_F(\alpha)(d): \tilde{V}_F(G)(d) \to \tilde{V}_F(G')(d)$ of $\mathcal{C}$.

We sometimes do not write parentheses for functor applications of the form $\tilde{V}_F(D)$ or $\tilde{F}(D')$. For example, we write $\tilde{V}_F D$ and $\tilde{F}\tilde{V}_F D$ for $\tilde{V}_F(D)$ and $\tilde{F}(\tilde{V}_F(D))$, respectively.

Definition 9 [101] *Let $\mathcal{C}$ be a category. The* diagram category $Dg(\mathcal{C})$ *has as objects functors $T : \mathcal{P} \to \mathcal{C}$, where $\mathcal{P}$ is a small category. If $T : \mathcal{P} \to \mathcal{C}$ and $T' : \mathcal{P}' \to \mathcal{C}$ are objects, then a morphism $(R, \varrho): T \to T'$ consists of a functor $R: \mathcal{P} \to \mathcal{P}'$ and a natural transformation $\varrho: T \to T' \cdot R$.*

$$\begin{tikzcd}
\mathcal{P} \arrow[r, "T"] \arrow[d, "R"'] \arrow[dr, phantom, "\Downarrow\varrho"] & \mathcal{C} \\
\mathcal{P}' \arrow[ur, "T'"'] &
\end{tikzcd}$$

The composition of morphisms (R, ϱ) and (R', ϱ'), as depicted in the figure below, is given by the morphism $(R' \cdot R,\ \varrho' R \cdot \varrho)$.

$$\begin{tikzcd}
\mathcal{P} \arrow[d, "R"'] \arrow[dr, "T"] \arrow[d, phantom, shift left=4ex, "\Downarrow\varrho"] & \\
\mathcal{P}' \arrow[r, "T'"] \arrow[d, "R'"'] \arrow[d, phantom, shift left=4ex, "\Downarrow\varrho'"] & \mathcal{C} \\
\mathcal{P}'' \arrow[ur, "T''"'] &
\end{tikzcd}$$

Note that, given objects $T: \mathcal{P} \to \mathcal{C}$ and $T' : \mathcal{P}' \to \mathcal{C}$ in $Dg(\mathcal{C})$, a morphism $(R, \varrho) : T \to T'$, and a functor $F: \mathcal{C} \to \mathcal{D}$ then $(R, F \cdot \varrho) : F \cdot T \to F \cdot T'$ is a morphism in $Dg(\mathcal{D})$. That is, composition on the left with a functor $F: \mathcal{C} \to \mathcal{D}$ induces a functor $Dg(F): Dg(\mathcal{C}) \to Dg(\mathcal{D})$.

Theorem 10 *Let $\mathcal{C}$ be a cocomplete category. Then, the category $Dg(\mathcal{C})$ is also cocomplete.*

We split the proof of Theorem 10 in two separate propositions. In Proposition 11 we show that if $\mathcal{C}$ is cocomplete then $Dg(\mathcal{C})$ has pushouts, and in Proposition 12 we show that $Dg(\mathcal{C})$ has coproducts. Since a category with pushouts and coproducts has all colimits, the cocompleteness of $Dg(\mathcal{C})$ follows from these two propositions. The detailed proof of Proposition 11 gives a very concrete and precise construction for pushouts of structured theories. Such a construction is useful as a precise specification that can then be used as a correctness criteria for implementation purposes.

Proposition 11 *If $\mathcal{C}$ is cocomplete, then $Dg(\mathcal{C})$ has pushouts.*

Proof
Given small categories $\mathcal{P}_0$, $\mathcal{P}_1$, and $\mathcal{P}_2$, diagrams D_0 in $\mathcal{C}^{\mathcal{P}_0}$, D_1 in $\mathcal{C}^{\mathcal{P}_1}$, and D_2 in $\mathcal{C}^{\mathcal{P}_2}$, and diagram morphisms $(F, \alpha) : D_0 \to D_1$ and $(J, \beta) : D_0 \to D_2$, we need to construct a pushout object D_3 in $\mathcal{C}^{\mathcal{P}_3}$, and corresponding morphisms in

$Dg(\mathcal{C})$, as depicted in the following figure:

$$\begin{tikzcd}
D_2 \arrow[r, dotted, "(F',\delta)"] & D_3 \\
D_0 \arrow[u, "(J,\beta)"] \arrow[r, "(F,\alpha)"'] & D_1 \arrow[u, dotted, "(J',\gamma)"']
\end{tikzcd}$$

Going into some detail, this figure can also be depicted as this other equivalent one:

$$\begin{tikzcd}
\mathcal{P}_2 \arrow[rr, dotted, "F'"] \arrow[dr, "D_2"] & \overset{\delta}{\Rightarrow} & \mathcal{P}_3 \arrow[dl, dotted, "D_3"'] \\
\arrow[r, phantom, "\beta\Uparrow"] & \mathcal{C} & \arrow[l, phantom, "\gamma\Uparrow"] \\
\mathcal{P}_0 \arrow[uu, "J"] \arrow[ur, "D_0"'] \arrow[rr, "F"'] & \overset{\alpha}{\Rightarrow} & \mathcal{P}_1 \arrow[ul, "D_1"] \arrow[uu, dotted, "J'"']
\end{tikzcd}$$

First, we define the small category $\mathcal{P}_3$, with $J' : \mathcal{P}_1 \to \mathcal{P}_3$ and $F' : \mathcal{P}_2 \to \mathcal{P}_3$, as the pushout of F and J in **Cat**.

The intuitive idea in order to build up the diagram D_3 is the following: if the D_i were all in some category $\mathcal{C}^{\mathcal{P}}$, that is, if all the functors were taken over the same small category $\mathcal{P}$, then the pushout could be constructed pointwise. Since we want to deal with this more general construction, we proceed by taking the Kan extensions for each of these diagrams, "moving" them to $\mathcal{C}^{\mathcal{P}_3}$, and then taking there the pushout pointwise.

The morphism $F : \mathcal{P}_0 \to \mathcal{P}_1$ gives us the forgetful functor $\tilde{F} : \mathcal{C}^{\mathcal{P}_1} \to \mathcal{C}^{\mathcal{P}_0}$ by simply composing with F, that is, $\tilde{F}D_1 = D_1 \cdot F$. By Theorem 7, there exists a left adjoint to $\tilde{F}$, denoted $\tilde{V}_F$. The adjunction is then given as

$$\langle \tilde{V}_F,\ \tilde{F},\ \eta^F,\ \epsilon^F \rangle :\ \mathcal{C}^{\mathcal{P}_0} \rightharpoonup \mathcal{C}^{\mathcal{P}_1},$$

where η^F and ϵ^F are, respectively, the unit and the counit. Given, for example, a diagram D_0 in $\mathcal{C}^{\mathcal{P}_0}$, we denote by $\eta^F_{D_0}$ the component of η^F corresponding to D_0.

By definition of $\tilde{F}$, the natural transformation $\alpha : D_0 \to D_1 \cdot F$ in $\mathcal{C}^{\mathcal{P}_0}$ can be rewritten as $\alpha : D_0 \to \tilde{F}D_1$, and, because of the adjunction $\tilde{V}_F \dashv \tilde{F}$, there is a unique arrow $\bar{\alpha} : \tilde{V}_F D_0 \to D_1$ in $\mathcal{C}^{\mathcal{P}_1}$ such that $\tilde{F}\bar{\alpha} \cdot \eta^F_{D_0} = \alpha$. Thus, we have

$$\begin{tikzcd}
 & \tilde{F}D_1 & \\
D_0 \arrow[ur, "\alpha"] \arrow[rr, "\eta^F_{D_0}"'] & = & \tilde{F}\tilde{V}_F D_0 \arrow[ul, "\tilde{F}\bar{\alpha}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \tilde{V}_F D_0 \arrow[dl, "\bar{\alpha}"'] \arrow[dr, "\tilde{V}_F\alpha"] & \\
D_1 & = & \tilde{V}_F\tilde{F}D_1 \arrow[ll, "\epsilon^F_{D_1}"]
\end{tikzcd}$$

The same reasoning with the morphism (J, β) gives the adjunction

$$\langle \tilde{V}_J,\ \tilde{J},\ \eta^J,\ \epsilon^J \rangle : \mathcal{C}^{\mathcal{P}_0} \rightharpoonup \mathcal{C}^{\mathcal{P}_2},$$

with $\tilde{V}_J$ the left adjoint to the forgetful functor $\tilde{J}$, having unit and counit η^J and ϵ^J, respectively. A natural transformation $\beta : D_0 \to \tilde{J}D_2$ in $\mathcal{C}^{\mathcal{P}_0}$ induces a unique $\bar{\beta}\colon \tilde{V}_J D_0 \to D_2$ in $\mathcal{C}^{\mathcal{P}_2}$ such that $\tilde{J}\bar{\beta} \cdot \eta^J_{D_0} = \beta$:

$$
\begin{tikzcd}
 & \tilde{J}D_2 & \\
D_0 \arrow[ur, "\beta"] \arrow[rr, "\eta^J_{D_0}"'] & {} \arrow[u, phantom, "="] & \tilde{J}\tilde{V}_J D_0 \arrow[ul, "\tilde{J}\bar{\beta}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \tilde{V}_J D_0 \arrow[dl, "\bar{\beta}"'] \arrow[dr, "\tilde{V}_J\beta"] & \\
D_2 & {} \arrow[u, phantom, "="] & \tilde{V}_J\tilde{J}D_2 \arrow[ll, "\epsilon^J_{D_2}"]
\end{tikzcd}
$$

Similarly, the functors F' and J' define forgetful functors $\tilde{F}'$ and $\tilde{J}'$, with $\tilde{F} \cdot \tilde{J}' = \tilde{J} \cdot \tilde{F}'$, because $F' \cdot J = J' \cdot F$. Among all the isomorphic adjoints to each one of them, we can choose $\tilde{V}_{F'}$ and $\tilde{V}_{J'}$ such that $\tilde{V}_{F'} \cdot \tilde{V}_J = \tilde{V}_{J'} \cdot \tilde{V}_F$. So, the following two square diagrams, one of forgetful functors and another of their left adjoints, commute:

$$
\begin{tikzcd}
\mathcal{C}^{\mathcal{P}_2} \arrow[r, shift left, "\tilde{V}_{F'}"] \arrow[d, shift left, "\tilde{J}"] & \mathcal{C}^{\mathcal{P}_3} \arrow[l, shift left, "\tilde{F}'"] \arrow[d, shift right, "\tilde{J}'"'] \\
\mathcal{C}^{\mathcal{P}_0} \arrow[r, shift right, "\tilde{V}_F"'] \arrow[u, shift left, "\tilde{V}_J"] & \mathcal{C}^{\mathcal{P}_1} \arrow[l, shift right, "\tilde{F}"'] \arrow[u, shift right, "\tilde{V}_{J'}"']
\end{tikzcd}
$$

For the new adjunctions $\tilde{V}_{F'} \dashv \tilde{F}'$ and $\tilde{V}_{J'} \dashv \tilde{J}'$, we can choose their units in such a way that the units of the two composite adjunctions[1] $\tilde{V}_{F'} \cdot \tilde{V}_J \dashv \tilde{J} \cdot \tilde{F}'$ and $\tilde{V}_{J'} \cdot \tilde{V}_F \dashv \tilde{F} \cdot \tilde{J}'$ are identical. Therefore, given the new adjunctions

$$\langle \tilde{V}_{F'},\ \tilde{F}',\ \eta^{F'},\ \epsilon^{F'} \rangle : \mathcal{C}^{\mathcal{P}_2} \rightharpoonup \mathcal{C}^{\mathcal{P}_3} \text{ and}$$

$$\langle \tilde{V}_{J'},\ \tilde{J}',\ \eta^{J'},\ \epsilon^{J'} \rangle : \mathcal{C}^{\mathcal{P}_1} \rightharpoonup \mathcal{C}^{\mathcal{P}_3},$$

we have composite adjunctions

$$\langle \tilde{V}_{J'} \cdot \tilde{V}_F,\ \tilde{F} \cdot \tilde{J}',\ \tilde{F}\eta^{J'}_{\tilde{V}_F} \cdot \eta^F,\ \epsilon^{J'} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J}'} \rangle : \mathcal{C}^{\mathcal{P}_0} \rightharpoonup \mathcal{C}^{\mathcal{P}_3} \text{ and}$$

$$\langle \tilde{V}_{F'} \cdot \tilde{V}_J,\ \tilde{J} \cdot \tilde{F}',\ \tilde{J}\eta^{F'}_{\tilde{V}_J} \cdot \eta^J,\ \epsilon^{F'} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F}'} \rangle : \mathcal{C}^{\mathcal{P}_0} \rightharpoonup \mathcal{C}^{\mathcal{P}_3},$$

with $\tilde{F} \cdot \tilde{J}' = \tilde{J} \cdot \tilde{F}'$, $\tilde{V}_{J'} \cdot \tilde{V}_F = \tilde{V}_{F'} \cdot \tilde{V}_J$,

$$\tilde{F}\eta^{J'}_{\tilde{V}_F} \cdot \eta^F = \tilde{J}\eta^{F'}_{\tilde{V}_J} \cdot \eta^J, \text{ and}$$

$$\epsilon^{J'} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J}'} = \epsilon^{F'} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F}'}.$$

Applying $\tilde{V}_{J'}$ to $\bar{\alpha}$, we can *move* $\bar{\alpha}$ to $\mathcal{C}^{\mathcal{P}_3}$, obtaining the morphism

$$\tilde{V}_{J'}\bar{\alpha}\colon \tilde{V}_{J'}\tilde{V}_F D_0 \to \tilde{V}_{J'} D_1.$$

We can, similarly, *move* $\bar{\beta}$ to $\mathcal{C}^{\mathcal{P}_3}$, obtaining the morphism

$$\tilde{V}_{F'}\bar{\beta}\colon \tilde{V}_{F'}\tilde{V}_J D_0 \to \tilde{V}_{F'} D_2.$$

---

[1](From [70], page 101) Given two adjunctions

$$\langle F,\ G,\ \eta,\ \epsilon \rangle : X \rightharpoonup A, \qquad \langle \bar{F},\ \bar{G},\ \bar{\eta},\ \bar{\epsilon} \rangle : A \rightharpoonup D,$$

the composite functors yield an adjunction

$$\langle \bar{F} \cdot F,\ G \cdot \bar{G},\ G\bar{\eta}_F \cdot \eta,\ \bar{\epsilon} \cdot \bar{F}\eta_{\bar{G}} \rangle : X \rightharpoonup D.$$

Since $\tilde{V}_{F'}\tilde{V}_J D_0 = \tilde{V}_{J'}\tilde{V}_F D_0$, we can take the pushout of $\tilde{V}_{J'}\bar{\alpha}$ and $\tilde{V}_{F'}\bar{\beta}$ in $\mathcal{C}^{\mathcal{P}_3}$, which, since $\mathcal{C}$ is cocomplete, can be constructed pointwise [70].

$$\begin{tikzcd}
\tilde{V}_{F'}D_2 \arrow[rr, dotted, "\bar{\delta}"] & \arrow[d, phantom, "p.o."] & D_3 \\
\tilde{V}_{F'}\tilde{V}_J D_0 = \tilde{V}_{J'}\tilde{V}_F D_0 \arrow[u, "\tilde{V}_{F'}\bar{\beta}"] \arrow[rr, "\tilde{V}_{J'}\bar{\alpha}"'] & & \tilde{V}_{J'}D_1 \arrow[u, dotted, "\bar{\gamma}"']
\end{tikzcd}$$

Given these $\bar{\delta}\colon \tilde{V}_{F'}D_2 \to D_3$ and $\bar{\gamma}\colon \tilde{V}_{J'}D_1 \to D_3$ in $\mathcal{C}^{\mathcal{P}_3}$, there are unique arrows $\delta\colon D_2 \to \tilde{F}'D_3$ in $\mathcal{C}^{\mathcal{P}_2}$, and $\gamma\colon D_1 \to \tilde{J}'D_3$ in $\mathcal{C}^{\mathcal{P}_1}$, such that $\epsilon^{F'}_{D_3} \cdot \tilde{V}_{F'}\delta = \bar{\delta}$ and $\epsilon^{J'}_{D_3} \cdot \tilde{V}_{J'}\gamma = \bar{\gamma}$.

$$\begin{tikzcd}
 & \tilde{V}_{F'}D_2 \arrow[dl, "\bar{\delta}"'] \arrow[dr, "\tilde{V}_{F'}\delta"] \arrow[d, phantom, "="] & \\
D_3 & & \tilde{V}_{F'}\tilde{F}'D_3 \arrow[ll, "\epsilon^{F'}_{D_3}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \tilde{V}_{J'}D_1 \arrow[dl, "\bar{\gamma}"'] \arrow[dr, "\tilde{V}_{J'}\gamma"] \arrow[d, phantom, "="] & \\
D_3 & & \tilde{V}_{J'}\tilde{J}'D_3 \arrow[ll, "\epsilon^{J'}_{D_3}"]
\end{tikzcd}$$

We claim that this object D_3 in $\mathcal{C}^{\mathcal{P}_3}$ together with the morphisms (F', δ) and (J', γ) constitutes a pushout of (F, α) and (J, β). To prove this claim, we will show first that the square diagram commutes, and then that it is universal.

Commutativity.

Given the adjunction $\tilde{V}_{J'} \cdot \tilde{V}_F \dashv \tilde{F} \cdot \tilde{J}'$ with counit $\epsilon^{J'} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J}'}$, and given the morphism $\bar{\gamma} \cdot \tilde{V}_{J'}\bar{\alpha} : \tilde{V}_{J'}\tilde{V}_F D_0 \to D_3$ in $\mathcal{C}^{\mathcal{P}_3}$, there is a unique morphism $\tilde{F}\gamma \cdot \alpha\colon D_0 \to \tilde{F}\tilde{J}'D_3$ in $\mathcal{C}^{\mathcal{P}_0}$ with $\epsilon^{J'}_{D_3} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J}'D_3} \cdot \tilde{V}_{J'}\tilde{V}_F(\tilde{F}\gamma \cdot \alpha) = \bar{\gamma} \cdot \tilde{V}_{J'}\bar{\alpha}$.

$$\begin{tikzcd}
 & & \tilde{V}_{J'}\tilde{V}_F D_0 \arrow[dl, "\tilde{V}_{J'}\tilde{V}_F\alpha"'] \arrow[dr, "\tilde{V}_{J'}\bar{\alpha}"] \arrow[d, phantom, "="] & & \\
 & \tilde{V}_{J'}\tilde{V}_F\tilde{F}D_1 \arrow[rr, "\tilde{V}_{J'}\epsilon^F_{D_1}"] \arrow[dl, "\tilde{V}_{J'}\tilde{V}_F\tilde{F}\gamma"'] & \arrow[d, phantom, "="] & \tilde{V}_{J'}D_1 \arrow[dl, "\tilde{V}_{J'}\gamma"'] \arrow[dr, "\bar{\gamma}"] \arrow[d, phantom, "="] & \\
\tilde{V}_{J'}\tilde{V}_F\tilde{F}\tilde{J}'D_3 \arrow[rr, "\tilde{V}_{J'}\epsilon^F_{\tilde{J}'D_3}"] & & \tilde{V}_{J'}\tilde{J}'D_3 \arrow[rr, "\epsilon^{J'}_{D_3}"] & & D_3
\end{tikzcd}$$

In the same way, given the adjunction $\tilde{V}_{F'} \cdot \tilde{V}_J \dashv \tilde{J} \cdot \tilde{F}'$ with counit $\epsilon^{F'} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F}'}$, and given the morphism $\bar{\delta} \cdot \tilde{V}_{F'}\bar{\beta} : \tilde{V}_{F'}\tilde{V}_J D_0 \to D_3$ in $\mathcal{C}^{\mathcal{P}_3}$, there is a unique morphism $\tilde{J}\delta \cdot \beta\colon D_0 \to \tilde{J}\tilde{F}'D_3$ in $\mathcal{C}^{\mathcal{P}_0}$ with $\epsilon^{F'}_{D_3} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F}'D_3} \cdot \tilde{V}_{F'}\tilde{V}_J(\tilde{J}\delta \cdot \beta) = \bar{\delta} \cdot \tilde{V}_{F'}\bar{\beta}$.

$$\begin{tikzcd}
 & & \tilde{V}_{F'}\tilde{V}_J D_0 \arrow[dl, "\tilde{V}_{F'}\tilde{V}_J\beta"'] \arrow[dr, "\tilde{V}_{F'}\bar{\beta}"] \arrow[d, phantom, "="] & & \\
 & \tilde{V}_{F'}\tilde{V}_J\tilde{J}D_2 \arrow[rr, "\tilde{V}_{F'}\epsilon^J_{D_2}"] \arrow[dl, "\tilde{V}_{F'}\tilde{V}_J\tilde{J}\delta"'] & \arrow[d, phantom, "="] & \tilde{V}_{F'}D_2 \arrow[dl, "\tilde{V}_{F'}\delta"'] \arrow[dr, "\bar{\delta}"] \arrow[d, phantom, "="] & \\
\tilde{V}_{F'}\tilde{V}_J\tilde{J}\tilde{F}'D_3 \arrow[rr, "\tilde{V}_{F'}\epsilon^J_{\tilde{F}'D_3}"] & & \tilde{V}_{F'}\tilde{F}'D_3 \arrow[rr, "\epsilon^{F'}_{D_3}"] & & D_3
\end{tikzcd}$$

We have chosen the left adjoints such that $\tilde{V}_{J'} \cdot \tilde{V}_F = \tilde{V}_{F'} \cdot \tilde{V}_J$, and the counits such that $\epsilon^{F'} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F}'} = \epsilon^{J'} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J}'}$; then, because $\bar{\delta} \cdot \tilde{V}_{F'}\bar{\beta} = \bar{\gamma} \cdot \tilde{V}_{J'}\bar{\alpha}$, these unique morphisms given by the adjunction must be the same one. Therefore, $\tilde{J}\delta \cdot \beta = \tilde{F}\gamma \cdot \alpha$.

$$\begin{tikzcd}
\tilde{J}D_2 \arrow[r, "\tilde{J}\delta"] & \tilde{J}\tilde{F}'D_3 \arrow[r, phantom, "="] & \tilde{F}\tilde{J}'D_3 \\
D_0 \arrow[u, "\beta"] \arrow[rr, "\alpha"'] \arrow[urr, phantom, "="] & & \tilde{F}D_1 \arrow[u, "\tilde{F}\gamma"']
\end{tikzcd}$$

Together with the fact that $F' \cdot J = F \cdot J'$ by the pushout construction, this means that our construction commutes, that is, $(F', \delta) \cdot (J, \beta) = (J', \gamma) \cdot (F, \alpha)$.

Universal Property.

We have to show that, for any diagram D_4 in some category $\mathcal{C}^{\mathcal{P}_4}$, with arrows $(K_1, \psi_1) : D_1 \to D_4$ and $(K_2, \psi_2) : D_2 \to D_4$, such that $(K_1, \psi_1) \cdot (F, \alpha) = (K_2, \psi_2) \cdot (J, \beta)$, there must exist a unique arrow $(K, \psi) : D_3 \to D_4$ such that $(K, \psi) \cdot (J', \gamma) = (K_1, \psi_1)$ and $(K, \psi) \cdot (F', \delta) = (K_2, \psi_2)$.

$$\begin{tikzcd}
 & & D_4 \\
D_2 \arrow[r, "(F',\delta)"'] \arrow[urr, bend left, "(K_2,\psi_2)"] & D_3 \arrow[ur, dotted, "(K,\psi)"] & \\
D_0 \arrow[u, "(J,\beta)"] \arrow[r, "(F,\alpha)"'] & D_1 \arrow[u, "(J',\gamma)"] \arrow[uur, bend right, "(K_1,\psi_1)"'] &
\end{tikzcd}$$

Note that the commutativity of the outer diagram is equivalent to assuming $K_1 \cdot F = K_2 \cdot J$ and $\tilde{F}\psi_1 \cdot \alpha = \tilde{J}\psi_2 \cdot \beta$, which can be displayed as

$$\begin{tikzcd}
\tilde{J}D_2 \arrow[r, "\tilde{J}\psi_2"] & \tilde{J}\tilde{K}_2D_4 \arrow[r, phantom, "="] & \tilde{F}\tilde{K}_1D_4 \\
D_0 \arrow[u, "\beta"] \arrow[rr, "\alpha"'] \arrow[urr, phantom, "="] & & \tilde{F}D_1 \arrow[u, "\tilde{F}\psi_1"']
\end{tikzcd}$$

We have to show that there exists a unique functor $K : \mathcal{P}_3 \to \mathcal{P}_4$ such that $K \cdot J' = K_1$ and $K \cdot F' = K_2$, and a unique natural transformation $\psi \colon D_3 \to D_4 \cdot K$ such that $\tilde{J}'\psi \cdot \gamma = \psi_1$ and $\tilde{F}'\psi \cdot \delta = \psi_2$.

Let us now prove the desired universal property. First, since $F' : \mathcal{P}_2 \to \mathcal{P}_3$ and $J' : \mathcal{P}_1 \to \mathcal{P}_3$ are a pushout of $F \colon \mathcal{P}_0 \to \mathcal{P}_1$ and $J \colon \mathcal{P}_0 \to \mathcal{P}_2$ in **Cat**, there exists a unique functor $K \colon \mathcal{P}_3 \to \mathcal{P}_4$ such that $K \cdot J' = K_1$ and $K \cdot F' = K_2$.

$$\begin{tikzcd}
 & & \mathcal{P}_4 \\
\mathcal{P}_2 \arrow[r, "F'"'] \arrow[urr, bend left, "K_2"] & \mathcal{P}_3 \arrow[ur, dotted, "K"] & \\
\mathcal{P}_0 \arrow[u, "J"] \arrow[r, "F"'] \arrow[ur, phantom, "p.o."] & \mathcal{P}_1 \arrow[u, "J'"'] \arrow[uur, bend right, "K_1"'] &
\end{tikzcd}$$

The functors K, K_1, and K_2 generate forgetful functors $\tilde{K}$, $\tilde{K_1}$, and $\tilde{K_2}$ with $\tilde{J'} \cdot \tilde{K} = \tilde{K_1}$ and $\tilde{F'} \cdot \tilde{K} = \tilde{K_2}$. We can choose their left adjoints (Kan extensions), $\tilde{V}_K, \tilde{V}_{K_1}$, and $\tilde{V}_{K_2}$, which exist because of Theorem 7, in such a way that $\tilde{V}_J \cdot \tilde{V}_{F'} = \tilde{V}_{K_1}$ and $\tilde{V}_K \cdot \tilde{V}_{F'} = \tilde{V}_{K_2}$.

Since $\tilde{J}\psi_2 \cdot \beta = \tilde{F}\psi_1 \cdot \alpha$ in $\mathcal{C}^{\mathcal{P}_0}$, and $\tilde{V}_{F'} \cdot \tilde{V}_J = \tilde{V}_{J'} \cdot \tilde{V}_F$, we then have $\tilde{V}_{F'}\tilde{V}_J(\tilde{J}\psi_2 \cdot \beta) = \tilde{V}_{J'}\tilde{V}_F(\tilde{F}\psi_1 \cdot \alpha)$. This equality, together with the naturality of ϵ^F and ϵ^J and the way in which the counits of $\tilde{V}_{F'}$ and $\tilde{V}_{J'}$ were chosen so that $\epsilon^{J'} \cdot \tilde{V}_{J'}\epsilon^F_{\tilde{J'}} = \epsilon^{F'} \cdot \tilde{V}_{F'}\epsilon^J_{\tilde{F'}}$, gives us the following commutative diagram in $\mathcal{C}^{\mathcal{P}_3}$.

$$\begin{tikzcd}
\tilde{V}_{F'}D_2 \arrow[r, "\tilde{V}_{F'}\psi_2"] & \tilde{V}_{F'}\tilde{K}_2D_4 \arrow[r, phantom, "="] & \tilde{V}_{F'}\tilde{F'}\tilde{K}D_4 \arrow[r, "\epsilon^{F'}_{\tilde{K}D_4}"] & \tilde{K}D_4 \\
\tilde{V}_{F'}\tilde{V}_J\tilde{J}D_2 \arrow[u, "\tilde{V}_{F'}\epsilon^J_{D_2}"'] \arrow[r, "\tilde{V}_{F'}\tilde{V}_J\tilde{J}\psi_2"'] & \tilde{V}_{F'}\tilde{V}_J\tilde{J}\tilde{K}_2D_4 \arrow[r, phantom, "="] & \tilde{V}_{F'}\tilde{V}_J\tilde{J}\tilde{F'}\tilde{K}D_4 \arrow[u, "\tilde{V}_{F'}\epsilon^J_{\tilde{F'}\tilde{K}D_4}"'] & \\
 & & \tilde{V}_{J'}\tilde{V}_F\tilde{F}\tilde{J'}\tilde{K}D_4 \arrow[u, equal] \arrow[r, "\tilde{V}_{J'}\epsilon^F_{\tilde{J'}\tilde{K}D_4}"] & \tilde{V}_{J'}\tilde{J'}\tilde{K}D_4 \arrow[uu, "\epsilon^{J'}_{\tilde{K}D_4}"'] \\
 & & \tilde{V}_{J'}\tilde{V}_F\tilde{F}\tilde{K}_1D_4 \arrow[u, equal] & \tilde{V}_{J'}\tilde{K}_1D_4 \arrow[u, equal] \\
\tilde{V}_{F'}\tilde{V}_JD_0 \arrow[uuu, "\tilde{V}_{F'}\tilde{V}_J\beta"'] \arrow[uuuu, bend left, "\tilde{V}_{F'}\bar{\beta}"] & & & \\
\tilde{V}_{J'}\tilde{V}_FD_0 \arrow[u, equal] \arrow[rr, "\tilde{V}_{J'}\tilde{V}_F\alpha"] \arrow[rrr, bend right, "\tilde{V}_{J'}\bar{\alpha}"'] & & \tilde{V}_{J'}\tilde{V}_F\tilde{F}D_1 \arrow[uu, "\tilde{V}_{J'}\tilde{V}_F\tilde{F}\psi_1"] \arrow[r, "\tilde{V}_{J'}\epsilon^F_{D_1}"] & \tilde{V}_{J'}D_1 \arrow[uu, "\tilde{V}_{J'}\psi_1"']
\end{tikzcd}$$

By pasting all the pieces of this commutative diagram, we obtain

$$\epsilon^{F'}_{\tilde{K}D_4} \cdot \tilde{V}_{F'}\psi_2 \cdot \tilde{V}_{F'}\bar{\beta} = \epsilon^{J'}_{\tilde{K}D_4} \cdot \tilde{V}_{J'}\psi_1 \cdot \tilde{V}_{J'}\bar{\alpha}.$$

Since by construction $\bar{\delta}$ and $\bar{\gamma}$ are a pushout of $\tilde{V}_{F'}\bar{\beta}$ and $\tilde{V}_{J'}\bar{\alpha}$ in $\mathcal{C}^{\mathcal{P}_3}$, then there exists a unique morphism $\psi \colon D_3 \to \tilde{K}D_4$ such that $\epsilon^{F'}_{\tilde{K}D_4} \cdot \tilde{V}_{F'}\psi_2 = \psi \cdot \bar{\delta}$ and $\epsilon^{J'}_{\tilde{K}D_4} \cdot \tilde{V}_{J'}\psi_1 = \psi \cdot \bar{\gamma}$.

$$\begin{tikzcd}
 & & & \tilde{K}D_4 \\
\tilde{V}_{F'}D_2 \arrow[rr, "\bar{\delta}"'] \arrow[rrru, bend left, "\epsilon^{F'}_{\tilde{K}D_4} \cdot \tilde{V}_{F'}\psi_2"] & & D_3 \arrow[ur, dotted, "\psi"] & \\
\tilde{V}_{F'}\tilde{V}_JD_0 \arrow[u, "\tilde{V}_{F'}\bar{\beta}"] \arrow[r, phantom, "="] \arrow[urr, phantom, "p.o."] & \tilde{V}_{J'}\tilde{V}_FD_0 \arrow[r, "\tilde{V}_{J'}\bar{\alpha}"'] & \tilde{V}_{J'}D_1 \arrow[u, "\bar{\gamma}"'] \arrow[uur, bend right, "\epsilon^{J'}_{\tilde{K}D_4} \cdot \tilde{V}_{J'}\psi_1"'] &
\end{tikzcd}$$

We have to check that this natural transformation ψ, together with the functor K, is indeed our desired universal morphism. By applying $\tilde{F'}$ to the equality $\epsilon^{F'}_{\tilde{K}D_4} \cdot \tilde{V}_{F'}\psi_2 = \psi \cdot \bar{\delta}$, and by the naturality of $\eta^{F'}$, we obtain the following commutative diagram.

$$
\begin{tikzcd}
\tilde K_2 D_4 = \tilde F'\tilde K D_4 \arrow[r, "\eta^{F'}_{\tilde F'\tilde K D_4}"'] \arrow[rr, bend left, "id_{\tilde F'\tilde K D_4}"] & \tilde F'\tilde V_{F'}\tilde K_2 D_4 = \tilde F'\tilde V_{F'}\tilde F'\tilde K D_4 \arrow[r, "\tilde F'\epsilon^{F'}_{\tilde K D_4}"'] & \tilde F'\tilde K D_4 \\
D_2 \arrow[u, "\psi_2"] \arrow[r, "\eta^{F'}_{D_2}"] \arrow[rr, bend right, "\delta"'] & \tilde F'\tilde V_{F'} D_2 \arrow[u, "\tilde F'\tilde V_{F'}\psi_2"'] \arrow[r, "\tilde F'\bar\delta"] & \tilde F' D_3 \arrow[u, "\tilde F'\psi"']
\end{tikzcd}
$$

Therefore, by pasting of commutative diagrams, we obtain $\tilde F'\psi \cdot \delta = \psi_2$.

Similarly, by applying $\tilde J'$ to $\epsilon^{J'}_{\tilde K D_4} \cdot \tilde V_{J'}\psi_1 = \psi \cdot \bar\gamma$, and by naturality of $\eta^{J'}$, we obtain the following commutative diagram.

$$
\begin{tikzcd}
\tilde J' D_3 \arrow[r, "\tilde J'\psi"] & \tilde J'\tilde K D_4 \\
\tilde J'\tilde V_{J'} D_1 \arrow[u, "\tilde J'\bar\gamma"'] \arrow[r, "\tilde J'\tilde V_{J'}\psi_1"] & \tilde J'\tilde V_{J'}\tilde K_1 D_4 = \tilde J'\tilde V_{J'}\tilde J'\tilde K D_4 \arrow[u, "\tilde J'\epsilon^{J'}_{\tilde K D_4}"] \\
D_1 \arrow[u, "\eta^{J'}_{D_1}"'] \arrow[r, "\psi_1"] \arrow[uu, bend left=60, "\gamma"] & \tilde K_1 D_4 = \tilde J'\tilde K D_4 \arrow[u, "\epsilon^{J'}_{\tilde J'\tilde K D_4}"] \arrow[uu, bend right=60, "id_{\tilde J'\tilde K D_4}"']
\end{tikzcd}
$$

By pasting of commutative diagrams, we obtain the equality $\tilde J'\psi \cdot \gamma = \psi_1$.

Thus, we have a natural transformation $\psi\colon D_3 \to \tilde K D_4$ such that $\tilde F'\psi \cdot \delta = \psi_2$ and $\tilde J'\psi \cdot \gamma = \psi_1$. We still have to show that this ψ is unique satisfying these conditions. Let us assume the existence of another morphism $\xi : D_3 \to \tilde K D_4$ with $\tilde F'\xi \cdot \delta = \psi_2$ and $\tilde J'\xi \cdot \gamma = \psi_1$.

Moving $\tilde J\bar\delta \cdot \beta = \tilde F\bar\gamma \cdot \alpha$ to $\mathcal{C}^{\mathcal{P}_3}$, since $\epsilon^{F'} \cdot \tilde V_{F'}\epsilon^{J}_{\tilde F'} = \epsilon^{J'} \cdot \tilde V_{J'}\epsilon^{F}_{\tilde J'}$, and using the naturality of $\epsilon^{F'}$ and ϵ^{F}, we obtain the commutative diagram of Figure 2.1.

Again, by pasting of commutative diagrams, we have that there are two morphisms $\psi, \xi\colon D_3 \to \tilde K D_4$ with $\epsilon^{F'}_{\tilde K D_4} \cdot \tilde V_{F'}\psi_2 = \psi \cdot \bar\delta$ and $\epsilon^{J'}_{\tilde K D_4} \cdot \tilde V_{J'}\psi_1 = \psi \cdot \bar\gamma$, and $\epsilon^{F'}_{\tilde K D_4} \cdot \tilde V_{F'}\psi_2 = \xi \cdot \bar\delta$ and $\epsilon^{J'}_{\tilde K D_4} \cdot \tilde V_{J'}\psi_1 = \xi \cdot \bar\gamma$. Since $\bar\delta$ and $\bar\gamma$ are a pushout of $\tilde V_{F'}\bar\beta$ and $\tilde V_{J'}\bar\alpha$, ψ and ξ must be identical. This means that our construction satisfies the universal property, with (K, ψ) the universal morphism as desired.
□

Proposition 12 *For any category $\mathcal{C}$, $Dg(\mathcal{C})$ has coproducts.*

Proof

Let $\{D_i : J_i \to \mathcal{C}\}_{i \in I}$ be an I-indexed family of diagrams for any set I. Let $\coprod_{i \in I} J_i$ be the coproduct of $\{J_i\}_{i \in I}$ in **Cat**, and let us denote the inclusion morphisms by $\iota_i\colon J_i \to \coprod_{i \in I} J_i$.

Note that for $I = \emptyset$, we have that $Dg(\mathcal{C})$ has an initial object. This initial object is the unique functor $\emptyset \to \mathcal{C}$ with $\emptyset$ the empty category with no objects and no arrows.

We claim that the induced functor $D\colon \coprod_{i \in I} J_i \to \mathcal{C}$, with the induced family of inclusion morphisms $\{(\iota_i, 1^{D_i}) : D_i \to D \cdot \iota_i\}_{i \in I}$, is the coproduct of the indexed family of diagrams $\{D_i\}_{i \in I}$ in $Dg(\mathcal{C})$.

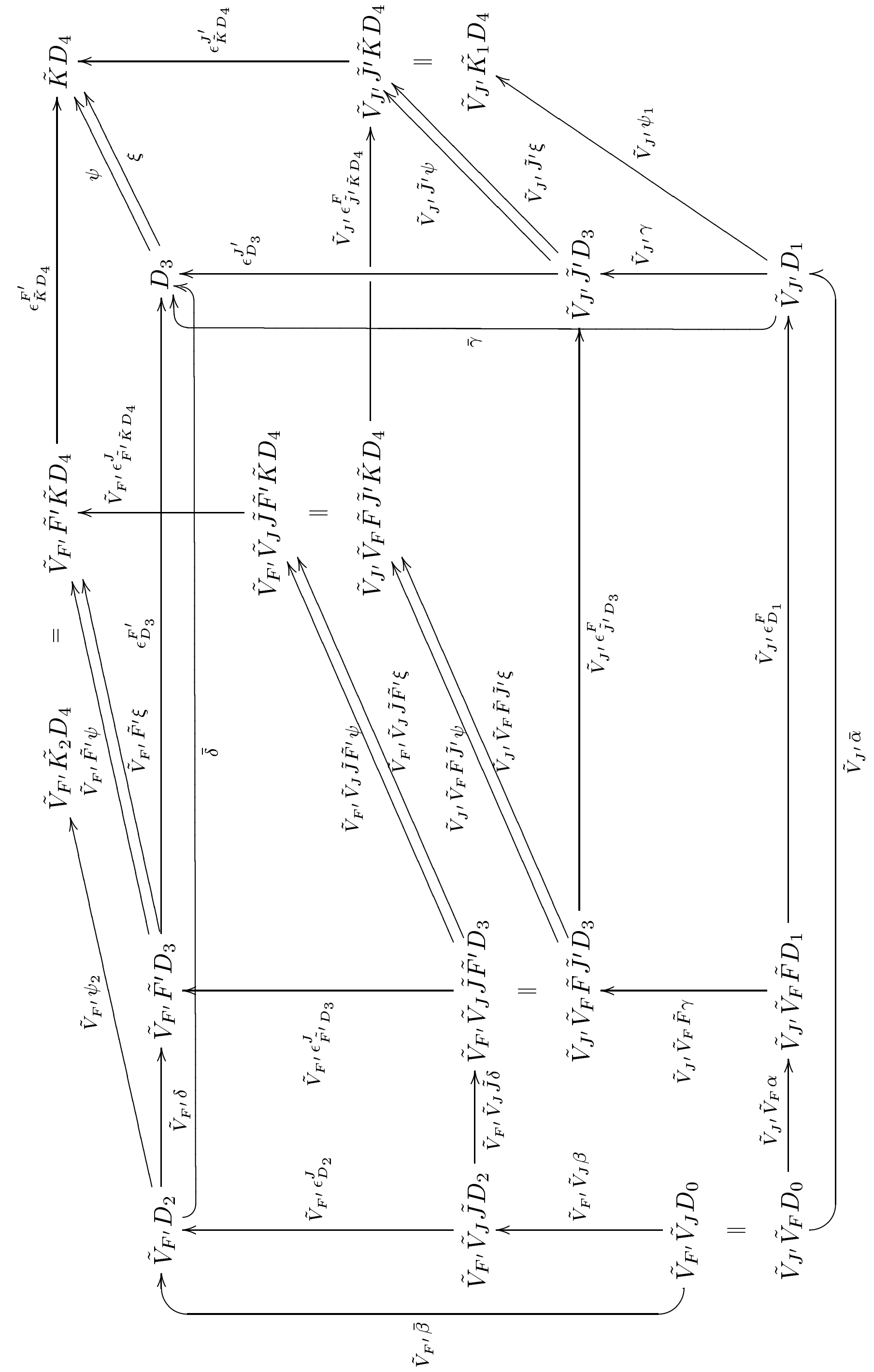


Figure 2.1: ψ is equal to ξ by uniqueness of the universal morphism.

$$\begin{tikzcd}
J_i \arrow[rr, "\iota_i"] \arrow[dr, "D_i"'] & \arrow[d, phantom, "\overset{1^{D_i}}{\Rightarrow}"] & \coprod_{i \in I} J_i \arrow[dl, dashed, "D"] \\
& \mathcal{C} &
\end{tikzcd}$$

Given a diagram $E : \Lambda \to \mathcal{C}$ for some small category Λ, with an indexed family of morphisms $\{(K_i, \sigma^i) \colon D_i \to E\}_{i \in I}$, we have to show that there is a unique diagram morphism $(K, \sigma) \colon D \to E$ such that $(K, \sigma) \cdot (\iota_i, 1^{D_i}) = (K_i, \sigma^i)$ for all $i \in I$.

Since $\coprod_{i \in I} J_i$ is the coproduct of the family $\{J_i\}_{i \in I}$, then there is a unique morphism $K \colon \coprod_{i \in I} J_i \to \Lambda$ making $K \cdot \iota_i = K_i$.

Let the construction of $\coprod_{i \in I} J_i$ be given by objects (i, a) for $a \in J_i$, and arrows $(i, f) : (i, a) \to (i, b)$ for $f : a \to b$ in J_i. Then, we have $D(i, a) = D_i(a)$, and $E(K(i, a)) = E(K_i(a))$. Similarly, we have $D(i, f) = D_i(f)$, and $E(K(i, f)) = E(K_i(f))$.

Let us define the natural transformation $\sigma \colon D \to E \cdot K$ by choosing

$$\sigma_{(i,a)} = \sigma^i_a \colon D(i, a) \to E(K(i, a))$$

for each $(i, a) \in \coprod_{i \in I} J_i$. Its naturality follows from the naturality of the natural transformations $\{\sigma^i \colon D_i \to E \cdot K_i\}_{i \in I}$. Indeed, note that, since $K_i = K \cdot \iota_i$, for each morphism $(i, f) \colon (i, a) \to (i, b)$ in $\coprod_{i \in I} J_i$, the diagram

$$\begin{tikzcd}
D(i,a) \arrow[r, "\sigma_{(i,a)}"] \arrow[d, "D(i,f)"'] & E(K(i,a)) \arrow[d, "E(K(i,f))"] \\
D(i,b) \arrow[r, "\sigma_{(i,b)}"'] & E(K(i,b))
\end{tikzcd}$$

commutes because it coincides with the diagram

$$\begin{tikzcd}
D_i(a) \arrow[r, "\sigma^i_a"] \arrow[d, "D_i(f)"'] & E(K_i(a)) \arrow[d, "E(K_i(f))"] \\
D_i(b) \arrow[r, "\sigma^i_b"'] & E(K_i(b))
\end{tikzcd}$$

which commutes by the naturality of σ^i.

Furthermore, by construction we have $\sigma \cdot 1^{D_i} = \sigma^i$ for all $i \in I$.

Let us see now that it is unique. Assume another natural transformation $\theta : D \to E \cdot K$ such that $\theta \cdot 1^{D_i} = \sigma^i$ for all $i \in I$. Since we must have $\sigma^i_a = \theta_{(i,a)} \cdot 1^{D_i}_a$ for all $(i, a) \in \coprod_{i \in I} J_i$, θ must be defined as

$$\theta_{(i,a)} = \sigma^i_a \colon D(i, a) \to E(K(i, a))$$

for all $(i, a) \in \coprod_{i \in I} J_i$, and therefore $\sigma = \theta$. □

2.3 Preservation of Colimits

The main result of this section is given in Theorem 16, where we show that given a colimit-preserving functor $U : \mathcal{C} \to \mathcal{A}$, with $\mathcal{C}$ and $\mathcal{A}$ cocomplete categories, then the induced functor $Dg(U)\colon Dg(\mathcal{C}) \to Dg(\mathcal{A})$ also preserves colimits. Since any colimit can be constructed out of pushouts and coproducts, the preservation of colimits follows from showing the preservation of pushouts and coproducts. We begin by introducing the definition of preservation of colimits by a functor, and by giving some previous results.

Definition 13 *Let $\mathcal{J}$ be a small category. A functor $U : \mathcal{C} \to \mathcal{A}$ is said to* preserve the $\mathcal{J}$-colimits *of functors $F : \mathcal{J} \to \mathcal{C}$ when every colimiting cone $\alpha\colon F \dot{\to} c$ in $\mathcal{C}$ for a functor F yields by composition with U a colimiting cone $U \cdot \alpha\colon U \cdot F \dot{\to} U \cdot c$ in $\mathcal{D}$.*

A functor $U : \mathcal{C} \to \mathcal{A}$ preserves colimits *if it preserves all $\mathcal{J}$-colimits for every $\mathcal{J}$.*

We say that a functor $U\colon \mathcal{C} \to \mathcal{A}$ preserves pushouts *when it preserves the colimits of functors $F\colon \mathcal{J} \to \mathcal{C}$ where $\mathcal{J}$ is the category with three objects whose nonidentity arrows are arranged as* $\begin{smallmatrix}\bullet & \\ \uparrow & \\ \bullet & \to \bullet\end{smallmatrix}$.

We say that a functor $U\colon \mathcal{C} \to \mathcal{A}$ preserves coproducts *when it preserves the colimits of functors $F\colon \mathcal{J} \to \mathcal{C}$ where $\mathcal{J}$ is a small category whose only arrows are identities.*

Note that this requires not only that U takes each colimit object which exists in $\mathcal{C}$ to a colimit object in $\mathcal{A}$, but also that U takes colimiting cones to colimiting cones[2].

Let $F\colon \mathcal{B} \to \mathcal{D}$ and $G\colon \mathcal{B} \to \mathcal{C}$ be functors, with $\mathcal{B}$ small and $\mathcal{C}$ cocomplete. Let $U\colon \mathcal{C} \to \mathcal{A}$ be a functor which preserves colimits and let $\mathcal{A}$ be cocomplete. The functor $F\colon \mathcal{B} \to \mathcal{D}$ induces a functor $\tilde{F}\colon \mathcal{C}^{\mathcal{D}} \to \mathcal{C}^{\mathcal{B}}$ by the rule $T \mapsto T \cdot F$, $\varrho \mapsto \varrho \cdot F$, for T and ϱ, respectively, an object and a morphism in $\mathcal{C}^{\mathcal{D}}$. Similarly, F induces another functor $\hat{F}\colon \mathcal{A}^{\mathcal{D}} \to \mathcal{A}^{\mathcal{B}}$ by the rule $T' \mapsto T' \cdot F$, $\varrho' \mapsto \varrho' \cdot F$, for T' and ϱ', respectively, an object and a morphism in $\mathcal{A}^{\mathcal{D}}$.

Let $\tilde{V}_F\colon \mathcal{C}^{\mathcal{B}} \to \mathcal{C}^{\mathcal{D}}$ be a left adjoint to $\tilde{F}\colon \mathcal{C}^{\mathcal{D}} \to \mathcal{C}^{\mathcal{B}}$, and let $\hat{V}_F\colon \mathcal{A}^{\mathcal{B}} \to \mathcal{A}^{\mathcal{D}}$ be a left adjoint to $\hat{F}\colon \mathcal{A}^{\mathcal{D}} \to \mathcal{A}^{\mathcal{B}}$, both of which exist by Theorem 7. Since $\tilde{F}$ and $\hat{F}$ are both defined "by composing with F on the right," we have $\hat{F}(U \cdot H) = U \cdot \tilde{F}(H)$ for any object H of $\mathcal{C}^{\mathcal{D}}$. We claim that we can choose $\hat{V}_F$ and $\tilde{V}_F$ such that we also have $\hat{V}_F(U \cdot G) = U \cdot \tilde{V}_F(G)$ for each object G of $\mathcal{C}^{\mathcal{B}}$. The situation can be depicted as follows.

$$\begin{tikzcd}
\mathcal{B} \arrow[rr, "F"] \arrow[dr, "G"'] & & \mathcal{D} \arrow[dl, dashed, "\tilde{V}_F(G)"'] \arrow[ddl, dashed, "\hat{V}_F(U\cdot G)"] \\
& \mathcal{C} \arrow[d, "U"'] & \\
& \mathcal{A} &
\end{tikzcd}$$

Proposition 14 *Let $F\colon \mathcal{B} \to \mathcal{D}$, $G\colon \mathcal{B} \to \mathcal{C}$, and $U\colon \mathcal{C} \to \mathcal{A}$ be functors, with $\mathcal{B}$ small, $\mathcal{C}$ and $\mathcal{A}$ cocomplete, and U preserving colimits. Let $\tilde{V}_F(G) : \mathcal{D} \to \mathcal{C}$ be the left Kan extension of G along F and let $\hat{V}_F(U \cdot G) : \mathcal{D} \to \mathcal{A}$ be the left Kan extension of $U \cdot G$ along F. Then, $\hat{V}_F(U \cdot G) \simeq U \cdot \tilde{V}_F(G)$, and $\hat{V}_F$ can be chosen such that $\hat{V}_F(U \cdot G) = U \cdot \tilde{V}_F(G)$.*

---

[2]In [53, page 108], Goguen and Burstall call *reflection of colimits* what we here call, following Mac Lane's terminology [70], *preservation of colimits*.

Proof
By Theorem 8, $\tilde{V}_F(G)$ and $\hat{V}_F(U \cdot G)$ are defined pointwise for each $a \in \mathcal{D}$ by

$$\tilde{V}_F(G)(a) = \text{colim } G \cdot \Pi_a \text{ and}$$
$$\hat{V}_F(U \cdot G)(a) = \text{colim } U \cdot G \cdot \Pi_a.$$

Each $g: a \to a'$ in $\mathcal{D}$ induces unique arrows

$$\tilde{V}_F(G)(g): \text{colim } G \cdot \Pi_a \to \text{colim } G \cdot \Pi_{a'} \text{ and}$$
$$\hat{V}_F(U \cdot G)(g): \text{colim } U \cdot G \cdot \Pi_a \to \text{colim } U \cdot G \cdot \Pi_{a'},$$

commuting with the respective colimiting cones. Since U preserves colimits, we have colim $U \cdot G \cdot \Pi_a \simeq U \cdot$ colim $G \cdot \Pi_a$ for each $a \in \mathcal{D}$. But we have more than that. Since U preserves colimits, given any $a \in \mathcal{D}$ and a colimiting cone $\alpha: G \cdot \Pi_a \dot{\to} \text{colim } G \cdot \Pi_a$, then $U \cdot \alpha: U \cdot G \cdot \Pi_a \dot{\to} U \cdot \text{colim } G \cdot \Pi_a$ is a colimiting cone for $U \cdot G \cdot \Pi_a$. Therefore, given $\tilde{V}_F(G)(a)$, we can choose $\hat{V}_F(U \cdot G)(a)$ such that $\hat{V}_F(U \cdot G)(a) = U \cdot \tilde{V}_F(G)(a)$.

Now, given $g : a \to a'$ in $\mathcal{D}$ and colimiting cones $\alpha : G \cdot \Pi_a \dot{\to} \text{colim } G \cdot \Pi_a$ and $\alpha' : G \cdot \Pi_{a'} \dot{\to} \text{colim } G \cdot \Pi_{a'}$ in $\mathcal{C}$, $\tilde{V}_F(G)(g)$ is uniquely determined such that $\tilde{V}_F(G)(g) \cdot \alpha_n = \alpha'_n$ for each $n \in F \downarrow a$.

Since U preserves colimits, the cones $U \cdot \alpha: U \cdot G \cdot \Pi_a \dot{\to} U \cdot \text{colim } G \cdot \Pi_a$ and $U \cdot \alpha' : U \cdot G \cdot \Pi_{a'} \dot{\to} U \cdot \text{colim } G \cdot \Pi_{a'}$ are colimiting cones for $U \cdot G \cdot \Pi_a$ and $U \cdot G \cdot \Pi_{a'}$ in $\mathcal{A}$, respectively. Since $\hat{V}_F(U \cdot G)(g)$ is uniquely determined such that $\hat{V}_F(U \cdot G)(g) \cdot U(\alpha) = U(\alpha')$, and we have $U(\tilde{V}_F(G)(g)) \cdot U(\alpha) = U(\alpha')$ because $\tilde{V}_F(G)(g) \cdot \alpha = \alpha'$, then, for these cones, we have $U(\tilde{V}_F(G)(g)) = \hat{V}_F(U \cdot G)(g)$. □

Corollary 15 *Let $G, G' : \mathcal{B} \to \mathcal{C}$, $F : \mathcal{B} \to \mathcal{D}$, and $U : \mathcal{C} \to \mathcal{A}$ be functors, with $\mathcal{B}$ small, $\mathcal{C}$ and $\mathcal{A}$ cocomplete categories, and U preserving colimits. Let $\tilde{V}_F(\alpha): \tilde{V}_F(G) \to \tilde{V}_F(G')$ be the left Kan extension of the natural transformation $\alpha : G \to G'$ along F, and let $\hat{V}_F(U \cdot \alpha) : \hat{V}_F(U \cdot G) \to \hat{V}_F(U \cdot G')$ be the left Kan extension of $U \cdot \alpha$ along F. Then, $\hat{V}_F$ can be chosen such that $\hat{V}_F(U \cdot \alpha) = U \cdot \tilde{V}_F(\alpha)$.*

Proof
Since by Proposition 14 we can choose $\hat{V}_F$ such that $\hat{V}_F(U \cdot G) = U \cdot \tilde{V}_F(G)$ and $\hat{V}_F(U \cdot G') = U \cdot \tilde{V}_F(G')$, and $\hat{V}_F(U \cdot \alpha)$ is defined as a function assigning to each object d of $\mathcal{D}$ an arrow $\hat{V}_F(U \cdot \alpha)(d): U(\tilde{V}_F(G)(d)) \to U(\tilde{V}_F(G')(d))$, then, for this particular choice, $\hat{V}_F(U \cdot \alpha) = U \cdot \tilde{V}_F(\alpha)$. □

Theorem 16 *Let $\mathcal{C}$ and $\mathcal{A}$ be cocomplete categories, and let $U: \mathcal{C} \to \mathcal{A}$ preserve colimits. Then $Dg(U): Dg(\mathcal{C}) \to Dg(\mathcal{A})$ preserves colimits.*

Since any colimit can be constructed out of pushouts and coproducts, it is enough to show that $Dg(U)$ preserves these constructs. Given a colimit-preserving functor $U: \mathcal{C} \to \mathcal{A}$, with $\mathcal{C}$ and $\mathcal{A}$ cocomplete, we show in Propositions 17 and 18 that $Dg(U): Dg(\mathcal{C}) \to Dg(\mathcal{A})$ preserves, respectively, pushouts and coproducts.

Proposition 17 *Let $\mathcal{C}$ and $\mathcal{A}$ be cocomplete categories, and let $U : \mathcal{C} \to \mathcal{A}$ preserve colimits. Then $Dg(U): Dg(\mathcal{C}) \to Dg(\mathcal{A})$ preserves pushouts.*

Proof
By Proposition 11, since $\mathcal{C}$ and $\mathcal{A}$ are cocomplete, $Dg(\mathcal{C})$ and $Dg(\mathcal{A})$ have pushouts.

Let us assume here all the definitions introduced in Proposition 11, and let $(F', \delta)\colon D_2 \to D_3$ and $(J', \gamma)\colon D_1 \to D_3$ be a pushout of $(F, \alpha)\colon D_0 \to D_1$ and $(J, \beta)\colon D_0 \to D_2$ as constructed in Proposition 11.

$$\begin{tikzcd}
D_2 \arrow[r, dotted, "{(F',\delta)}"] & D_3 \\
D_0 \arrow[u, "{(J,\beta)}"] \arrow[r, "{(F,\alpha)}"'] & D_1 \arrow[u, dotted, "{(J',\gamma)}"']
\end{tikzcd}$$

Thus, we have that $J' : \mathcal{P}_1 \to \mathcal{P}_3$ and $F' : \mathcal{P}_2 \to \mathcal{P}_3$ are a pushout in **Cat** of F and J, and that $\delta : D_2 \to \tilde{F}'(D_3)$ and $\gamma : D_1 \to \tilde{J}'(D_3)$ are the unique arrows such that $\epsilon^{F'}_{D_3} \cdot \tilde{V}_{F'}(\delta) = \bar{\delta}$ and $\epsilon^{J'}_{D_3} \cdot \tilde{V}_{J'}(\gamma) = \bar{\gamma}$, with $\bar{\delta}\colon \tilde{V}_{F'}(D_2) \to D_3$ and $\bar{\gamma} : \tilde{V}_{J'}(D_1) \to D_3$ a pushout of $\tilde{V}_{J'}(\bar{\alpha}) : \tilde{V}_{J'}(\tilde{V}_F(D_0)) \to \tilde{V}_{J'}(D_1)$ and $\tilde{V}_{F'}(\bar{\beta})\colon \tilde{V}_{F'}(\tilde{V}_J(D_0)) \to \tilde{V}_{F'}(D_2)$ constructed pointwise in $\mathcal{C}^{\mathcal{P}_3}$.

$$\begin{tikzcd}
\tilde{V}_{F'}(D_2) \arrow[rr, dotted, "\bar{\delta}"] & \arrow[d, phantom, "p.o."] & D_3 \\
\tilde{V}_{F'}(\tilde{V}_J(D_0)) = \tilde{V}_{J'}(\tilde{V}_F(D_0)) \arrow[u, "\tilde{V}_{F'}(\bar{\beta})"] \arrow[rr, "\tilde{V}_{J'}(\bar{\alpha})"'] & & \tilde{V}_{J'}(D_1) \arrow[u, dotted, "\bar{\gamma}"']
\end{tikzcd}$$

Since U preserves colimits, and $\mathcal{A}^{\mathcal{P}_3}$ is cocomplete because $\mathcal{A}$ is cocomplete [70], the following diagram is a pushout in $\mathcal{A}^{\mathcal{P}_3}$.

$$\begin{tikzcd}
U \cdot \tilde{V}_{F'}(D_2) \arrow[rr, dotted, "U\cdot\bar{\delta}"] & \arrow[d, phantom, "p.o."] & U \cdot D_3 \\
U \cdot \tilde{V}_{F'}(\tilde{V}_J(D_0)) = U \cdot \tilde{V}_{J'}(\tilde{V}_F(D_0)) \arrow[u, "U\cdot\tilde{V}_{F'}(\bar{\beta})"] \arrow[rr, "U\cdot\tilde{V}_{J'}(\bar{\alpha})"'] & & U \cdot \tilde{V}_{J'}(D_1) \arrow[u, dotted, "U\cdot\bar{\gamma}"']
\end{tikzcd}$$

Let $\hat{F} : \mathcal{A}^{\mathcal{P}_1} \to \mathcal{A}^{\mathcal{P}_0}$, $\hat{J} : \mathcal{A}^{\mathcal{P}_2} \to \mathcal{A}^{\mathcal{P}_0}$, $\hat{F}' : \mathcal{A}^{\mathcal{P}_3} \to \mathcal{A}^{\mathcal{P}_2}$, and $\hat{J}' : \mathcal{A}^{\mathcal{P}_3} \to \mathcal{A}^{\mathcal{P}_1}$ be the induced forgetful functors associated, respectively, to F, J, F', and J'. By Proposition 14, we can choose their respective left adjoints $\hat{V}_F : \mathcal{A}^{\mathcal{P}_0} \to \mathcal{A}^{\mathcal{P}_1}$, $\hat{V}_J : \mathcal{A}^{\mathcal{P}_0} \to \mathcal{A}^{\mathcal{P}_2}$, $\hat{V}_{F'} : \mathcal{A}^{\mathcal{P}_2} \to \mathcal{A}^{\mathcal{P}_3}$, and $\hat{V}_{J'} : \mathcal{A}^{\mathcal{P}_1} \to \mathcal{A}^{\mathcal{P}_3}$ in such a way that the following equations hold:

$$\begin{aligned}
\hat{V}_F(U \cdot D_0) &= U \cdot \tilde{V}_F(D_0), \\
\hat{V}_J(U \cdot D_0) &= U \cdot \tilde{V}_J(D_0), \\
\hat{V}_{F'}(U \cdot D_2) &= U \cdot \tilde{V}_{F'}(D_2), \\
\hat{V}_{F'}(U \cdot \tilde{V}_J(D_0)) &= U \cdot \tilde{V}_{F'}(\tilde{V}_J(D_0)), \\
\hat{V}_{J'}(U \cdot D_1) &= U \cdot \tilde{V}_{J'}(D_1), \text{ and} \\
\hat{V}_{J'}(U \cdot \tilde{V}_F(D_0)) &= U \cdot \tilde{V}_{J'}(\tilde{V}_F(D_0)).
\end{aligned}$$

Let $\hat{\eta}^J : id \to \hat{J} \cdot \hat{V}_J$ be the unit of the adjunction $\hat{V}_J \dashv \hat{J}$, and let

$$\overline{U \cdot \beta}\colon \hat{V}_J(U \cdot D_0) \to U \cdot D_2$$

be the unique morphism in $\mathcal{A}^{\mathcal{P}_2}$ such that $U \cdot \beta = \hat{J}(\overline{U \cdot \beta}) \cdot \hat{\eta}^J_{U \cdot D_0}$.

The components of η^J and $\hat{\eta}^J$ corresponding to D_0 and $U \cdot D_0$ are, respectively,

$$\eta^J_{D_0} : D_0 \to \tilde{J}(\tilde{V}_J(D_0)) \text{ and}$$

$$\hat{\eta}^J_{U \cdot D_0} : U \cdot D_0 \to \hat{J}(\hat{V}_J(U \cdot D_0)).$$

By composing $\eta^J_{D_0}$ with U, we have

$$U \cdot \eta^J_{D_0} : U \cdot D_0 \to U \cdot \tilde{J}(\tilde{V}_J(D_0)).$$

Therefore, since $\hat{V}_J$ has been chosen such that $\hat{V}_J(U \cdot D_0) = U \cdot \tilde{V}_J(D_0)$, and $\hat{J}(U \cdot \tilde{V}_J(D_0)) = U \cdot \tilde{J}(\tilde{V}_J(D_0))$, we have

$$\hat{\eta}^J_{U \cdot D_0} = U \cdot \eta^J_{D_0}.$$

Since $\beta = \tilde{J}(\bar{\beta}) \cdot \eta^J_{D_0}$, we have $U(\beta) = U(\tilde{J}(\bar{\beta})) \cdot U(\eta^J_{D_0})$ by applying U. Since $\hat{\eta}^J_{U \cdot D_0} = U \cdot \eta^J_{D_0}$ and $\hat{J}(U \cdot \bar{\beta}) = U \cdot \tilde{J}(\bar{\beta})$, then we have $U \cdot \beta = \hat{J}(U \cdot \bar{\beta}) \cdot \eta^J_{U \cdot D_0}$. Therefore, since $\overline{U \cdot \beta}$ is unique, we must have $\overline{U \cdot \beta} = U \cdot \bar{\beta}$.

Let $\hat{\eta}^F : id \to \hat{F} \cdot \hat{V}_F$ be the unit of the adjunction $\hat{V}_F \dashv \hat{F}$, and let

$$\overline{U \cdot \alpha} : \hat{V}_F(U \cdot D_0) \to U \cdot D_1$$

be the unique morphism in $\mathcal{A}^{\mathcal{P}_1}$ such that $U \cdot \alpha = \hat{F}(\overline{U \cdot \alpha}) \cdot \hat{\eta}^F_{U \cdot D_0}$.

As above, from the choice of $\hat{V}_F$ it follows

$$\hat{\eta}^F_{U \cdot D_0} = U \cdot \eta^F_{D_0},$$

and then, by uniqueness of $\overline{U \cdot \alpha}$, we have $\overline{U \cdot \alpha} = U \cdot \bar{\alpha}$.

Similarly for $\hat{V}_{F'}$ and $\hat{V}_{J'}$, let $\hat{\eta}^{F'} : id \to \hat{F'} \cdot \hat{V}_{F'}$ and $\hat{\eta}^{J'} : id \to \hat{J'} \cdot \hat{V}_{J'}$ be the units of the adjunctions $\hat{V}_{F'} \dashv \hat{F'}$ and $\hat{V}_{J'} \dashv \hat{J'}$, respectively, and let

$$\overline{U \cdot \delta} : \hat{V}_{F'}(U \cdot D_2) \to U \cdot D_3 \text{ and}$$

$$\overline{U \cdot \gamma} : \hat{V}_{J'}(U \cdot D_1) \to U \cdot D_3$$

be the unique morphisms in $\mathcal{A}^{\mathcal{P}_3}$ such that $U \cdot \delta = \hat{F'}(\overline{U \cdot \delta}) \cdot \hat{\eta}^{F'}_{U \cdot D_2}$ and $U \cdot \gamma = \hat{J'}(\overline{U \cdot \gamma}) \cdot \hat{\eta}^{J'}_{U \cdot D_1}$.

From the choice of $\hat{V}_{F'}$ and $\hat{V}_{J'}$ it follows

$$\hat{\eta}^{F'}_{U \cdot D_2} = U \cdot \eta^{F'}_{D_2} \text{ and } \hat{\eta}^{J'}_{U \cdot D_1} = U \cdot \eta^{J'}_{D_1},$$

and then, by uniqueness of $\overline{U \cdot \delta}$ and $\overline{U \cdot \gamma}$, we have $\overline{U \cdot \delta} = U \cdot \bar{\delta}$ and $\overline{U \cdot \gamma} = U \cdot \bar{\gamma}$.

By Corollary 15, given $\hat{V}_{F'}(U \cdot D_2) = U \cdot \tilde{V}_{F'}(D_2)$ and $\hat{V}_{F'}(U \cdot \tilde{V}_J(D_0)) = U \cdot \tilde{V}_{F'}(\tilde{V}_J(D_0))$, we have $\hat{V}_{F'}(U \cdot \bar{\beta}) = U \cdot \tilde{V}_{F'}(\bar{\beta})$. Since $\overline{U \cdot \beta} = U \cdot \bar{\beta}$, then

$$U \cdot \tilde{V}_{F'}(\bar{\beta}) = \hat{V}_{F'}(\overline{U \cdot \beta}).$$

Similarly, since $\hat{V}_{J'}(U \cdot \tilde{V}_F(D_0)) = U \cdot \tilde{V}_{J'}(\tilde{V}_F(D_0))$, $\hat{V}_{J'}(U \cdot D_1) = U \cdot \tilde{V}_{J'}(D_1)$, and $\overline{U \cdot \alpha} = U \cdot \bar{\alpha}$, then

$$U \cdot \tilde{V}_{J'}(\bar{\alpha}) = \hat{V}_{J'}(\overline{U \cdot \alpha}).$$

Finally, since

$$\hat{V}_{F'}(\hat{V}_J(U \cdot D_0)) = U \cdot \tilde{V}_{F'}(\tilde{V}_J(D_0)) = U \cdot \tilde{V}_{J'}(\tilde{V}_F(D_0)) = \hat{V}_{J'}(\hat{V}_F(U \cdot D_0)),$$

we have

$$\hat{V}_{F'}(\hat{V}_J(U \cdot D_0)) = \hat{V}_{J'}(\hat{V}_F(U \cdot D_0)).$$

Therefore, from the above pushout it follows that the following diagram is a pushout as well.

$$\begin{tikzcd}
\hat{V}_{F'}(U \cdot D_2) \arrow[rr, dotted, "\overline{U\cdot\delta}"] \arrow[drr, phantom, "p.o."] & & U \cdot D_3 \\
\hat{V}_{F'}(\hat{V}_J(U \cdot D_0)) \arrow[u, "\hat{V}_{F'}(\overline{U\cdot\beta})"] \arrow[r, phantom, "="] & \hat{V}_{J'}(\hat{V}_F(U \cdot D_0)) \arrow[r, "\hat{V}_{J'}(\overline{U\cdot\alpha})"'] & \hat{V}_{J'}(U \cdot D_1) \arrow[u, dotted, "\overline{U\cdot\gamma}"']
\end{tikzcd}$$

And, by the proof of Proposition 11, the following diagram is also a pushout, as desired.

$$\begin{tikzcd}
U \cdot D_2 \arrow[r, dotted, "(F', U\cdot\delta)"] & U \cdot D_3 \\
U \cdot D_0 \arrow[u, "(J, U\cdot\beta)"] \arrow[r, "(F, U\cdot\alpha)"'] & U \cdot D_1 \arrow[u, dotted, "(J', U\cdot\gamma)"']
\end{tikzcd}$$

□

Proposition 18 *Given any functor $U : \mathcal{C} \to \mathcal{A}$, then $Dg(U) : Dg(\mathcal{C}) \to Dg(\mathcal{A})$ preserves coproducts.*

Proof

Let $\{D_i : J_i \to \mathcal{C}\}_{i \in I}$ be an I-indexed family of diagrams for any set I, and let the functor $D : \coprod_{i \in I} J_i \to \mathcal{C}$, with the induced family of inclusion morphisms $\{(\iota_i, 1^{D_i}) : D_i \to D \cdot \iota_i\}_{i \in I}$, be the coproduct of the indexed family of diagrams $\{D_i\}_{i \in I}$ in $Dg(\mathcal{C})$, as constructed in Proposition 12.

We claim that the functor $U \cdot D : \coprod_{i \in I} J_i \to \mathcal{A}$, with the induced family of inclusion morphisms $\{(\iota_i, U1^{D_i}) : U \cdot D_i \to U \cdot D \cdot \iota_i\}_{i \in I}$, is a coproduct of the indexed family of diagrams $\{U \cdot D_i\}_{i \in I}$ in $Dg(\mathcal{A})$

$$\begin{tikzcd}
J_i \arrow[rr, "\iota_i"] \arrow[dr, "D_i"'] \arrow[rr, phantom, shift right=4ex, "\overset{1^{D_i}}{\Rightarrow}"] & & \coprod_{i \in I} J_i \arrow[dl, dashed, "D"] \\
& \mathcal{C} \arrow[d, "U"] & \\
& \mathcal{A} &
\end{tikzcd}$$

Given a diagram $F : \Lambda \to \mathcal{A}$ with an indexed family of morphisms $\{(K_i, \lambda^i) : U \cdot D_i \to F\}_{i \in I}$, we have to show that there is a unique diagram morphism $(K, \lambda) : U \cdot D \to F$ such that $(K, \lambda) \cdot (\iota_i, U1^{D_i}) = (K_i, \lambda^i)$ for all $i \in I$.

Since $\coprod_{i \in I} J_i$ is the coproduct of the family $\{J_i\}_{i \in I}$, then there is a unique morphism $K : \coprod_{i \in I} J_i \to \Lambda$ making $K \cdot \iota_i = K_i$ for all $i \in I$.

Let the construction of $\coprod_{i \in I} J_i$ be given by objects (i, a) for $a \in J_i$, and arrows $(i, f) : (i, a) \to (i, b)$ for $f : a \to b$ in J_i. Then, we have $D(i, a) = D_i(a)$, and $F(K(i, a)) = F(K_i(a))$. Similarly, we have $D(i, f) = D_i(f)$, and $F(K(i, f)) = F(K_i(f))$.

Let us define the natural transformation $\lambda : U \cdot D \to F \cdot K$ by defining

$$\lambda_{(i,a)} = \lambda^i_a : U \cdot D(i, a) \to F(K(i, a))$$

for each $(i,a) \in \coprod_{i\in I} J_i$. Its naturality follows from the naturality of the natural transformations $\{\lambda^i : U \cdot D_i \to F \cdot K_i\}_{i\in I}$.

Furthermore, by construction we have $\lambda \cdot U1^{D_i} = \lambda^i$ for all $i \in I$.

Let us see now that it is unique. Assume another natural transformation $\gamma : U \cdot D \to F \cdot K$ such that $\gamma \cdot U1^{D_i} = \lambda^i$ for all $i \in I$. Since we must have $\gamma_{(i,a)} \cdot U1^{D_i}_a = \lambda^i_a$ for all $(i,a) \in \coprod_{i\in I} J_i$, γ must be defined as

$$\gamma_{(i,a)} = \lambda^i_a : D(i,a) \to F(K(i,a))$$

for each $(i,a) \in \coprod_{i\in I} J_i$, and therefore $\lambda = \gamma$. □

2.4 Structured Theories

In this section we define the institution $\mathcal{S}(\mathcal{I})$ of structured theories over a given institution $\mathcal{I}$, and give some results about the cocompleteness of its categories of signatures and theories, and about the liberality of the institution.

Specifically, we show that the cocompleteness of the categories of signatures and theories is preserved for the corresponding categories of structured entities, that is, we show that if the category $\mathbf{Sign}_{\mathcal{I}}$ of signatures and signature morphisms for the institution $\mathcal{I}$ is cocomplete then the category $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ of structured signatures and structured signature morphisms of $\mathcal{S}(\mathcal{I})$ is also cocomplete. Moreover, from a result by Goguen and Burstall in [53] valid for any institution, it then follows that the category $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ of structured theories is cocomplete if $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ is cocomplete. We also show that even when $\mathbf{Sign}_{\mathcal{I}}$ is not cocomplete, if $\mathbf{Th}_{\mathcal{I}}$ is cocomplete then $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ is also cocomplete. We finish Section 2.4.2 showing that if $\mathcal{I}$ has a cocomplete category of signatures, and is liberal and exact, then $\mathcal{S}(\mathcal{I})$ is also liberal. In Section 2.4.3 we use a general result by Goguen and Burstall in [53] to add freeness constraints to our setting.

2.4.1 The Institution of Structured Theories

A structured signature can be formalized as a functor $D : I \to \mathbf{Sign}_{\mathcal{I}}$ from a small category I to the category $\mathbf{Sign}_{\mathcal{I}}$ of signatures and signature morphisms in a given institution $\mathcal{I}$. This is of course a quite general notion. One can specialize the concept to the more familiar concept of hierarchy of signatures by requiring that I is a poset and that all the arrows in the diagram are inclusions in an appropriate subcategory of inclusion morphisms. Based on this definition, we build the institution $\mathcal{S}(\mathcal{I})$ of structured theories by defining functors $\mathrm{sen}_{\mathcal{S}(\mathcal{I})}$ and $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}$ associating to each structured signature D in $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ a set of D-sentences and a category of D-models, respectively. Then, we give a satisfaction relation for it and show that the satisfaction condition holds.

Definition 19 *Let us denote by* $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ *the category* $Dg(\mathbf{Sign}_{\mathcal{I}})$ *of diagrams over the category of signatures in the institution* $\mathcal{I}$*. We shall call structured signatures the objects of* $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$*. Therefore, structured signatures are given by a small category* I *and a functor* $D : I \to \mathbf{Sign}_{\mathcal{I}}$*. We shall denote a structured signature by its corresponding functor* D*. We call* structured signature morphisms *the morphisms in* $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$.

Definition 20 *The functor* $\mathrm{sen}_{\mathcal{S}(\mathcal{I})} : \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Set}$*, associating to each structured signature* $D : I \to \mathbf{Sign}_{\mathcal{I}}$ *a set of sentences and to each structured* *signature morphism* $(K,H)\colon D \to D'$ *a corresponding translation at the level of sentences, is defined as follows:*

$$\mathrm{sen}_{\mathcal{S}(\mathcal{I})}(D) = \coprod_{i\in I} \mathrm{sen}_{\mathcal{I}}(D(i))$$

$$\mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H)) = \coprod_{i\in I} \mathrm{sen}_{\mathcal{I}}(H_i)$$

It is easy to see that $\mathrm{sen}_{\mathcal{S}(\mathcal{I})}\colon \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Set}$ *preserves identities and composition.*

We can see each of the sentences of D as a pair (i,φ), where φ is a sentence in $\mathrm{sen}_{\mathcal{I}}(D(i))$. Note that, given a structured signature morphism $(K,H)\colon D \to D'$ and a sentence (i,φ) of D, we have

$$\mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H))((i,\varphi)) = (K(i), \mathrm{sen}_{\mathcal{I}}(H_i)(\varphi)).$$

Definition 21 *Given a structured signature* $D : I \to \mathbf{Sign}_{\mathcal{I}}$*, its category of models* $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D)$ *has as objects families* $M = \{M_i\}_{i\in I}$ *with* M_i *in* $\mathrm{Mod}_{\mathcal{I}}(D(i))$*, such that for each* $\phi\colon i \to j$ *in* I*,* $\mathrm{Mod}_{\mathcal{I}}(D(\phi))(M_j) = M_i$*. A morphism between two such models* $f\colon M \to M'$ *is given by a family* $\{f_i\colon M_i \to M'_i\}_{i\in I}$ *with* f_i *in* $\mathrm{Mod}_{\mathcal{I}}(D(i))$ *such that for each* $\phi\colon i \to j$ *in* I*,* $\mathrm{Mod}_{\mathcal{I}}(D(\phi))(f_j) = f_i$*.*

Let $D : I \to \mathbf{Sign}_{\mathcal{I}}$ be a structured signature, and let $f : M \to M'$ and $f' : M' \to M''$ be D-model morphisms given, respectively, by $f = \{f_i\}_{i\in I}$ and $f' = \{f'_i\}_{i\in I}$. The composition of two such morphisms is defined by

$$f \cdot f' = \{f_i \cdot f'_i\}_{i\in I}.$$

Definition 22 *The functor*

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}\colon \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Cat}^{\mathrm{op}}$$

assigns to each structured signature $D : I \to \mathbf{Sign}_{\mathcal{I}}$ *its category of models* $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D)$*, and to each structured signature morphism* $(K,H) : D \to D'$ *the model functor*

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))\colon \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D') \to \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D),$$

defined as follows:

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(\{M'_j\}_{j\in I'}) = \{\mathrm{Mod}_{\mathcal{I}}(H_i)(M'_{K(i)})\}_{i\in I}$$

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(\{f'_j\}_{j\in I'}) = \{\mathrm{Mod}_{\mathcal{I}}(H_i)(f'_{K(i)})\}_{i\in I}$$

Definition 23 *Given a structured signature* D*, a* D*-model* $M = \{M_i\}_{i\in I}$ *satisfies a* D*-sentence* (i,φ) *if and only if* $M_i \models_{D(i)} \varphi$*. In this case, we write* $M \models_D (i,\varphi)$*.*

Proposition 24 (Satisfaction Condition) *Let* $D : I \to \mathbf{Sign}_{\mathcal{I}}$ *and* $D' : I' \to \mathbf{Sign}_{\mathcal{I}}$ *be structured signatures, and let* $(K,H)\colon D \to D'$ *be a structured signature morphism. Given a* D*-sentence* (i,φ) *and a* D'*-model* M'*, then*

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(M') \models_D (i,\varphi) \iff M' \models_{D'} \mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H))((i,\varphi)).$$

Proof
$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(M')$ is a D-model, namely, the family of models

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(M') = \{\mathrm{Mod}_{\mathcal{I}}(H_i)(M'_{K(i)})\}_{i\in I}.$$

Since φ is a sentence in $D(i)$, we have

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(M') \models_D (i,\varphi) \iff \mathrm{Mod}_{\mathcal{I}}(H_i)(M'_{K(i)}) \models_{D(i)} \varphi.$$

On the other hand, since $\mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H))((i,\varphi)) = (K(i), \mathrm{sen}_{\mathcal{I}}(H_i)(\varphi))$, and thus $\mathrm{sen}_{\mathcal{I}}(H_i)(\varphi)$ is a sentence in $D'(K(i))$, we have

$$M' \models_{D'} \mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H))((i,\varphi)) \iff M'_{K(i)} \models_{D'(K(i))} \mathrm{sen}_{\mathcal{I}}(H_i)(\varphi).$$

By the satisfaction condition for the institution $\mathcal{I}$, we also have

$$\mathrm{Mod}_{\mathcal{I}}(H_i)(M'_{K(i)}) \models_{D(i)} \varphi \iff M'_{K(i)} \models_{D'(K(i))} \mathrm{sen}_{\mathcal{I}}(H_i)(\varphi),$$

and therefore,

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))(M') \models_D (i,\varphi) \iff M' \models_{D'} \mathrm{sen}_{\mathcal{S}(\mathcal{I})}((K,H))((i,\varphi)).$$

□

Definition 25 *Let $\mathcal{S}(\mathcal{I})$ be the institution with:*

- $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ *as category of signatures,*
- *the functor* $\mathrm{sen}_{\mathcal{S}(\mathcal{I})} : \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Set}$, *of Definition 20, giving the set of D-sentences over any structured signature D in* $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$*,*
- *the functor* $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})} : \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Cat}^{\mathrm{op}}$*, of Definition 22, giving for each structured signature D a category of D-models and D-model morphisms, and*
- *the satisfaction relation given in Definition 23, for which the satisfaction condition holds as shown in Proposition 24.*

Since $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})} = Dg(\mathbf{Sign}_{\mathcal{I}})$, there should be a close and systematic relationship between the category $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ of theories in the institution $\mathcal{S}(\mathcal{I})$ and the diagram category $Dg(\mathbf{Th}_{\mathcal{I}})$. We proceed to express this relationship as an adjunction with particularly good properties.

Let $J \colon \mathbf{Th}_{\mathcal{S}(\mathcal{I})} \to Dg(\mathbf{Th}_{\mathcal{I}})$ be the functor defined on objects by the rule

$$(D,\Gamma) \mapsto D_{\Gamma^*},$$

where if $D : I \to \mathbf{Sign}_{\mathcal{I}}$ is a structured signature, then $D_{\Gamma^*} : I \to \mathbf{Th}_{\mathcal{I}}$ has $D_{\Gamma^*}(i) = (D(i), \Gamma^*_i)$ and $D_{\Gamma^*}(\phi\colon i \to j) = D(\phi)$, where

$$\Gamma^*_i = \{\varphi \in \mathrm{sen}_{\mathcal{I}}(D(i)) \mid \forall M \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma), M_i \models_{D(i)} \varphi\}.$$

Note that then $D(\phi)\colon D_{\Gamma^*}(i) \to D_{\Gamma^*}(j)$ is a theory morphism because

$$\begin{aligned}
\varphi \in \Gamma^*_i \quad & \Leftrightarrow \quad \forall M \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma),\ M_i \models_{D(i)} \varphi \\
& \Leftrightarrow \quad (\text{by satisfaction, with } \mathrm{Mod}_{\mathcal{I}}(\phi)(M_j) = M_i) \\
& \qquad\ \forall M \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma),\ M_j \models_{D(j)} \mathrm{sen}_{\mathcal{I}}(D(\phi))(\varphi) \\
& \Rightarrow \quad \mathrm{sen}_{\mathcal{I}}(D(\phi))(\varphi) \in \Gamma^*_j
\end{aligned}$$

Therefore, $\mathrm{sen}_{\mathcal{I}}(D(\phi))(\Gamma_i^*) \subseteq \Gamma_j^*$ and $D(\phi)$ is a theory morphism.

The definition of J on morphisms assigns to each theory morphism $(K, H)\colon (D, \Gamma) \to (D', \Gamma')$ in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ the diagram morphism $(K, \tilde{H})\colon D_{\Gamma^*} \to D'_{\Gamma'^*}$ with $\tilde{H}_i = H_i$ for each $i \in I$, which is well-defined by observing that $\Gamma'^{\bullet} = \coprod_{j \in I'} \Gamma'^*_j$, and using the fact that (K, H) is a theory morphism, so that for each $(i, \varphi) \in \Gamma$ we have $(K(i), \mathrm{sen}_{\mathcal{I}}(H_i)(\varphi)) \in \Gamma'^{\bullet}$.

Let $R\colon Dg(\mathbf{Th}_{\mathcal{I}}) \to \mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ be the functor defined on objects by the rule

$$D \mapsto (\mathrm{sign}_{\mathcal{I}} \cdot D,\ \coprod_{i \in I} \mathrm{ax}(D(i))),$$

where, for (Σ, Γ) a theory, we use the notation $\mathrm{ax}(\Sigma, \Gamma) = \Gamma$.

Note that for any theory (D, Γ) in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ we have a natural isomorphism $RJ(D, \Gamma) \overset{\varepsilon_{(D,\Gamma)}}{\simeq} (D, \Gamma)$. Indeed, by construction we have $RJ(D, \Gamma) = (D, \Gamma^{\bullet})$, where

$$\Gamma^{\bullet} = \{\psi \in \mathrm{sen}_{\mathcal{S}(\mathcal{I})}(D) \mid \forall M \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D, \Gamma),\ M \models_D \psi\}.$$

Therefore, both theories are isomorphic, with the identity signature morphism as the isomorphism.

We do not need to define R on morphisms, since such a definition follows automatically from the adjunction result below.

Proposition 26 *The functor $J\colon \mathbf{Th}_{\mathcal{S}(\mathcal{I})} \to Dg(\mathbf{Th}_{\mathcal{I}})$ is full and faithful, with R left adjoint to J and ε as the counit.*

Proof

By Theorem 1 of Section IV.3 in [70], if ε is a counit and is an isomorphism, J is full and faithful. So we just have to check the adjunction.

The unit of the adjunction is the $Dg(\mathbf{Th}_{\mathcal{I}})$-morphism

$$\eta_D = (1_I, \eta)\colon D \to JRD$$

that is, the diagram morphism

$$\begin{tikzcd} I \arrow[rr, "1_I"] \arrow[dr, "D"'] & \arrow[d, phantom, "\overset{\eta}{\Rightarrow}"] & I \arrow[dl, "JRD"] \\ & \mathbf{Th}_{\mathcal{I}} & \end{tikzcd}$$

where, for $D(i) = (\Sigma_i, \Gamma_i)$,

$$\eta_i\colon (\Sigma_i, \Gamma_i) \to (\Sigma_i,\ (\coprod_{i \in I} \mathrm{ax}(D(i)))_i^*)$$

is the identity signature morphism on Σ_i.

Note that, since

$$\forall \varphi \in \Gamma_i,\ \forall M \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(\mathrm{sign}_{\mathcal{I}} \cdot D,\ \coprod_{i \in I} \mathrm{ax}(D(i))),\ M_i \models_{D(i)} \varphi,$$

we have $\varphi \in (\coprod_{i \in I} \mathrm{ax}(D(i)))_i^*$, and therefore η_i is indeed a theory morphism in $\mathbf{Th}_{\mathcal{I}}$.

Let $(K,\beta) : D \to J(D',\Gamma')$ be a diagram morphism in $Dg(\mathbf{Th}_{\mathcal{I}})$ with $(D',\Gamma') \in |\mathbf{Th}_{\mathcal{S}(\mathcal{I})}|$. We have to show that there is a unique theory morphism $\overline{(K,\beta)}\colon RD \to (D',\Gamma')$ such that

$$\begin{tikzcd}
D \arrow[r, "\eta_D"] \arrow[dr, "(K,\beta)"'] \arrow[dr, phantom, shift left=2, "="] & JRD \arrow[d, dashed, "J\overline{(K,\beta)}"] & RD \arrow[d, dashed, "\overline{(K,\beta)}"] \\
& J(D',\Gamma') & (D',\Gamma')
\end{tikzcd}$$

Uniqueness is clear, because if we have (K,β) with, say,

$$\begin{tikzcd}
I \arrow[rr, "K"] \arrow[dr, "D"'] \arrow[rr, phantom, shift right=4, "\overset{\beta}{\Rightarrow}"] & & I' \arrow[dl, "J(D',\Gamma')"] \\
& \mathbf{Th}_{\mathcal{I}} &
\end{tikzcd}$$

then the signature morphism part of $\overline{(K,\beta)}$, if it exists, must, by construction, coincide with the morphism

$$(K,\ \mathrm{sign}_{\mathcal{I}} \cdot \beta)\colon \mathrm{sign}_{\mathcal{I}} \cdot D \to \mathrm{sign}_{\mathcal{I}} \cdot J(D',\Gamma').$$

So we just need to check that this signature morphism is indeed a theory morphism

$$\overline{(K,\beta)}\colon RD \to (D',\Gamma')$$

in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$. That is, for each $(i,\varphi) \in \coprod_{i\in I} \mathrm{ax}(D(i))$ we have to show that

$$\forall M' \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D',\Gamma'),\ M'_{K(i)} \models_{D'(K(i))} \mathrm{sen}_{\mathcal{S}(\mathcal{I})}(\beta_i)(\varphi),$$

which is equivalent to showing that

$$\forall M' \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D',\Gamma'),\ \mathrm{sen}_{\mathcal{S}(\mathcal{I})}(\beta_i)(\varphi) \in {\Gamma'}^*_{K(i)}.$$

But by the hypothesis that there is a morphism $(K,\beta)\colon D \to J(D',\Gamma')$ we know that

$$\forall \varphi \in \mathrm{ax}(D(i)),\ \mathrm{sen}_{\mathcal{S}(\mathcal{I})}(\beta_i)(\varphi) \in ({\Gamma'}^*_{K(i)})^\bullet,$$

where, for (Σ,Γ) a theory in $\mathbf{Th}_{\mathcal{I}}$,

$$\Gamma^\bullet = \{\varphi \in \mathrm{sen}_{\mathcal{I}}(\Sigma) \mid \forall M \in \mathrm{Mod}_{\mathcal{I}}(\Sigma,\Gamma),\ M \models_\Sigma \varphi\}.$$

Then, it is enough to show that $({\Gamma'}^*_{K(i)})^\bullet = {\Gamma'}^*_{K(i)}$. Suppose that $\varphi \in ({\Gamma'}^*_{K(i)})^\bullet$, then,

$$\forall M \in \mathrm{Mod}_{\mathcal{I}}(D'(K(i))),\ M \models_{D'(K(i))} {\Gamma'}^*_i \Rightarrow M \models_{D'(K(i))} \varphi.$$

Since, by construction, we have

$$\forall M' \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D',\Gamma'),\ M'_{K(i)} \models_{D'(K(i))} {\Gamma'}^*_i,$$

we then have

$$\forall M' \in \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D',\Gamma'),\ M'_{K(i)} \models_{D'(K(i))} \varphi,$$

and therefore $\varphi \in {\Gamma'}^*_{K(i)}$, as desired.

We thus have that R is left adjoint to J with η as unit. To see that ε is the counit is straightforward. Given the following situation, $\overline{1_{J(D,\Gamma)}}$ is the counit $\varepsilon_{(D,\Gamma)}$ of the adjunction, which is, obviously, an isomorphism. Since for the signature part $\overline{1_{J(D,\Gamma)}}$ is an identity signature morphism, $\overline{1_{J(D,\Gamma)}}$ coincides with $\varepsilon_{(D,\Gamma)}$, as originally defined, and therefore, ε is the counit as desired.

$$\begin{tikzcd}
 & J(D,\Gamma) & \\
J(D,\Gamma) \arrow[ur, "1_{J(D,\Gamma)}"] \arrow[rr, "\eta_{J(D,\Gamma)}"'] & {=} & JRJ(D,\Gamma) \arrow[ul, "J\cdot\overline{1_{J(D,\Gamma)}}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
(D,\Gamma) & \\
 & RJ(D,\Gamma) \arrow[ul, "\overline{1_{J(D,\Gamma)}}"']
\end{tikzcd}$$

□

2.4.2 Cocompleteness and Liberality

Given the institution $\mathcal{S}(\mathcal{I})$, we now present some results on the cocompleteness of its categories of signatures and theories, and on the liberality of the institution.

Theorem 27 *If* $\mathbf{Sign}_{\mathcal{I}}$ *is cocomplete then* $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ *is cocomplete.*

Proof
Since $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})} = Dg(\mathbf{Sign}_{\mathcal{I}})$, this follows from Theorem 10. □

By the following result from [53], it follows that, given any institution $\mathcal{I}$, its category of theories is cocomplete if its category of signatures is cocomplete.

Theorem 28 [53] *If* $\mathcal{I}$ *is an institution such that* $\mathbf{Sign}_{\mathcal{I}}$ *is cocomplete, then* $\mathbf{Th}_{\mathcal{I}}$ *is also cocomplete and the forgetful functor* $\mathrm{sign}_{\mathcal{I}} \colon \mathbf{Th}_{\mathcal{I}} \to \mathbf{Sign}_{\mathcal{I}}$ *preserves colimits.*

Since $\mathcal{S}(\mathcal{I})$ is an institution, we have the following corollary of Theorem 27.

Corollary 29 *If* $\mathbf{Sign}_{\mathcal{I}}$ *is cocomplete then* $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ *is cocomplete, and the functor* $\mathrm{sign}_{\mathcal{S}(\mathcal{I})} \colon \mathbf{Th}_{\mathcal{S}(\mathcal{I})} \to \mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ *preserves colimits.*

The result by Goguen and Burstall gives a simple criterion to show when all colimits in the category of theories exist. However, it may not be sufficient. We can have institutions with cocomplete categories of theories and whose categories of signatures lack some colimits. For example, if we consider signature morphisms in many-sorted equational logic that can map *operations* to *terms* (as it is allowed for *views* in OBJ and in Maude, for example), the category of signatures fails to be cocomplete. The following shows that, independently of the cocompleteness of $\mathbf{Sign}_{\mathcal{S}(\mathcal{I})}$ for some institution $\mathcal{S}(\mathcal{I})$, if $\mathbf{Th}_{\mathcal{I}}$ is cocomplete then $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ is also cocomplete.

Theorem 30 *If* $\mathbf{Th}_{\mathcal{I}}$ *is cocomplete then* $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ *is cocomplete.*

Proof
If $\mathbf{Th}_{\mathcal{I}}$ is cocomplete, then $Dg(\mathbf{Th}_{\mathcal{I}})$ is cocomplete by Theorem 10. Then, since ε is a natural isomorphism, any diagram Ω in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ is isomorphic to the diagram $RJ\Omega$. But, since R is a left adjoint to J, and therefore preserves colimits, and since $Dg(\mathbf{Th}_{\mathcal{I}})$ has colimits, we have $R(\mathrm{colim}\ J\Omega) = \mathrm{colim}\ (RJ\Omega) = \mathrm{colim}\ (\Omega)$. □

Note that, assuming that $\mathbf{Sign}_{\mathcal{I}}$ is cocomplete, Theorem 28 (applied to $\mathcal{I}$), the proofs of Theorem 27 and Theorem 30, and the preservation of colimits result in Theorem 16 give us an alternative way of seeing that $\mathrm{sign}_{\mathcal{S}(\mathcal{I})}$ preserves colimits. Consider a diagram Ω in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ and the commutative diagram

$$\begin{tikzcd}
Dg(\mathbf{Th}_{\mathcal{I}}) \arrow[d, "Dg(\mathrm{sign}_{\mathcal{I}})"'] & \mathbf{Th}_{\mathcal{S}(\mathcal{I})} \arrow[l, "J"'] \arrow[d, "\mathrm{sign}_{\mathcal{S}(\mathcal{I})}"] & Dg(\mathbf{Th}_{\mathcal{I}}) \arrow[l, "R"'] \arrow[d, "Dg(\mathrm{sign}_{\mathcal{I}})"] \\
Dg(\mathbf{Sign}_{\mathcal{I}}) \arrow[r, "="'] & \mathbf{Sign}_{\mathcal{S}(\mathcal{I})} & Dg(\mathbf{Sign}_{\mathcal{I}}) \arrow[l, "="']
\end{tikzcd}$$

Then we have

$$\begin{aligned}
\mathrm{sign}_{\mathcal{S}(\mathcal{I})}(\mathrm{colim}\ \Omega) &\simeq \mathrm{sign}_{\mathcal{S}(\mathcal{I})}(\mathrm{colim}\ RJ\Omega) \\
&\simeq \mathrm{sign}_{\mathcal{S}(\mathcal{I})}R(\mathrm{colim}\ J\Omega) \\
&= Dg(\mathrm{sign}_{\mathcal{I}})(\mathrm{colim}\ J\Omega) \\
&= \mathrm{colim}\ (Dg(\mathrm{sign}_{\mathcal{I}})J\Omega) \\
&= \mathrm{colim}\ (\mathrm{sign}_{\mathcal{S}(\mathcal{I})}\Omega).
\end{aligned}$$

So far, we have only considered the category $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D)$ of models of a structured theory D over a fixed institution $\mathcal{I}$. We shall see in Section 2.4.3 how we can easily add freeness constraints to this setting. We could also deal with other kinds of constraints (see, e.g., [53, 45, 96, 15, 52]) in the same way. Liberality of an institution is an important property for initiality constraints. Since it is needed in what follows, we first introduce the definition of creation of limits by a functor.

Definition 31 *A functor $U : \mathcal{C} \to \mathcal{D}$* creates limits *for a functor $F : \mathcal{J} \to \mathcal{C}$ if for every limiting cone $\alpha : U \cdot F \dot{\to} d$ in $\mathcal{D}$ there is exactly a limiting cone $\beta : F \dot{\to} c$ in $\mathcal{C}$ such that $Uc = d$ and $U\beta = \alpha$.*

Theorem 32 *If an institution $\mathcal{I}$ is liberal and exact and $\mathbf{Th}_{\mathcal{I}}$ is cocomplete, then $\mathcal{S}(\mathcal{I})$ is liberal.*

Proof
First of all we observe that, for any $(D,\Gamma) \in |\mathbf{Th}_{\mathcal{S}(\mathcal{I})}|$ we have the isomorphism

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma) \simeq \lim\ (\mathrm{Mod}_{\mathcal{I}} \cdot D_{\Gamma^\bullet}). \qquad (\dagger)$$

Indeed, since $\Gamma^\bullet = \coprod_{i\in I} \Gamma_i^*$, (D,Γ) and $(D,\Gamma^\bullet)$ are isomorphic theories. Therefore, for each $M \in |\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D)|$ we have $M \models_D \Gamma$ iff $M \models_D \Gamma^\bullet$ iff for each $i \in I$, $M_i \models_{D(i)} \Gamma_i^*$. Therefore, $M \in |\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma)|$ iff

(i). $\forall i \in I$, $M_i \in |\mathrm{Mod}_{\mathcal{I}}(D(i),\Gamma_i^*)|$, and

(ii). $\forall \phi : i \to j$ in I, $\mathrm{Mod}_{\mathcal{I}}(D(\phi))(M_j) = M_i$,

and similarly, $f : M \to M'$ is a morphism in $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D,\Gamma)$ iff

(i). $\forall i \in I$, $f_i : M_i \to M_i'$ is a morphism in $\mathrm{Mod}_{\mathcal{I}}(D(i),\Gamma_i^*)$, and

(ii). $\forall \phi : i \to j$ in I, $\mathrm{Mod}_{\mathcal{I}}(D(\phi))(f_j) = f_i$.

The above isomorphism (†) then follows either by an explicit limit construction in **Cat**, or, more easily, as a consequence of the following observations, that reduce such a limit construction to a construction of limits in **Set**:

(1). The functor $\mathbf{Cat} \to \mathbf{Graph}$ sending each category to its underlying graph is monadic [1] and therefore creates limits [70].

(2). **Graph** is the functor category $\mathbf{Set}^{\bullet \rightrightarrows \bullet}$, and therefore limits are computed pointwise.

(3). If $D : I \to \mathbf{Set}$ is a functor, then $\lim D$ is a set with elements families $\{a_i\}_{i \in I}$ with $a_i \in D(i)$ such that for each $\phi : i \to j$ in I, $D(\phi)(a_i) = a_j$ [101].

But since $\mathbf{Th}_{\mathcal{I}}$ is cocomplete, the colimit of D_{Γ^*} exists, and by exactness of $\mathcal{I}$ we have the isomorphism

$$\lim\ (\mathrm{Mod}_{\mathcal{I}} \cdot D_{\Gamma^*}) \simeq \mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ D_{\Gamma^*}),$$

which combined with the isomorphism (†) gives us the isomorphism

$$\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D, \Gamma) \simeq \mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ D_{\Gamma^*}).$$

Notice also that, by cocompleteness of $\mathbf{Th}_{\mathcal{I}}$, there is a functor

$$\mathrm{colim}\ : Dg(\mathbf{Th}_{\mathcal{I}}) \to \mathbf{Th}_{\mathcal{I}}$$

such that for any morphism $(K, H) : (D, \Gamma) \to (D', \Gamma')$ in $\mathbf{Th}_{\mathcal{S}(\mathcal{I})}$ we have, thanks to the above isomorphism, the following commutative diagram.

$$\begin{tikzcd}
\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D, \Gamma) \arrow[d, phantom, "\wr"] & \mathrm{Mod}_{\mathcal{S}(\mathcal{I})}(D', \Gamma') \arrow[l, "\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K,H))"'] \arrow[d, phantom, "\wr"] \\
\mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ D_{\Gamma^*}) & \mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ D'_{\Gamma'^*}) \arrow[l, "\mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ J(K,H))"']
\end{tikzcd}$$

By liberality of the institution $\mathcal{I}$ there is a left adjoint to $\mathrm{Mod}_{\mathcal{I}}(\mathrm{colim}\ J(K, H))$, which, composed with the vertical isomorphisms, gives rise to a left adjoint to $\mathrm{Mod}_{\mathcal{S}(\mathcal{I})}((K, H))$. □

2.4.3 Freeness Constraints

One of the key motivations for making structured theories a direct object of study is dealing with *freeness constraints*, called *constraints* or *data constraints* in [53]. They are crucial for the notion of *parameterized module* (see Section 3.6) in which the model of the parameterized module's body should be a *free extension* of the model of the parameter theory. In languages like OBJ, CafeOBJ, and Maude, this leads to a distinction between *theories*, with loose semantics, and *modules*, with initial or, more generally, free extension semantics.

Both theories and modules can be parameterized[3], but in the case of parameterized modules, a freeness constraint between models of the parameter and models of the body is enforced.

Intuitively, freeness constraints are associated to particular theory maps appearing in the diagram of a structured theory. Suppose that $\mathcal{I}$ is liberal and

[3]In the present module algebra for Maude, described in Chapters 3 and 4, parameterized theories are not supported yet, but see Section 2.5 for several examples of parameterized theories.

that (D, Γ) is a structured theory with $D\colon I \to \mathbf{Sign}_{\mathcal{I}}$, and consider a morphism $\phi\colon i \to j$ in I. Then, we can associate a freeness constraint to the theory map

$$D(\phi)\colon D_{\Gamma^*}(i) \to D_{\Gamma^*}(j)$$

by requiring that the models M of (D, Γ), in addition to satisfying the axioms Γ, satisfy the constraint

$$M_j \simeq F_{D(\phi)}(M_i)$$

for $F_{D(\phi)} : \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(i)) \to \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(j))$ the left adjoint to the forgetful functor $\mathrm{Mod}_{\mathcal{I}}(D(\phi)) : \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(j)) \to \mathrm{Mod}_{\mathcal{I}}(D_{\Gamma^*}(i))$. For example, $D_{\Gamma^*}(i)$ may be the theory `TRIV`, specifying just one sort `Elt`, and $D_{\Gamma^*}(j)$ may be the theory `LIST` with a sort `List`, specifying lists formed with data elements from the `Elt` parameter sort. Then, the freeness constraint requires that the models of `LIST` are really lists, freely generated from the data elements.

For purposes of translating freeness constraints along signature maps it turns out that the above notion of freeness constraints should in fact be generalized somewhat, to allow an extra signature map bringing the model to the context in which the constraint is applied. This leads us to the following definition, due to Goguen and Burstall.

Definition 33 [53] *Let $\mathcal{I}$ be an institution. Then a* freeness constraint *on a signature Σ is a pair*

$$c = (H\colon T'' \to T',\ G\colon \mathrm{sign}(T') \to \Sigma)$$

such that a Σ-model M satisfies such a constraint if and only if $\mathrm{Mod}_{\mathcal{I}}(G)(M)$ satisfies T' and is H-free, which means that $\mathrm{Mod}_{\mathcal{I}}(H)(\mathrm{Mod}_{\mathcal{I}}(G)(M))$ has a free extension along H such that the corresponding component of the counit of the adjunction $\varepsilon_{\mathrm{Mod}_{\mathcal{I}}(G)(M)} : F_H(\mathrm{Mod}_{\mathcal{I}}(H)(\mathrm{Mod}_{\mathcal{I}}(G)(M))) \to \mathrm{Mod}_{\mathcal{I}}(G)(M)$ is an isomorphism; in this case we write $M \models_{\Sigma} c$.

Our intuitive notion of freeness constraint in a structured theory can then be recovered by two special cases of the above definition. The case of a *parameterized module*, illustrated by the theory inclusion `TRIV` $\hookrightarrow$ `LIST` in our previous example, corresponds to freeness constraints of the form

$$(D(\phi)\colon D_{\Gamma^*}(i) \to D_{\Gamma^*}(j),\ 1_{D(j)} : D(j) \to D(j)),$$

whereas the case of an *unparameterized module*, like `NAT` or `BOOL`, for which we want an *initial model semantics*, corresponds to a freeness constraint of the form

$$(\emptyset_{D_{\Gamma^*}(j)} : \emptyset \to D_{\Gamma^*}(j),\ 1_{D(j)} : D(j) \to D(j)),$$

where $\emptyset$ is the initial object in the category of signatures of an exact liberal institution $\mathcal{I}$ (so that $\mathrm{Mod}_{\mathcal{I}}(\emptyset)$ has only one model, let us call it also $\emptyset$) because then the initial model of $D_{\Gamma^*}(j)$ coincides with $F_{\emptyset_{D_{\Gamma^*}(j)}}(\emptyset)$.

The need for the more general notion of freeness constraint in Definition 33 has to do with translation of constraints by composition with signature morphisms. We think of a constraint $c = (H : T'' \to T',\ G : \mathrm{sign}(T') \to \Sigma)$ on Σ as a *sentence* associated to the signature Σ. Then, if $Q : \Sigma \to \Omega$ is a signature morphism, we can associate to the constraint c the following constraint on Ω:

$$\mathrm{sen}_{\mathcal{I}}(Q)(c) = (H\colon T'' \to T',\ Q \cdot G\colon \mathrm{sign}(T') \to \Omega).$$

The key point is that, as shown by Goguen and Burstall in [53], the satisfaction condition holds for freeness constraints translated along signature morphisms.

Lemma 34 (Constraint satisfaction [53]) *If $Q : \Sigma \to \Omega$ is a signature morphism, if c is a freeness constraint on Σ, and if M' is a Σ'-model, then*

$$M' \models_{\Sigma'} \mathrm{sen}_{\mathcal{I}}(Q)(c) \quad \text{iff} \quad \mathrm{Mod}_{\mathcal{I}}(Q)(M') \models_{\Sigma} c.$$

Goguen and Burstall exploit this satisfaction lemma to give a general construction associating to an institution $\mathcal{I}$ another institution $\mathcal{C}(\mathcal{I})$ (cf. [53, Proposition 23]) with the same category of signatures and the same model functor as $\mathcal{I}$, and with $\mathrm{sen}_{\mathcal{C}(\mathcal{I})}(\Sigma)$ the disjoint union of the sets $\mathrm{sen}_{\mathcal{I}}(\Sigma)$ and of the set[4] of all freeness constraints on Σ. Then, by the general result in Theorem 28, if $\mathbf{Sign}_{\mathcal{I}}$ is cocomplete, then $\mathbf{Th}_{\mathcal{C}(\mathcal{I})}$ is a cocomplete category.

Although the construction of $\mathcal{C}(\mathcal{I})$ is given by Goguen and Burstall for an institution $\mathcal{I}$ whose signatures and theories are unstructured, we have pointed out above how the notion of freeness constraint finds its natural home as additional constraints added to specific components of a structured theory. The way of explicitly combining freeness constraints and structured theories is then straightforward.

Definition 35 *Given an institution $\mathcal{I}$, the institution of $\mathcal{I}$-*structured theories with freeness constraints *is by definition the institution $\mathcal{S}(\mathcal{C}(\mathcal{I}))$.*

Therefore, theories in $\mathcal{S}(\mathcal{C}(\mathcal{I}))$ are pairs (D, Γ), with $D : I \to \mathbf{Sign}_{\mathcal{I}}$ a diagram, and with Γ sentences of the form (i, φ), with φ either a sentence in $\mathrm{sen}_{\mathcal{I}}(D(i))$ or a freeness constraint on the signature $D(i)$. In this way, after the appropriate desugaring in the case of object-oriented modules, we can interpret all the modules and theories in Maude as structured theories with initiality constraints in the institution $\mathcal{S}(\mathcal{C}(\boldsymbol{MRWLogic}))$, where $\boldsymbol{MRWLogic}$ is the institution of rewriting logic with membership equational logic[5] as its underlying equational fragment.

Notice that, although they are of course related constructions, the institution $\mathcal{S}(\mathcal{C}(\mathcal{I}))$ defined above is different from the institution $\mathcal{C}(\mathcal{S}(\mathcal{I}))$, that, by Theorem 32 and by the general construction of $\mathcal{C}(\mathcal{I})$ of Goguen and Burstall, could be defined for a liberal institution $\mathcal{I}$ with $\mathbf{Th}_{\mathcal{I}}$ cocomplete. Intuitively speaking, in $\mathcal{S}(\mathcal{C}(\mathcal{I}))$ the freeness constraints are *local* to specific components of structured theories, whereas in $\mathcal{C}(\mathcal{S}(\mathcal{I}))$ the freeness constraints are *global*, in the sense of involving pairs of structured theories.

2.5 Structured Theories in Practice

In this section we illustrate the use of the categorical constructions presented in the previous sections by giving several examples of structured theories. We use for that the Full Maude language[6], and in particular its membership equational logic institution. We present structured theories as diagrams of equational theory *inclusions*, leaving implicit the application of the functor $R\colon Dg(\mathbf{Th}_{\mathcal{I}}) \to \mathbf{Th}_{\mathcal{S}(\mathcal{I})}$.

[4] There are foundational questions about the size of the closure of a constraint theory that we will ignore here; as pointed out in [53], they can be solved, for example, by limiting the size of the category of signatures used in the original institution.

[5] More precisely, the freeness constraints should be understood with *partial* membership equational logic as the underlying equational fragment [82]. In the total case, because of the fact that new error elements may be added to the kinds when performing a free extension, the notion of satisfaction of a freeness constraint by a model should be made relative to sorts only, and not to kinds.

[6] In fact we use an extension of Full Maude not implemented, since parameterized theories are not available in Full Maude yet.

Specifically, we present the equational theory of modules over a ring as a structured theory, which is parameterized by the theory of rings and the theory of commutative groups. In the construction of the structure of the theory MODULE we encounter specifications of theories for monoids, groups, commutative monoids and groups, semirings, rings, monoid actions, and semimodules.

Let us begin by introducing the functional theory TRIV, which requires just a sort.

```
(fth TRIV is
   sort Elt .
 endfth)
```

The theory of monoids, with an associative binary operator with identity element 1, can be expressed as follows.

```
(fth MONOID is
   including TRIV .
   op 1 : -> Elt .
   op __ : Elt Elt -> Elt [assoc id: 1] .
 endfth)
```

The theory of commutative monoids can be defined similarly. Now, the _+_ operator is declared associative, commutative, and with 0 as its identity element.

```
(fth +MONOID is
   including TRIV .
   op 0 : -> Elt .
   op _+_ : Elt Elt -> Elt [assoc comm id: 0] .
 endfth)
```

A commutative group can be expressed as a commutative monoid with an inverse operator -_.

```
(fth +GROUP is
   including +MONOID .
   op -_ : Elt -> Elt .
   var X : Elt .
   eq X + - X = 0 .
 endfth)
```

The theories of rings and semirings can be expressed as follows.

```
(fth SEMIRING is
   including MONOID .
   including +MONOID .
   vars X Y Z : Elt .
   eq X (Y + Z) = (X Y) + (X Z) .
   eq (X + Y) Z = (X Z) + (Y Z) .
   eq 0 X = 0 .
   eq X 0 = 0 .
 endfth)

(fth RING is
   including SEMIRING .
   including +GROUP .
 endfth)
```

$$\begin{tikzcd} & \texttt{ACTION} & \\ \texttt{M::MONOID} \arrow[ur, hook] & & \texttt{X::TRIV} \arrow[ul, hook'] \\ \texttt{M::TRIV} \arrow[u, hook] & & \end{tikzcd}$$

Figure 2.2: Structure of the parameterized theory `ACTION`.

Next, we define the theory of (left) actions of a monoid on a set. We define it as a functional theory parameterized by the theory of monoids and sets, as indicated below.

In the case of parameterized theories, we regard the relationship between the body of the parameterized theory and the interface of its parameters not as an inclusion, but as a theory constructor which is evaluated, generating renamed copies of the parameters, which are then included. For example, for the interface of the theory `ACTION`, a copy of the theory `MONOID` with name `M :: MONOID` and a copy of the theory `TRIV` with name `X :: TRIV` are generated. In such copies of parameter theories, sorts are renamed so that, if Z is the label of a parameter theory T, then each sort S in T is renamed to $S.Z$. This is the reason why all occurrences of these sorts in the parameterized theory must mention their corresponding renaming.

```
(fth ACTION[M :: MONOID, X :: TRIV] is
   op __ : Elt.M Elt.X -> Elt.X .
   vars A B : Elt.M .
   var Y : Elt.X .
   eq 1 Y = Y .
   eq (A B) Y = A (B Y) .
 endfth)
```

If a parameter theory is structured, as for example the theory `MONOID` in the theory `ACTION` above, this renaming process for parameter theories is carried out not only at the top of the structured parameter, but on the whole structure of the parameter theory[7]. Representing by $\hookrightarrow$ the inclusion relations between theories, we can depict the structure of the parameterized theory `ACTION` as in Figure 2.2.

The instantiation of a parameterized theory requires the definition of a *view*, that is, a theory morphism for each of the formal parameters. We use views to assert how a particular target theory is claimed to satisfy each of the formal parameters, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds in the target theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target.

Given a theory T which is included in another theory T', let us adopt the convention of naming the view from T to T' defined by the inclusion $T \hookrightarrow T'$ by the name of the "supertheory" T'.

Theories instantiated by other theories could still be further instantiated. To 'link' the formal parameters of a parameterized theory to the formal param-

[7]In the more general case in which both modules with initial semantics and theories with loose semantics coexist, the renaming process is carried out on the "theory part" only, that is, modules are not renamed.

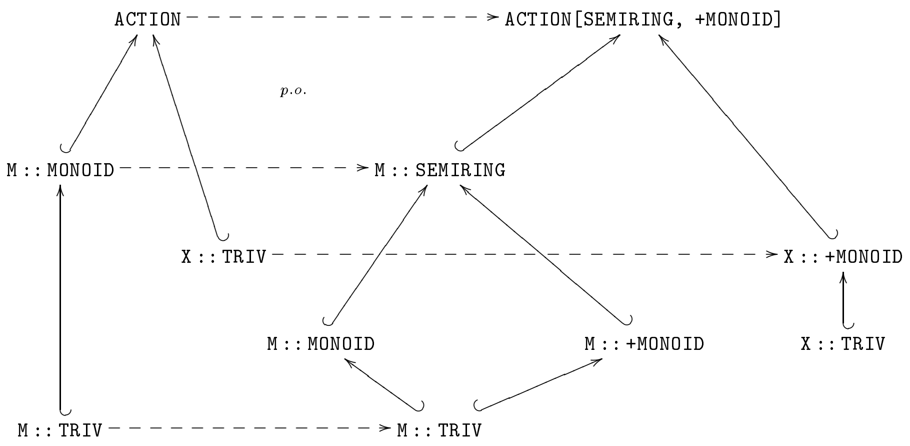


Figure 2.3: Instantiation of the parameterized theory `ACTION`.

eters of another parameterized theory being imported, we use the labels in the interface of the theory, in the OBJ3 style, as indicated in the following theory `SEMIMODULE`. In fact, we can see the labels in the interface as denoting identity views for the theories with which they are associated. For example, given the parameter `S :: SEMIRING` below, the label `S` denotes the identity view for the theory `SEMIRING`, or more precisely, it denotes the view from the structured theory `M :: SEMIRING` to the structured theory `S :: SEMIRING`, which is just an isomorphic copy obtained by renaming.

Given the theory `ACTION` above, the result of instantiating it with views `SEMIRING` and `+MONOID` is a theory with name `ACTION[SEMIRING, +MONOID]` and interface `[M :: SEMIRING, X :: +MONOID]`. Linking the parameters of this theory to the formal parameters of the structured theory `SEMIMODULE` in which it is included, or, in other words, instantiating it with views `S` and `T`, results in a structured theory with name `ACTION[SEMIRING, +MONOID][S, T]` and interface `[S :: SEMIRING, T :: +MONOID]`.

The semantics of the instantiation of theories is given by the pushouts in the category of structured theories discussed in Section 2.4.2, which can be obtained, using the functor J, from pushouts in $Dg(\mathbf{Th}_{\mathcal{I}})$ thanks to Theorem 30. We can depict the instantiation of the theory `ACTION` by views `SEMIRING` and `+MONOID` by the diagram in Figure 2.3. In this case, we consider the pushout of the inclusion morphism of the structured theory with tops `M :: MONOID` and `X :: TRIV` in the structure with `ACTION` as top, and of the morphism defined by the views `SEMIRING` and `+MONOID` from the structure with tops `M :: MONOID` and `X :: TRIV` to the structure with theories `M :: SEMIRING` and `X :: +MONOID` as tops. The structure of the theory `SEMIMODULE` and the different instantiations involved are depicted in Figure 2.4.

```
(fth SEMIMODULE[S :: SEMIRING, T :: +MONOID] is
   including ACTION[SEMIRING, +MONOID][S, T] .
   vars A B : Elt.S .
   vars X Y : Elt.T .
   eq X (A + B) = (X A) + (X B) .
   eq (X + Y) A = (X A) + (Y A) .
   eq X 0 = 0 .
 endfth)
```

The theory of modules can now be defined as parameterized by theories `RING`

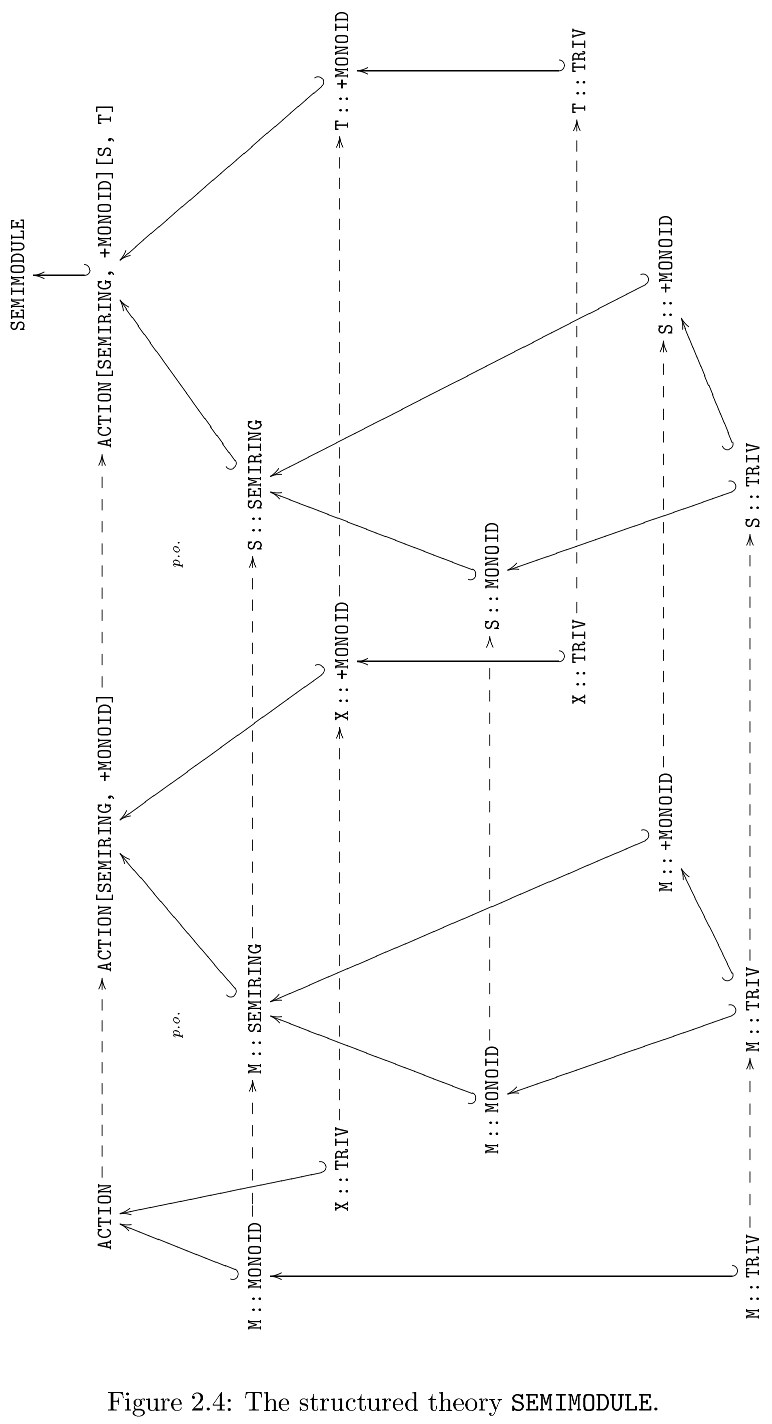


Figure 2.4: The structured theory SEMIMODULE.

and +GROUP, and including the theory SEMIMODULE instantiated by views RING and +GROUP as follows.

```
(fth MODULE[R :: RING, G :: +GROUP] is
   including SEMIMODULE[RING, +GROUP][R, G] .
 endfth)
```

The structure of the theory MODULE, together with the different theories and pushouts involved in its construction, is depicted in Figure 2.5.

2.6 Concluding Remarks

We have presented some basic results on category theory dealing with the existence of colimits in the category $Dg(\mathcal{C})$ of diagrams over a cocomplete category $\mathcal{C}$. We have shown how the existence of colimits in $\mathcal{C}$ implies the existence of colimits in $Dg(\mathcal{C})$, and how any colimit-preserving functor $U : \mathcal{C} \to \mathcal{A}$, with $\mathcal{A}$ and $\mathcal{C}$ cocomplete, induces a colimit-preserving functor $Dg(U) : Dg(\mathcal{C}) \to Dg(\mathcal{A})$.

Using the categorical constructions discussed in the present chapter we have given a precise semantics to structured theories and to some theory-building operations on structured theories. Specifically, we have shown that the addition of structured theories to an institution $\boldsymbol{\mathcal{I}}$ results in an institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$, and that if the category of signatures $\mathbf{Sign}_{\boldsymbol{\mathcal{I}}}$ has colimits, then the categories of signatures and theories of $\boldsymbol{\mathcal{S}(\mathcal{I})}$ both have colimits. We have also shown some other basic results on the cocompleteness of the categories of signatures and theories of $\boldsymbol{\mathcal{S}(\mathcal{I})}$, and on the liberality of the institution $\boldsymbol{\mathcal{S}(\mathcal{I})}$. We have also presented a very simple way of using a general result by Goguen and Burstall to add freeness constraints to our setting, resulting in institutions $\boldsymbol{\mathcal{S}(\mathcal{C}(\mathcal{I}))}$ and $\boldsymbol{\mathcal{C}(\mathcal{S}(\mathcal{I}))}$.

In fact, the operations to which we have given semantics are more general than what is available in Full Maude at present. We have also given a precise semantics to the notions of parameterized theories and parameterized views, which we plan to support in the Full Maude system in the future.

Another natural extension of the current categorical setting would be to generalize it to give support to multi-formalism specifications. In the case of rewriting logic and Maude, different institutions coexist thanks to the embedding of membership equational logic into rewriting logic. However, this approach does not give a concrete meaning to what happens to specifications in different institutions when they are combined. A necessary prerequisite for a sensible use of several logical systems together is that they are related somehow. In the work of institutions this is captured by various notions, such as institution morphisms [53], maps of institutions [75], etc. Some recent research has addressed the heterogeneity problem [35, 105]. The CafeOBJ [36] module system updates in an interesting way the original Clear/OBJ concepts to the situation of multi-paradigm systems involving theory morphisms across institution embeddings [35, 39].

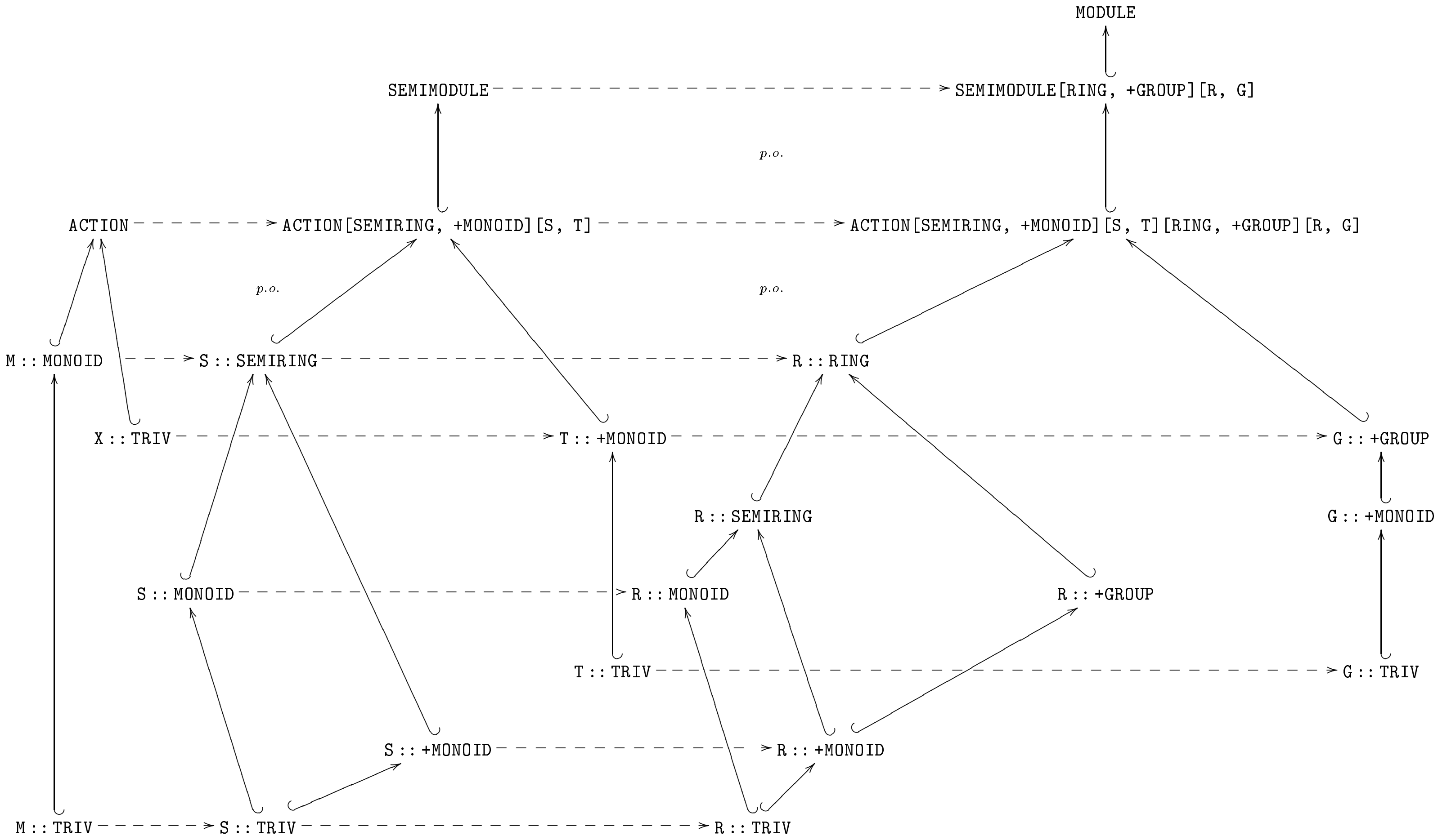


Figure 2.5: The structured theory MODULE.

Chapter 3

The Design of Full Maude

3.1 The Full Maude Execution Environment

During the development of the Maude system we have put special emphasis on the creation of metaprogramming facilities to allow the generation of execution environments for a wide variety of languages and logics. The first most obvious area where Maude can be used as a metalanguage is in building language extensions for Maude itself. Our experience in this regard—first reported in [41], and further documented in [21, 42] and in this thesis—is very encouraging. We have been able to define in Core Maude a language extension with notation for object-oriented programming, parameterized modules, views (for module instantiation), and module expressions [41]. Furthermore, using the predefined `META-LEVEL` and `LOOP-MODE` modules (see Section 1.3.4 and [21]), we have also been able to define in Core Maude all the additional functionality required for parsing, evaluating, and pretty printing modules in the extended language, as already discussed in Section 1.3.4.

Thanks to the efficient implementation of the rewrite engine, the parser, and the module `META-LEVEL`, such a language extension executes with reasonable efficiency. In the future, however, we may support in Core Maude a significant part of the functionality currently supported by Full Maude. Full Maude contains Core Maude as a sublanguage, so that Core Maude modules can also be entered at the Full Maude level. However, at present there are a few syntactic restrictions that have to be satisfied by modules and commands in order to be acceptable inputs at the Full Maude level. These syntactic restrictions will be explained as they appear along the presentation; they will be removed in the future.

We give the concrete syntax of the different constructs of the language by giving the declarations of the signature used in the parsing of the inputs to Full Maude. This syntax is given in its entirety in Appendix A.

Maude essentially contains the OBJ3 [60] language as an equational sublanguage. We have also followed it quite closely in the design of the module operations specified in Full Maude. In particular, the notion of *module expression* plays a central rôle in Maude as it does in OBJ3. We borrow from OBJ3 the syntax for the definition of parameterized modules and for the different module expressions, including the different kinds of renamings. In OBJ3, sorts and operators can be qualified by the name of the module they come from. In Maude we have taken a different approach. Although the *flattening* of structures is still based on the categorical concept of colimit, instead of considering the category of specifications and specification morphisms, flattening is understood as the colimit in the category of specifications and inclusions of specifications. The colimit of a diagram in this category coincides with the set-theoretic union of the theories in the diagram. We thus have a situation similar to what we have in languages like ASL [99, 111] or CASL [31].

The specification of Full Maude uses the `LOOP-MODE` module in order to allow the entering of modules into the system, and to maintain a persistent database in which to store all the modules, theories, and views being introduced. Let us recall from Section 1.3.4 that to get something into the `LOOP-MODE` system, the text *has to be enclosed in parentheses.* This means that any module or command intended for Full Maude has to be written enclosed in parentheses.

3.2 Functional and System Modules

A Core Maude module, such as those presented in Sections 1.3.1 and 1.3.2, can be entered in Full Maude by enclosing it in parentheses. For example, the module PATH given in Section 1.3.1 can be entered to Full Maude as follows.

```
Maude> (fmod PATH is
          protecting MACHINE-INT .

          sorts Edge Path Path? Node .
          subsorts Edge < Path < Path? .

          ops n1 n2 n3 n4 n5 : -> Node .
          ops a b c d e : -> Edge .
          op _;_ : Path? Path? -> Path? [assoc] .
          ops source target : Path -> Node .
          op length : Path -> MachineInt .

          var E : Edge .
          var P : Path .

          cmb E ; P : Path if target(E) == source(P) .

          ceq source(E ; P) = source(E) if E ; P : Path .
          ceq target(P ; E) = target(E) if P ; E : Path .
          eq length(E) = 1 .
          ceq length(E ; P) = 1 + length(P) if E ; P : Path .

          eq source(a) = n1 .
          eq target(a) = n2 .
          eq source(b) = n1 .
          eq target(b) = n3 .
          eq source(c) = n3 .
          eq target(c) = n4 .
          eq source(d) = n4 .
          eq target(d) = n2 .
          eq source(e) = n2 .
          eq target(e) = n5 .
        endfm)
```

As in Core Maude, we can enter any module or command by writing it directly after the prompt, or by having it in a file and then using the `in` command of Core Maude. Also, as in Core Maude, we can write several Full Maude modules or commands in a file and then enter all of them with a single `in` command; but each of the modules or commands has to be enclosed in parentheses. In Full Maude, in addition to functional and system modules and some of the Core Maude commands, we can enter object-oriented modules, parameterized modules, theories, views, and some additional commands. We discuss all these concepts in the coming sections.

As we shall discuss in Section 3.5, we can do some reduction or rewriting using a syntax for commands such as that of Core Maude (although with some minor differences).

```
Maude> (red b ; c ; d .)
Result Path? : b ; c ; d

Maude> (red length(b ; c ; d) .)
```

```
Result NzMachineInt : 3

Maude> (red a ; b ; c .)
Result Path? : a ; b ; c

Maude> (red source(a ; b ; c) .)
Result errorSort(Node) : source(a ; b ; c)

Maude> (red target(a ; b ; c) .)
Result errorSort(Node) : target(a ; b ; c)

Maude> (red length(a ; b ; c) .)
Result errorSort(MachineInt) : length(a ; b ; c)
```

In the rest of this chapter we describe the syntax of Full Maude giving the declarations of sorts, subsort relations, and operators included in the actual module used in the parsing of the inputs to Full Maude. We use the techniques described in Section 1.3.6 to parse these inputs. In addition to the declarations for `Token`, `Bubble`, and `NeTokenList` introduced in Section 1.3.6, we need to add declarations for two new kinds of tokens, namely `ViewToken` and `SortToken`. These two sorts are just particular cases of tokens which exclude several identifiers.

```
op viewToken : Qid -> ViewToken
       [special (id-hook Bubble        (1 1)
                 op-hook qidBaseSymbol (<Qids> : -> Qid)
                 id-hook Exclude (assoc associative
                                  comm commutative
                                  idem idempotent))] .
op sortToken : Qid -> SortToken
       [special (id-hook Bubble        (1 1)
                 op-hook qidBaseSymbol (<Qids> : -> Qid)
                 id-hook Exclude ([ ] < to : , . ( )))] .
```

To be able to use parentheses around elements of bubble sorts and of other sorts in the same connected components, explicit declarations need to be introduced. Since we want to allow the possibility of using parentheses, for example, to enclose the names of the operators in their declarations, or with module expressions, we introduce the declaration[1]

```
op ((_)) : Token -> Token .
```

With this declaration we can declare an operator `_+_` as

```
op (_+_) : Nat Nat -> Nat .
```

or have a module expression such as

```
(LIST * (op __ to _._))[Nat]
```

which uses a module `LIST` having an operator `__`, where `LIST` is parameterized by the `TRIV` theory, and where `Nat` is a view from `TRIV` to a module `NAT`.

[1] Note that to declare an operator as `(_)` it has to be declared enclosed in a second pair of parentheses [21].

The basic syntax for the declarations of sorts, subsort relations, and operators[2] in Full Maude modules is given by the following declarations[3].

```
subsorts SortToken < Sort < SortList .
subsort Attr < AttrList .

op __ : SortList SortList -> SortList [assoc] .
op sort_. : SortList -> SortDecl .
op sorts_. : SortList -> SortDecl .

op _<_ : SortList SortList -> SubsortRel .
op _<_ : SortList SubsortRel -> SubsortRel .
op subsort_. : SubsortRel -> SubsortDecl .
op subsorts_. : SubsortRel -> SubsortDecl .

op assoc : -> Attr .
op comm : -> Attr .
op id:_ : Bubble -> Attr .
op left id:_ : Bubble -> Attr .
op right id:_ : Bubble -> Attr .
op strat(_) : NeTokenList -> AttrList .
op prec_ : Token -> Attr .
op gather(_) : NeTokenList -> Attr .
op idem : -> Attr .
op __ : AttrList AttrList -> AttrList [assoc] .

op op_: ->_. : Token Sort -> OpDecl .
op op_: ->_[_]. : Token Sort AttrList -> OpDecl .
op op_:_->_. : Token SortList Sort -> OpDecl .
op op_:_->_[_]. : Token SortList Sort AttrList -> OpDecl .
op ops_: ->_. : NeTokenList Sort -> OpDecl .
op ops_: ->_[_]. : NeTokenList Sort AttrList -> OpDecl .
op ops_:_->_. : NeTokenList SortList Sort -> OpDecl .
op ops_:_->_[_]. : NeTokenList SortList Sort AttrList -> OpDecl .
```

Full Maude supports not only module hierarchies, as discussed in Section 1.3.3 for Core Maude, but also parameterized programming techniques in the OBJ3 style. In particular, Full Maude supports importations of modules in `including` and `protecting` modes, not only for user-defined modules, but, as we will see in Section 3.6.4, for module expressions as well. The syntax for importation declarations is as follows.

```
subsort Token < ModExp .

op including_. : ModExp -> ImportDecl .
op protecting_. : ModExp -> ImportDecl .
```

[2]Note that the declarations of operators have an element of sort `Token` (or `NeTokenList` for multiple declarations) as first argument. Therefore, in operator declarations, operator names have to be single identifiers. This does not mean that we cannot use multi-identifier names. We can, but their equivalent single form must be used instead in their declarations. All Maude identifiers have an equivalent single identifier form given by placing a backquote in front of each of the component of the identifier. For example, an operator `_less than or equal_` has to be declared using its equivalent identifier ``_less`than`or`equal_``. However, multi-identifier operators can still be used in expressions with their usual mixfix form. For example, assuming that the operator above was declared on integers, we can have a term like `3 less than or equal 2`.

[3]We do not give all the declarations in this chapter. The complete set of declarations can be found in Appendix A.

The syntax for the declaration of variables, membership axioms, equations, and rules is the following.

```
op vars_:_. : NeTokenList Sort -> VarDecl .
op var_:_. : NeTokenList Sort -> VarDecl .

op mb_:_. : Bubble Sort -> MembAxDecl .
op cmb_:_if_. : Bubble Sort Bubble -> MembAxDecl .

op eq_=_. : Bubble Bubble -> EquationDecl .
op ceq_=_if_. : Bubble Bubble Bubble -> EquationDecl .

op rl[_]:_=>_. : Token Bubble Bubble -> RuleDecl .
op crl[_]:_=>_if_. : Token Bubble Bubble Bubble -> RuleDecl .
```

The remaining top-level syntax for functional and system modules is given by the following declarations.

```
subsort VarDecl < VarDeclList .
op __ : VarDeclList VarDeclList -> VarDeclList [assoc] .

subsorts ImportDecl SortDecl SubsortDecl OpDecl
         MembAxDecl EquationDecl VarDeclList < FDeclList .
op __ : FDeclList FDeclList -> FDeclList [assoc] .

subsorts RuleDecl FDeclList < SDeclList .
op __ : SDeclList SDeclList -> SDeclList [assoc] .

subsort Token < ModuleName .
op fmod_is_endfm : ModuleName FDeclList -> PreUnit .
op mod_is_endm : ModuleName SDeclList -> PreUnit .
```

3.3 Object-Oriented Modules

Full Maude supports object-oriented notation as described by Meseguer in [77]. In a concurrent object-oriented system, the concurrent state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages that evolve by concurrent ACU-rewriting[4] using rules that describe the effects of *communication events* between some objects and messages. Intuitively, we can think of messages as "traveling" to come into contact with the objects to which they are sent, and then causing "communication events" by application of rewrite rules. Therefore, we can view concurrent object-oriented computation as *deduction* in rewriting logic; in this way, the configurations S that are *reachable* from a given initial configuration S_0 are exactly those such that the sequent $S_0 \longrightarrow S$ is *provable* in rewriting logic using the rewrite rules that specify the behavior of the given object-oriented system.

An *object* in a given state is represented as a term

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ a_1 \texttt{:}\ v_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n \texttt{:}\ v_n\ \texttt{>}$$

where O is the object's name or identifier, C is its class identifier, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. In particular, an object with no attributes can be represented as

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ \texttt{>}$$

[4] We call rewriting modulo associativity, commutativity, and identity *ACU-rewriting*.

Messages do not have a fixed syntactic form. Such syntactic form can be defined by the user for each application. The concurrent state of an object-oriented system is then a multiset of objects and messages, called a `Configuration`, with multiset union described with empty syntax `__`.

The following module `CONFIGURATION` defines the basic concepts of concurrent object systems. Note that the sorts `Msg` and `Attribute`—as well as the sorts `Oid` and `Cid` of object and class identifiers—are for the moment left unspecified. They will become fully defined when the `CONFIGURATION` module is extended by specific object-oriented definitions in a given object-oriented module.

```
fmod CONFIGURATION is
  sorts Oid Cid Attribute AttributeSet
        Object Msg Configuration .
  subsorts Object Msg < Configuration .
  subsort Attribute < AttributeSet .

  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
          [assoc comm id: none] .

  op <_:_| > : Oid Cid -> Object .
  op <_:_|_> : Oid Cid AttributeSet -> Object .

  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
          [assoc comm id: none] .
endfm
```

In Full Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod`—using a syntax more convenient than that of system modules because it assumes acquaintance with the basic entities, such as objects, messages, and configurations, and supports linguistic distinctions appropriate for the object-oriented case. In particular, all object-oriented modules implicitly include the above `CONFIGURATION` module and assume its syntax. For example, the `ACCNT` object-oriented module below specifies the concurrent behavior of objects in a very simple class `Accnt` of bank accounts, each having a `bal`(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts.

```
(omod ACCNT is
  protecting QID .
  protecting MACHINE-INT .

  subsort Qid < Oid .

  class Accnt | bal : MachineInt .

  msgs credit debit : Oid MachineInt -> Msg .
  msg transfer_from_to_ : MachineInt Oid Oid -> Msg .

  vars A B : Oid .
  vars M N N' : MachineInt .

  rl [credit] : credit(A, M) < A : Accnt | bal : N >
     => < A : Accnt | bal : (N + M) > .
```

```
  crl [debit] : debit(A, M) < A : Accnt | bal : N >
     => < A : Accnt | bal : (N - M) >
        if N > M .
  crl [transfer] : (transfer M from A to B)
     < A : Accnt | bal : N > < B : Accnt | bal : N' >
     => < A : Accnt | bal : (N - M) >
        < B : Accnt | bal : (N' + M) >
        if N > M .
endom)
```

3.3.1 The Syntax of Object-Oriented Modules

Classes are defined with the keyword `class`, followed by the name of the class C, and by a list of attribute declarations separated by commas. Each attribute declaration has the form $a : S$, where a is an attribute identifier and S is the sort in which the values of the attribute identifier range. That is, class declarations have the form

$$\texttt{class}\ C\ \texttt{|}\ a_1 \texttt{:}\ S_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n \texttt{:}\ S_n\ \texttt{.}$$

In particular, we can also declare classes without attributes using syntax

$$\texttt{class}\ C\ \texttt{.}$$

The basic syntax for class declarations is given by the following operators and subsort relations.

```
subsort AttrDecl < AttrDeclList .
op _:_ : Token Sort -> AttrDecl [prec 40] .
op _,_ : AttrDeclList AttrDeclList -> AttrDeclList [assoc] .

op class_|_. : Sort AttrDeclList -> ClassDecl .
op class_. : Sort -> ClassDecl .
```

In the example in the previous section, the only attribute of an account is its `bal`(ance), which is declared to be a value in `MachineInt`, a sort declared in the predefined module `MACHINE-INT`. The three kinds of messages involving accounts are `credit`, `debit`, and `transfer` messages, whose user-definable syntax is introduced by the keyword `msg`. Notice the use of `msgs` to define multiple messages with the same arity in a single declaration.

The syntax for message declarations[5] is given by the following operators:

```
op msg_:_->_. : Token SortList Sort -> MsgDecl .
op msgs_:_->_. : NeTokenList SortList Sort -> MsgDecl .
```

The rewrite rules in the module specify in a declarative way the behavior associated with the messages. The multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the rules [77]. For example, we can rewrite a simple configuration consisting of an account and a message as follows.

```
Maude> (rew < 'Peter : Accnt | bal : 2000 >
            debit('Peter, 1000) .)

Result Object : < 'Peter : Accnt | bal : 1000 >
```

[5] The names used in the declarations of messages have the same restrictions as operator names as explained in footnote 2, page 80.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` in an object-oriented module is just a particular case of a subsort declaration `C < C'`. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses as well as the newly defined attributes, messages, and rules of the subclass characterize the structure and behavior of the objects in the subclass.

For example, we can define an object-oriented module `SAV-ACCNT` of saving accounts introducing a subclass `SavAccnt` of `Accnt` with a new attribute `rate` recording the interest rate of the account. We leave unspecified the rules for computing and crediting the interest of an account according to its rate whose proper expression should introduce a real-time[6] attribute in account objects.

```
(omod SAV-ACCNT is
  including ACCNT .
  class SavAccnt | rate : MachineInt .
  subclass SavAccnt < Accnt .
 endom)
```

In this example, there is only one class immediately above `SavAccnt`, namely, `Accnt`. In general, however, a class C may be defined as a subclass of several classes $D_1, \ldots, D_k$, i.e., *multiple inheritance* is supported. If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass. Indeed, all such attributes are *inherited*. In the case of multiple inheritance, the only requirement that is made is that if an attribute occurs in two different superclasses, then the sort attributed to it in each of those superclasses must be the same. In summary, a class inherits all the attributes, messages, and rules from all its superclasses. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages, and rules in the subclass.

As for subsort relations, we can declare multiple subclass relations in the same declaration. Thus, given, for example, classes `A`, ..., `H`, we can have a declaration such as

```
subclasses A B C < D E < F G H .
```

Since class names have the same form as sorts, in the signature used to parse Full Maude given in Appendix A, we use the sort `Sort` to parse them, and the sort `SortList` for lists of names of classes. The syntax for subclass declarations is given by the following operators.

```
op subclass_. : SubsortRel -> SubclassDecl .
op subclasses_. : SubsortRel -> SubclassDecl .
```

Objects in the class `SavAccnt` will have an attribute `bal` and can receive messages debiting, crediting, and transferring funds exactly as any other object in the class `Accnt`. We can now rewrite a configuration, obtaining the following result.

```
Maude> (rew < 'Paul : SavAccnt | bal : 5000, rate : 3 >
            < 'Peter : Accnt | bal : 2000 >
            < 'Mary : SavAccnt | bal : 9000, rate : 3 >
            debit('Peter, 1000)
            credit('Paul, 1300)
```

[6]See [90] for a general method to specify real-time systems in rewriting logic.

```
            credit('Mary, 200) .)

Result Configuration :
            < 'Peter : Accnt | bal : 1000 >
            < 'Paul : SavAccnt | bal : 6300 , rate : 3 >
            < 'Mary : SavAccnt | bal : 9200 , rate : 3 >
```

The remaining top-level syntax for object-oriented modules is given by the following declarations.

```
subsorts SDeclList MsgDecl SubclassDecl ClassDecl < ODeclList .

op omod_is_endom : ModuleName ODeclList -> PreUnit .
```

3.3.2 Transforming Object-Oriented Modules into System Modules

The best way to understand classes and class inheritance in Maude is by making explicit the full structure of an object-oriented module, which is left somewhat implicit in the syntactic conventions adopted for them. Indeed, although Maude's object-oriented modules provide convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules. We can regard the special syntax reserved for object-oriented modules as syntactic sugar. In fact, each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module. We will describe in Section 4.6.6 how this transformation is accomplished.

However, although Maude's object-oriented modules can in this way be reduced to system modules, there are of course important conceptual advantages provided by the syntax of object-oriented modules, because it allows the user to think and express his or her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be lost if only system modules were provided.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the `CONFIGURATION` functional module defined at the beginning of this section. The translation of a given object-oriented module extends this structure with the classes, messages, and rules introduced by the module. For example, the following system module is the translation of the `ACCNT` module introduced earlier. Note that a subsort `Accnt` of `Cid` is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `Accnt`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `Accnt` is the class identifier `Accnt`.

```
mod ACCNT is
   including MACHINE-INT .
   including QID .
   including CONFIGURATION .
   sorts Accnt .
   subsort Qid < Oid .
   subsort Accnt < Cid .
   op Accnt : -> Accnt .
   op credit : Oid MachineInt -> Msg .
   op debit : Oid MachineInt -> Msg .
   op transfer_from_to_ : MachineInt Oid Oid -> Msg .
```

```
    op bal :_ : MachineInt -> Attribute .
    var A : Oid .
    var B : Oid .
    var M : MachineInt .
    var N : MachineInt .
    var N' : MachineInt .
    var V@Accnt : Accnt .
    var ATTS@0 : AttributeSet .
    var V@Accnt1 : Accnt .
    var ATTS@2 : AttributeSet .
    rl [credit] : credit(A, M)
            < A : V@Accnt | bal : N, none, ATTS@0 >
      => < A : V@Accnt | bal : (N + M), ATTS@0 > .
    crl [debit] : debit(A, M)
            < A : V@Accnt | bal : N, none, ATTS@0 >
      => < A : V@Accnt | bal : (N - M), ATTS@0 >
         if N > M = true .
    crl [transfer] : (transfer M from A to B)
            < A : V@Accnt | bal : N, none, ATTS@0 >
            < B : V@Accnt1 | bal : N', none, ATTS@2 >
      => < A : V@Accnt | bal : (N - M), ATTS@0 >
         < B : V@Accnt1 | bal : (N' + M), ATTS@2 >
         if N > M = true .
endm
```

We can describe the desired transformation from an object-oriented module to a system module as follows[7]:

- The module `CONFIGURATION` is imported.
- For each class declaration of the form `class` C `|` a_1`:`S_1`,` ... `,`a_n`:`S_n, the following has to be introduced: a subsort C of sort `Cid`, a constant C of sort C, and declarations of operators a_i `:_ :` S_i `-> Attribute` for each attribute a_i.
- For each subclass relation C `<` C' a subsort declaration

  `subsort` C `<` C' `.`

  is introduced, and the set of attributes for objects of class C is completed with those of C'.
- The membership axioms, equations, and rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given. The membership axioms, equations, and rules are then "inherited" by all objects in their subclasses by replacing the class identifiers in the objects in the axioms by variables of the corresponding class sort. Variables of sort `AttributeSet` are also introduced to range over the additional attributes that may appear in objects of a subclass. That is, each object `<` O `:` C `|` ... `>` appearing in a membership axiom, equation, or rule, is translated into `<` O `:` X `|` ... `,` *Atts* `>` where the new variable X is declared of sort C, and the new variable *Atts* has sort `AttributeSet`.

[7]Notice that we have simplified the transformation of object-oriented modules into system modules that originally appeared in [77].

- As described in [77], we simplify the notation used in object-oriented modules by giving the user the possibility of not mentioning in a given membership axiom, equation, or rule those attributes of an object that are not relevant for that axiom. To explain this convention, let $\overline{a : v}$ denote the attribute-value pairs $a_1 : v_1, \ldots , a_n : v_n$, where the $\overline{a}$ are the attribute identifiers of a given class C (after completing it with all the attributes in its superclasses) having $\overline{S}$ as the corresponding sorts of values prescribed for those attributes. Then, in object-oriented modules we allow equations and rules where the attributes for an object O, mentioned in the lefthand and righthand sides of such an equation or rule, need not exhaust all the object's attributes, but can instead be in any two arbitrary subsets of the object's attributes. For rules, for example, we can picture this as follows

$$\ldots\langle O : C \mid \overline{al : vl}, \overline{ab : vb}\rangle \ldots \;\longrightarrow\; \ldots\langle O : C \mid \overline{ab : vb'}, \overline{ar : vr}\rangle \ldots$$

  where $\overline{al}$ are the attributes appearing only on the *left*, $\overline{ab}$ are the attributes appearing on *both* sides, and $\overline{ar}$ are the attributes appearing only on the *right*. In the transformation into a system module, this rule is translated into

$$\begin{array}{l}\ldots\langle O : X \mid \overline{al : vl}, \overline{ab : vb}, \overline{ar : x}, \overline{ac : x'}, Atts\rangle \ldots \\ \quad\longrightarrow\; \ldots\langle O : X \mid \overline{al : vl}, \overline{ab : vb'}, \overline{ar : vr}, \overline{ac : x'}, Atts\rangle \ldots\end{array}$$

  where X is a variable of sort C, $\overline{ac}$ are the attributes defined in the class C that do not appear in $\overline{al}$, $\overline{ab}$, or $\overline{ar}$, the $\overline{x}$ and $\overline{x'}$ are new variables of the appropriate sorts, and *Atts* matches the remaining attribute-value pairs. Although the form of the rule obtained is slightly different from the one given in [77], the convention is similar to the convention presented there: The attributes mentioned only on the left are preserved unchanged, the original values of attributes mentioned only on the right do not matter, and all attributes not explicitly mentioned are left unchanged.

The rewrite rules given in the original `ACCNT` module are interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of `Accnt` that could be defined later. Thus, `SavAccnt` inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for `Accnt`.

Let us illustrate the treatment of class inheritance with the system module resulting from the transformation of the module `SAV-ACCNT` introduced in page 84.

```
mod SAV-ACCNT is
  including ACCNT .
  sort SavAccnt .
  subsort SavAccnt < Cid .
  subsort SavAccnt < Accnt .
  op SavAccnt : -> SavAccnt .
  op rate :_ : MachineInt -> Attribute .
endm
```

3.4 Structured Specifications and Extensions of `META-LEVEL`

As explained for Core Maude in Section 1.3.3, in Full Maude we can use the keywords `protecting` and `including`, or `pr` and `inc` in abbreviated form, to define structured specifications. All the predefined modules introduced in Section 1.3.4, except the module `LOOP-MODE`, are also available in Full Maude. As we explain in Section 3.6, Full Maude supports not only the importation of modules, but the importation of module expressions as well.

In metalevel computations it is very convenient to be able to refer by name to the metarepresentations of modules already entered into the system. We discussed in Section 1.3.6 several ways of having access to these metarepresentations. To make it more convenient, Full Maude allows importation declarations of the form

$$\texttt{protecting META-LEVEL[}Id_1, \ldots , Id_n\texttt{] .}$$

where $Id_1, \ldots , Id_n$ is a list of names of user-defined modules. The effect of this declaration is that constants $Id_1, \ldots , Id_n$ of sort `Module` are declared, and equations making each constant Id_i equal to the metalevel representation of the module with name Id_i declared previously by the user, for $i = 1 \ldots n$, are added. Thus, for example, we can first enter the module

```
(fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
  var N M : Nat .
  eq s N + s M = s s (N + M) .
 endfm)
```

and then we can declare a module that imports `META-LEVEL[NAT]` and defines a function to extract the set of operator declarations of that functional module as follows.

```
(fmod META-NAT is
  protecting META-LEVEL[NAT] .

  op getOpDeclSet : FModule -> OpDeclSet .

  var QI : Qid .
  var IL : ImportList .
  var SD : SortDecl .
  var SSDS : SubsortDeclSet .
  var ODS : OpDeclSet .
  var VDS : VarDeclSet .
  var MAS : MembAxSet .
  var EqS : EquationSet .

  eq getOpDeclSet(fmod QI is IL SD SSDS ODS VDS MAS EqS endfm)
    = ODS .
 endfm)
```

Note that, when importing a module at the object level in the metarepresentation of a module as described in Section 1.3.6, we do not have access to the metarepresentation itself. We only can use modules obtained in such a way in calls to built-in functions. In Full Maude, the above mechanism for extending `META-LEVEL` with the metarepresentation of other modules does not only simplify the use of modules at the metalevel, but in fact allows access to the metarepresentation of the imported module. For example, we can now apply the `getOpDeclSet` function to the constant `NAT`, which in `META-NAT` has been declared to be equal to the metarepresentation of the above module NAT, as follows.

```
Maude> (red getOpDeclSet(NAT) .)
Result OpDeclSet :
   op '0 : nil -> 'Nat [none] .
   op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id({'0}'Nat)] .
   op 's_ : 'Nat -> 'Nat [none] .
```

We can also use the descent functions discussed in Section 1.3.4.

```
Maude> (red meta-reduce(NAT, '_+_[{'0}'Nat, 's_[{'0}'Nat]]) .)
Result Term : 's_[{'0}'Nat]
```

Note that we have written the actual metarepresentation of the term `0 + s 0` instead of using the more intuitive notation $\overline{\texttt{0 + s 0}}$ used in Section 1.3.4. However, in Full Maude, we can use the `up` function to avoid the cumbersome task of explicitly writing the metarepresentation of a term or the metarepresentation of a module. For example, to obtain the metarepresentation of a term such as `s 0` in the module NAT, which we denote by $\overline{\texttt{s 0}}$, we can write

```
Maude> (red up(NAT, s 0) .)
Result Term : 's_[{'0}'Nat]
```

Therefore, instead of explicitly writing the metarepresentation $\overline{\texttt{0 + s 0}}$ in the above reduction we can write

```
Maude> (red meta-reduce(NAT, up(NAT, 0 + s 0)) .)
Result Term : 's_[{'0}'Nat]
```

Note that the module name is the first argument of the `up` function, with the term of that module to be metarepresented as the second argument. Since the same term can be parsed in different ways in different modules, and therefore can have different metarepresentations depending on the module in which it is considered, the module to which the term belongs has to be used to obtain the correct metarepresentation. Note also that the above reduction only makes sense at the metalevel, that is, in a module importing the module META-LEVEL.

The `up` function also gives us a second way of accessing the metarepresentation of any module in the database. In any module importing the module META-LEVEL we can evaluate the `up` function with the name of any module in the database as argument. We then obtain the metarepresentation of such a module. Thus, assuming that the previous module NAT has been entered in Full Maude, and therefore it is in the database, we can get its metarepresentation, which we denote by $\overline{\texttt{NAT}}$, by evaluating in META-LEVEL, or in any other module importing META-LEVEL, the following command.

```
Maude> (red up(NAT) .)
Result FModule :
    fmod 'NAT is
       nil
       sorts 'Nat .
       none
       op '0 : nil -> 'Nat [none] .
       op '_+_ : 'Nat 'Nat -> 'Nat [assoc comm id({'0}'Nat)] .
       op 's_ : 'Nat -> 'Nat [none] .
       var 'M : 'Nat .
```

```
        var 'N : 'Nat .
        none
        eq '_+_['s_['N], 's_['M]] = 's_['s_['_+_['N, 'M]]].
     endfm
```

This facility can be used to write reductions of terms as those presented in Section 1.3.4, for example of $\texttt{meta-reduce(}\overline{\texttt{NAT}}\texttt{, }\overline{\texttt{s s 0 + s s s 0}}\texttt{)}$, as follows.

```
Maude> (red meta-reduce(up(NAT), up(NAT, s s 0 + s s s 0)) .)
Result Term : 's_['s_['s_['s_['s_[{'0}'Nat]]]]]
```

The result of a metalevel computation that may use several levels of reflection can be a term or module metarepresented one or more times, which may be hard to read. Therefore, to display the output in a more readable form we can use the `down` command, which is in a sense inverse to `up`, since it gives us back the term from its metarepresentation. The `down` command takes two arguments. The first argument is the name of the module to which the term to be returned belongs. The metarepresentation of the desired output term should be the result of the command given as second argument. The syntax of the `down` command is as follows.

```
op down_:_ : ModExp PreCommand -> PreCommand .
```

Thus, we can give the following command.

```
Maude> (down NAT :
           red-in META-NAT :
              meta-reduce(NAT, up(NAT, 0 + s 0)) .)
Result Nat : s 0
```

Notice that this is equivalent to what we wrote in section 1.3.4 as

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{NAT}}\texttt{, }\overline{\texttt{s 0 + 0}}\texttt{) .}$$
$$\texttt{result Term: }\overline{\texttt{s 0}}$$

The use of `up` and `down` can be iterated with as many levels of reflection as we wish. For example, in the module

```
(fmod META-META-NAT is
   protecting META-LEVEL[META-NAT] .
 endfm)
```

we can give the command

```
Maude> (down NAT :
           down META-NAT :
               red meta-reduce(META-NAT,
                       up(META-NAT,
                          meta-reduce(NAT,
                              up(NAT, 0 + s 0)))) .)
Result Nat : s 0
```

This is equivalent to what we would have written using the overline notation as

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{META-NAT}}\texttt{, }\overline{\texttt{meta-reduce(}\overline{\texttt{NAT}}\texttt{, }\overline{\texttt{s 0 + 0}}\texttt{)}}\texttt{) .}$$
$$\texttt{result Term: }\overline{\overline{\texttt{s 0}}}$$

3.5 Commands and the Module Database

As with modules, all commands at the Full Maude level should be entered enclosed in parentheses. In this way the system can distinguish between commands at the Core Maude level—that in this context are "system programming" commands in the module `FULL-MAUDE`—and commands to be handled by Full Maude.

The `reduce` and `rewrite` commands have the same effect in Full Maude specifications as their homonymous commands in Core Maude.

The syntax for the `reduce` commands is given by the following declarations. Notice that, as in Core Maude, `red` is used as an abbreviation for `reduce`. However, the command to reduce a term in a given module has a somewhat different syntax from the one used in Core Maude[8].

```
op red_. : Bubble -> PreCommand .
op red-in_:_. : ModExp Bubble -> PreCommand .
```

Similarly, the following are the declarations defining the syntax of the rewrite commands.

```
op rew_. : Bubble -> PreCommand .
op rew-in_:_. : ModExp Bubble -> PreCommand .
```

Full Maude maintains a database with all the modules that have been introduced since the beginning of the session. Except for the built-in modules, Core Maude and Full Maude keep independent module stores.

As in Core Maude, when a module is not explicitly specified the system uses a module by default, that in general is the last module introduced, although one can select another module from the database with the `select` command.

```
op select_. : ModExp -> PreCommand .
```

There are also several `show` commands as in Core Maude. There are commands to show a module or theory[9] as introduced by the user, to show the flattened version of any module in the database, and to show some of the components in a module in the same way as the commands for Core Maude (see [21]). For any of these commands the name of a module can be given. If no name is specified, the default current module is used. The syntax of these commands is as follows.

```
op show module . : -> PreCommand .
op show module_. : ModExp -> PreCommand .
op show all . : -> PreCommand .
op show all_. : ModExp -> PreCommand .
op show sorts . : -> PreCommand .
op show sorts_. : ModExp -> PreCommand .
op show ops . : -> PreCommand .
```

[8]To have, for example, the command `red in_:_.` we would need the operator declaration

```
op red in_:_. : ModExp Bubble -> PreCommand .
```

However, given a command like `red in NAT : s 0 .` there would be two possible parses: `red in` $\boxed{\texttt{NAT}}$ `:` $\boxed{\texttt{s 0}}$ `.` and `red` $\boxed{\texttt{in NAT : s 0}}$ `..` Both of them are correct parses, but the `meta-parse` function returns only one of them. In case of ambiguity, one of the possible parses is arbitrarily chosen, preventing us from the possibility of taking the right one. This syntactic limitation, as well as those discussed along this chapter, will be overcome in a future version.

[9]Theories are discussed in Section 3.6.1.

```
op show ops_. : ModExp -> PreCommand .
op show vars . : -> PreCommand .
op show vars_. : ModExp -> PreCommand .
op show mbs . : -> PreCommand .
op show mbs_. : ModExp -> PreCommand .
op show eqns . : -> PreCommand .
op show eqns_. : ModExp -> PreCommand .
op show rls . : -> PreCommand .
op show rls_. : ModExp -> PreCommand .
```

The `show view` command prints the view[10] with the specified name.

```
op show view_. : ViewExp -> PreCommand .
```

The `show modules` and `show views` commands print, respectively, the list of the names of all modules, and the list of the names of all views, present in the database.

```
op show modules . : -> PreCommand .
op show views . : -> PreCommand .
```

3.6 Parameterized Programming

Parameterized modules, theories, and views are the basic building blocks of parameterized programming [15, 60]. As in OBJ3, a theory defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. The instantiation of the formal parameters of a parameterized module with actual parameter modules requires a *view* from the formal interface theory to the corresponding actual module. That is, views provide the interpretation of the actual parameters.

3.6.1 Theories

Theories are used, among other things, to declare the interfaces of parameterized modules, that is, the syntactic and semantic properties that must be satisfied by the actual parameter modules used in an instantiation. As for modules, Full Maude supports three different types of theories: functional theories, system theories, and object-oriented theories. Their structure is the same as that of their module counterparts. All of them can have declarations of sorts, subsort relations, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can have rules as well, and object-oriented theories can have declarations of classes, subclass relations, and messages. However, in the present version of Full Maude, theories cannot be parameterized by other theories.

As functional modules, *functional theories* are membership equational logic theories, but they do not need to be Church-Rosser; they have a *loose* interpretation, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model.

Theories are then allowed to contain rules and equations with variables in their righthand sides or conditions that may not appear in their corresponding lefthand sides. Similarly, conditional membership axioms may have variables in their conditions that do not appear in their membership assertions. Also, the

[10] Views are discussed in Section 3.6.3.

lefthand side may be a single variable. In the current version, theories are not executed and, as already mentioned, cannot be parameterized.

Functional theories are declared with the keywords `fth ... endfth`, system theories with the keywords `th ... endth`, and object-oriented theories with the keywords `oth ... endoth`. The syntax for the declaration of theories is as follows.

```
op fth_is_endfth : ModuleName FDeclList -> PreUnit .
op th_is_endth : ModuleName SDeclList -> PreUnit .
op oth_is_endoth : ModuleName ODeclList -> PreUnit .
```

Let us begin by introducing the functional theory `TRIV`, which requires just a sort.

```
(fth TRIV is
  sort Elt .
 endfth)
```

The theory of partially ordered sets with an anti-reflexive and transitive binary operator can be expressed in the following way.

```
(fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false .
  ceq X < Z = true if X < Y and Y < Z .
 endfth)
```

The theory of totally ordered sets, that is, posets in which all pairs of distinct elements have to be related, can be given as follows.

```
(fth TOSET is
  including POSET .
  vars X Y : Elt .
  eq X < Y or Y < X or X == Y = true .
 endfth)
```

The `including` importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore must be interpreted with an initial semantics[11], then that initial semantics is maintained by the importation. For example, in the definition of the `POSET` theory, the declaration `protecting BOOL` ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement. This requirement is then preserved by `TOSET` when `POSET` is included. In fact, we are dealing with a structure in which part of it, not only the top theory, has a loose semantics, while other parts contain modules with an initial semantics. The kind of semantics of a module or theory is determined by the keyword used in its definition and the importation mode.

As an example of a system theory, let us consider the theory `CHOICE` of multisets of elements with a choice operator defined on the multisets by a rewrite rule that nondeterministically picks up one of the elements in the multiset. We can express this theory as indicated below, where we have a sort `MSet` declared as a supersort of the sort `Elt`.

[11] In Full Maude, the importation of a module into a theory is supported only in `protecting` mode.

```
(th CHOICE is
  sort MSet Elt .
  subsort Elt < MSet .
  op mt : -> MSet .
  op __ : MSet MSet -> MSet [assoc comm id: mt] .
  var E : Elt .
  var L : MSet .
  rl [choice] : E L => E .
 endth)
```

Our last example is an object-oriented theory, namely, the theory of classes with at least one attribute of any sort. It is defined as follows.

```
(oth CELL is
  sort Elt .
  class Cell | contents : Elt .
 endoth)
```

This last theory could have been more naturally expressed as a parameterized theory. We could have defined `CELL` with a parameter `TRIV` to capture the idea of defining cells in a generic way. However, the present Full Maude implementation does not support parameterized theories. We plan to extend the language to support not only parameterized theories, but also *parameterized views*.

3.6.2 Parameterized Modules

Theories can be used to declare the interface requirements of parameterized modules. As explained in Section 1.3.1, the semantics of an unparameterized functional module is the *initial algebra* specified by its theory. The semantics of a *parameterized* functional module is the free functor associated to the inclusion of the parameter theory[12] into the body of the parameterized module [82]. For example, a parameterized list module `LIST[X :: TRIV]` forms lists of models of the trivial parameter theory `TRIV` with one sort `Elt`, whose models are sets of elements and its semantics is the functor sending each set to the algebra of lists of the set. Similarly, a sorting module `SORTING[Y :: POSET]` sorts lists whose elements belong to a model of the `POSET` functional theory, that is, the data type of elements must have a partial order and its semantics is the functor sending each poset to the algebra of lists for that poset with a sorting function. All this is entirely similar to the semantics of "objects" and theories in OBJ [60].

Modules can be parameterized by one or more theories. All theories appearing in the interface have to be labeled in such a way that their sorts can be uniquely identified. The general form for the interface of a parameterized module is `[`X_1 `::` T_1 `,` ... `,` X_n `::` T_n`]`, where $X_1 \ldots X_n$ are the labels and $T_1 \ldots T_n$ are the names of the parameter theories. Thus, the syntax of the interface of parameterized modules is given by the following declarations.

```
op _::_ : Token ModExp -> Parameter [prec 40 gather (e &)] .

subsort Parameter < ParameterList .
op _,_ : ParameterList ParameterList -> ParameterList [assoc] .

op _[_] : Token ParameterList -> ModuleName .
```

[12] Of course, if the parameterized module has several parameter theories, we should form their colimit, and consider instead the inclusion of such a colimit into the body.

As mentioned above, we follow the parameter labeling convention of OBJ3.

In the current version of Full Maude all the sorts coming from theories in the interface must be qualified by their labels, even if there is no ambiguity. If Z is the label of a parameter theory T, then each sort S in T has to be qualified as S.Z. Since, as we will see in Section 3.6.3, operator maps affect entire families of subsort-overloaded operators, subsort overloading between an operator declared in a theory being used as parameter of a parameterized module and an operator declared in the body of the parameterized module, or between operators declared in two parameter theories of the same parameterized module, is forbidden. Thus, the parameterized module SIMPLE-SET, with TRIV as interface can be defined as follows.

```
(fmod SIMPLE-SET[X :: TRIV] is
  sorts Set NeSet .
  subsorts Elt.X < NeSet < Set .
  op mt : -> Set .
  op __ : Set Set -> Set [assoc comm id: mt] .
  op __ : NeSet NeSet -> NeSet [assoc comm id: mt] .
  var E : Elt.X .
  eq E E = E .
 endfm)
```

Note that, as discussed in Section 3.1, in Maude—unlike OBJ3—sorts are not systematically qualified by their module name. In OBJ3, importing, for example, sets or lists of different elements introduces repeated sorts named Set or List, and set or list operators. To avoid ambiguous situations such sorts and operators must then be qualified by the names of the submodules they come from, that is, by module expressions often of considerable length. Of course, in OBJ3 it is possible to rename all these items. But this means that, to avoid the burden of long qualifications by module expressions, we have to perform explicitly many more renamings than we would like.

The convention of not qualifying sorts may be particularly weak when dealing with parameterized modules. However, given that Maude supports ad-hoc overloading and that constants can be qualified in order to be disambiguated, the problem of ambiguity in a signature is reduced to collisions of sorts. Our proposal consists in qualifying parameterized sorts, not with the module expression they belong to, but with the much shorter name of the view or views used in the instantiation of the parameterized module. In the current version of Full Maude, we assume that all views are named, and that these names are the ones used in the qualification. Specifically, in the body of a parameterized module M[X_1 :: T_1, ..., X_n :: T_n], any sort S declared there can be written in the form S[X_1, ..., X_n]. When the module is instantiated with views V_1, ..., V_n then the corresponding instantiated sort becomes S[V_1, ..., V_n] [13]. Note that the parameterization of sorts is optional. The above SIMPLE-SET specification, for example, is perfectly valid even though the sorts Set and NeSet are not qualified by the parameter X.

A notion similar to our parameterized sorts has been used in languages like Larch [61], LPG [6], and CASL [31]. However, none of these approaches qualifies

[13] In Maude, a term t can be qualified by a sort S by writing (t).S. If we want to use the sort S[V_1, ..., V_n] to qualify the sort of a term t, for parsing or evaluating purposes, then, since qualifications of terms by their sorts—as well as parentheses, polymorphism, and sort tests—are directly handled by Core Maude, and Core Maude does not know about parameterized sorts, the user should qualify such a term t by the sort name, not as he or she has defined it, but in its equivalent single identifier form. Thus, if we have, for example, a sort List[Nat] and an overloaded constant nil in it, if necessary, it should be qualified as (nil).List`[Nat`].

sorts in a systematic way as we do. The use of views as "parameters" of the sorts improves readability and reduces the possibility of undesired collisions of the names of sorts.

The declarations needed to allow parameterized sorts are the following.

```
subsort ViewToken < ViewExp .
op _,_ : ViewExp ViewExp -> ViewExp [assoc] .

op _[_] : Sort ViewExp -> Sort [prec 40] .
```

Thus, the previous module to define sets could instead have been defined as follows.

```
(fmod SET[X :: TRIV] is
  sorts Set[X] NeSet[X] .
  subsorts Elt.X < NeSet[X] < Set[X] .
  op mt : -> Set[X] .
  op __ : Set[X] Set[X] -> Set[X] [assoc comm id: mt] .
  op __ : NeSet[X] NeSet[X] -> NeSet[X] [assoc comm id: mt] .
  var E : Elt.X .
  eq E E = E .
 endfm)
```

In the coming sections we will see how this qualification convention for the sorts of a parameterized module avoids many unintended collisions of sort names, thus making renaming practically unnecessary.

The module `SET` has only one parameter. In general, however, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory, that is, we can have an interface of the form `[X :: TRIV, Y :: TRIV]`, involving the theory `TRIV` twice. Therefore, parameters cannot be treated as normal submodules, since we do not want them to be shared when their labels are different. We regard the relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating renamed copies of the parameters, which are then included. For the above interface, two copies of the theory `TRIV` are generated, with names `X :: TRIV` and `Y :: TRIV`. In such copies of parameter theories sorts are renamed as follows: If Z is the label of a parameter theory T, then each sort S in T (for `TRIV` just the sort `Elt`) is renamed to S.Z. This is the reason why all occurrences of these sorts in the parameterized module must mention their corresponding renaming. In a future version of the system, this qualification will be necessary only in case of ambiguity.

The idea of renaming part of the structure of the parameter appears in CafeOBJ [36] as well. However, in CafeOBJ one can distinguish between what they call "share" and "non-share" cases, stating the kind of sharing one wants for the parameters. In the "non-share" case it is still possible to explicitly share some submodule in the structure of the parameter.

Let us consider as an example the following module `PAIR`. Notice the use of the qualifications for the sorts coming from each of the parameters, and notice also the qualification of the sort `Pair[X, Y]`.

```
(fmod PAIR[X :: TRIV, Y :: TRIV] is
  sort Pair[X, Y] .
  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] .
  op 1st : Pair[X, Y] -> Elt.X .
```

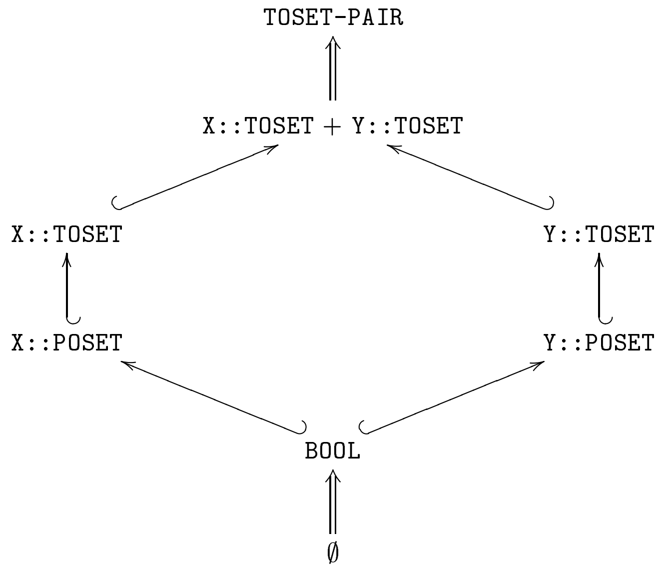


Figure 3.1: Structure of TOSET-PAIR.

```
  op 2nd : Pair[X, Y] -> Elt.Y .
  var A : Elt.X .
  var B : Elt.Y .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
endfm)
```

If a parameter theory is structured, this renaming process for parameter theories is carried out not only at the top level, but for the whole "theory part", that is, not renaming modules. Consider, for example, the following parameterized module defining a lexicographical ordering on pairs of elements, in which the components of the pairs may belong to different ordered sets.

```
(fmod TOSET-PAIR[X :: TOSET, Y :: TOSET] is
  sort Pair[X, Y] .
  op <_;_> : Elt.X Elt.Y -> Pair[X, Y] .
  op _<_ : Pair[X, Y] Pair[X, Y] -> Bool .
  op 1st : Pair[X, Y] -> Elt.X .
  op 2nd : Pair[X, Y] -> Elt.Y .
  vars A A' : Elt.X .
  vars B B' : Elt.Y .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
  eq < A ; B > < < A' ; B' >
    = (A < A') or (A == A' and B < B') .
endfm)
```

Representing by $\hookrightarrow$ the inclusion relations between modules and theories, and by $\Rightarrow$ the initiality constraints, we can depict the resulting structure as in Figure 3.1, where we have two copies not only of TOSET but also of the POSET subtheory.

We now give an object-oriented parameterized module defining a stack of elements. We define a class `Stack[X]`[14] as a linked sequence of node objects. Objects of class `Stack[X]` only have an attribute `first`, containing the identifier of the first node in the stack. If the stack is empty the value of the `first` attribute is `null`. Each object of class `Node[X]` has an attribute `next` holding the identifier of the next node—which will be `null` if there is no next node—and an attribute `contents` to store a value of sort `Elt.X`. Notice that the identifiers of the nodes are of the form `o(`S`,`N`)`, where S is the identifier of the stack object to which the node belongs, and N is a natural number. The messages `push`, `pop`, and `top` have as their first argument the identifier of the object to which they are addressed, and will cause, respectively, the insertion at the top of the stack of a new element, the deletion of the top element, and the sending of a response message `elt` containing the element at the top of the stack to the object making the request.

```
(omod STACK[X :: TRIV] is
  protecting MACHINE-INT .
  protecting QID .
  subsort Qid < Oid .
  class Node[X] | next : Oid, contents : Elt.X .
  class Stack[X] | first : Oid .
  msg _push_ : Oid Elt.X -> Msg .
  msg _pop : Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg _elt_ : Oid Elt.X -> Msg .

  op null : -> Oid .
  op o : Oid MachineInt -> Oid .

  vars O O' O'' : Oid .
  var E : Elt.X .
  var N : MachineInt .

  rl [top] : *** top on a nonempty stack
     < O : Stack[X] | first : O' >
     < O' : Node[X] | contents : E >
     (O top O'')
     => < O : Stack[X] | >
        < O' : Node[X] | >
        (O'' elt E) .

  rl [push1] : *** push on a nonempty stack
     < O : Stack[X] | first : o(O, N) >
     (O push E)
     => < O : Stack[X] | first : o(O, N + 1) >
        < o(O, N + 1) : Node[X] | contents : E, next : o(O, N) > .

  rl [push2] : *** push on an empty stack
     < O : Stack[X] | first : null >
     (O push E)
     => < O : Stack[X] | first : o(O, 0) >
        < o(O, 0) : Node[X] | contents : E, next : null > .

  rl [pop] : *** pop on a nonempty stack
     < O : Stack[X] | first : O' >
```

[14]Notice that naming of parameterized classes follows the same conventions discussed above for parameterized sorts.

```
      < O' : Node[X] | next : O'' >
      (O pop)
      => < O : Stack[X] | first : O'' > .
  endom)
```

We may want to define stacks not storing data elements of a particular sort, but actually objects in a particular class. We can define an object-oriented module with the intended behavior as a parameterized module `STACK2` parameterized by the object-oriented theory `CELL`, presented in Section 3.6.1, as follows:

```
(omod STACK2[X :: CELL] is
   protecting MACHINE-INT .
   protecting QID .
   subsort Qid < Oid .
   class Node[X] | next : Oid, node : Oid .
   class Stack[X] | first : Oid .
   msg _push_ : Oid Oid -> Msg .
   msg _pop : Oid -> Msg .
   msg _top_ : Oid Oid -> Msg .
   msg _elt_ : Oid Elt.X -> Msg .

   op null : -> Oid .
   op o : Oid MachineInt -> Oid .

   vars O O' O'' O''' : Oid .
   var E : Elt.X .
   var N : MachineInt .

   rl [top] : *** top on a nonempty stack
      < O : Stack[X] | first : O' >
      < O' : Node[X] | node : O'' >
      < O'' : Cell.X | contents : E >
      (O top O''')
      => < O : Stack[X] | >
         < O' : Node[X] | >
         < O'' : Cell.X | >
         (O''' elt E) .

   rl [push1] : *** push on a nonempty stack
      < O : Stack[X] | first : o(O, N) >
      (O push O')
      => < O : Stack[X] | first : o(O, N + 1) >
         < o(O, N + 1) : Node[X] | next : o(O, N), node : O' > .

   rl [push2] : *** push on an empty stack
      < O : Stack[X] | first : null >
      (O push O')
      => < O : Stack[X] | first : o(O, 0) >
         < o(O, 0) : Node[X] | next : null, node : O' > .

   rl [pop] : *** pop on a nonempty stack
      < O : Stack[X] | first : O' >
      < O' : Node[X] | next : O'' >
      (O pop)
      => < O : Stack[X] | first : O'' > .
  endom)
```

3.6.3 Views

We use views to assert how a particular target module or theory is claimed to satisfy[15] a source theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target.

In the current version of Full Maude, default views are not supported. Therefore, all views have to be defined explicitly, and all of them must have a name. As any theory or module, views should have been defined before they are used.

In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort, function, class, and message in the source theory. Although the current version does not support default views in the style of OBJ3, "obvious" parts of a mapping do not need to be explicitly given, namely, any identical mapping of an operator, message, or attribute f to itself such that its arity and coarity are mapped to those of an operator, message, or attribute with the same name in the target can be omitted. However, maps for all sorts in the source theory have to be given, even when they are identity maps.

The mapping of a sort S in the source theory to a sort S' in the target is expressed with syntax

$$\texttt{sort}\ S\ \texttt{to}\ S'\ \texttt{.}$$

The mapping of operators is expressed with syntax

$$\texttt{op}\ O\ \texttt{to}\ O'\ \texttt{.}$$

where O is an operator identifier or an operator identifier together with its arity and value sort. An operator map in which explicit arity and coarity are given affects not only the operators with such arity and coarity, but the entire family of subsort-overloaded operators[16] associated to the given operator. The target operators can be derived operators, that is, they can be terms with variables. Therefore, we can map an operator symbol not only to another operator symbol, but also to an expression. Consider, for example, the case in which we want to define a view from a theory in which we have a "less or equal" operator `_<=_`, defined with reflexivity, symmetry, and transitivity equations, to a module in which such an operator does not exist, but we have instead a "less than" operator `_<_`. Then, we can define a view with a map

```
op X <= Y to term X < Y or X == Y .
```

The mapping of a class C in the source theory to a class C' in the target is expressed with syntax

$$\texttt{class}\ C\ \texttt{to}\ C'\ \texttt{.}$$

Attribute maps have the form

$$\texttt{attr}\ A\ \texttt{.}\ C\ \texttt{to}\ A'\ \texttt{.}$$

[15] Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [24].

[16] See [21, Section 2.1.1] for a discussion on the overloading supported in Maude or [56] for a general discussion on order-sorted algebra.

where A is the name of an attribute of class C in the source theory and A' is an attribute of the image class of C under the view. The mapping of messages is expressed with syntax

$$\texttt{msg } M \texttt{ to } M' \texttt{ .}$$

where M is a message identifier or a message identifier together with its arity and value sort. As for operators, a message map in which explicit arity and coarity are given affects the entire family of subsort-overloaded message declarations associated to the declaration of the given message.

The syntax for views is given by the following declarations.

```
op op_to term_. : Bubble Bubble -> ViewDecl .
op op_to_. : Token Token -> ViewDecl .
op op_:_->_to_. : Token SortList Sort Token -> ViewDecl .
op op_: ->_to_. : Token Sort Token -> ViewDecl .
op sort_to_. : Sort Sort -> ViewDecl .
op class_to_. : Sort Sort -> ViewDecl .
op attr_._to_. : Token Sort Token -> ViewDecl .
op msg_to_. : Token Token -> ViewDecl .
op msg_:_->_to_. : Token SortList Sort Token -> ViewDecl .
op msg_: ->_to_. : Token Sort Token -> ViewDecl .

subsorts VarDecl < ViewDecl < ViewDeclList .
subsort VarDeclList < ViewDeclList .

op __ : ViewDeclList ViewDeclList -> ViewDeclList [assoc] .

op view_from_to_is_endv :
      ViewToken ModExp ModExp ViewDeclList -> PreView .
```

Thus, we can have a view

```
(view MachineInt from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
 endv)
```

which defines a view from the theory `TRIV` to the module `MACHINE-INT`. We encourage the convention of naming the views with `TRIV` as source by the name of the *sort* to which `Elt` is mapped. Although it is not necessary to follow this convention, it can add understandability to the specifications. What has to be avoided is using the labels in interfaces of parameterized modules as names of views, since this can sometimes generate ambiguities.

We can have views between theories, which is particularly useful to compose instantiations of views to link the formal parameter of some parameterized module to some actual parameter via some intermediate formal parameter of another parameterized module. We will give some examples in the coming sections. An example of a view whose target is a theory is the following.

```
(view Toset from TRIV to TOSET is
   sort Elt to Elt .
 endv)
```

As already mentioned, in some cases it is useful to be able to express mappings of operators, not to other operators but to expressions. For example, the `_<_` relation of a toset can be mapped to an expression using the "less than or equal" operator `_<=_` and the inequality operator `_=/=_` as follows.

```
(view MachIntAsToset from TOSET to MACHINE-INT is
   sort Elt to MachineInt .
   vars X Y : Elt .
   op X < Y to term X <= Y and X =/= Y .
 endv)
```

Notice that, when dealing with parameterized modules with structured theories as parameters, as in the TOSET-PAIR example discussed in Section 3.6.1, we have to give a view not only for the top theory but for the entire "loose part," that is, for all other subtheories imported by the theory. In the near future we plan to handle parameterized theories and views as well, as a way to give more structure to both theories and views.

3.6.4 Module Expressions

As in Clear [15], OBJ [60], and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations, that is, according to a specific *module algebra*. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications.

The current version of Full Maude supports two types of module expression: instantiation of a module expression with a view expression, and renaming of a module expression with a set of mappings. The syntax used for both of them is the same one as in OBJ3, namely

```
op _[_] : ModExp ViewExp -> ModExp .
```

for instantiation of parameterized modules, and

```
op _*(_) : ModExp MapList -> ModExp .
```

for renamings. As we saw in Section 3.6.2, a view expression can be a single view name, or a sequence of view names separated by commas in case the module being instantiated has several parameters. We shall explain in Section 4.10 some other module expressions and how new module expressions can be added.

Module Instantiation

Instantiation is the process by which actual parameters are bound to the formal parameters of a parameterized module and a new module is created as a result. This can be seen in fact as the evaluation of a module expression. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the actual parameters to the corresponding sorts, operators (or expressions), etc. in the target. We shall give other details about the instantiation of parameterized modules when we discuss their executable specification in Section 4.6.9.

The instantiation of a parameterized module has to be made with views explicitly defined previously. Thus, we can have a set of machine integers with the module expression SET[MachineInt], or a pair of machine integers as tosets with `TOSET-PAIR[MachIntAsToset, MachIntAsToset]`, using the view `MachIntAsToset` defined in Section 3.6.3.

As mentioned in Section 3.6.3, we can define views from theories to theories. Using such views we can, for example, instantiate the module `SET` with the view `Toset` given in the previous section. The result is a module `SET[Toset]` which is still parameterized, but now by the theory `TOSET`. We can instantiate it again with a view from `TOSET` to some other theory or module, for example, `MachIntAsToset`, obtaining the module `SET[Toset][MachIntAsToset]`, which is just a set of machine integers. For example, we can give a more concise definition of the parameterized module `TOSET-PAIR[X :: TOSET, Y :: TOSET]` using these ideas as follows.

```
(fmod TOSET-PAIR[X :: TOSET, Y :: TOSET] is
   protecting PAIR[Toset, Toset][X, Y] .
   op _<_ : Pair[Toset, Toset][X, Y] Pair[Toset, Toset][X, Y] -> Bool .
   vars A A' : Elt.X .
   vars B B' : Elt.Y .
   eq < A ; B > < < A' ; B' >
     = (A < A') or (A == A' and B < B') .
 endfm)
```

Let us consider now the following module `MAX`, parameterized by the theory `TOSET`. Given a set of elements in this toset, the function `max` returns the maximum element in the set.

```
(fmod MAX[T :: TOSET] is
  protecting SET[Toset][T] .
  op max : NeSet[Toset][T] -> Elt.T .
  var E : Elt.T .
  var S : NeSet[Toset][T] .
  eq max(E) = E .
  ceq max(E S) = E if max(S) < E .
  ceq max(E S) = max(S) if not (max(S) < E) .
 endfm)
```

Module expressions can be arguments of a `protecting` or `including` importation, or can be used as the module in which to reduce or rewrite by a `red` or `rew` command. In general we can use module expressions in any place where the name of a module is expected. In fact, in Full Maude the name of a module is given by a module expression, and each time a new module expression is entered, the module expression is evaluated, and a module is generated with this module expression as its name. It is this module that is passed to the engine to get the intended result.

Let us see how module expressions can be used in `red` commands using the instantiation of `MAX` with the view `MachIntAsToset` presented in Section 3.6.3.

```
Maude> (red-in MAX[MachIntAsToset] : max(5 4 8 4 6 5) .)
Result : 8
```

Notice that, if we have several parameters, we can instantiate the parameterized module with some views going to theories and others going to modules. The result in this case is the expected one, that is, we get a module parameterized by the targets of those views going to theories.

Module Renaming

Module renaming can be considered as a function that, given a module M and a list of mappings S, returns a copy of the module in which the names of the sorts, operators, etc., are changed as indicated by the mappings. More precisely, given a structured specification, the renaming not only causes the creation of a copy of the top module in the structure, but renames also the part of the submodule structure that is affected by the renaming. For any other submodule M' in the structure which is affected by the mappings, a renamed copy of it is generated with name `M' * (S')`, where S' is the subset of mappings in S that affect M'. We shall explain in some more detail the renaming module expression when we discuss its executable specification in Section 4.6.10.

The complete syntax for renaming maps is as follows.

```
op op_to_ : Token Token -> Map .
op op_:_->_to_ : Token SortList Sort Token -> Map .
op op_: ->_to_ : Token Sort Token -> Map .
op op_to_[_] : Token Token AttrList -> Map .
op op_:_->_to_[_] : Token SortList Sort Token AttrList -> Map .
op op_: ->_to_[_] : Token Sort Token AttrList -> Map .
op sort_to_ : Sort Sort -> Map .
op label_to_ : Token Token -> Map .
op class_to_ : Sort Sort -> Map .
op attr_._to_ : Token Sort Token -> Map .
op msg_to_ : Token Token -> Map .
op msg_:_->_to_ : Token SortList Sort Token -> Map .
op msg_: ->_to_ : Token Sort Token -> Map .

subsort Map < MapList .
op _,_ : MapList MapList -> MapList [assoc prec 42] .
```

Notice that we also allow the renaming of rule labels, which may be useful for metalevel applications.

A set of attributes can also be given in the renaming of an operator. This allows changing syntactic attributes, such as the precedence values and the gathering patterns, which may be of practical relevance when dealing with mixfix syntax—for example, when a change in the syntax of the operator could cause a parsing different from the intended one. Let us see an example in which modifying the grammatical attributes of an operator is useful. Suppose that we want to change the syntax of the function `max` in the module `MAX` presented above, to `maximum_`. We can do the following reduction.

```
Maude> (red-in MAX[MachIntAsToset]
                * (op max : NeSet[Toset][MachIntAsToset] -> MachineInt
                        to maximum_)
          : maximum 5 4 8 4 6 5 .)
Result : 4 5 6 8
```

This result may seem strange, but makes perfect sense. In fact the system indicates the term that it has reduced:

```
Reduce in MAX[MachIntAsToset]
           * (op max : NeSet[Toset][MachIntAsToset] -> MachineInt
                 to maximum_)
      : (maximum 5) 4 8 4 6 5 .
```

What has happened is that the precedence value given by default to the operator with this new syntax is the same as that given to the operator `__`, and therefore, by the default gathering patterns, this is a valid parse. Notice that the other elements passed to the function `maximum` have "disappeared" by the equations in SET. We can obtain the intended result by placing parentheses around the set of numbers, but it is more convenient to change the precedence values of the attributes. We can, for example, raise the precedence value of maximum_.

```
Maude> (red-in MAX[MachIntAsToset]
                * (op max : NeSet[Toset][MachIntAsToset] -> MachineInt
                        to maximum_
                     [prec 41])
          : maximum 5 4 8 4 6 5 .)
Reduce in MAX[MachIntAsToset]
              * (op max : NeSet[Toset][MachIntAsToset] -> MachineInt
                      to maximum_
                   [prec 41])
        : maximum (5 4 8 4 6 5) .
Result : 8
```

More examples can be found in [21]. We finish this section with an example involving object-oriented modules, namely, a stack of banking accounts.

```
(view Accnt from CELL to ACCNT is
  sort Elt to MachineInt .
  class Cell to Accnt .
  attr contents . Cell to bal .
 endv)
```

Now we can do the following rewriting.

```
Maude> (rew-in STACK2[Accnt] :
          < 'stack : Stack[Accnt] | first : null >
          < 'paul : Accnt | bal : 5000 >
          < 'peter : Accnt | bal : 2000 >
          < 'mary : Accnt | bal : 15000 >
          ('stack push 'paul)
          ('stack push 'peter)
          ('stack push 'mary)
          ('stack top 'peter)
          ('stack pop) .)
Result Configuration :
   ('peter elt 2000)
   < 'stack : Stack[Accnt] | first : o('stack, 1) >
   < 'paul : Accnt | bal : 5000 >
   < 'peter : Accnt | bal : 2000 >
   < 'mary : Accnt | bal : 15000 >
   < o('stack, 0) : Node[Accnt] |
         next : null, node : 'peter >
   < o('stack, 1) : Node[Accnt] |
         next : o('stack, 0), node : 'mary >
```

3.7 Concluding Remarks

We have explained in this chapter the design of Full Maude, focusing on the object-oriented notation and on the module operations with which Full Maude has been endowed.

Although we have followed quite closely OBJ3 [60] in the overall design, each one of the constructs in the language was carefully studied. When a construct we were interested in appeared in OBJ3, its approach was considered as one of the possibilities. Other languages have been studied in detail, and other proposals were taken in consideration, as shown by the fact that some central decisions, like the one regarding qualification of sorts, were influenced by other approaches.

We follow OBJ3 in considering module expressions as a central notion. This has influenced quite strongly not only the design of the language itself, but also its specification, as we shall explain in Chapter 4. We follow OBJ3 in the way of defining parameterized modules, and in the labeling of the parameter theories, not only to qualify with them the sorts defined in parameter theories used in the body of the parameterized module, but also to be able to create renamed copies of the structured parameters. The labeling of the formal parameters is in fact quite important. Although it forces to qualify the sorts defined in the parameters, this approach not only allows us to use a theory multiple times to define different formal parameters of the same module, but also improves reusability. Indeed, in the approaches followed by ACT ONE [44] or by CASL [31], for example, if one wants to use the same theory for different parameters of the same module, the user is forced to explicitly give renamed copies of the parameter theory in question. Note that, in addition to this, in the instantiation of such a parameterized module, a different view will have to be given for each of these copies.

We have adopted the convention of not qualifying sorts and operators with the name of the module in which they are defined, as in OBJ3. We have instead opted for the simplification of the language, although paying the price of forcing explicit renaming in order to avoid undesired name collisions. However, we have seen how, thanks to the support for overloaded operators, the problem is mainly reduced to the collision of sorts. We have proposed a solution for systematically reducing these collisions in the case of parameterized modules by using parameterized sorts.

We plan to continue working on the design of the language to overcome the syntactic restrictions mentioned at some points in the chapter, and to further improve its usability by further integrating it with Core Maude. We also plan to extend the language with new module operations and facilities. In addition to parameterized theories and parameterized views, it would be useful to endow Full Maude with conventions for the automatic generation of views, perhaps in the line of OBJ3's *default views*. It would also be useful to extend the language with additional features, like hiding of sorts and operators, and with constructs to redefine or remove the rewrite rules for messages in object-oriented modules as proposed by Meseguer in [77].

One of the more interesting characteristics of the proposed module algebra is its flexibility and extensibility, which makes possible experimenting with new module operations. We explain in Chapter 4 the executable formal specification of Full Maude, and illustrate the extensibility of the module algebra with several examples. A more substantial example is explained in Chapter 5.

Chapter 4

The Maude Specification of Full Maude

4.1 Full Maude in Maude

The efforts required for building adequate tools for formal specification languages are considerable. Such efforts can be particularly intense when such languages are *executable*, since a good execution engine must also be developed to have a complete execution environment. The methods we propose to implement tools for formal specification languages in Maude (see also [22, 26]) can be used in practice to develop tools and environments for formal specification languages, including executable ones, and to endow such languages with a powerful module algebra of specification-combining operations.

We have applied these methods to the design and implementation of Maude itself. The most basic parts of the language—supporting module hierarchies of functional and system modules and some predefined modules—are implemented in C++, giving rise to a sublanguage called Core Maude. This is extended by special syntax for object-oriented specifications, and by a rich *module algebra* of parameterized modules and module composition in the Clear/OBJ style [15, 60], as described in Chapter 3, giving rise to the Full Maude language.

All of Full Maude has been formally specified in Core Maude [41, 42]. This executable formal specification—a Maude specification of about 7,000 lines—is in fact its implementation in the Maude 1.00 system. Our experience in this regard is very encouraging in several respects. Firstly, because of how quickly we were able to develop Full Maude. Secondly, because of how easy it will be to maintain it, modify it, and extend it with new features and new module operations (see Section 4.10 and Chapter 5). Thirdly, because of the competitive performance with which we can carry out very complex module composition and module transformation operations, that makes the user interaction with Full Maude quite reasonable.

But the reflective methods described here, that underly our development of Full Maude, are much more general. They can equally be used to develop high-performance executable environments for other formal specification languages with much less effort and much greater flexibility, maintainability, and extensibility than what would be required in conventional implementations. For example, an executable environment for Action Semantics [87] is currently being considered, and it would be quite interesting to explore Maude implementations, for example, for a next-generation CafeOBJ [36] and for CASL [31]. Therefore, before discussing in detail the executable specification of Full Maude in Maude, we explain the more general methodology for building formal tools for different logics in Maude, of which the Full Maude specification in this chapter and the Church-Rosser checker in Chapter 5 are particular applications.

4.1.1 Building Formal Tools in Maude

Suppose that we want to build a theorem prover for a logic, or an executable formal specification language. We can do so by representing the logic $\mathcal{L}$ of the theorem prover or specification language in question in rewriting logic by means of a representation map

$$\Psi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

Such representations are typically very simple and natural. Using reflection we can, as already explained in Section 1.3.5, *internalize* such a map as an executable equationally defined function $\overline{\Psi}$. In Maude this is accomplished using the module `META-LEVEL` and its sort `Module`. Specifically, we can reify the above representation map Ψ by defining an abstract data type $\texttt{Module}_{\mathcal{L}}$

representing theories in the logic $\mathcal{L}$. We can then internalize the representation map Φ as an equationally defined function

$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}$$

in a module extending `META-LEVEL`. Then, we can use the functions `meta-reduce`, `meta-apply`, and `meta-rewrite`, or more complex strategies that use such functions, to execute in Maude the metarepresentation $\overline{\Psi}(\overline{T})$ of a theory T in $\mathcal{L}$. In other words, we can in this way *execute* $\mathcal{L}$ in Maude.

But we need more. To build a usable formal tool we need to build an *environment* for it, including not only the execution aspect just described, but parsing, pretty printing, and input/output. If we had instead considered formal translator generator uses of Maude—that is, translations of the form $\Phi : \mathcal{L} \to \mathcal{L}'$ between two logics, that can be reified in Maude as functions $\bar{\Phi} : \texttt{Module}_{\mathcal{L}} \to \texttt{Module}_{\mathcal{L}'}$—we would have observed entirely similar needs, since we need to get the specifications in different logics—originating from, or going to, different tools—in and out of Maude by appropriate parsing, pretty printing, and input-output functions. In Maude, these additional metalanguage features are supported as follows:

- The *syntax definition* for $\mathcal{L}$ is accomplished by defining the data type $\texttt{Grammar}_{\mathcal{L}}$. In Maude this can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of an existing tool supporting $\mathcal{L}$.

- Particularities at the *lexical* level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question (see Section 1.3.6). This is very useful when the theories in $\mathcal{L}$ have user-definable syntax, since then we can use such bubble sorts to represent strings of symbols corresponding to expressions that can only be analyzed after the user-defined syntax has been analyzed.

- Parsing and pretty printing for $\mathcal{L}$ is accomplished by the `meta-parse` and `meta-pretty-print` functions in `META-LEVEL`, in conjunction with the bubble sorts defined for $\mathcal{L}$ (see Section 1.3.4).

- Input/output of theory definitions, and of commands for execution in $\mathcal{L}$ is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop (see Section 1.3.4).

As we saw in Section 1.3.4, given any input enclosed in parentheses, the read-eval-print loop gives us access to a list of quoted identifiers $\mathcal{I}$ obtained automatically by putting a quote in front of each of the identifiers in the input. Similarly, any list of quoted identifiers placed in the "output channel" of the loop will be printed in the terminal after removing the quotes. Calling the function `meta-parse` with the input $\mathcal{I}$ and the metarepresentation of the signature $\texttt{Grammar}_{\mathcal{L}}$ in which we want to parse it, if there is a parse, we get the corresponding *parse tree* as a term $\mathcal{T}_{\mathcal{I}}$ of sort `Term`. The reverse process is accomplished by the `meta-pretty-print` function, which takes a term of sort `Term`, and returns its representation as a list of quoted identifiers.

The term $\mathcal{T}_{\mathcal{I}}$ can be seen as the representation of the input $\mathcal{I}$ as a term of sort `Term`. We could manipulate this term directly, applying any transformation to it and obtaining some other term representing the desired result, or we could apply some function to it to perform some kind of analysis or computation. Our experience, however, is that it is simpler and more flexible to transform this term into another term in some data type more appropriate for representing the input.

Suppose, for example, that we want to specify the transformation from modules in some language or logic $\mathcal{L}$ into modules in some language or logic $\mathcal{L}'$. First, we define abstract data types $\mathtt{Module}_{\mathcal{L}}$ and $\mathtt{Module}_{\mathcal{L}'}$. Then, we equationally define a function

$$\Lambda : \mathtt{Term} \longrightarrow \mathtt{Module}_{\mathcal{L}}$$

returning a term of sort $\mathtt{Module}_{\mathcal{L}}$ for each term $\mathcal{T}_{\mathcal{I}}$ of sort `Term` corresponding to an input $\mathcal{I}$. Then, we can compose this function with the also equationally defined function

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}_{\mathcal{L}'}$$

reifying a map of logics of the form

$$\Phi : \mathcal{L} \longrightarrow \mathcal{L}'.$$

The whole picture can be completed with a user-defined `meta-pretty-print` function extending the predefined one.

$$\mathtt{QidList} \xrightarrow[\mathtt{Grammar}_{\mathcal{L}}]{\mathtt{meta-parse}} \mathtt{Term} \xrightarrow{\Lambda} \mathtt{Module}_{\mathcal{L}} \xrightarrow{\overline{\Phi}} \mathtt{Module}_{\mathcal{L}'} \xrightarrow[\mathtt{Grammar}_{\mathcal{L}}]{\mathtt{meta-pretty-print}} \mathtt{QidList}$$

In the case of an *executable* environment, this picture may have to be extended to meet additional requirements. In the case of Full Maude, for example, we not only want to apply some transformation on some particular input, or to just give some command or module expression with some particular set of arguments and get some result. We also want to be able to interact with the system, entering modules, theories, views, and commands of different types.

In addition, such modules can be highly structured and parameterized. That is, there can be a rich collection of module composition operations endowing $\mathcal{L}$ with a *module algebra*. In such cases we typically have two data types of modules, a data type $\mathtt{Module}_{\mathcal{L}}$ of *flat* or unstructured modules, and a more general data type $\mathtt{StrModule}_{\mathcal{L}}$ of structured modules.

All the module algebra operations for $\mathcal{L}$ can be defined within Maude as an extension of the module `META-LEVEL`. In this way, the environment that we can build for $\mathcal{L}$ using Maude can also support all the module composition operations of $\mathcal{L}$. Among those module operations a common and important one is *flattening*, that is, the process of passing from a structured module to its unstructured flat form. This can be understood as a function

$$\mathtt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^{\flat}} \mathtt{Module}_{\mathcal{L}}.$$

Since modularity constructs can change from language to language, it may be simpler to represent $\mathcal{L}$ in rewriting logic by representing only its flat modules, that is, by a function $\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module}$ which makes the language $\mathcal{L}$ executable on top of Maude. But using the function $(\_)^{\flat}$ we can also make structured modules in $\mathcal{L}$ executable by means of the function composition

$$\mathtt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^{\flat}} \mathtt{Module}_{\mathcal{L}} \xrightarrow{\overline{\Phi}} \mathtt{Module}.$$

We may also want to be able to *refer by name* to modules, theories, and views in any other module, theory, view, or command. In Full Maude, for example, modules can be parameterized by theories, and both of them can import other modules or theories, or combinations of them given by *module expressions*. A module importing some module expressions can be seen as denoting a structured module with more or less complex relationships among its component submodules. We want to be able to store modules, theories, and views in a *database*, so that they can be referred later in order to evaluate module expressions, evaluate commands, and so on. Given a data type `Database` for the database of modules, theories, and views, the processing of a term obtained from a call to `meta-parse` with an input corresponding, for example, to a module can be seen as a function

$$\texttt{Term} \times \texttt{Database} \xrightarrow{\texttt{processModule}} \texttt{Database}.$$

In the case of commands, we can see their processing as the following function.

$$\texttt{Term} \times \texttt{Database} \xrightarrow{\texttt{processCommand}} \texttt{Database} \times \texttt{QidList}.$$

Note that some module, theory, or view in the database can be referred to in a command. There might be commands changing the database itself, for example, a command to remove a module from the database. The execution of a command may also generate some output, which will have to be conveniently meta-pretty-printed, so that it can be passed to the loop.

4.1.2 The Full Maude Execution Environment

In the case of Full Maude, there are several data types and several transformations taking place in the system. The main data types are: `Unit`, for the different types of theories and modules; `View` for views; and `Database` for the database of module, theory, and view definitions. We also have subsorts of `Unit` for the different types of theories and modules. For example, we have sorts `StrOModule` and `StrSModule` for structured object-oriented modules and structured system modules, respectively, and the transformation from object-oriented modules to system modules, described in Section 3.3.2, is reified as a function

```
objectOrientedToSystemModule : StrOModule Database -> StrSModule
```

We discuss the details about the data types `Unit`, `View`, and `Database` in Sections 4.3, 4.4, and 4.5, respectively, and the details about the function `objectOrientedToSystemModule` in Section 4.6.6.

To be able to *execute* some command using functions such as `meta-rewrite` or `meta-reduce`, or to be able to use any of the other predefined functions in the `META-LEVEL` module on the modules entered to Full Maude, we need to map terms of sort `Unit` into terms of sort `Module`, which will then be compiled into the rewrite engine. Note, however, that terms of sort `Unit` are structured modules, where the structure is given by module expressions that refer by name to other modules, or combinations of modules, which are stored in the database. This function in fact takes a unit and a database and returns the database after including the resulting module in it.

```
evalUnit : Unit Database -> Database
```

Terms of sort `Module` can only contain importation declarations of predefined modules, or of modules entered at the object level of Core Maude. Therefore, structured modules have to be "partially flattened," leaving only importations of built-ins[1]. In our approach, transforming a module from its possibly complex

[1]Although still partially structured, we shall refer to this "partially flattened" module as a module in *unstructured form* or in *flattened form*.

structured version to its unstructured form is a two-step process. First, the evaluation of the module expressions results in an intermediate form, in which only simple inclusion relationships appear among the modules; this first step can be seen as the reduction of a structured specification to its *structured normal form*. The normalization of the structure is carried out by the `normalization` function. It takes place on the database, and gives as result a possibly modified version of the database state. Then, in a second step, this structured normal form is flattened into an unstructured specification. Note that, instead of flattening the modules before evaluating the expressions, as it is done in many systems, we prefer to define operations on structured modules that return new structured modules. Besides simplicity and extensibility, the main advantage of dealing with structured modules is that the sharing of sub-structures is made explicit, avoiding the introduction of duplicated modules, or the cost of checking for it. We explain this process in detail in Section 4.6.

Since units can contain bubbles, which have to be parsed using the signature of the unit itself, the function Λ mentioned in Section 4.1.1 can be split in three stages. As we saw in Section 1.3.6, to be able to parse the bubbles in a module, we first need, in a first stage, to extract the signature. Then, in a second stage, a function Λ_1 replaces each bubble by its parsed term using the extracted signature. Then, in a third stage, we map the resulting term of sort `Term` without bubbles to a term of sort $\texttt{Module}_{\mathcal{L}}$ using a function Λ_2.

$$\texttt{Term} \xrightarrow{(1_{\texttt{Term}} \times \texttt{getSignature})} \texttt{Term} \times \texttt{Sign}_{\mathcal{L}} \xrightarrow{\Lambda_1} \texttt{Term} \xrightarrow{\Lambda_2} \texttt{Module}_{\mathcal{L}}$$

However, this process is computationally quite expensive. Note that building the signature can be almost as expensive as building the whole module. We shall see in detail in Section 4.6.7 how this process is efficiently accomplished. The idea is to build at the same time a term of sort $\texttt{Module}_{\mathcal{L}}$ with bubbles, which we call a *premodule*, and the signature, using finally the signature to parse the bubbles in the premodule and returning the module without bubbles in it. We can see the process as the functional composition

$$\texttt{Term} \longrightarrow \texttt{Sign}_{\mathcal{L}} \times \texttt{PreModule}_{\mathcal{L}} \longrightarrow \texttt{Module}_{\mathcal{L}}.$$

Notice that in the case of Full Maude we are dealing with structured modules, which are used to construct the signature and to evaluate a module expression in the database. In Full Maude, this process is accomplished by the `evalPreUnit` function. We will discuss it in Section 4.6.7 as well. In fact, we shall see that the functions `evalUnit` and `evalPreUnit` are quite similar.

The rest of the chapter is organized as follows. Section 4.2 presents the way in which the signature of Full Maude, presented in Chapter 3 and in Appendix A, is related to the rest of the specification. Sections 4.3.5, 4.4.3, and 4.5.2 explain, respectively, the data types `Unit`, `View`, and `Database`. The evaluation of units and views is described, respectively, in Sections 4.6 and 4.7. In Section 4.8 we discuss the processing of the input/output of the system, and in Section 4.9.1 we describe the top-level handling of the persistent database and its interaction with the read-eval-print loop presented in Section 1.3.4. Finally, in Section 4.10 we illustrate the flexibility and extensibility of the Full Maude system by introducing some new module expressions. Another extension of Full Maude to a theorem-proving tool for checking the Church-Rosser property will be discussed in detail in Chapter 5.

We do not intend to present here all the details of the executable specification of Full Maude. See [42] for this discussion. Here, we present an overview of the whole system, focusing on its most important aspects and its main functionality. We present the different modules with their main declarations so that a general idea of the whole system can be gained. Declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions and some rules, are omitted. We sometimes add dots "`...`" to indicate the missing equations and declarations.

All the declarations of the variables used are presented. However, some explanation of the conventions followed may ease the readability of the specification. We have followed very simple conventions for naming variables, sorts, and operators. As for the predefined modules, sort names are given starting with a capital letter. When the sort name is the concatenation of different words, or parts of words, each of the parts starts with a capital letter. Thus, we have, for example, sorts `UnitList` and `SubclassDecl` for lists of units and for declarations of subclass relations, respectively. Variables are named using the capital letters of the names of the sorts in which they are defined. For example, we have variables `B` of sort `Bool` or `VDS` of sort `VarDeclSet`. However, we have made some exceptions to this rule when there was a possibility of confusion, or seemed more natural. For example, `QI`, `F`, `V`, and `L` are used as names of variables of sort `Qid` when they are used as any quoted identifier, as an operator name, as a variable name, or as a label, respectively, and there are variables `Eq`, `Rl`, and `AtS` for elements of sorts `Equation`, `Rule`, and `AttrSet`, respectively.

4.2 The Signature of Full Maude

The signature of Full Maude is given in its entirety in Appendix A, and has already been discussed in detail in Chapter 3 and in [21]. We focus here on the way in which the signature is used in the specification, and in the way in which it can be extended in future extensions of Full Maude.

As explained in Section 1.3.6, to parse some input using the built-in function `meta-parse`, we need to give the metarepresentation of the signature in which the input is going to be parsed. But we do not need to give the complete metarepresentation of such a module. As we saw in Section 1.3.6, in modules including `META-LEVEL` it is possible to define terms of sort `Module` that import built-in modules or any module introduced at the "object level" of Core Maude. In this way, it is possible to get the equivalent effect of having the explicit metarepresentation of a module by declaring a constant and adding an equation identifying such a constant with the metarepresentation of an extended module that imports the original module at the object level.

As also mentioned in Section 1.3.6, the declaration of constructors for bubble sorts at the object level is not supported in the current version of Core Maude. The `special` attributes linking the constructors for the bubble sorts to the built-in ones are only supported at the metalevel, that is, the declarations of the constructor operators for bubble sorts have to be given in the metarepresentation of a module.

To allow the greatest generality and flexibility in future extensions of Full Maude, we have declared its signature as a module `FULL-MAUDE-SIGN` (see Appendix A). Then, in the following module `META-FULL-MAUDE-SIGN` we declare a constant `GRAMMAR` of sort `FModule`, and using the technique described in Section 1.3.6 to import the metarepresentation of a module by name, we give an equation identifying such constant with the metarepresentation of a module `GRAMMAR` in which there is a declaration importing `FULL-MAUDE-SIGN`. Declarations for the constructors of the bubble sorts are also included in this module. Note that the bubble sorts `Token`, `Bubble`, `ViewToken`, `SortToken`, and `NeTokenList` are declared in the module `SIGN&VIEW-EXPR`, which is imported by `FULL-MAUDE-SIGN`. These sorts are used in the declarations describing the syntax of the system.

```
fmod META-FULL-MAUDE-SIGN is
  including META-LEVEL .

  op GRAMMAR : -> FModule .

  eq GRAMMAR
    = (fmod 'GRAMMAR is
        including 'QID-LIST .
        including 'FULL-MAUDE-SIGN .
        sorts none .
        none
        op 'token : 'Qid -> 'Token
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        op 'viewToken : 'Qid -> 'ViewToken
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, 'assoc 'associative 'comm
                                  'commutative 'idem 'idempotent)))] .
        op 'sortToken : 'Qid -> 'SortToken
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '`[ '`] '< 'to ': '`, '. '`( '`))))] .
        op 'neTokenList : 'QidList -> 'NeTokenList
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        op 'bubble : 'QidList -> 'Bubble
             [special(
               (id-hook('Bubble, '1 '-1 '`( '`))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        none
        none
        none
     endfm) .
endfm
```

The `GRAMMAR` module will be used in calls to the `meta-parse` function in order to get the input parsed in this signature. Note that this module is not the data type in which we shall represent the inputs. From the call to `meta-parse` we shall get a term representing the parse tree of the input. This term will then be transformed into terms of other appropriate data types if necessary.

Future extensions to Full Maude will require extending the signature as well. The addition of new commands, new module expressions, or additions of any other kind will require adding new declarations to the present Full Maude signature and defining the corresponding extensions to the data types and functions to deal with the new cases introduced by the extensions. We shall see in Section 4.10 how the system can be easily extended to handle some new module expressions in addition to the renaming and instantiation module expressions presented in Section 3.6.4 and further discussed in Sections 4.6.10 and 4.6.9. In Chapter 5 we shall explain how to combine a proving tool like a Church-Rosser checker [25, 24] with Full Maude.

4.3 The Abstract Data Type `Unit`

In this section we present the abstract data type `Unit`, which can be seen as an extension of the predefined sort `Module` in several ways. There are constructors for functional, system, and object-oriented modules and theories, which can be parameterized and can import module expressions. There can also be parameterized sorts in Full Maude modules, and therefore, the constructors for the different declarations that can appear in a module have to be appropriately extended.

The section is structured as follows. After introducing some modules defining some functions on the predefined sorts `Bool` and `QidList` in Section 4.3.1, we present in Sections 4.3.2 and 4.3.3 the data types for extended sorts and extended declarations. In Section 4.3.4 we introduce module expressions and module names, and in Section 4.3.5 the abstract data type `Unit` itself.

4.3.1 Extensions to `BOOL` and `QID-LIST`

The module `EXT-BOOL` extends the predefined `BOOL` module with operations `_and-then_` and `_or-else_`. The semantics of `_and-then_` and `_or-else_` are, respectively, as the semantics of the boolean operators `_and_` and `_or_`, but they are not declared to be associative, and their second arguments are evaluated lazily thanks to the evaluation strategy `(1 0 2 0)` (see [60, 46, 21] for more on evaluation strategies).

```
fmod EXT-BOOL is
  pr BOOL .

  op _and-then_ : Bool Bool -> Bool [strat (1 0 2 0) prec 55] .
  op _or-else_ : Bool Bool -> Bool [strat (1 0 2 0) prec 59] .

   ...

endfm
```

The conversion of lists of quoted identifiers into single quoted identifiers by concatenating them is heavily used in the coming modules. This is the task of the `qidListToQid` function, which is introduced in the following module `EXT-QID-LIST` extending the predefined module `QID-LIST`.

```
fmod EXT-QID-LIST is
  pr QID-LIST .

  op qidListToQid : QidList -> Qid .

   ...

endfm
```

4.3.2 View Expressions and Extended Sorts

To allow the use of parameterized sorts, or sorts qualified by the view expression with which the parameterized module in which the given sorts appear is instantiated (see Section 3.6.2), we add the sort `ESort` of "extended sorts" as a supersort of the predefined sort `Sort`. View expressions and extended sorts are introduced in the following modules.

View Expressions

A view expression is given by a single quoted identifier, by a sequence of view expressions (at the user level, separated by commas), or by the composition of view expressions. In the current version, the composition of view expressions is only used internally; we plan to make it available to the user with syntax `_;_` in the future. View expressions are used in the instantiation of parameterized modules and in parameterized sorts. We plan to support parameterized views in the future as well. We use operators `_|_` and `_;;_` to represent, respectively, sequences and composition of view expressions.

The `VIEW-EXPR` module below also includes declarations for sets of view expressions, which will be used in the database structure to hold the list of the names of the views in it. The elements of sort `ViewExpSet`, for sets of view expressions, are given using the constructor `_#_`, which is declared to be associative, commutative, and with identity element `noneViewExpSet`.

The function `_inViewExpSet_` checks whether the view expression given as first argument is in the set of view expressions given as second argument or not.

```
fmod VIEW-EXPR is
  pr EXT-BOOL .
  pr QID .

  sort ViewExp .
  subsort Qid < ViewExp .
  op nullViewExp : -> ViewExp .
  op _|_ : ViewExp ViewExp -> ViewExp [assoc id: nullViewExp] .
  op _;;_ : ViewExp ViewExp -> ViewExp [assoc] .

  sort ViewExpSet .
  subsort ViewExp < ViewExpSet .
  op noneViewExpSet : -> ViewExpSet .
  op _#_ : ViewExpSet ViewExpSet -> ViewExpSet
        [assoc comm id: noneViewExpSet] .

  op _inViewExpSet_ : ViewExp ViewExpSet -> Bool .

   ...

endfm
```

Since the Core Maude engine does not know about view expressions, or, as we shall see, about extended sorts, extended module expressions, extended modules, and other declarations that we introduce, to be able to use them with built-in functions such as `sameComponent`, `leastSort`, `meta-reduce`, etc., we shall have to convert them into terms which only use the built-in constructors. Thus, for example, view expressions in sort `ViewExp` will be converted into quoted identifiers of sort `Qid` by means of function `viewExpToQid`, or, similarly, elements of sorts `ESort`, `ESortList`, and `ESortSet` are transformed into elements of sorts

`Qid`, `QidList`, and `QidSet`, respectively, with functions `eSortToQid` defined on the appropriate sorts.

```
fmod VIEW-EXPR-TO-QID is
  pr VIEW-EXPR .
  pr EXT-QID-LIST .

  op viewExpToQid : ViewExp -> Qid .
  op viewExpToQidList : ViewExp -> QidList .

   ...

endfm
```

Parameterized Sorts

In addition to the `ESort` sort, in the following module `EXT-SORT` we also define sorts `ESortList` and `ESortSet`.

The operator `eSort` is declared to be a constructor for extended sorts.

As for lists and sets of quoted identifiers, we declare `__` and `_;_` as constructors for sorts `SortList` and `ESortList`, and `ESortSet`, respectively.

```
fmod EXT-SORT is
  pr META-LEVEL .
  pr VIEW-EXPR-TO-QID .

  sort ESort .
  subsort Sort < ESort .
  op eSort : ESort ViewExp -> ESort .

  sort SortList .
  subsort SortList ESort < ESortList .
  op __ : SortList SortList -> SortList [assoc id: nil] .

  sort ESortList .
  subsort QidList Sort < SortList .
  op __ : ESortList ESortList -> ESortList [assoc id: nil] .

  sort ESortSet .
  subsort QidSet ESort < ESortSet .
  op _;_ : ESortSet ESortSet -> ESortSet [assoc comm id: none] .

endfm
```

Transforming Extended Sorts into Quoted Identifiers

Since eventually all units will have to be given as arguments in some of the calls to the built-in functions, and since the engine does not know about parameterized sorts, we need to transform all parameterized sorts appearing in any of the declarations that can appear in a unit into quoted identifiers. This is the task of the `eSortToQid` function, which is defined in the module `EXT-SORT-TO-QID`. These functions will be used with terms not containing error sorts[2]. Although the user cannot use error sorts, calls to built-in functions may generate such error

[2]The Core Maude system automatically adds error supersorts above each of the connected components of the poset of sorts declared by the user, using the set of maximal sorts in each connected component to qualify the corresponding error sort. Such error sorts are called *kinds* in the theory of membership algebras [82, 12].

sorts, and some built-in functions can handle them. The function eSortToSort presented in the previous module will be used in the calls to the built-in functions that may use such error sorts.

```
fmod EXT-SORT-TO-QID is
  pr EXT-SORT .

  op eSortToQid : ESort -> Qid .
  op eSortToQid : ESortSet -> QidSet .
  op eSortToQid : ESortList -> QidList .

   ...

endfm
```

4.3.3 Extended Declarations

In this section we discuss modules EXT-DECL and O-O-DECL which introduce, respectively, the declarations extending the sorts and constructors for declarations of the predefined data type Module in the META-LEVEL module to allow the use of extended sorts in them, and the declarations appearing in object-oriented units, namely class declarations, subclass relation declarations, and message declarations.

Declarations of Functional and System Units

In the following module EXT-DECL, we introduce the declarations extending those in META-LEVEL to allow the use of extended sorts in declarations of sorts, subsort relations, operators, variables, and membership axioms. The extension is accomplished by adding new supersorts for each of the sorts in META-LEVEL involved, and by adding new constructors for these new sorts.

We start introducing the declarations for the supersorts and their corresponding constructors. The EXT-DECL module also contains the declarations for sets of such declarations.

```
fmod EXT-DECL is
  pr EXT-SORT .

  sort ESortDecl .
  subsort SortDecl < ESortDecl .
  op sorts_. : ESortSet -> ESortDecl .

  sorts ESubsortDecl ESubsortDeclSet .
  subsort SubsortDecl < ESubsortDecl .
  subsorts SubsortDeclSet ESubsortDecl < ESubsortDeclSet .
  op subsort_<_. : ESort ESort -> ESubsortDecl .
  op __ : ESubsortDeclSet ESubsortDeclSet -> ESubsortDeclSet
        [comm assoc id: none] .

  sorts EOpDecl EOpDeclSet .
  subsort OpDecl < EOpDecl .
  subsorts OpDeclSet EOpDecl < EOpDeclSet .
  op op_:_->_[_]. : Qid ESortList ESort AttrSet -> EOpDecl .
  op __ : EOpDeclSet EOpDeclSet -> EOpDeclSet
        [assoc comm id: none] .
```

```
  sorts EVarDecl EVarDeclSet .
  subsort VarDecl < EVarDecl .
  subsorts VarDeclSet EVarDecl < EVarDeclSet .
  op var_:_. : Qid ESort -> EVarDecl .
  op __ : EVarDeclSet EVarDeclSet -> EVarDeclSet
        [assoc comm id: none] .

  sorts EMembAx EMembAxSet .
  subsort MembAx < EMembAx .
  subsorts EMembAx MembAxSet < EMembAxSet .
  op mb_:_. : Term ESort -> EMembAx .
  op cmb_:_if_=_. : Term ESort Term Term -> EMembAx .
  op __ : EMembAxSet EMembAxSet -> EMembAxSet
        [assoc comm id: none] .
```

The `sortSet` function returns the set of sorts from a declaration of sorts.

```
  op sortSet : ESortDecl -> ESortSet .
```

There are also some other functions to deal with variables. The function `varDeclSetInTerm` returns the set of the variable declarations in the given module for those variables that appear in the term given as second argument. The function `varDeclSetDiff` computes the set-theoretic difference between the two sets of variables given as arguments. Note that the difference is not between the declarations themselves, but just between the names of the variables in the declarations; the sorts with which they are declared are not taken into consideration.

```
  op varDeclSetInTerm : EVarDeclSet Term -> EVarDeclSet .

  op varDeclSetDiff : EVarDeclSet EVarDeclSet -> EVarDeclSet .

   ...

endfm
```

Declarations for Object-Oriented Units

In the `O-O-DECL` module we introduce the sorts and constructors for declarations of classes, subclass relations, and messages in object-oriented units.

Note that we follow the same naming conventions for classes as for extended sorts (see Section 3.6.2), and therefore we use the sort `ESort` for class identifiers, and `ESortList` and `ESortSet` for lists and sets of class identifiers, respectively. We use the operator `attr_:_` as a constructor for declarations of attributes.

```
fmod O-O-DECL is
  pr EXT-SORT .

  sorts AttrDecl AttrDeclSet .
  subsort AttrDecl < AttrDeclSet .
  op attr_:_ : Qid ESort -> AttrDecl .
  op none : -> AttrDeclSet .
  op _,_ : AttrDeclSet AttrDeclSet -> AttrDeclSet
        [assoc comm id: none] .

  sorts ClassDecl ClassDeclSet .
  subsort ClassDecl < ClassDeclSet .
```

```
  op class_|_. : ESort AttrDeclSet -> ClassDecl .
  op none : -> ClassDeclSet .
  op __ : ClassDeclSet ClassDeclSet -> ClassDeclSet
       [assoc comm id: none] .

  sorts SubclassDecl SubclassDeclSet .
  subsort SubclassDecl < SubclassDeclSet .
  op subclass_<_. : ESort ESort -> SubclassDecl .
  op none : -> SubclassDeclSet .
  op __ : SubclassDeclSet SubclassDeclSet -> SubclassDeclSet
        [assoc comm id: none] .

  sorts MsgDecl MsgDeclSet .
  subsort MsgDecl < MsgDeclSet .
  op msg_:_->_. : Qid ESortList ESort -> MsgDecl .
  op none : -> MsgDeclSet .
  op __ : MsgDeclSet MsgDeclSet -> MsgDeclSet
       [assoc comm id: none] .

   ...

endfm
```

'Desugaring' of Extended Sorts in Declarations

In the following module we define the `eSortToQid` function on the sorts for the different declarations introduced in the previous modules.

```
fmod DECL-EXT-SORT-TO-QID is
  pr EXT-SORT-TO-QID .
  pr EXT-DECL .
  pr O-O-DECL .

  op eSortToQid : ESubsortDeclSet -> SubsortDeclSet .
  op eSortToQid : EOpDeclSet -> OpDeclSet .
  op eSortToQid : EVarDeclSet -> VarDeclSet .
  op eSortToQid : EMembAxSet -> MembAxSet .
  op eSortToQid : ClassDeclSet -> ClassDeclSet .
  op eSortToQid : SubclassDeclSet -> SubclassDeclSet .
  op eSortToQid : MsgDeclSet -> MsgDeclSet .
  op eSortToQid : AttrDeclSet -> AttrDeclSet .

   ...

endfm
```

4.3.4 Module Expressions and Module Names

The abstract syntax for writing specifications in Maude can be seen as given by module expressions, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations. All module expressions will be evaluated generating modules with such module expressions as names. In the case of parameterized modules, each of the parameters in an interface will be used as the name of a new module created as a renamed copy of the parameter theory.

Module Expressions

In the following MOD-EXPR module the sort ModExp is introduced as a supersort of Qid. Module expressions for module renaming and for module instantiation will be introduced, respectively, in the modules RENAMING-EXPR and INST-EXPR, in Sections 4.6.10 and 4.6.9, which will be defined as extensions of MOD-EXPR. We shall see in Section 4.10 how new module combining and/or transforming operators can be easily added, showing the flexibility and extensibility of the module algebra being defined.

```
fmod MOD-EXPR is
  pr QID .

  sort ModExp .
  subsort Qid < ModExp .

endfm
```

Module Names

As we shall see in the coming sections, the evaluation of module expressions may produce the creation of new modules, whose *names* are given by the module expressions themselves. If there is already a module in the database with the module expression being evaluated as name, the evaluation of such module expression does not produce any change in the database. However, the evaluation of a module expression may involve the evaluation of some other module expressions contained in the modules involved, which in turn may generate new modules.

Given a parameterized module $\texttt{N[L}_1\texttt{ :: T}_1\texttt{, ..., L}_n\texttt{ :: T}_n\texttt{]}$, with $\texttt{L}_1 \ldots \texttt{L}_n$ labels and $\texttt{T}_1 \ldots \texttt{T}_n$ theory identifiers, we say that N is the name of the module and that $\texttt{[L}_1\texttt{ :: T}_1\texttt{, ..., L}_n\texttt{ :: T}_n\texttt{]}$ is its *interface*. As we shall see in Sections 4.6.9 and 4.8.3, for each parameter $\texttt{L}_i\texttt{ :: T}_i$ in the interface of a module, a new module is generated with such a parameter expression as its name, and a declaration importing it in the parameterized module is added. We regard the relationship between the body of a parameterized module and the parameters in its interface, not as an inclusion, but as mediated by a module constructor that generates renamed copies of the parameters, which are then included. Therefore, the sort Parameter is declared as a subsort of ModName, that is, terms of sort Parameter are considered to be module names. The constructor operator for the sort Parameter is par_::_.

The MOD-NAME module also includes a sort ModNameSet, for sets of module names. The constructors of sort ModNameSet are noneModNameSet and _&_. The function _inModNameSet_ checks whether the module name given as first argument is in the set of module names given as second argument.

```
fmod MOD-NAME is
  pr MOD-EXPR .
  pr EXT-BOOL .

  sort Parameter .
  op par_::_ : Qid ModExp -> Parameter .

  sorts ModName ModNameSet .
  subsorts Parameter ModExp < ModName < ModNameSet .
  op noneModNameSet : -> ModNameSet .
```

```
  op _&_ : ModNameSet ModNameSet -> ModNameSet
        [assoc comm id: noneModNameSet] .
  op nullModName : -> ModName .

  op _inModNameSet_ : ModName ModNameSet -> Bool .

   ...
endfm
```

Since the Core Maude engine assumes that module names are identifiers and does not know about term-structured module names (such as parameterized module interfaces or module expressions), for evaluation purposes we need to transform them into quoted identifiers. The functions `modNameToQid` and `modNameToQidList` in the module `MOD-NAME-TO-QID` below accomplish this transformation. In any language extensions, new equations for the function `modNameToQidList` should be added for each new module expression constructor introduced. In Sections 4.6.10 and 4.6.9 we shall see how the corresponding equalities are added for renaming and instantiation expressions, and in Section 4.10 for other new module expressions in extensions of Full Maude.

```
fmod MOD-NAME-TO-QID is
  pr MOD-NAME .
  pr EXT-QID-LIST .

  op modNameToQid : ModName -> Qid .
  op modNameToQidList : ModName -> QidList .

  vars QI X : Qid .
  var ME : ModExp .
  var MN : ModName .

  eq modNameToQidList(QI) = QI .
  eq modNameToQidList(par X :: ME)
    = (X ':: modNameToQidList(ME)) .
  eq modNameToQidList(nullModName) = ' .

  eq modNameToQid(MN)
    = qidListToQid(modNameToQidList(MN)) .

endfm
```

4.3.5 Units

We handle six different types of units: functional, system, and object-oriented modules, and functional, system, and object-oriented theories. Modules and theories of any kind are considered to be elements in specific subsorts of the sort `Unit`. A constructor `unitError` is also included to represent incorrect units. `unitError` has a list of quoted identifiers as argument, which is used to report the error. Besides considering functional and system theories and object-oriented theories and modules, the declarations presented in the following module extend the declarations for sort `Module` in the `META-LEVEL` module in three different ways:

- the name of a module can be any term of sort `ModName`,
- parameterized modules are handled, for which a list of parameters is added to the constructors of modules,

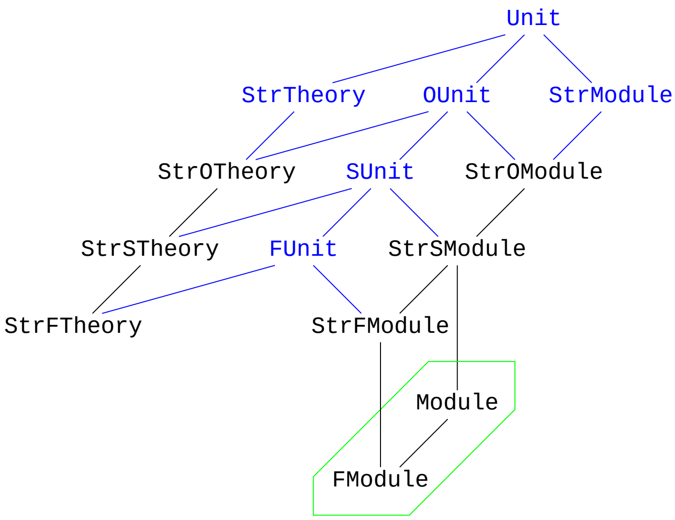


Figure 4.1: Hierarchy of Unit Sorts.

- the importation declaration is extended to module names, and
- parameterized sorts are supported.

```
fmod UNIT is
  pr EXT-DECL .
  pr O-O-DECL .
  pr MOD-NAME-TO-QID .
  inc META-LEVEL .
```

We start by introducing declarations for sorts `EImport` and `EImportList`, which are declared as supersorts of `Import` and `ImportList`, respectively. We declare the constructor `including_.` for importation declarations on module names. We shall see in Section 4.8.1 that importations in both `protecting` and `including` modes are represented using this constructor. The sort for lists of parameters is `ParameterList`, which is defined with constructors `parameterList` and `nilParameterList`.

```
  sorts EImport EImportList .
  subsort Import < EImport  .
  subsorts ImportList EImport < EImportList .
  op including_. : ModName -> EImport .
  op __ : EImportList EImportList -> EImportList
        [assoc id: nil] .

  sort ParameterList .
  subsort Parameter < ParameterList .
  op nilParameterList : -> ParameterList .
  op parameterList : ParameterList ParameterList -> ParameterList
        [assoc id: nilParameterList] .
```

Next, we introduce the different sorts for the different types of modules and theories, with the subsort relation among them, and their constructors. The structure of the hierarchy of sorts can be seen in Figure 4.1.

```
sorts Unit FUnit SUnit OUnit
      StrFModule StrSModule StrOModule StrModule
      StrFTheory StrSTheory StrOTheory StrTheory .
subsorts FModule < StrFModule < FUnit StrSModule .
subsorts Module < StrSModule < SUnit StrOModule .
subsorts StrFTheory < StrSTheory FUnit .
subsorts StrSTheory < StrOTheory SUnit .
subsort FUnit < SUnit .
subsort StrOTheory < StrTheory .
subsorts StrOTheory SUnit StrOModule < OUnit .
subsort StrOModule < StrModule .
subsorts StrTheory OUnit StrModule < Unit .

op noModule : -> Module .
op unitError : QidList -> Unit .

op fmod_is________endfm :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet -> StrFModule
      [gather (& & & & & & & & &)] .
op fth_is________endfth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet -> StrFTheory
      [gather (& & & & & & & & &)] .
op mod_is_________endm :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet RuleSet -> StrSModule
      [gather (& & & & & & & & & &)] .
op th_is_________endth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      EOpDeclSet EVarDeclSet EMembAxSet EquationSet RuleSet -> StrSTheory
      [gather (& & & & & & & & & &)] .
op omod_is____________endom :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      ClassDeclSet SubclassDeclSet EOpDeclSet MsgDeclSet EVarDeclSet
      EMembAxSet EquationSet RuleSet -> StrOModule
      [gather (& & & & & & & & & & & & &)] .
op oth_is____________endoth :
      ModName ParameterList EImportList ESortDecl ESubsortDeclSet
      ClassDeclSet SubclassDeclSet EOpDeclSet MsgDeclSet EVarDeclSet
      EMembAxSet EquationSet RuleSet -> StrOTheory
      [gather (& & & & & & & & & & & & &)] .
```

In addition to the constructor operators, the following functions are introduced in the UNIT module:

- A function _in_ to check whether a given importation declaration is in a set of importation declarations or not.

```
op _in_ : EImport EImportList -> Bool .
```

- Selector functions for the different components of a unit.

```
op name : Unit -> ModName .
op importList : Unit -> EImportList .
op parameterList : Unit -> ParameterList .
op sortDecl : Unit -> ESortDecl .
```

```
op subsortDeclSet : Unit -> ESubsortDeclSet .
op opDeclSet : Unit -> EOpDeclSet .
op varDeclSet : Unit -> EVarDeclSet .
op membAxSet : Unit -> EMembAxSet .
op equationSet : Unit -> EquationSet .
op ruleSet : Unit -> RuleSet .
op classDeclSet : Unit -> ClassDeclSet .
op subclassDeclSet : Unit -> SubclassDeclSet .
op msgDeclSet : Unit -> MsgDeclSet .
```

- Functions to change the value of each of the components of a unit.

```
op setName : Unit ModName -> Unit .
op setParameterList : Unit ParameterList -> Unit .
op setImportList : Unit EImportList -> Unit .
op setSortDecl : Unit ESortDecl -> Unit .
op setSubsortDeclSet : Unit ESubsortDeclSet -> Unit .
op setOpDeclSet : Unit EOpDeclSet -> Unit .
op setVarDeclSet : Unit EVarDeclSet -> Unit .
op setMembAxSet : Unit EMembAxSet -> Unit .
op setEquationSet : Unit EquationSet -> Unit .
op setRuleSet : Unit RuleSet -> Unit .
op setClassDeclSet : Unit ClassDeclSet -> Unit .
op setSubclassDeclSet : Unit SubclassDeclSet -> Unit .
op setMsgDeclSet : Unit MsgDeclSet -> Unit .
```

- Functions to add new declarations to the set of declarations already in a unit.

```
op addImportList : EImportList Unit -> Unit .
op addSortSet : ESortSet Unit -> Unit .
op addSubsortDeclSet : ESubsortDeclSet Unit -> Unit .
op addOpDeclSet : EOpDeclSet Unit -> Unit .
op addVarDeclSet : EVarDeclSet Unit -> Unit .
op addMembAxSet : EMembAxSet Unit -> Unit .
op addEquationSet : EquationSet Unit -> Unit .
op addRuleSet : RuleSet Unit -> Unit .
op addClassDeclSet : ClassDeclSet Unit -> Unit .
op addSubclassDeclSet : SubclassDeclSet Unit -> Unit .
op addMsgDeclSet : MsgDeclSet Unit -> Unit .
```

- There are functions and constants to create empty units of the different types. For example, the function `emptyStrFTheory` returns an empty structured functional theory. There is also a function `empty` which takes a unit as argument and returns an empty unit of the same type.

```
op emptyFModule : ModName -> FModule .
op emptyStrFModule : -> StrFModule .
op emptyStrSModule : -> StrSModule .
op emptyStrOModule : -> StrOModule .
op emptyStrFTheory : -> StrFTheory .
op emptyStrSTheory : -> StrSTheory .
op emptyStrOTheory : -> StrOTheory .
op empty : Unit -> Unit .
```

- A function `addDecls` which returns the unit resulting from adding all the declarations in the unit passed as second argument to the unit passed as first argument.

```
op addDecls : Unit Unit -> Unit .
```

Note that some of the 'set' and 'add' functions are partial functions.
...

```
endfm
```

4.4 The Abstract Data Type `View`

In this section we present the data type `View` for views. Basically, the data elements of sort `View` are composed by the name of the view, the names of the source and target units, and a set of maps representing the maps asserting how the given target unit is claimed to satisfy the source theory (see Section 3.6.3).

Internally, renaming maps are considered to be a particular case of view maps. The sort `ViewMap` is declared as a supersort of `Map`. The only kind of maps in sort `ViewMap` not in sort `Map` are maps of operators going to derived operators. We start introducing the declarations for renaming maps and sets of renaming maps in Section 4.4.1, we then introduce view maps and sets of view maps in Section 4.4.2, and finally we introduce the sort `View`, its constructor, and some operations on it in Section 4.4.3.

4.4.1 Renaming Maps

We introduce the different types of renaming maps in the following module `MAP`. A sort is introduced for each of these types of maps, with the appropriate constructors for each sort (see Section 3.6.4). All these sorts are declared to be subsorts of the sort `Map`. A sort for sets of maps (`MapSet`) is then declared as supersort of `Map` with constructors `none` and `_,_`.

```
fmod MAP is
  pr EXT-SORT .

  sorts OpMap SortMap LabelMap ClassMap MsgMap AttrMap Map .
  subsorts OpMap SortMap LabelMap ClassMap MsgMap AttrMap < Map .
  op op_to_[_] : Qid Qid AttrSet -> OpMap .
  op op_:_->_to_[_] : Qid ESortList ESort Qid AttrSet -> OpMap .
  op sort_to_ : ESort ESort -> SortMap .
  op label_to_ : Qid Qid -> LabelMap .
  op class_to_ : ESort ESort -> ClassMap .
  op attr_._to_ : Qid ESort Qid -> AttrMap .
  op msg_to_ : Qid Qid -> MsgMap .
  op msg_:_->_to_ : Qid ESortList ESort Qid -> MsgMap .

  sort MapSet .
  subsort Map < MapSet .
  op none : -> MapSet .
  op _,_ : MapSet MapSet -> MapSet [assoc comm id: none] .

endfm
```

4.4.2 View Maps

In addition to the maps of sort `Map` declared in the `MAP` module, in views there can also be maps from operators to derived operators, that is, terms with variables (see Section 3.6.3). Maps of this kind are given with the constructor `termMap`, which, in addition to the source and target terms, takes the set of variable declarations for the variables used in the map. The source term must be of the form $\mathtt{F(X}_1\mathtt{,}\ldots\mathtt{,X}_n\mathtt{)}$, where `F` is an operator name declared with n arguments of sorts in the connected components of the variables $\mathtt{X}_1 \ldots \mathtt{X}_n$, respectively. We will see in Section 4.8.3 how in the initial processing of a view the variables declared in it are associated to each of the maps in which they are used.

```
fmod VIEW-MAP is
  pr MAP .
  pr EXT-DECL .

  sort TermMap .
  op termMap : EVarDeclSet Term Term -> TermMap .

  sorts ViewMap ViewMapSet .
  subsorts Map TermMap < ViewMap .
  subsorts ViewMap MapSet < ViewMapSet .
  op _,_ : ViewMapSet ViewMapSet -> ViewMapSet [assoc comm id: none] .

endfm
```

4.4.3 Views

The `View` sort is introduced in the following module `VIEW`. In addition to the constructor for views (`view`), selector functions are added for each of the components of a view (`name`, `source`, `target`, and `mapSet`), and a constant `emptyView`, which is identified in an equation with the empty view, is defined.

Although the declaration of the constructor for views includes an argument for the list of parameters, parameterized views are not handled yet, so at present this argument must be set to the `nilParameterList`.

```
fmod VIEW is
  pr UNIT .
  pr VIEW-MAP .

  sort View .
  op view : ViewExp ParameterList ModExp ModExp ViewMapSet -> View .

  op name : View -> ViewExp .
  op source : View -> ModExp .
  op target : View -> ModExp .
  op mapSet : View -> ViewMapSet .

  op emptyView : Qid ModExp ModExp -> View .

  eq emptyView(QI, ME, ME')
    = view(QI, nilParameterList, ME, ME', none) .

   ...

endfm
```

4.5 The Abstract Data Type Database

In this section we present the data type Database, which will be used to store information about the units and views in the system. Before discussing this data type in Section 4.5.2, we present the predefined units added in Full Maude to those already available in Core Maude.

4.5.1 Non-Built-In Predefined Modules

As we shall see in the following section, except for the LOOP-MODE module, all the predefined modules that are available in Core Maude are also available in Full Maude. In addition to these Core Maude predefined modules, in Full Maude we have some additional predefined units. In the present system, the only units with which the database is initialized (see Section 4.9.2) are the functional theory TRIV, the module CONFIGURATION (see Section 3.3), and the module UP, which will be used to evaluate the up functions (see Sections 3.4 and 4.6.4). We shall see in Section 4.9.2 how new predefined modules can be added to the initial database.

```
fmod PREDEF-UNITS is
  pr UNIT .

  op TRIV : -> StrFTheory .
  eq TRIV = (fth 'TRIV is
               nilParameterList
               nil
               sorts 'Elt .
               none
               none
               none
               none
               none
            endfth) .

  op CONFIGURATION : -> StrFModule .
  eq CONFIGURATION
    = (fmod 'CONFIGURATION is
         nilParameterList
         nil
         sorts 'Oid ; 'Cid ; 'Attribute ; 'AttributeSet ;
               'Configuration ; 'Object ; 'Msg .
         subsort 'Attribute < 'AttributeSet .
         subsort 'Object < 'Configuration .
         subsort 'Msg < 'Configuration .
         op 'none : nil -> 'AttributeSet [none] .
         op '_`,_ : 'AttributeSet 'AttributeSet -> 'AttributeSet
               [comm assoc id({'none}'AttributeSet)] .
         op 'none : nil -> 'Configuration [none] .
         op '__ : 'Configuration 'Configuration -> 'Configuration
               [comm assoc id({'none}'Configuration)] .
         op '<_:_|_> : 'Oid 'Cid 'AttributeSet -> 'Object [none] .
         op '<_:_|`> : 'Oid 'Cid -> 'Object [none] .
         var 'O : 'Oid . var 'C : 'Cid .
         none
         eq '<_:_|`>['O, 'C] = '<_:_|_>['O, 'C, {'none}'AttributeSet] .
      endfm) .
```

The following module UP contains the necessary declarations to be able to parse the up functions presented in Section 3.4. We shall see in Section 4.6.7 how a declaration importing the following module UP is added to all the modules importing the predefined module META-LEVEL. With this declaration, it is possible to parse the up commands in the bubbles of such modules or in commands being evaluated in such modules. We shall see in Section 4.6.4 how these commands are then evaluated.

```
  op UP : -> StrFModule .
  eq UP
    = (fmod 'UP is
          nilParameterList
          including 'QID-LIST .
          including 'META-LEVEL .
          sorts 'Token ; 'Bubble .
          none
          op 'token : 'Qid -> 'Token
               [special(
                 (id-hook('Bubble, '1 '1)
                  op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'bubble : 'QidList -> 'Bubble
               [special(
                 (id-hook('Bubble, '1 '-1 '`( '`))
                  op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                  op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
          op 'up : 'Token 'Bubble -> 'Term [none] .
          op 'up : 'Token -> 'Module [none] .
          none
          none
          none
       endfm) .

endfm
```

4.5.2 The Database

In order to be able to refer to modules by name, which is extremely useful for module definition purposes at the user level, the evaluation of module expressions takes place in the context of a database, in which we keep information about the modules already introduced in the system, and also about those modules generated internally. This information is stored as a set of elements of sort UnitInfo and ViewInfo, in which we hold, respectively, the information concerning units and views. For each unit we save:

- Its original form, as introduced by the user, or, in case of an internally generated unit, as generated from the original form of some other unit.
- Its internal representation, in which variables have been renamed to avoid collisions with the names of variables in other units in the same hierarchy. In the case of object-oriented units, we store its equivalent system module, that is, the result of transforming it into a system module.
- Its signature, which is given as a functional module of sort FModule with no axioms, ready to be used in calls to meta-parse. There can only be importation declarations including built-in modules in this module. These are the only inclusions handled by the Core Maude engine.

- Its flattened version, for which, as for signatures, only the importation of built-in modules is left unevaluated.

For each view we keep its name and the view itself.

As a simple mechanism to keep the database consistent, for each unit we maintain the list of names of all the units and views "depending" on it. Similarly, for each view we maintain the list of names of all the units "depending" on it. The idea is that if a unit or view is redefined or removed, all those units and/or views depending on it will also be removed. This dependency does not only mean direct importation. For example, the module resulting from the renaming of some module also depends on the module being renamed; the instantiation of a parameterized module also depends on the parameterized module and on all the views used in its instantiation; a view depends on its source and target units, etc. This dependency is transitive: if a module, theory, or view has to be removed, all the units and/or views depending on them will be removed as well. The dependencies derived from the module expressions themselves are established by the function `setUpModExpDependencies`. The function `setUpUnitDependencies` calls `setUpModExpDependencies`, and then `setUpImportSetDependencies` to add the *back references* in the modules being imported. The function `setUpViewDependencies` sets up the back references for the views being introduced.

In addition to this set of information cells for units and views, we also keep lists with the names of all the units and views in the database, and a list of quoted identifiers in which we store the messages generated during the process of treatment of the inputs in order to simplify the communication with the read-eval-print loop process.

```
fmod DATABASE is
  pr VIEW .

  sort UnitInfo .
  op <_;_;_;_;_;_;_> : ModName Unit Unit Unit Module
        ModNameSet ViewExpSet -> UnitInfo .

  sort ViewInfo .
  op <_;_;_> : ViewExp View ModNameSet -> ViewInfo .

  sort InfoSet .
  subsort UnitInfo ViewInfo < InfoSet .
  op emptyInfoSet : -> InfoSet .
  op __ : InfoSet InfoSet -> InfoSet
        [assoc comm id: emptyInfoSet] .

  sort Database .
  op database : InfoSet ModNameSet ViewExpSet QidList -> Database .

  vars QI X : Qid .
  vars QIL QIL' : QidList .
  var VE : ViewExp .
  vars VES VES' : ViewExpSet .
  var IS : InfoSet .
  vars MNS MNS' MNS'' : ModNameSet .
  var PL : ParameterList .
  vars ME ME' : ModExp .
  var VI : View .
  var VMAPS : ViewMapSet .
```

```
vars U U' U'' : Unit .
var DB : Database .
vars M M' M'' : Module .
var EIL : EImportList .
vars MN MN' : ModName .
```

The constant `emptyDatabase` denotes the empty database, and there are predicates `viewInDatabase` and `unitInDatabase` to check, respectively, whether a view and a unit are in a database or not.

```
op emptyDatabase : -> Database .
eq emptyDatabase
  = database(emptyInfoSet, noneModNameSet, noneViewExpSet, nil) .

op unitInDatabase : ModName Database -> Bool .
op viewInDatabase : ViewExp Database -> Bool .
```

If a module, theory, or view is being redefined, that is, if there was already in the database a module, theory, or view with the same name, then all the units and/or views depending on it are removed using the functions `deleteUnits` and `deleteViews`. Removing a view or a unit from the database means removing its info cell from the set of cells in the database.

```
op deleteUnits : ModNameSet Database -> Database .
op deleteViews : ViewExpSet Database -> Database .
```

The `warning` function allows us to place messages (warning, error, or any other kind of messages) in the last argument of the database constructor. These messages are given in the form of quoted identifier lists, and will be passed to the third argument of the read-eval-print loop, to be printed in the terminal.

```
op warning : Database QidList -> Database .
eq warning(database(IS, MNS, VES, QIL), QIL')
  = database(IS, MNS, VES, (QIL QIL')) .
```

The constant `builtIns` denotes the set of identifiers of the predefined modules of Core Maude. It will be used to check whether a particular module is a built-in Core Maude module or not.

```
op builtIns : -> ModNameSet .
eq builtIns
  = 'TRUTH-VALUE & 'TRUTH & 'BOOL & 'IDENTICAL & 'MACHINE-INT &
    'QID & 'QID-LIST & 'META-LEVEL & 'LOOP-MODE .
```

Core Maude built-in modules are handled in a special way in the current version of the system. They are not explicitly defined in the Full Maude database; their importation is directly handled by Core Maude. This has some drawbacks: Core Maude built-in modules cannot be renamed; they cannot be directly used with built-in functions, such as `meta-reduce` or `sameComponent`, although they can be imported in modules being used in the calls to these functions; and, in general, any function taking as argument or returning as result the metarepresentation of a module cannot take one of these built-in modules as argument. This is the case, for example, for the `up` function presented in Section 4.6.2, or for functions or commands in which the name of a module has to be specified, as the `select` or `down` commands, or the `up` function presented in Section 3.4. Nevertheless, there are also some advantages: The flattening of the built-in part of the structure is accomplished more efficiently, and, since these modules do not have to be stored in the database of Full Maude, the size of the database is reduced.

Our plan is to have in the future a hybrid solution. Once we have some way of storing the modules entered to Full Maude in Core Maude's database, it will be enough to keep in the Full Maude database just the original form of the top of all the modules, including built-ins, leaving all the importation declarations to be resolved by the engine. The structures will be normalized as they are now, so that the engine will have to deal just with inclusions, but it will be possible to use the predefined modules as any other module. Moreover, the Full Maude database will be relatively smaller and the flattening will be computed more efficiently.

When a new module or theory is entered, the names of all the modules, theories, and views depending on it are included in its lists of dependencies with functions `setUpUnitDependencies` and `setUpViewDependencies`. Notice that if new module expressions are defined, the function `setUpModExpDependencies` will have to be extended accordingly.

```
op setUpUnitDependencies : Unit Database -> Database .
op setUpModExpDependencies : ModName Database -> Database .
op setUpImportSetDependencies :
      ModName EImportList Database -> Database .

eq setUpUnitDependencies(U, DB)
  = setUpImportSetDependencies(name(U), importList(U),
      setUpModExpDependencies(name(U), DB)) .

eq setUpModExpDependencies((par X :: ME),
     database((< ME ; U ; U' ; M ; M' ; MNS ; VES > IS),
       MNS', VES', QIL))
  = database(
      (< ME ; U ; U' ; M ; M' ; MNS & (par X :: ME) ; VES > IS),
      MNS', VES', QIL) .
ceq setUpModExpDependencies((par X :: ME), DB)
  = warning(DB,
      ('\n 'ERROR:
       'module modNameToQidList(ME) 'not 'in 'database '.))
    if not unitInDatabase(ME, DB) .
eq setUpModExpDependencies(QI, DB) = DB .

eq setUpImportSetDependencies(MN, ((including MN' .) EIL),
     database((< MN' ; U ; U' ; M ; M' ; MNS' ; VES > IS),
       MNS'', VES', QIL))
  = setUpImportSetDependencies(MN, EIL,
       database((< MN' ; U ; U' ; M ; M' ; MNS' & MN ; VES > IS),
          MNS'', VES', QIL)) .
ceq setUpImportSetDependencies(MN, ((including MN' .) EIL), DB)
  = if MN' inModNameSet builtIns
    then DB
    else warning(DB,
           ('\n 'ERROR:
            'module modNameToQidList(MN') 'not 'in 'database '.))
    fi
    if not unitInDatabase(MN', DB) .
eq setUpImportSetDependencies(MN, nil, DB) = DB .
```

```
op setUpViewDependencies : ModExp ViewExp Database -> Database .

eq setUpViewDependencies(ME, VE,
      database(
        (< ME ; U ; U' ; M ; M' ; MNS' ; VES > IS),
        MNS'', VES', QIL))
  = database(
       (< ME ; U ; U' ; M ; M' ; MNS' ; VES # VE > IS),
       MNS'', VES', QIL) .
ceq setUpViewDependencies(ME, VE, DB)
  = if ME inModNameSet builtIns
    then DB
    else warning(DB,
           ('\n 'ERROR:
            'module modNameToQidList(ME) 'not 'in 'database '.))
    fi
    if not unitInDatabase(ME, DB) .
```

There are functions to insert and extract a view in the database.

```
op insertView : View Database -> Database .
op getView : ViewExp Database -> View .

eq insertView(view(VE, PL, ME, ME', VMAPS),
     database((< VE ; VI ; MNS > IS), MNS', VES, QIL))
  = setUpViewDependencies(ME, VE,
      setUpViewDependencies(ME', VE,
        deleteUnits(MNS,
          database(
            (< VE ; view(VE, PL, ME, ME', VMAPS) ; noneModNameSet > IS),
            MNS', VES, QIL)))) .
ceq insertView(view(VE, PL, ME, ME', VMAPS),
      database(IS, MNS, VES, QIL))
  = setUpViewDependencies(ME, VE,
      setUpViewDependencies(ME', VE,
        database(
          (< VE ; view(VE, PL, ME, ME', VMAPS) ; noneModNameSet > IS),
          MNS, (VE # VES), QIL)))
    if not (VE inViewExpSet VES) .

eq getView(VE, database((< VE ; VI ; MNS > IS), MNS', VES, QIL))
  = VI .
```

There are functions to insert the different versions of a unit, and to extract them. We only give here the equations for the insertion of top units to illustrate the way in which the consistency of the database is maintained. We assume that when the internal version, the signature, or the flat version of a module is entered in the database, its corresponding top module is already present in it.

```
op insertTopUnit : Unit Database -> Database .
op insertInternalTopUnit : Unit Database -> Database .
op insertSignature : ModName Module Database -> Database .
op insertFlatUnit : ModName Module Database -> Database .
op getTopUnit : ModName Database -> Unit .
op getInternalTopUnit : ModName Database -> Unit .
op getSignature : ModName Database -> Unit .
op getFlatUnit : ModName Database -> Unit .
```

```
  ceq insertTopUnit(U, database(IS, MNS, VES, QIL))
    = setUpUnitDependencies(U,
        database(
          (< name(U) ; U ; noModule ; noModule ;
             noModule ; noneModNameSet ; noneViewExpSet > IS),
          (name(U) & MNS), VES, QIL))
      if not (name(U) inModNameSet MNS) .
  ceq insertTopUnit(U,
        database((< MN ; U' ; U'' ; M ; M' ; MNS ; VES > IS),
          MNS', VES', QIL))
    = setUpUnitDependencies(U,
        deleteUnits(MNS,
          deleteViews(VES,
             database(
               (< MN ; U ; noModule ; noModule ;
                  noModule ; noneModNameSet ; noneViewExpSet > IS),
               MNS', VES',
               (QIL ('\n 'ADVISORY:
                     'Module modNameToQidList(MN) 'redefined)))))
      if name(U) == MN .

   ...

endfm
```

4.6 The Evaluation of Units

The general principle for the evaluation of units in our design consists in first evaluating any module expression, reducing it to a canonical form in which only unit inclusions appear, that is, to a unit hierarchy, which can be seen as a partial order of unit inclusions. The design of the Full Maude system has been based upon the principle of evaluating all module expressions to irreducible structured units, and on using the flat version of the units only for execution purposes. We have then two different processes clearly distinguished: a first step in which the structured unit is evaluated and reduced to its normal form, and a second step in which this normal form is flattened. We illustrate both stages of this process in Section 4.6.1 with a very simple example, namely the importation of a renamed module.

As explained in Section 4.1.1, the process of evaluation to normal form is also responsible for the parsing of the bubbles in the premodules, which is accomplished once the signature has been built. The parsing of bubbles is discussed in Section 4.6.4. To be able to handle the `up` function and the `down` command presented in Section 3.4, it is necessary to be able to move terms and modules from one level of reflection to another. The functionality to move between levels is presented in Section 4.6.2, where functions `up` and `down` on sorts `Module` and `Term` are defined. The transformation of object-oriented modules into system modules in discussed in Section 4.6.6. The evaluation of module expressions is discussed in Sections 4.6.5, 4.6.8, 4.6.9, and 4.6.10.

4.6.1 Overview of the Unit Evaluation Process

Let us consider the following specification `NAT` of natural numbers, as a functional module with a sort `Nat` and operations `zero` and `suc` with the usual meaning, and a specification `NAT3` of the natural numbers modulo 3, given by importing a renamed copy of the module NAT and adding the corresponding congruence equation. Note that this module is imported in *including* mode, that is, nothing is guaranteed about the relationship between the initial models of NAT and NAT3.

```
(fmod NAT is
   sort Nat .
   op zero : -> Nat .
   op suc : Nat -> Nat .
 endfm)

(fmod NAT3 is
   including NAT *(sort Nat to Nat3) .
   eq suc(suc(suc(zero))) = zero .
 endfm)
```

The renaming process is summarized in the following picture. Basically, we create a copy of the module NAT, with name NAT *(sort Nat to Nat3), in which the sort Nat has been renamed to Nat3. This amounts to a module transformation denoted by ϕ in the picture below. Then, the renamed module is included as a submodule of NAT3, in which the congruence equation is added. It is important to highlight the fact that there is no inclusion relationship between the modules NAT and NAT3. The "confusion" associated to the congruence modulo 3 equation has only been introduced in the sort Nat3, which has nothing to do with the sort Nat.

$$\begin{tikzcd} & \mathtt{NAT3} \\ \mathtt{NAT} \arrow[r, dashed, "\phi"'] & \mathtt{NAT\ *(sort\ Nat\ to\ Nat3)} \arrow[u, hook] \end{tikzcd}$$

The metarepresentations of the modules NAT and NAT3 before being evaluated are given by the following terms of sort StrFModule:

```
fmod 'NAT is
  nilParameterList
  nil
  sorts 'Nat .
  none
  op 'zero : nil -> 'Nat [none] .
  op 'suc : 'Nat -> 'Nat [none] .
  none
  none
  none
endfm

fmod 'NAT3 is
  nilParameterList
  including 'NAT *(sort 'Nat to 'Nat3) .
  sorts none .
  none
  none
  none
  none
  eq 'suc['suc['suc[{'zero}'Nat3]]] = {'zero}'Nat3 .
endfm
```

Normalization of Extended Modules

For `NAT3`, the first stage of the module evaluation process consists in the reduction of the module to its structured normal form. This normalization step can be understood as the evaluation of all module expressions in it. Since the normal forms of all previously defined modules have been computed and have been stored in the database, the module expressions have to be evaluated only in the module at the top of the hierarchy. However, some of the operations may affect the whole structure, or part of it. This happens, for example, with the renaming operation when some of the renamings affect sorts or operators in a submodule. We will discuss the details about the evaluation of the renaming module expression in Section 4.6.10. Note that in this case, as in general for all the module operations we deal with, given a structured module as input, we evaluate it without flattening, and return a structured module as result.

The evaluation of the only module expression appearing in the module `NAT3`, namely `NAT *(sort Nat to Nat3)`, results in a new module, whose *name* is given by the module expression itself. This module is then introduced in the database, in such a way that if the module expression appears again, it will not be recomputed. The representation of this module at the metalevel is

```
fmod 'NAT *(sort Nat to Nat3) is
  nilParameterList
  nil
  sorts 'Nat3 .
  none
  op 'zero : nil -> 'Nat3 [none] .
  op 'suc : 'Nat3 -> 'Nat3 [none] .
  none
  none
  none
endfm
```

Note that the name of the module is the module expression itself. Thus, the normalization process does not change the text of the module being normalized, such as `NAT3` in our example. Instead, each imported module expression is evaluated and a new module with that expression as name is entered into the database. In this way, the normalization yields a structured module in which only inclusions appear. The following picture shows the form of this structure for the example.

$$\begin{tikzcd} \texttt{NAT3} \\ \texttt{NAT *(sort Nat to Nat3)} \arrow[u, hook] \end{tikzcd}$$

Flattening

The flattening of the normalized structure is accomplished following the tradition of the Clear/OBJ family of languages, in which specification structuring is based on the categorical concept of *colimit* [15, 38, 52]. However, instead of considering the category of specifications and specification morphisms [15], flattening is understood as a colimit in the category of specifications and inclusions of specifications. The colimit of a diagram in this category coincides with the set-theoretic union of the theories in the diagram. This is accomplished by the function

$$\texttt{evalUnit} : \texttt{Unit Database} \rightarrow \texttt{Database}$$

that takes a unit U and a database and returns the database after having evaluated all the module expressions appearing in U. The corresponding flat form of the module is also included in the database.

The resulting flat specification for NAT3 is:

```
fmod 'NAT3 is
  nilParameterList
  nil
  sorts 'Nat3 .
  none
  op 'zero : nil -> 'Nat3 [none] .
  op 'suc : 'Nat3 -> 'Nat3 [none] .
  none
  none
  eq 'suc['suc['suc[{'zero}'Nat3]]]
    = {'zero}'Nat3 .
endfm
```

Note that in the flattening of a structured module we are assuming that sort names are unique in each structure, although not necessarily in the entire database. This proposal is simpler than the solution taken, for example, in OBJ3 [60], in which sorts and operations are qualified by module names. However, although simpler, this solution would not be fully satisfactory without some mechanism to help the user avoid the systematic renaming of repeated sorts, which—as discussed in Section 3.6.2—is a common occurrence in the presence of parameterized modules. This is accomplished by parameterized sorts, already introduced in Section 4.3.2 and further discussed in Section 3.6.2.

4.6.2 Changing Levels

Moving terms of sorts `Term` and `Module` from one level of reflection to another is possible thanks to the `up` and `down` functions, which are defined, respectively, in the following modules `MOVE-UP` and `MOVE-DOWN`.

The `up` Function

Given a term of sort `Module` or `Term`, the `up` function, defined in the following module `MOVE-UP`, returns the term metarepresenting it. The function is also defined on the different components of a unit. The `up` function on units is defined recursively by applying it to the different components of a unit.

The semantics of the `up` functions can be inferred from the following example. The result of applying the `up` function to the metarepresentation of the flat form of the module `NAT3` presented in Section 4.6.1 is the following.

```
red up(fmod 'NAT3 is
          nil
          sorts 'Nat3 .
          none
          op 'zero : nil -> 'Nat3 [none] .
          op 'suc : 'Nat3 -> 'Nat3 [none] .
          none
          none
          eq 'suc['suc['suc[{'zero}'Nat3]]]
            = {'zero}'Nat3 .
        endfm) .
```

```
result Term:
  'fmod_is_______endfm[
     {''NAT3}'Qid,
     {'nil}'ImportList,
     'sorts_.[{''Nat3}'Qid],
     {'none}'SubsortDeclSet,
     '__['op_:_->_`[_`].[{''zero}'Qid, {'nil}'QidList, {''Nat3}'Qid,
            {'none}'AttrSet],
         'op_:_->_`[_`].[{''suc}'Qid, {''Nat3}'Qid, {''Nat3}'Qid,
            {'none}'AttrSet]],
     {'none}'VarDeclSet,
     {'none}'MembAxSet,
     'eq_=_.['_`[_`][{''suc}'Qid,
               '_`[_`][{''suc}'Qid,
                 '_`[_`][{''suc}'Qid,
                   '`{_`}_[{''zero}'Qid, {''Nat3}'Qid]]]],
             '`{_`}_[{''zero}'Qid, {''Nat3}'Qid]]]
```

We shall see in Section 4.6.4 how the up function is used to evaluate the homonymous function discussed in Section 3.4. In Section 4.6.9 we shall discuss how the up function is used to evaluate the META-LEVEL module expression (see Section 3.4).

```
fmod MOVE-UP is
  pr UNIT .
  op up : Module -> Term .
  op up : Term -> Term .

   ...

endfm
```

The down Function

Given a term of sort Term metarepresenting a term of sort Term or Module, the down function can be seen as the inverse of the up function discussed in the previous section, that is, it returns the original term that had been metarepresented. There are also down functions for terms metarepresenting terms in other sorts. We present here only some of them.

We assume that the down functions are called with valid metarepresentations. In fact, these functions should be declared as partial functions going to error sorts when their arguments are invalid.

The main application of the down functions is in the evaluation of the down command (see Section 3.4). However, they are also used in other tasks, as for example in the parsing of some inputs.

```
fmod MOVE-DOWN is
  pr UNIT .
  op downTerm : Term -> Term .
  op downModule : Term -> Module .
  op downQid : Term -> Qid .
  op downQidList : Term -> QidList .
  op downQidSet : Term -> QidSet .
  op downModName : Term -> ModName .
  op downMachineInt : Term -> MachineInt .

   ...

endfm
```

4.6.3 Renaming of Variables

To avoid the clash of names of variables appearing in different units in the same unit hierarchy, which can appear very easily when such a hierarchy is flattened to a single unit, variables are systematically renamed. The function `renameVars` takes a unit and returns it with all the variables in it renamed following the convention of adding the character '`@`' plus the name of its sort after being converted to a `Qid`. Thus, for example, a variable `'N` of sort `'Nat` is renamed to `'N@Nat`, and a variable `'L` of sort `'List['X]` is renamed to ``'L@List`[X`]``.

```
fmod VBLE-RENAMING is
  pr UNIT .
  pr EXT-SORT-TO-QID .

  op renameVars : Unit -> Unit .

   ...

endfm
```

4.6.4 Parsing of Bubbles

As discussed in Section 4.1, in Full Maude, the parsing process is split into two phases. In a first stage, the input is parsed using the top-level grammar for Full Maude modules, theories, views, and commands. Once this first stage is completed, we get a term with bubbles in it, which is converted into a module, theory, or view. This unit or view may still have the bubbles in it. We say that a module with bubbles is a premodule, a view with bubbles a preview, and so on. The second stage of the process consists in taking this preunit or preview and converting the bubbles in it into terms by parsing them in the appropriate signatures, obtaining a 'valid' unit or view out of it, or otherwise a parsing error. In the case of commands, if they contain any bubble, the same will have to be done. All bubbles have to be parsed in the appropriate signature before any further processing can be done with the module, view, or command in which they appear.

Parsing of Bubbles

In the following module `BUBBLE-PARSING`, the definitions for the basic processing of bubbles are introduced. In it we declare a function `solveBubbles` which takes a bubble and some other arguments and returns the term resulting from parsing it.

```
fmod BUBBLE-PARSING is
  pr DATABASE .
  pr MOVE-UP .
  pr MOVE-DOWN .
  pr EXT-SORT-TO-QID .

  var T : Term .
  var M : Module .
  var B : Bool .
  var QIL : QidList .
  var DB : Database .
```

As we shall see in Section 4.6.7, a declaration importing the predefined module `UP` (see Section 4.5.1) is added to all modules importing the `META-LEVEL` module. The `solveBubbles` function is called with a 'flag' indicating whether the module can contain calls to the `up` function or not. Thus, when we call `meta-parse` with some bubble and the module in which such bubble has to be parsed, if there are occurrences of the function `up` in it, they will be of the form `'up['token[T]]` or `'up['token[T], 'bubble[T']]` for terms `T` and `T'`. The function `solveUps` will evaluate them.

```
  op solveBubbles : Term Module Bool Database -> Term .
  op solveBubbles1 : QidList Module Bool Database -> Term .
  op solveUps : TermList Database -> TermList .
```

The function `solveBubbles` calls the function `solveBubbles1` after lowering the list of quoted identifiers of the bubble.

```
  eq solveBubbles('bubble[T], M, B, DB)
    = solveBubbles1(downQidList(T), M, B, DB) .
```

The `solveBubbles1` function is in charge of calling the function `meta-parse`. The flag indicating the inclusion of the module `META-LEVEL` in the module in which the term appears decides whether the function `solveUps` is called or not, so the extra price of searching for calls to the `up` function is paid only when an occurrence of the function is possible. This function takes care of the occurrences of the `up` function that may exist in such bubbles.

The function `solveUps` goes through the term looking for a term with `'up` as top operator and `'token` as top operator of its unique argument if there is only one argument, or with `'token` and `'bubble` as top operators of its first and second arguments, respectively, if there are two. If a term of the form `'up['token[T]]` is reached, it is replaced by the metarepresentation of the flat version of the module in the database with the name given by the token. If a term of form `'up['token[T], 'bubble[T']]` is reached, the metarepresentation of the result of parsing the bubble in the signature of the module with the name given by the token, after solving possible nested calls to the `up` function, is returned.

```
  eq solveBubbles1(QIL, M, true, DB) = solveUps(meta-parse(M, QIL), DB) .
  eq solveBubbles1(QIL, M, false, DB) = meta-parse(M, QIL) .

    ...

endfm
```

Parsing the Bubbles in a Unit

The `solveBubbles` function defined in the `UNIT-BUBBLE-PARSING` module takes a term of sort `Unit` (a preunit in fact) and a signature, and returns the unit resulting from the evaluation (parsing) of all the bubbles in it.

```
fmod UNIT-BUBBLE-PARSING is
  pr BUBBLE-PARSING .
  pr DATABASE .
  pr MOVE-UP .
  pr MOVE-DOWN .
  pr EXT-SORT-TO-QID .

  op solveBubbles : Unit Module Bool Database -> Unit .

    ...

endfm
```

4.6.5 Module Expression Evaluation

So far we have not introduced more module expressions than those given by simple quoted identifiers. We will introduce some later, but the scheme followed for evaluating them is very simple and can be presented in a generic way. Given a module expression and a database state, the evaluation of a module expression results in the generation of a new module, which is introduced in the database, with the module expression as its name. The resulting database is then returned. If there is already a module in the database with that name, the function returns the original database without any change. The evaluation of a module expression may produce the evaluation of other module expressions contained in the modules involved in the process. This is the case, for example, for the renaming of modules, in which not only the top module is renamed but, perhaps, some of its submodules as well; it is also the case for the instantiation of parameterized modules, where the module being instantiated may contain submodules which are parameterized by some of the parameter theories of the parameterized module in which are imported. We shall discuss in more detail the renaming and instantiation of module expressions in Sections 4.6.10 and 4.6.9, respectively.

We saw in Section 3.6.4 how it is possible to import a module expression in which a parameterized module is instantiated by some of the formal parameters of the parameterized module into which it is imported. To be able to evaluate this kind of module expression, the list of parameters of the module in which the module expression appears has to be given.

```
fmod MOD-EXPR-EVAL is
  pr DATABASE .

  op evalModExp : ModExp Database -> Database .
  op evalModExp : ModExp ParameterList Database -> Database .

  var QI : Qid .
  var ME : ModExp .
  var PL : ParameterList .
  var DB : Database .

  eq evalModExp(ME, DB)
    = evalModExp(ME, nilParameterList, DB) .

  ceq evalModExp(QI, PL, DB)
    = if QI inModNameSet builtIns
      then DB
      else warning(DB, ('ERROR: 'module QI 'not 'in 'database))
      fi
      if not unitInDatabase(QI, DB) .
  ceq evalModExp(ME, PL, DB)
    = DB
      if unitInDatabase(ME, DB) .

endfm
```

4.6.6 The Transformation of Object-Oriented Modules to System Modules

The transformation of object-oriented modules into system modules has already been discussed in Section 3.3.2, and also in [77, 41, 21]. We focus here on the part of the process accomplished by each of the main functions involved in the transformation.

The transformation discussed in [41] assumed that object-oriented modules were flattened before being transformed into system modules. However, doing it in this way, the transformations already made for the modules in the structure were not reused. In the current system, the transformation is done only for the module being introduced, the top of the structure, and using the 'internal' representations of the submodules stored in the database for the rest of the structure.

This approach requires gathering all class and subclass relation declarations in the structure before starting with the transformation process itself. The function `prepareClasses` collects all these declarations in the structure, and completes all the declarations of classes with the attributes inherited from their superclasses.

Once all the class declarations in the structure have been collected and completed, the transformation is accomplished in two stages. First, the function `objectOrientedToSystemModuleAux` carries out the following tasks:

- For each class declaration of the form `class` C `|` a_1`:`S_1 `,` ... `,`a_n`:`S_n, the following items are introduced: a subsort C of sort `Cid`, a constant C of sort C, and declarations of operations a_i `:_` `:` S_i `-> Attribute` for each attribute a_i (the function `attributeOpDeclSet` creates these declarations).

- For each subclass relation of the form `subclass` C `<` C', a subsort declaration `subsort` C `<` C' is introduced.

- For each message declaration of the form `msg F : ESL -> ES`, an operator declaration `op F : ESL -> ES` is added.

When this process has been completed, the function `prepareAxioms` is called. This function applies to the membership axioms, equations, and rewriting rules in the module the transformations indicated in Section 3.3.2, so that they become applicable to all the objects of the given class and of their subclasses. The set of attributes of the objects appearing in the membership axioms, equations, and rewriting rules are completed, so that the default convention of not having to exhaustively mention the set of attributes of a class is supported.

Note that in Meseguer's paper [77] a parallel hierarchy of sorts was defined to deal with objects in different classes, and membership axioms constraining the objects to their corresponding sorts were added. The transformation could be easily completed with sorts, subsort relations, and membership constraints as indicated there. In fact, these declarations were added in an initial version and were then removed because they were computationally expensive. However, there are examples in which it would be interesting to have them; when needed, these declarations can be explicitly added by the user in the current version.

```
fmod O-O-TO-SYSTEM-MOD-TRANSF is
  pr DATABASE .
  pr VBLE-RENAMING .

  var DB : Database .
  var I : MachineInt .
  var MN : ModName .
  var F : Qid .
  vars ES C C' : ESort .
```

```
var ESS : ESortSet .
var ESL : ESortList .
vars T T' T'' T''' : Term .
var PL : ParameterList .
vars EIL EIL' : EImportList .
var ESD : ESortDecl .
vars CDS CDS' : ClassDeclSet .
var ADS : AttrDeclSet .
var ESSDS : ESubsortDeclSet .
var SCDS : SubclassDeclSet .
var EODS : EOpDeclSet .
var MDS : MsgDeclSet .
var EVDS : EVarDeclSet .
vars EMAS EMAS' : EMembAxSet .
vars EqS EqS' : EquationSet .
vars RlS RlS' : RuleSet .
```

The function `prepareClasses` completes all classes in the module with all the attributes they inherit from their superclasses.

```
op prepareClasses :
     ClassDeclSet SubclassDeclSet EImportList Database -> ClassDeclSet .
```

Given a set of attribute declarations, the `attributeOpDeclSet` function returns a set of operator declarations as indicated above. That is, for each attribute $a\!:\!S$, an operator of the form a `:_ :` S `-> Attribute` is declared.

```
op attributeOpDeclSet : AttrDeclSet -> OpDeclSet .
```

The function `prepareLHS` takes the term in the lefthand side of a rule, equation, or membership axiom, and replaces each object `<` O `:` C `|` ADS `>` in it—with O of sort `Oid`, C the name of a class, and ADS a set of attributes with their corresponding values—by an object `<` O `:` V `|` $ADS\ ADS'\ Atts$ `>` where the identifier of the class is replaced by a variable V of sort C, which is not used in the axiom, and where the set of attributes is completed with attributes ADS' as indicated in Section 3.3.2, so that each attribute declared in class C or in any of its superclasses is added with a new variable as value. *Atts* is a new variable of sort `AttributeSet`, which is used to range over the additional attributes that may appear in objects of a subclass.

The function `prepareLHS` takes as arguments a term (in the initial call, the term in the lefthand side of a rule, equation, or membership axiom), the set of variable declarations of those variables declared in the module that are not used in the axiom—new variables are created only if there are no variables in the module with the appropriate sort—the set of class declarations in the module—to be able to complete the set of attributes in the occurrences of the objects—and an index—to make sure that the variables being added have not been added previously. In the initial call this index is set to zero. `prepareLHS` gives as result a tuple composed of the resulting term, the set of objects in the term (so that the modification of the objects in the righthand side of the rule is simplified), the set of variable declarations corresponding to the new added variables, the set of variable declarations of the variables in the module that have not been used, and the index for the creation of new variables.

The set of objects in the lefthand side will be given as a set of terms. The sort `TermSet` is defined as a supersort of the sort `Term`, and with constructors `emptyTermSet` and `termSet` as follows.

```
sort TermSet .
subsort Term < TermSet .
op emptyTermSet : -> TermSet .
op termSet : TermSet TermSet -> TermSet
      [comm assoc id: emptyTermSet] .
```

Terms of sort `PrepareLHSResult` are built with constructor `<_;_;_;_;_>`, and have selectors for their different components.

```
sort PrepareLHSResult .
op <_;_;_;_;_> : TermList TermSet EVarDeclSet EVarDeclSet MachineInt
     -> PrepareLHSResult .
op term : PrepareLHSResult -> TermList .
op objects : PrepareLHSResult -> TermSet .
op oldVbles : PrepareLHSResult -> EVarDeclSet .
op newVbles : PrepareLHSResult -> EVarDeclSet .
op index : PrepareLHSResult -> MachineInt .

op prepareLHS : TermList EVarDeclSet ClassDeclSet MachineInt
     -> PrepareLHSResult .
```

In the case of equations and rules, the function `prepareAxioms` calls the function `prepareLHS` with the term in the lefthand side of the axiom, and then use the generated set of objects to call the `prepareRHS` function. For conditional equations, rules, and membership axioms, this set of terms representing the objects in the lefthand side is also used in the calls to `prepareRHS` with each of the terms in the conditions. The term in the lefthand side of the equation, rule, or membership axiom is replaced by the term returned by `prepareLHS`. The variables generated are added to the variables in the module, and the index is used in the recursive calls to `prepareAxioms`.

`prepareLHS` returns as second argument the set of objects (as a set of terms) appearing in it. These objects are returned after extending their set of attributes by those of the class to which they belong not already specified.

```
op prepareAxioms : Unit MembAxSet EquationSet RuleSet
     ClassDeclSet MachineInt -> Unit .
```

We illustrate the process presenting the equation corresponding to the case of conditional equations.

```
eq prepareAxioms(
     mod MN is PL EIL ESD ESSDS EODS EVDS EMAS EqS RlS endm,
     EMAS', ((ceq T = T' if T'' = T''' .) EqS'), RlS', CDS, I)
  = prepareAxioms(
      mod MN is
         PL EIL ESD ESSDS EODS
         (EVDS newVbles(
                 prepareLHS(T,
                   varDeclSetDiff(EVDS,
                     (varDeclSetInTerm(EVDS, T)
                      varDeclSetInTerm(EVDS, T'))),
                   CDS, I)))
         EMAS
         ((ceq term(
                 prepareLHS(T,
                   varDeclSetDiff(EVDS,
                     (varDeclSetInTerm(EVDS, T)
```

```
                  varDeclSetInTerm(EVDS, T'))),
                CDS, I))
            = prepareRHS(
                objects(
                  prepareLHS(T,
                    varDeclSetDiff(EVDS,
                      (varDeclSetInTerm(EVDS, T)
                       varDeclSetInTerm(EVDS, T'))),
                    CDS, I)),
                T')
            if prepareRHS(
                 objects(
                   prepareLHS(T,
                     varDeclSetDiff(EVDS,
                       (varDeclSetInTerm(EVDS, T)
                        varDeclSetInTerm(EVDS, T'))),
                     CDS, I)), T'')
               = prepareRHS(
                   objects(
                     prepareLHS(T,
                       varDeclSetDiff(EVDS,
                         (varDeclSetInTerm(EVDS, T)
                          varDeclSetInTerm(EVDS, T'))),
                       CDS, I)), T''') .) EqS)
         RlS
     endm,
     EMAS', EqS', RlS', CDS,
     index(
       prepareLHS(T,
         varDeclSetDiff(EVDS,
           (varDeclSetInTerm(EVDS, T) varDeclSetInTerm(EVDS, T'))),
         CDS, I))) .
```

After completing the set of classes in the module with the attributes from their superclasses, the function `objectOrientedToSystemModule` calls the function `objectOrientedToSystemModuleAux` with the same module and the set of class declarations. The definition of the `objectOrientedToSystemModule` function is given by the five equations below.

```
op objectOrientedToSystemModule :
     StrOModule Database -> StrSModule .
op objectOrientedToSystemModuleAux :
     StrOModule ClassDeclSet -> StrSModule .

eq objectOrientedToSystemModule(
     omod MN is PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS endom,
     DB)
  = objectOrientedToSystemModuleAux(
      omod MN is
         PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
      endom,
      prepareClasses(CDS, SCDS, EIL, DB)) .

eq objectOrientedToSystemModuleAux(
     omod MN is
        PL EIL (sorts ESS .) ESSDS ((class C | ADS .) CDS)
        SCDS EODS MDS EVDS EMAS EqS RlS endom,
```

```
      CDS')
    = objectOrientedToSystemModuleAux(
        omod MN is
           PL EIL (sorts (ESS ; C) .)
           (subsort C < 'Cid . ESSDS)
           CDS SCDS
           ((op eSortToQid(C) : nil -> C [none] .)
            attributeOpDeclSet(ADS) EODS)
           MDS EVDS EMAS EqS RlS
        endom,
        CDS') .
  eq objectOrientedToSystemModuleAux(
       omod MN is
          PL EIL ESD ESSDS CDS ((subclass C < C' .) SCDS)
          EODS MDS EVDS EMAS EqS RlS
       endom,
       CDS')
    = objectOrientedToSystemModuleAux(
        omod MN is
           PL EIL ESD ((subsort C < C' .) ESSDS)
           CDS SCDS EODS MDS EVDS EMAS EqS RlS
        endom,
        CDS') .
  eq objectOrientedToSystemModuleAux(
        omod MN is
           PL EIL ESD ESSDS CDS SCDS EODS ((msg F : ESL -> ES .) MDS)
           EVDS EMAS EqS RlS
        endom,
        CDS')
    = objectOrientedToSystemModuleAux(
         omod MN is
            PL EIL ESD ESSDS CDS SCDS ((op F : ESL -> ES [none] .) EODS)
            MDS EVDS EMAS EqS RlS
         endom,
         CDS') .
  eq objectOrientedToSystemModuleAux(
       omod MN is
          PL EIL ESD ESSDS none none EODS none EVDS EMAS EqS RlS
       endom, CDS)
    = prepareAxioms(
        mod MN is PL EIL ESD ESSDS EODS EVDS none none none endm,
        EMAS, EqS, RlS, CDS, 0) .

   ...

endfm
```

4.6.7 Evaluation of Modules and Theories

As explained in Section 4.6.1, in our approach transforming a module from its possibly complex structured version to its unstructured form is a two-step process. First, all module expressions are evaluated, generating an intermediate form in which there are only simple inclusion relationships among the modules. This first step can be seen as the reduction of a structured specification to its structured *normal form*. Then, in a second step, this structured normal form is flattened into an unstructured specification. Note, however, that the importation of built-in modules is left explicit in the flattened form. The function normalization is in charge of normalizing the structure.

The process of evaluation of a preunit has to take into account the possibility of bubbles being contained in it. Depending on whether it is dealing with a preunit or with a unit, the evaluation process is accomplished by two different functions, namely, evalPreUnit and evalUnit. One function or the other will be called in each case. Evaluating a module already in the database, which is done by evalUnit, does not require bubble handling. Besides this difference, both functions proceed in a similar way. Before presenting the functions evalPreUnit and evalUnit we introduce some auxiliary declarations.

```
fmod EVALUATION is
  pr O-O-TO-SYSTEM-MOD-TRANSF .
  pr MOD-EXPR-EVAL .
  pr UNIT-BUBBLE-PARSING .
  pr DECL-EXT-SORT-TO-QID .

  sort UnitList .
  subsort Unit < UnitList .

  op nil : -> UnitList .
  op __ : UnitList UnitList -> UnitList [assoc id: nil] .

  var QIL : QidList .
  var M : Module .
  vars PU U U' U'' : Unit .
  var UL : UnitList .
  var DB : Database .
  var ME : ModExp .
  var P : Parameter .
  var PL : ParameterList .
  var EIL : EImportList .
  var ESD : ESortDecl .
  var CDS : ClassDeclSet .
  var ESSDS : ESubsortDeclSet .
  var SCDS : SubclassDeclSet .
  var EODS : EOpDeclSet .
  var MDS : MsgDeclSet .
  var EVDS : EVarDeclSet .
  var EMAS : EMembAxSet .
  var EqS : EquationSet .
  var RlS : RuleSet .
```

The subunitImportList function returns the list of all the subunits of a given unit. It is called with the list of importations of the given unit as first argument, and proceeds recursively through its structure collecting all the subunits in it.

```
  op subunitImportList : EImportList Database -> EImportList .
```

The function getNonBuiltInUnitList returns the list of those units in the list of importations given as argument which are not built-in.

```
  op getNonBuiltInUnitList : EImportList Database -> UnitList .
```

The function selectBuiltInImports returns the sublist of the importations of built-in modules.

```
  op selectBuiltInImports : EImportList -> EImportList .
```

The normalization of a structure consists in evaluating each of the module expressions appearing in it. Note that, if the `evalModExp` function generates new modules, they will be evaluated using the `evalUnit` function, producing recursive calls on the part of the structure not previously normalized.

```
op normalization : EImportList ParameterList Database -> Database .
eq normalization(((including P .) EIL), PL, DB)
  = normalization(EIL, PL, DB) .
eq normalization(((including ME .) EIL), PL, DB)
  = normalization(EIL, PL, evalModExp(ME, PL, DB)) .
eq normalization(nil, PL, DB) = DB .
```

`checkSortClashes` checks whether the intersection of the two sets of sorts given as arguments is empty or not. If it is nonempty, then there is a clash of names, and a warning message is passed to the database. The check is very simple, and only reports the name of one of the modules from which the sorts come. Only the name of the module from which the sorts given as second argument come is known at this point. This is the module name given as first argument.

```
op checkSortClashes : ModName ESortSet ESortSet Database -> Database .
```

In the current system, the only transformation handled by the `transform` function is the one from object-oriented modules to system modules, which is accomplished by the `objectOrientedToSystemModule` function presented in Section 4.6.6. However, `transform` has been defined as a general transformation that could affect other kinds of modules in a future extension.

```
op transform : Unit Database -> Unit .

eq transform(
     omod MN is
        PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
     endom, DB)
  = objectOrientedToSystemModule(
      omod MN is
         PL EIL ESD ESSDS CDS SCDS EODS MDS EVDS EMAS EqS RlS
      endom, DB) .
```

The function `signature` generates a functional module of sort `FModule`, without equations, by "forgetting" the appropriate declarations and converting extended sorts and module names into quoted identifiers.

```
op signature : Unit -> FModule .
eq signature(unitError(QIL)) = unitError(QIL) .
ceq signature(U)
  = fmod modNameToQid(name(U)) is
       importList(U)
       sorts eSortToQid(sortSet(sortDecl(U))) .
       eSortToQid(subsortDeclSet(U))
       eSortToQid(opDeclSet(U))
       eSortToQid(varDeclSet(U))
       none
       none
    endfm
    if (U : StrTheory) or (U : StrModule) .
```

The function `flatUnit` generates a module of sort `Module` "forgetting" declarations and converting extended sorts and module identifiers into quoted identifiers.

```
op flatUnit : Unit -> Module .
eq flatUnit(unitError(QIL)) = unitError(QIL) .
ceq flatUnit(U)
  = if U : FUnit
    then (fmod modNameToQid(name(U)) is
              importList(U)
              sorts eSortToQid(sortSet(sortDecl(U))) .
              eSortToQid(subsortDeclSet(U))
              eSortToQid(opDeclSet(U))
              eSortToQid(varDeclSet(U))
              eSortToQid(membAxSet(U))
              equationSet(U)
           endfm)
    else (mod modNameToQid(name(U)) is
              importList(U)
              sorts eSortToQid(sortSet(sortDecl(U))) .
              eSortToQid(subsortDeclSet(U))
              eSortToQid(opDeclSet(U))
              eSortToQid(varDeclSet(U))
              eSortToQid(membAxSet(U))
              equationSet(U)
              ruleSet(U)
           endm)
    fi
    if (U : StrTheory) or (U : StrModule) .
```

The evaluation process for units without bubbles is as follows. After normalizing the structure, the function `evalUnit` calls `evalUnit1` with an empty copy of the module to which the list of declarations of importations of built-in modules is added, and with the list of its nonbuilt-in subunits.

`evalUnit1` accumulates all the declarations in all the nonbuilt-in submodules in the copy of the module passed as second argument. The top module is then introduced in the database, and, after calling the `transform` function and renaming all the variables in it, the internal version of such a module is entered in the database as well.

Finally, `evalUnit2` generates the signature and the flat version of the module and enters them in the database.

```
op evalUnit : Unit Database -> Database .
op evalUnit1 : Unit Unit UnitList Database -> Database .
op evalUnit2 : Unit Unit Database -> Database .

eq evalUnit(U, DB)
  = evalUnit1(U,
      setImportList(empty(U),
        selectBuiltInImports(
          subunitImportList(importList(U),
            normalization(importList(U), parameterList(U), DB)))),
      getNonBuiltInUnitList(
        subunitImportList(importList(U),
          normalization(importList(U), parameterList(U), DB)),
        normalization(importList(U), parameterList(U), DB)),
      normalization(importList(U), parameterList(U), DB)) .
```

```
eq evalUnit1(U, U', (U'' UL), DB)
  = evalUnit1(U, addDecls(U', setImportList(U'', nil)), UL,
      checkSortClashes(
        name(U), sortSet(sortDecl(U')),
        sortSet(sortDecl(U'')), DB)) .
eq evalUnit1(U, U', nil, DB)
  = evalUnit2(setImportList(transform(U, DB), nil), U',
      checkSortClashes(
        name(U'), sortSet(sortDecl(U)), sortSet(sortDecl(U')),
        insertInternalTopUnit(renameVars(transform(U, DB)),
          insertTopUnit(U, DB)))) .

eq evalUnit2(U, U', DB)
  = insertFlatUnit(name(U),
      flatUnit(addDecls(U, U')),
      insertSignature(name(U),
        signature(addDecls(U, U')), DB)) .
```

The function `evalPreUnit` has to take care of the bubbles in the unit. As we explained in Section 4.6.1, both the signature and the flattened version of the module are created simultaneously, completing the parsing of the bubbles once the signature has been built, and then completing the flattened module.

The `evalPreUnit` function takes as arguments two copies of the module and a database. We shall see in Section 4.8.3 how these two modules are generated; the one passed as first argument has still bubbles in it, while the other one, which will be used to build the signature, does not contain any bubbles. This module without bubbles is the result of removing the bubbles from the declarations in it, or of removing the declarations themselves when they contain bubbles, as in the case of equations, for example.

The `evalPreUnit` function is quite similar to the function `evalUnit`. First, the structure is normalized by calling the `normalization` function, and then all the subunits in the structure are collected (accomplished by `subunitImportList` and `getNonBuiltInUnitList`) and the list of importations is updated with the sublist of importations of built-in modules (`selectBuiltInImports`). Second, the structure of all the subunits below the top is flattened to a single unit. This unit is used to create a first version of the signature (without identity elements of operators) in which all the bubbles in the top preunit are parsed (`solveBubbles`). The final version of the signature and the flat unit are generated once the bubbles have been parsed. The 'internal' version of the module is also generated by renaming the variables in it (`renameVars`). All these versions of the module are finally entered in the database.

Note that if the `META-LEVEL` module is imported in the module being evaluated, a declaration importing the predefined module `UP` (see Section 4.5.1) is added. With the declarations in this module it will be possible to parse bubbles containing calls to the `up` functions (see Section 3.4) in them.

```
op evalPreUnit : Unit Unit Database -> Database .
op evalPreUnit1 : Unit Unit UnitList Unit Database -> Database .
op evalPreUnit2 : Unit Unit Module Database -> Database .
op evalPreUnit3 : Unit Unit Module Database -> Database .

eq evalPreUnit(PU, U, DB)
  *** PU : top module with bubbles
  *** U  : top module without bubbles
```

```
  = evalPreUnit1(
      (if ((including 'META-LEVEL .) in
              subunitImportList(importList(PU),
                normalization(importList(PU), parameterList(PU), DB)))
       then addImportList((including 'UP .), PU)
       else PU
       fi),
      setName(empty(U), name(U)),
      getNonBuiltInUnitList(
        subunitImportList(
          importList(
            if ((including 'META-LEVEL .) in
                   subunitImportList(importList(PU), nil,
                     normalization(importList(PU),
                       parameterList(PU), DB)))
            then addImportList((including 'UP .), PU)
            else PU
            fi),
          normalization(importList(PU), parameterList(PU), DB)),
        normalization(importList(PU), parameterList(PU), DB)),
      setImportList(U,
        selectBuiltInImports(
          subunitImportList(importList(PU),
            normalization(importList(PU), parameterList(PU), DB)))),
      normalization(importList(PU), parameterList(PU), DB)) .

eq evalPreUnit1(PU, U, (U' UL), U'', DB)
  = evalPreUnit1(PU, addDecls(U, U'), UL, U'',
      checkSortClashes(
        name(U''), sortSet(sortDecl(U)), sortSet(sortDecl(U')), DB)) .
ceq evalPreUnit1(PU, U, nil, U', DB)
  = evalPreUnit2(PU, U,
      signature(addDecls(transform(U', DB), setImportList(U, nil))),
      checkSortClashes(
        name(U'), sortSet(sortDecl(U)), sortSet(sortDecl(U')), DB))
    if (U : StrTheory) or (U : StrModule) .
eq evalPreUnit1(PU, unitError(QIL), UL, U', DB) = warning(DB, QIL) .

eq evalPreUnit2(PU, U, M, DB)
  *** PU : top module with bubbles
  *** U  : everything below
  *** M  : complete signature (with variables of the top module)
  = evalPreUnit3(
      solveBubbles(PU, addImportList((including 'BOOL .), M),
        ((including 'META-LEVEL .) in importList(M)), DB),
      U, M,
      insertTopUnit(
        solveBubbles(PU, addImportList((including 'BOOL .), M),
          ((including 'META-LEVEL .) in importList(M)), DB), DB)) .

eq evalPreUnit3(PU, U, M, DB)
  *** PU : top module without bubbles
  *** U  : everything below
  *** M  : complete signature (with variables of the top module)
  = insertFlatUnit(name(PU),
      flatUnit(
        setImportList(addDecls(transform(PU, DB), U), importList(M))),
```

```
        insertSignature(name(PU),
          setOpDeclSet(M,
            opDeclSet(
              flatUnit(
                setImportList(
                  addDecls(transform(PU, DB), U), importList(M))))),
          insertInternalTopUnit(renameVars(transform(PU, DB)), DB))) .

  ...

endfm
```

Note that in both `evalUnit` and `evalPreUnit`, the function `transform` has to be invoked to transform the module into a functional or system module. In the current system, the only transformation available is from object-oriented modules to system modules.

4.6.8 Application of Map Sets

The following two modules deal with the application of a set of renaming maps to a module. Except for the proof obligations and additional checks associated with views—almost none of these checks are performed, and none of these proof obligations is generated in the current version—the way of applying a renaming map and a view map on a module is the same. Internally, they are treated in the same way; the only difference between them consists in the way of calling the function to accomplish this application.

Note that there might be some 'interference' between sort maps, and operator maps and message maps when they are applied. Let us consider for example a module with an operator declaration

```
  op f : Foo -> Foo .
```

and a renaming map set

```
  (sort Foo to Bar, op f : Foo -> Foo to g)
```

These renamings have to be applied carefully to avoid unintended behaviors. Depending on which of the maps is applied first, the other will be applicable or not. All the maps must be applied to the original module. To avoid the interference between the sort maps and other maps, the map set is divided into two sets: The first one contains the sort maps, and the second one contains the other maps.

We assume that there are no ambiguous mappings, that is, that we do not have, for example, maps `op f to g` and `op f to h`. In case of such ambiguity, one of the maps will be arbitrarily chosen.

Map Sets on Terms

The application of a set of view maps to a term is defined in the following module `VIEW-MAP-SET-APPL-ON-TERM`. The function `applyMapSetToTerm` is used to apply a given view map set to terms appearing in equations, rules, identity element declarations, and membership axioms, as part of the process of applying a map set to a unit.

```
fmod VIEW-MAP-SET-APPL-ON-TERM is
  pr UNIT .
  pr VIEW-MAP .
  pr EXT-SORT-TO-QID .
```

Note that all maps introduced in Sections 4.4.1 and 4.4.2, except for label maps, may affect a term. For example, sort maps will be applied to the qualifications of terms, and class and attribute maps have to be applied to the objects appearing in the term. Operator and message maps in which an explicit arity and coarity is given, and operator maps going to derived operators (see Section 3.6.3) must be applied to the complete family of subsort-overloaded operators.

The function `applyMapSetToTerm` takes as arguments two sets of view maps (the first set for sort maps, and the second for the other maps), the term to which the maps will be applied, and a module to be used in the matching of terms, sort comparisons, etc. Its declaration is as follows.

```
  op applyMapSetToTerm : ViewMapSet ViewMapSet Term Unit -> Term .

   ...

endfm
```

We do not include here the equations defining the semantics of the function `applyMapSetToTerm`. Instead, we present an example illustrating the meaning of the function. Renaming maps and view maps were already discussed in Sections 3.6.3 and 3.6.4.

Let us consider the following configuration in the module `STACK2[Accnt]` presented in Section 3.6.4. In this configuration we have objects in the class `Accnt` which represent the accounts of different clients of a bank, which is represented as an object `'bank` of class `Stack[Accnt]`. The object `'bank` in the example configuration below keeps a stack with the accounts of the bank represented as a linked list of nodes, each of which corresponds to the account of one of the clients.

```
  ('bank push 'john)
  ('peter elt 2000)
  < 'bank : Stack[Accnt] | first : o ('bank, 1) >
  < 'paul : Accnt | bal : 5000 >
  < 'peter : Accnt | bal : 2000 >
  < 'mary : Accnt | bal : 7200 >
  < 'john : Accnt | bal : 100 >
  < o('bank, 0) : Node[Accnt] | node : 'peter, next : null >
  < o('bank, 1) : Node[Accnt] | node : 'mary, next : o('bank, 0) > .
```

Let us apply the following renaming to the previous term.

```
  (op o to id,
   class Stack[Accnt] to Bank,
   msg _push_ : Oid Oid -> Msg to open`account`in_to_,
   msg _pop to close`account`of_,
   msg _elt_ to _owns_dollars,
   attr node . Node[Accnt] to client,
   attr bal . Accnt to balance)
```

The resulting term is as follows.

```
  (open account in 'bank to 'john)
  ('peter owns 2000 dollars)
  < 'bank : Bank | first : id('bank, 1) >
  < 'paul : Accnt | balance : 5000 >
```

```
  < 'peter : Accnt | balance : 2000 >
  < 'mary : Accnt | balance : 7200 >
  < 'john : Accnt | balance : 100 >
  < id('bank, 0) : Node[Accnt] | client : 'peter, next : null >
  < id('bank, 1) : Node[Accnt] | client : 'mary, next : id('bank, 0) >
```

The function `applyMapSetToTerm` treats the object constructor `<_:_|_>` in a special way. It cannot be renamed, and, when an occurrence of such a constructor is found, class and attribute maps require a particular handling. Inside terms these maps are only triggered when this constructor is found, and they are applied in a very restricted way, according to the general pattern for objects. We assume that the operator `<_:_|_>` is only used for objects and that objects constructed using it are well-formed.

Map Sets on Units

The application of view maps to modules and theories of the different types is defined in the following module `VIEW-MAP-SET-APPL-ON-UNIT`. The function `applyMapSetToUnit` is defined recursively by applying it to the different components of a unit. When the terms in the different declarations are reached, the function `applyMapSetToTerm` is called. This call is made with the set of maps split conveniently, as explained above.

```
fmod VIEW-MAP-SET-APPL-ON-UNIT is
  pr VIEW-MAP-SET-APPL-ON-TERM .

  op applyMapSetToUnit : ViewMapSet Unit Module -> Unit .

   ...

endfm
```

4.6.9 Instantiation of Parameterized Modules and the `META-LEVEL` Module Expression

A parameterized module $\mathtt{M[L_1\ ::\ T_1,\ \ldots,\ L_n\ ::\ T_n]}$, with $\mathtt{L_1 \ldots L_n}$ labels and $\mathtt{T_1 \ldots T_n}$ theory identifiers, is represented as a module with name `M` which contains parameter declarations $\mathtt{par\ L_i\ ::\ T_i}$ for $1 \leq i \leq n$, and an importation declaration $\mathtt{including\ par\ L_i\ ::\ T_i\ .}$ for each parameter $\mathtt{L_i\ ::\ T_i}$ in its interface. Note that all modules are handled in a uniform way: nonparameterized modules and theories have their list of parameters set to `nilParameterList`.

The instantiation of the formal parameters of a parameterized module with actual modules or theories requires a view from each formal parameter theory to its corresponding actual unit. The process of instantiation results in the replacement of each interface theory by its corresponding actual parameter, using the views to bind actual names to formal names.

The naming conventions for sorts have to be taken into account in the instantiation process: every occurrence of a sort coming from a theory in the interface of a module must be qualified by its theory's label, and sorts defined in the body of a parameterized module can be parameterized by the labels in the interface of the module (see Section 3.6.2).

The labeling convention for theories and for the sorts coming from them is very useful to avoid collisions of sort names coming from the parameter theories, and also to allow different uses of the same theory several times in the interface of a module. We assume that all sorts coming from the theory part of the parameter theories are used in their qualified form to manipulate the maps defined in the views before being applied to the body of the module being instantiated. If the target of a view is a theory, the sorts from the theory part of the target theory appearing in the targets of the maps in the view will be qualified as well, following the same convention.

When a parameterized module $\texttt{M[L}_1\texttt{ :: T}_1\texttt{, ..., L}_n\texttt{ :: T}_n\texttt{]}$ is instantiated with views $\texttt{V}_1 \ldots \texttt{V}_n$, each parameterized sort $\texttt{S[L}_1\texttt{,...,L}_n\texttt{]}$ in the body of the parameterized module is renamed to $\texttt{S[V}_1\texttt{,...,V}_n\texttt{]}$.

The discussion on the qualification of sorts in views before being used in the instantiation process applies in a completely similar way to class names in parameterized object-oriented modules.

As we saw in Section 3.6.4, it is possible to import a module expression in which a parameterized module is instantiated by some of the formal parameters of the parameterized module in which it is imported. This is done by using the label of some of the parameters in the interface of a module, say $\texttt{L}_k\texttt{ :: T}_k$, in a module expression in which some parameterized module `N` with formal parameter $\texttt{T}_k$ is instantiated with $\texttt{L}_k$, that is, we have the module expression $\texttt{N[}\ldots\texttt{L}_k\ldots\texttt{]}$. In this case, $\texttt{L}_k$ is considered as the identity view for the theory $\texttt{T}_k$ with $\texttt{L}_k$ as name. Note that to be able to check whether a label in the interface of a module is used in an instantiation of this form, in the evaluation of a module expression the list of parameters of the module in which the module expression appears must be available. This is the reason why the `evalModExp` function was defined with `ParameterList` as one of the sorts in its arity (see Section 4.6.5). For module expressions appearing outside of any module, that is, in commands, etc., this list will be set to `nilParameterList`.

Note that this kind of instantiation may produce a 'cascade' effect. The module being instantiated may itself import module expressions in which labels of some of its parameter theories are used in the instantiation of some module expression in its body. This is handled by 'preparing' the module expressions appearing in the importation declarations of the module (`prepareImportList`). This process consists in changing the labels of the interface of the module being instantiated which are used in the importations of module expressions by the corresponding view names (`prepareModName`). After completing the generation of the module resulting from the evaluation of the module expression, this module will be evaluated with the `evalUnit` function, producing the evaluation of these new module expressions. In any extension of the language, new equations for the function `prepareModName` will have to be added for each new kind of module expression being defined.

In Sections 4.6.10 and 4.10 we shall see how new equations completing the semantics of `prepareModName` are added for each new module expression being defined. In the case of the renaming module expression, the renaming maps will have to be prepared as well, to adjust the sort names being renamed to the conventions discussed above.

As for any other module expression being defined, in addition to the operator declaration for the constructor of the instantiation module expression, equations completing the semantics of operators `evalModExp`, `modNameToQidList`, and `setUpModExpDependencies` have to be given.

```
fmod INST-EXPR is
  pr EVALUATION .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  inc MOD-EXPR .
  inc MOD-NAME .
```

We start by giving the new constructor for sort `ModExp`. Note that the modules `MOD-EXPR` and `MOD-NAME` have been imported in `including` mode.

```
op _<_> : ModExp ViewExp -> ModExp .

vars QI X Y W Z : Qid .
var QIL : QidList .
vars M M' : Module .
vars U U' : Unit .
vars ME ME' : ModExp .
var MN : ModName .
vars MNS MNS' : ModNameSet .
vars VE VE' VE'' : ViewExp .
vars VES VES' : ViewExpSet .
var IS : InfoSet .
var DB : Database .
vars PL PL' PL'' : ParameterList .
vars EIL EIL' EIL'' : EImportList .
var VMAPS : ViewMapSet .
```

In the input given by the user, the operator `_[_]` is used both for the instantiation of module expressions, and for expressions parameterizing the module `META-LEVEL` with a list of module names. The function `evalModExp` distinguishes these two cases, calling the function `instantiation` in the former and the function `prepareMetalevel` in the latter.

```
op instantiation : ModName ViewExp ParameterList Database -> Database .
op prepareMetalevel : ViewExp Database -> Database .

ceq evalModExp(_<_>(ME, VE), PL, DB)
  = if ME == 'META-LEVEL
    then prepareMetalevel(VE, DB)
    else instantiation(ME, VE, PL, evalModExp(ME, PL, DB))
    fi
    if not unitInDatabase(_<_>(ME, VE), DB) .
```

The function `prepareMetalevel` creates a new module with the module expression being evaluated as name, which imports the predefined `META-LEVEL` module. For each module name `I` in the list given as parameter of the expression, the declaration of a constant `I` of sort `Module` and an equation identifying such constant with the metarepresentation of the module with such name in the database are added to the module being created.

```
op prepareMetalevelAux : ViewExp Unit Database -> Database .

eq prepareMetalevel(VE, DB)
  = prepareMetalevelAux(VE,
      addImportList((including 'META-LEVEL .),
        setName(emptyStrFModule, _<_>('META-LEVEL, VE))), DB) .

eq prepareMetalevelAux(_|_(QI, VE), U, DB)
  = if QI inModNameSet builtIns
    then prepareMetalevelAux(VE, U,
           warning(DB,
             ('WARNING: 'META-LEVEL '`[ QI '`] 'not 'supported)))
    else prepareMetalevelAux(VE,
           addOpDeclSet((op QI : nil -> 'Module [none] .),
```

```
            addEquationSet(
              (eq {QI}'Module = up(getFlatUnit(QI, DB)) .), U)),
          DB)
    fi .
  eq prepareMetalevelAux(nullViewExp, U, DB) = evalUnit(U, DB) .
```

The function `labelInParameterList` checks whether the quoted identifier given as first argument is used as a label in the list of parameters given as second argument.

```
  op labelInParameterList : Qid ParameterList -> Bool .
```

The function `classDeclSetESortSet` returns the set of the names of the classes in a set of class declarations.

```
  op classDeclSetESortSet : ClassDeclSet -> ESortSet .
```

The following 'getTheory' functions return the corresponding elements in the theory part of the structure of the given unit. For example, the function `getTheorySortSet` returns the set of sorts declared in the "loose part" of the structure of the unit in the database having the name indicated as first argument.

```
  op getTheorySortSet : ModName Database -> ESortSet .
  op getTheoryClassSet : ModName Database -> ESortSet .
```

The 'get' functions return the corresponding elements in the structure of the given unit. For example, `getSortSet` returns all the sorts declared in the structure of the unit in the database having the name given as first argument.

```
  op getSortSet : ModName Database -> ESortSet .
  op getClassSet : ModName Database -> ESortSet .
```

As pointed out in Section 3.6.2, in a parameterized module all occurrences of sorts or classes coming from the parameter theories have to be qualified. `createCopy` is the function used for creating these renamed copies of the parameters. As also explained in Section 3.6.2, if a parameter theory is structured, the renaming is carried out not only at the top level, but for the entire "theory part" in the structure.

The function `createCopy` calls an auxiliary function, `prepareParameter`, which recursively proceeds through all the subtheories of the given theory. For each theory in the structure, the required set of maps is generated and applied to such a theory using the `applyMapSetToUnit` function discussed in Section 4.6.8, which is then evaluated and entered into the database. Note that the renamings to which a theory is subjected must also be applied to the theories importing it. The new database and the renaming maps applied to the theory will have to be returned by the function.

The function `prepareParameter` makes a copy of the theory specified by the name given as first argument and of all its subtheories (only theories, no modules), and qualifies all the sorts appearing in it with the label given in the declaration of the parameter, which is given as second argument.

```
  op createCopy : ModExp Parameter Database -> Database .
  op prepareParameter : ModName Qid Database -> Database .
```

```
eq createCopy(ME, (par X :: ME'), DB)
  = if ME =/= ME' or-else unitInDatabase((par X :: ME), DB)
    then DB
    else prepareParameter(ME, X, DB)
    fi .
```

When one of the labels of the interface of a module is being used in a module expression to instantiate some formal parameter of a module, then, in the evaluation of such module expression the qualification of all sorts and class names coming from the theory part of the parameter theory have to be changed according to such a label. In the evaluation of an instantiation module expression this is done by generating the corresponding renaming maps, which are then applied to the module being instantiated. Given labels `L` and `L'`, for each sort or class name `S` in the set given as argument, a map of the form `S.L to S.L'` is generated.

Since for the time being we do not handle parameterized theories, we assume that there can only be non-qualified sorts and classes in theories.

```
op generateMapSetQualifiedSortSet : Qid Qid QidSet -> MapSet .
op generateMapSetQualifiedClassSet : Qid Qid QidSet -> MapSet .
```

The function `prepareViewMapSet` takes the map set of a view and prepares it to be used in an instantiation by transforming sort and class names into their qualified form, if required (sorts and class names in a view have to be qualified only if they were defined in a theory).

The `prepareViewMapSet` function takes five arguments: The set of maps to be prepared, the label with which the sorts to be renamed have to be qualified, the set of sorts in the theory part of the source of the view, and the set of sorts and class names in the theory part of the target of the view.

Note that we assume that there is a sort map and a class map for each sort and class in the theory part of the source of the view. Therefore, sorts and class names appearing as sources of sort and class maps are systematically qualified. The sorts or class names used in the targets of the maps will be qualified only if they were declared in a theory. In maps for operators in which the arity and coarity are specified, or for those going to derived terms, the sorts appearing in the arity or coarity of an operator and those used to qualify terms, or in sort tests in terms, must also be qualified. However, in these cases the qualification cannot be done on all sorts, but only on those defined in the theory parts. This is the reason why the sets of sorts in the theory parts of the source and target and the set of class names in the target of the view are given when calling `prepareViewMapSet`.

```
op prepareViewMapSet :
     ViewMapSet Qid ESortSet ESortSet ESortSet -> ViewMapSet .
```

For each parameterized sort $\texttt{S[L}_1\texttt{,}\ldots\texttt{,L}_n\texttt{]}$ in the body of a parameterized module with $\texttt{L}_1 \ldots \texttt{L}_n$ the labels of the parameters in the interface of the module, a map of the form $\texttt{sort S[L}_1\texttt{,}\ldots\texttt{,L}_n\texttt{] to S[V}_1\texttt{,}\ldots\texttt{,V}_n\texttt{]}$ is generated, where $\texttt{V}_i$ is the name of the view associated to the label $\texttt{L}_i$ in the set of pairs given as argument.

As we shall see below, in the instantiation process a set of pairs is created. Each of these pairs is composed of the label of a parameter theory and the view expression with which it is being instantiated. The sorts `QidViewExp`, `QidViewExpPair`, and `QidViewExpPairSet` are defined with their respective constructors. This set of pairs is used by the functions `generateMapSetSortSet` and `generateMapSetClassSet` to generate the maps mentioned above.

```
sort QidViewExpPair .
op <_,_> : Qid ViewExp -> QidViewExpPair .
op qid : QidViewExpPair -> Qid .
op viewExp : QidViewExpPair -> ViewExp .
eq qid(< QI , VE >) = QI .
eq viewExp(< QI , VE >) = VE .

sort QidViewExpPairSet .
subsort QidViewExpPair < QidViewExpPairSet .
op none : -> QidViewExpPairSet .
op __ : QidViewExpPairSet QidViewExpPairSet -> QidViewExpPairSet
       [assoc comm id: none] .

vars QIVEPS QIVEPS' : QidViewExpPairSet .

op generateMapSetSortSet : ESortSet QidViewExpPairSet -> MapSet .
op generateMapSetClassSet : ESortSet QidViewExpPairSet -> MapSet .
```

The function prepareImportList takes a list of importation declarations and a set of pairs composed of a label and a view name, and returns the list of importations resulting from changing in each of the module expressions the occurrences of the labels of the interface of the module being instantiated by the names of the views associated to them in the list of pairs.

```
op prepareImportList : EImportList QidViewExpPairSet -> EImportList .
op prepareModName : ModName QidViewExpPairSet -> ModName .
op prepareModNameAux : ModName ViewExp ViewExp ViewExp
      QidViewExpPairSet QidViewExpPairSet -> ModName .

eq prepareImportList(((including MN .) EIL), QIVEPS)
  = ((including prepareModName(MN, QIVEPS) .)
     prepareImportList(EIL, QIVEPS)) .
eq prepareImportList(nil, QIVEPS) = nil .

eq prepareModName(_<_>(ME, VE), QIVEPS)
  = prepareModNameAux(ME, VE, nullViewExp, nullViewExp, QIVEPS, QIVEPS) .
eq prepareModName(QI, QIVEPS) = QI .
eq prepareModName((par X :: ME), QIVEPS) = (par X :: ME) .

eq prepareModNameAux(
     ME, _|_(X, VE), VE', VE'', (< Y , Z > QIVEPS), QIVEPS')
  = if X == Y
    then prepareModNameAux(ME, VE, _|_(VE', Z), VE'', QIVEPS', QIVEPS')
    else prepareModNameAux(ME, _|_(X, VE), VE', VE'', QIVEPS, QIVEPS')
    fi .
eq prepareModNameAux(
      ME, _|_(X, VE), VE', VE'', (< Y , _;;_(W, Z) > QIVEPS), QIVEPS')
  = if X == Y
    then prepareModNameAux(
           ME, VE, _|_(VE', W), _|_(VE'', Z), QIVEPS', QIVEPS')
    else prepareModNameAux(ME, _|_(X, VE), VE', VE'', QIVEPS, QIVEPS')
    fi .
eq prepareModNameAux(ME, _|_(X, VE), VE', VE'', none, QIVEPS')
  = prepareModNameAux(ME, VE, _|_(VE', X), VE'', QIVEPS', QIVEPS') .
eq prepareModNameAux(ME, nullViewExp, VE, VE', QIVEPS, QIVEPS')
  = if VE' == nullViewExp
    then _<_>(ME, VE)
```

```
    else _<_>(_<_>(ME, VE), VE')
    fi .
```

The function `instantiation` calls the auxiliary function `instantiationAux`, which proceeds recursively on each of the parameters in the interface of the module being instantiated. For each view, a set of maps to be applied to the module is generated, which are accumulated in the third argument of the function.

In the base case, when there are no more parameters and no more views, the maps for the parameterized sorts are also generated, and all maps are then applied.

`instantiationAux` proceeds accumulating also the list of parameters being modified, the list of importations, and a list of label-view pairs (`QidViewExpPair`) associating each label in the interface to the view used in the instantiation of the theory with such label. This list of pairs is used to generate the set of maps of the parameterized sorts and to 'prepare' the list of importations as indicated above.

```
op instantiationAux : Unit Unit ViewMapSet ParameterList ParameterList
      EImportList EImportList ViewExp QidViewExpPairSet ParameterList
      Database -> Database .

eq instantiation(ME, VE, PL, DB)
  = instantiationAux(setName(getTopUnit(ME, DB), _<_>(ME, VE)),
      getSignature(ME, DB), none,
      parameterList(getTopUnit(ME, DB)), nilParameterList,
      importList(getTopUnit(ME, DB)), nil, VE, none, PL, DB) .

eq instantiationAux(U, M, VMAPS,
     parameterList((par X :: ME), PL), PL',
     (EIL (including (par X :: ME) .) EIL'), EIL'',
     _|_(QI, VE), QIVEPS, PL'', DB)
  = if labelInParameterList(QI, PL'')
    then instantiationAux(U, M,
           (VMAPS,
            generateMapSetQualifiedSortSet(X, QI,
              getTheorySortSet(ME, DB)),
            generateMapSetQualifiedClassSet(X, QI,
              getTheoryClassSet(ME, DB))),
           PL, parameterList(PL', (par QI :: ME)), (EIL EIL'),
           (EIL'' (including (par QI :: ME) .)),
           VE, (< X , QI > QIVEPS), PL'',
           createCopy(ME, (par QI :: ME), DB))
    else if viewInDatabase(QI, DB)
         then if getTopUnit(target(getView(QI, DB)), DB) : StrTheory
              then instantiationAux(U, M,
                     (VMAPS,
                      prepareViewMapSet(
                        mapSet(getView(QI, DB)), X,
                        getTheorySortSet(ME, DB),
                        getTheorySortSet(target(getView(QI, DB)), DB),
                        getTheoryClassSet(target(getView(QI, DB)), DB))),
                     PL,
                     parameterList(PL', (par X :: target(getView(QI, DB)))),
                     (EIL EIL'),
                     (EIL''
                      (including (par X :: target(getView(QI, DB))) .)),
                     VE,
```

```
                      (< X , _;;_(QI, X) > QIVEPS),
                      PL'',
                      createCopy(target(getView(QI, DB)),
                        (par X :: target(getView(QI, DB))), DB))
               else instantiationAux(U, M,
                      (VMAPS,
                       prepareViewMapSet(
                         mapSet(getView(QI, DB)), X,
                         getTheorySortSet(ME, DB), none, none)),
                      PL,
                      parameterList(
                        parameterList(
                          getTopUnit(target(getView(QI, DB)), DB)),
                        PL'),
                      (EIL EIL'),
                      (EIL'' (including target(getView(QI, DB)) .)),
                      VE, (< X , QI > QIVEPS),
                      PL'', DB)
               fi
          else warning(DB, ('\n 'ERROR: 'view QI 'not 'in 'database))
          fi
     fi .
  eq instantiationAux(U, M, VMAPS, nilParameterList, PL, EIL, EIL',
       nullViewExp, QIVEPS, PL', DB)
    = evalUnit(
        setImportList(
          setParameterList(
            applyMapSetToUnit(
              (VMAPS,
               generateMapSetSortSet(
                 (sortSet(sortDecl(U)) ;
                  getSortSetAux(importList(U), DB)), QIVEPS),
               generateMapSetClassSet(
                 (classDeclSetESortSet(classDeclSet(U)) ;
                  getClassSetAux(importList(U), DB)), QIVEPS)),
              U, M),
            PL),
          (prepareImportList(EIL, QIVEPS) EIL')),
        DB) .
  eq instantiationAux(
       unitError(QIL), U, VMAPS, PL, PL', EIL, EIL', VE, QIVEPS, PL'', DB)
    = warning(DB, QIL) .
  eq instantiationAux(U, M, VMAPS, parameterList((par X :: ME), PL),
       PL', EIL, EIL', nullViewExp, QIVEPS, PL'', DB)
  = warning(DB, ('\n 'ERROR: 'incorrect 'module modNameToQidList(name(U)))) .
  eq instantiationAux(U, M, VMAPS, nilParameterList, PL, EIL, EIL',
       _|_(QI, VE), QIVEPS, PL', DB)
  = warning(DB, ('\n 'ERROR: 'incorrect 'module modNameToQidList(name(U)))) .
```

As pointed out in Section 4.3.4, for each new module expression constructor being introduced, we need to add equations for the operator `modNameToQid`. Since the function to transform view expressions into lists of quoted identifiers was already defined in Section 4.3.2, we just need to add the following equation.

```
  eq modNameToQidList(_<_>(ME, VE))
    = (modNameToQidList(ME) '`[ viewExpToQidList(VE) '`]) .
```

Similarly, we need to give equations for the `setUpModExpDependencies` function, presented in Section 4.5.2, on the instantiation module expression. Given a module expression of the form `ME < VE >` such that `ME` is in the database, we need to add `ME < VE >` to the set of names of the modules depending on `ME` and on `VE`. Since `VE` may be a composed view expression, we have to add the name of the module to each of the views in it. In this way, if `ME` or any of the views in `VE` is redefined or removed from the database, `ME < VE >` will be removed as well.

```
  eq setUpModExpDependencies(_<_>(ME, VE),
       database(
         (< ME ; U ; U' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
    = viewExpDependencies(_<_>(ME, VE), VE,
        database(
          (< ME ; U ; U' ; M ; M' ; MNS & _<_>(ME, VE) ; VES > IS),
          MNS', VES', QIL)) .
  ceq setUpModExpDependencies(_<_>(ME, VE), DB)
    = if ME inModNameSet builtIns
      then DB
      else warning(DB,
             ('\n 'ERROR:
              'module modNameToQidList(ME) 'not 'in 'database '.))
      fi
      if not unitInDatabase(ME, DB) .

  op viewExpDependencies : ModName ViewExp Database -> Database .

  ceq viewExpDependencies(MN, _|_(QI, VE),
        database((< QI ; VI ; MNS > IS), MNS', VES, QIL))
    = viewExpDependencies(MN, VE,
         database((< QI ; VI ; MNS & MN > IS), MNS', VES, QIL))
      if QI =/= nullViewExp .
  ceq viewExpDependencies(MN, _|_(QI, VE), DB)
    = DB
      if not viewInDatabase(QI, DB) .
  eq viewExpDependencies(MN, nullViewExp, DB)
    = DB .

   ...

endfm
```

4.6.10 Renaming of Units

In addition to the declaration of the constructor for renaming module expressions, the following module `RENAMING-EXPR` introduces equations to treat this new case in the definition of functions `evalModExp`, `modNameToQidList`, `prepareModName`, and `setUpUnitDependencies`.

A renaming expression is evaluated by applying the renaming maps, not only to the top unit, but also to the part of the structure *affected* by the maps. The renaming process propagates downwards in the unit hierarchy while the units in the structure are affected by the renamings. We say that a unit is affected by a set of maps (checked by the `moduleAffected` function) when any of the maps is applicable to any of the declarations in the unit, or in any of its subunits. The application of a set of maps to a single unit is accomplished by the `applyMapSetToUnit` function, discussed in Section 4.6.8.

```
fmod RENAMING-EXPR is
  pr DATABASE .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  pr EVALUATION .
  inc MOD-EXPR .
  pr MOD-EXPR-EVAL .
  pr INST-EXPR .
  pr MAP .
  pr DECL-EXT-SORT-TO-QID .

  op _*<_> : ModExp MapSet -> ModExp .

  vars ME ME' : ModExp .
  var MNS MNS' : ModNameSet .
  vars M M' : Module .
  vars U U' : Unit .
  var DB : Database .
  vars ES ES' : ESort .
  var QIL : QidList .
  vars VES VES' : ViewExpSet .
  var IS : InfoSet .
  var PL : ParameterList .
  vars EIL EIL' : EImportList .
  var VMAP : ViewMap .
  var VMAPS : ViewMapSet .
  var QIVEPS : QidViewExpPairSet .
```

The function `createCopyRenaming` creates a copy of the part of the structure of the specified module which is affected by the renaming, applying to each of the generated modules in the new structure the subset of maps affecting each one of them. The equation extending the `evalModExp` function to the renaming module expression is then reduced to a call to `createCopyRenaming` with the appropriate arguments.

```
  op createCopyRenaming : ModExp ModExp Unit Database -> Database .

  ceq evalModExp(_*<_>(ME, VMAPS), PL, DB)
    = createCopyRenaming(ME, _*<_>(ME, VMAPS),
        getSignature(ME, evalModExp(ME, PL, DB)),
        evalModExp(ME, PL, DB))
      if not unitInDatabase(_*<_>(ME, VMAPS), DB) .
```

The predicate `moduleAffected` checks whether the module with the name given as first argument in the database is affected by the set of maps given as second argument. A module is affected by a map set if any of the maps is applicable to the module or to any of its submodules.

```
  op moduleAffected : ModName ViewMapSet Module Database -> Bool .
```

The function `mapSetRestriction` returns the subset of the view maps given as second argument that affect the given module.

```
  op mapSetRestriction : Unit ViewMapSet Module Database -> ViewMapSet .

  eq mapSetRestriction(U, (VMAP, VMAPS), M, DB)
    = if moduleAffectedAux(U, VMAP, M, DB)
      then (VMAP, mapSetRestriction(U, VMAPS, M, DB))
```

```
    else mapSetRestriction(U, VMAPS, M, DB)
    fi .
eq mapSetRestriction(U, none, M, DB) = none .
```

We proceed downwards while the set of maps affects the module, but we do so restricting the set of maps to the subset affecting the module. Since operator and message maps in which arity and coarity are specified must be applied to the whole subsort-overloaded family of operators or messages, we have to carry along the signature of the module at the top to make all the calls to the engine. Note that we may have maps of operations or messages with the domain given by sorts that are not in the submodules but which have other sorts in the submodules in the same connected components.

```
op applyMapSetRecursively :
      MapSet EImportList EImportList Unit Module Database -> Database .

eq createCopyRenaming(ME, _*<_>(ME, VMAPS), M, DB)
  = if not unitInDatabase(_*<_>(ME, VMAPS), DB)
    then applyMapSetRecursively(
           VMAPS, importList(getTopUnit(ME, DB)), nil,
           setName(applyMapSetToUnit(VMAPS, getTopUnit(ME, DB), M),
             _*<_>(ME, VMAPS)),
           M, DB)
    else DB
    fi .
eq createCopyRenaming(ME, ME', unitError(QIL), DB) = warning(DB, QIL) .

eq applyMapSetRecursively(
     VMAPS, ((including MN .) EIL), EIL', U, M, DB)
 = if MN inModNameSet builtIns or-else
      not moduleAffectedAux(getTopUnit(MN, DB), VMAPS,
            setVarDeclSet(M,
              eSortToQid(varDeclSet(getTopUnit(MN, DB)))),
            DB)
   then applyMapSetRecursively(
          VMAPS, EIL, ((including MN .) EIL'), U, M, DB)
   else applyMapSetRecursively(
          VMAPS, EIL,
          ((including
              (MN *< mapSetRestriction(getTopUnit(MN, DB), VMAPS,
                       setVarDeclSet(M,
                         eSortToQid(
                           varDeclSet(getTopUnit(MN, DB)))), DB) >) .)
           EIL'), U, M,
          createCopyRenaming(MN,
            (MN *< mapSetRestriction(getTopUnit(MN, DB), VMAPS,
                     setVarDeclSet(M,
                       eSortToQid(
                         varDeclSet(getTopUnit(MN, DB)))), DB) >),
            setVarDeclSet(M,
              eSortToQid(varDeclSet(getTopUnit(MN, DB)))),
            DB))
    fi .
eq applyMapSetRecursively(VMAPS, nil, EIL, U, M, DB)
  = evalUnit(setImportList(U, EIL), DB) .
eq applyMapSetRecursively(VMAPS, EIL, EIL', unitError(QIL), M, DB)
  = warning(DB, QIL) .
```

The definition of the function `modNameToQidList` on the renaming module expression has to take care of transforming into a quoted identifier list the set of view maps given in the module expression.

```
  op mapSetToQidList : MapSet -> QidList .

  eq modNameToQidList(_*<_>(ME, MAPS))
    = ((modNameToQidList(ME) '* '`( mapSetToQidList(MAPS) '`))) .
```

Let us now give the equations for `setUpModExpDependencies` on the renaming module expression. Given a module expression of the form `ME *< VMAPS >` such that `ME` is in the database, we just need to add `ME *< VMAPS >` to the set of names of the modules depending on `ME`. In this way, if `ME` is redefined or removed from the database, `ME *< VMAPS >` will be removed as well.

```
  eq setUpModExpDependencies(_*<_>(ME, VMAPS),
       database((< ME ; U ; U' ; M ; M' ; MNS ; VES > IS), MNS', VES', QIL))
    = database(
        (< ME ; U ; U' ; M ; M' ; MNS & _*<_>(ME, VMAPS) ; VES > IS),
        MNS', VES', QIL) .
  ceq setUpModExpDependencies(_*<_>(ME, VMAPS), DB)
    = warning(DB,
        ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
      if not unitInDatabase(ME, DB) .
```

The definition of the `prepareModName` function on a renaming module expression must take into account the possibility of having parameterized sorts or parameterized class names in the maps of a renaming module expression. The preparation of a renaming module expression must take into account this fact and prepare accordingly all parameterized sorts and classes appearing in it.

```
  op prepareMapSet : ViewMapSet QidViewExpPairSet -> ViewMapSet .

  eq prepareModName(_*<_>(ME, VMAPS), QIVEPS)
    = _*<_>(prepareModName(ME, QIVEPS), prepareMapSet(VMAPS, QIVEPS)) .
```

For example, for sort maps the equation is as follows.

```
  eq prepareMapSet(((sort ES to ES'), VMAPS), QIVEPS)
    = ((sort prepareESort(ES, QIVEPS) to prepareESort(ES', QIVEPS)),
       prepareMapSet(VMAPS, QIVEPS)) .

   ...

endfm
```

4.7 The Evaluation of Views

Before being entered into the database, besides containing bubbles, views have a somewhat different structure from that of the views given in Section 4.4.3. We introduce in the following module a sort `PreView` with constructor `view`, which is declared as the constructor for views of sort `View`, but with an additional argument, namely, a set of variable declarations to hold the declarations of variables in the view. During the processing of views (see Section 4.8.3), which takes place once the parsing process has concluded, these variables are associated with the corresponding maps where they are used, generating a term of sort `View`.

We start by introducing in the following module `PRE-VIEW-MAP` the sorts `TermPreMap`, `PreViewMap`, and `PreViewMapSet`. A preview map is a view map with bubbles. Note that the bubbles can only appear in term maps. Elements of sort `TermPreMap` are built with the constructor `preTermMap`, which takes two terms of sort `Term`, that is, two bubbles. In the processing of views (see Section 4.8.3), elements of sort `PreTermMap` will be converted into elements of sort `TermMap` by parsing the bubbles in them, and by associating to them the variables in them defined in the view in which the maps appear.

```
fmod PRE-VIEW-MAP is
  pr VIEW-MAP .

  sort TermPreMap .
  op preTermMap : Term Term -> TermPreMap .

  sorts PreViewMap PreViewMapSet .
  subsorts Map TermPreMap < PreViewMap .
  subsorts PreViewMap ViewMapSet < PreViewMapSet .
  op _,_ : PreViewMapSet PreViewMapSet -> PreViewMapSet
        [assoc comm id: none] .
```

Given a set of maps, the function `sortMapSet` returns the subset of sort maps in it.

```
  op sortMapSet : PreViewMapSet -> MapSet .

   ...

endfm

fmod PRE-VIEW is
  pr UNIT .
  pr PRE-VIEW-MAP .

  sort PreView .
  op view : ViewExp ParameterList ModExp ModExp EVarDeclSet PreViewMapSet
          -> PreView .

  op name : PreView -> ViewExp .
  op source : PreView -> ModExp .
  op target : PreView -> ModExp .
  op varDeclSet : PreView -> EVarDeclSet .
  op mapSet : PreView -> PreViewMapSet .

  var QI : Qid .
  vars ME ME' : ModExp .
```

The following functions can be used to add new declarations to the set of declarations already in a preview.

```
  op addMapSet : PreViewMapSet PreView -> PreView .
  op addVarDeclSet : EVarDeclSet PreView -> PreView .

  op emptyPreView : Qid ModExp ModExp -> PreView .
  eq emptyPreView(QI, ME, ME')
    = view(QI, nilParameterList, ME, ME', none, none) .
```

```
  ...

endfm
```

The function solveBubbles defined in the following VIEW-BUBBLE-PARSING module parses the bubbles in a set of preview maps. It takes two modules, the signature of the view's source theory, with the variables declared in the view, to parse the source term in the term maps, and the target theory, with the mappings of the variable declarations in the view, to parse the target terms.

```
fmod VIEW-BUBBLE-PARSING is
  pr BUBBLE-PARSING .
  pr PRE-VIEW .

  op solveBubbles : PreViewMapSet EVarDeclSet Unit Unit -> ViewMapSet .

   ...

endfm
```

4.8 Input/Output Processing

In this section we discuss how the preterm resulting from the call to the function meta-parse with the input and the top-level signature of Full Maude is transformed into a term of sort Unit, representing a preunit or a term of sort PreView. In the case of commands, they are evaluated giving the corresponding results in the appropriate form.

4.8.1 Input Parsing

Let us recall here the example presented in Section 1.3.6. Calling meta-parse with the module NAT3 given there and the signature of Full Maude presented in Section 4.2, we obtain the following term.

```
'fmod_is_endfm[
   'token[{''NAT3}'Qid],
   '__['sort_.['token[{''Nat3}'Qid]],
       '__['op_:_->_.['token[{''s_}'Qid],
                      'neTokenList[{''Nat3}'Qid],
                      'token[{''Nat3}'Qid]],
           '__['op_:`->_.['token[{''0}'Qid],
                          'token[{''Nat3}'Qid]],
               'eq_=_.['bubble['__[{''s}'Qid, {''s}'Qid,
                                   {''s}'Qid, {''0}'Qid]],
                       'bubble[{''0}'Qid]]]]]]
```

Given each one of the subterms representing declarations in terms representing modules as the previous one, the function parseDecl generates the corresponding declaration, with no bubbles in it, and the corresponding predeclaration, with the bubbles appearing in the term. For example, for the term

```
'op_:_->_.['token[{''s_}'Qid],
           'neTokenList[{''Nat3}'Qid],
           'token[{''Nat3}'Qid]]
```

the following operator declaration is generated:

```
  op 's_ : 'Nat3 -> 'Nat3 [none] .
```

Note that in this case, since the operator is declared without identity element (the only place a bubble might appear), the declaration and the predeclaration generated by `parseDecl` coincide.

In the following sections we shall see how this approach is followed for declarations appearing in units and in views.

Parsing of Unit Declarations

The `parseDecl` function takes a term (which corresponds to a declaration that has to be parsed), a preunit (to which the parsed declaration with its bubbles in it will be added), and a unit (to which the parsed declaration without bubbles will be added to build up the signature). For example, a term corresponding to an unconditional equation, that is, a term of the form `'eq_=_.[T, T']` will be added to the set of equations of the preunit as `eq T = T' .`, but nothing will be added to the unit. Note that according to the signature used in the call to `meta-parse` (see Sections 4.2 and 4.9.2), `T` and `T'` are bubbles. Declarations of sorts, subsort relations, operators, classes, subclass relations, messages, and variables will be added to both of them. In the case of operator declarations, identity element attributes, which in general can be terms, are not included in the added declaration. As in Core Maude, declarations in a module can be given in any order, and therefore we follow a two-step approach consisting in first building the signature to parse the bubbles, and then generating the unit without bubbles in it. It could be different for other languages. For example, in some languages we may be able to assume that each operator and sort has been defined before being used, allowing then an incremental processing of the input.

```
fmod UNIT-DECL-PARSING is
  pr DATABASE .
  pr MOVE-DOWN .
  pr INST-EXPR .
  pr RENAMING-EXPR .

  vars PU U : Unit .
  vars T T' : Term .
```

Before introducing the `parseDecl` function, we present some auxiliary functions. For example, the following functions `parseSort`, `parseSortSet`, and `parseSortList` return, respectively, the sort, set of sorts, and list of sorts represented by the term given as argument. Note that these functions, as most of the functions in this module, are partial functions. We assume that the term given as argument is in fact the representation of, for example, a valid sort, or set of sorts, etc. In the case of `parseDecl` we assume that the term is the representation of a predeclaration.

```
  op parseSort : Term -> ESort .
  op parseSortSet : Term -> ESortSet .
  op parseSortList : Term -> ESortList .
  op parseViewExp : Term -> ViewExp .

  eq parseSort('sortToken[T]) = downQid(T) .
  eq parseSort('_`[_`][T, T'])
    = eSort(parseSort(T), parseViewExp(T')) .
```

```
eq parseSortSet('__[T, T'])
  = (parseSortSet(T) ; parseSortSet(T')) .
eq parseSortSet('sortToken[T]) = parseSort('sortToken[T]) .
eq parseSortSet('_`[_`][T, T']) = parseSort('_`[_`][T, T']) .

eq parseSortList('__[T, T'])
  = (parseSortList(T) parseSortList(T')) .
eq parseSortList('sortToken[T]) = parseSort('sortToken[T]) .
eq parseSortList('_`[_`][T, T']) = parseSort('_`[_`][T, T']) .

eq parseViewExp('viewToken[T]) = downQid(T) .
eq parseViewExp('_`,_[T, T'])
  = _|_(parseViewExp(T), parseViewExp(T')) .
```

The function parseModExp takes a term representing a module expression and returns the corresponding term in sort ModExp. In case of adding new constructors for module expressions, as it will be done in Section 4.10, new equations defining the semantics of the function on them will have to be given.

```
op parseModExp : Term -> ModExp .
op parseMapSet : Term -> MapSet .

eq parseModExp('token[T]) = downQid(T) .
eq parseModExp('`(_`)[T]) = parseModExp(T) .
eq parseModExp('_`[_`][T, T'])
  = _<_>(parseModExp(T), parseViewExp(T')) .
eq parseModExp('_*`(_`)[T, T'])
  = _*<_>(parseModExp(T), parseMapSet(T')) .
```

Given a term representing a machine integer, the function parseMachineInt returns the corresponding integer.

```
op parseMachineInt : Term -> MachineInt .
```

Given a term representing a declaration or a predeclaration, the function parseDecl must generate and update both the unit and the preunit that it takes as arguments. Note that in the case of rules, for example, only a prerule is generated.

Since the preunit and the unit may be modified, they have to be returned as a pair, which will be used to extract the corresponding arguments for the following calls. Note that the parseDecl functions are in fact partial functions. Each parsing function assumes that it is possible to parse the given term.

We include here only some of the equations needed.

```
sort parseDeclResult .
op <_;_> : Unit Unit -> parseDeclResult .
op preUnit : parseDeclResult -> Unit .
op unit : parseDeclResult -> Unit .

eq preUnit(< PU ; U >) = PU .
eq unit(< PU ; U >) = U .

op parseDecl : Term Unit Unit -> parseDeclResult .
```

In the case of importation declarations, since internally only the including mode is handled, all importations are generated in this mode, independently of the keyword used in the input.

```
  eq parseDecl('including_.[T], PU, U)
    = < addImportList((including parseModExp(T) .), PU) ; U > .
  eq parseDecl('protecting_.[T], PU, U)
    = < addImportList((including parseModExp(T) .), PU) ; U > .

  eq parseDecl('sorts_.[T], PU, U)
    = < addSortSet(parseSortSet(T), PU) ;
        addSortSet(parseSortSet(T), U) > .

  eq parseDecl('eq_=_.[T, T'], PU, U)
    = < addEquationSet((eq T = T' .), PU) ; U > .
  eq parseDecl('ceq_=_if_.[T, T', T''], PU, U)
    = < addEquationSet((ceq T = T' if T'' = {'true}'Bool .), PU) ; U > .

   ...

endfm
```

Parsing of View Declarations

A similar approach is followed for the parsing of declarations in views.

```
fmod VIEW-DECL-PARSING is
  pr PRE-VIEW .
  pr VIEW .
  pr UNIT .
  pr UNIT-DECL-PARSING .

  vars T T' : Term .
  var EODS : EOpDeclSet .
  var MDS : MsgDeclSet .
  var M : Module .
```

Operator and message name maps of the form `F to F'` are substituted by an equivalent set of maps of the form `F : ESL -> ES to F'`. One of these maps is added for each family of subsort-overloaded operators in the source theory of the view.

The following functions `generateOpMapSet` and `generateMsgMapSet` take, respectively, an operator and a message map of the form `F to F'`, a set of operator or message declarations, and a term of sort `Module`, and return, respectively, a set of operator maps and a set of message maps, with each of the members of those sets having the general form `F : ESL -> ES to F'`. One of these maps is generated for each family of subsort-overloaded operators or messages with name `F` in the module given as argument.

```
  op generateOpMapSet : Map EOpDeclSet Module -> MapSet .
  op generateMsgMapSet : Map MsgDeclSet Module -> MapSet .
```

In the case of views, the `parseDecl` function takes the term representing the corresponding declaration and a preview in which the declarations are introduced. Note that in the case of views, the approach followed in the evaluation is somewhat different. The only predeclarations in a preview correspond to the term premaps of sort `PreTermMap`, for which, in addition to solving the bubbles in them, we have to convert them into term maps of sort `TermMap` associating to them the set of declarations of variables in the view which are used in them (see Section 4.8.3).

The function parseDecl for declarations in views takes then the term representing such declaration and a preview in which the result of adding the declaration will be returned. To be able to generate the sets of equivalent operator and message maps as indicated above, the function takes also as parameters the sets of declarations of operators and messages in the theory part of the source theory of the view in question, and the signature of such theory to make the necessary sort comparisons.

```
  op parseDecl : Term PreView EOpDeclSet MsgDeclSet Module -> PreView .

  eq parseDecl('op_to`term_.[T, T'], PV, EODS, MDS, M)
    = addMapSet(preTermMap(T, T'), PV) .

  eq parseDecl('sort_to_.[T, T'], PV, EODS, MDS, M)
    = addMapSet((sort parseSort(T) to parseSort(T')), PV) .

   ...

endfm
```

4.8.2 Meta Pretty Printing

To be able to show to the user the modules, theories, views, and terms resulting from the different commands, the built-in function meta-pretty-print is extended in the modules in this section to deal with units and views.

Meta Pretty Printing of Declarations

The predefined function meta-pretty-print is extended in the following module DECL-META-PRETTY-PRINT to handle any declaration that can appear in a unit. Note that the following meta-pretty-print functions, as the built-in one, return a list of quoted identifiers, although only for the declarations containing terms—such as equations, rules, operator declarations with an identity attribute, etc.—they have been defined with a term of sort Module as argument. In the other cases the module is not necessary.

```
fmod DECL-META-PRETTY-PRINT is
  pr EXT-DECL .
  pr O-O-DECL .
  pr EXT-SORT-TO-QID .

  op meta-pretty-print : ESort -> QidList .
  op meta-pretty-print : ESortSet -> QidList .
  op meta-pretty-print : ESortList -> QidList .
  op meta-pretty-print : ESubsortDeclSet -> QidList .
  op meta-pretty-print : ClassDeclSet -> QidList .
  op meta-pretty-print : SubclassDeclSet -> QidList .
  op meta-pretty-print : Module EOpDeclSet -> QidList .
  op meta-pretty-print : MsgDeclSet -> QidList .
  op meta-pretty-print : EVarDeclSet -> QidList .
  op meta-pretty-print : Module EMembAxSet -> QidList .
  op meta-pretty-print : Module EquationSet -> QidList .
  op meta-pretty-print : Module RuleSet -> QidList .

   ...

endfm
```

Meta Pretty Printing of Units

In the following module, the `meta-pretty-print` function is defined on sort `Unit`.

```
fmod UNIT-META-PRETTY-PRINT is
  pr UNIT .
  pr DECL-META-PRETTY-PRINT .

  op meta-pretty-print : Unit Unit -> QidList .

    ...
endfm
```

The function `meta-pretty-print` on units is defined recursively, calling the `meta-pretty-print` functions for the different declarations in the unit defined in module `DECL-META-PRETTY-PRINT`.

Meta Pretty Printing of Maps and Views

We define in the following module the function `meta-pretty-print` on maps.

```
fmod MAP-SET-META-PRETTY-PRINT is
  pr DECL-META-PRETTY-PRINT .
  pr MAP .
  pr UNIT .

  op meta-pretty-print : MapSet -> QidList .

    ...
endfm
```

Finally, in the `VIEW-META-PRETTY-PRINT` module, the `meta-pretty-print` function is defined on views.

```
fmod VIEW-META-PRETTY-PRINT is
  pr DATABASE .
  pr MAP-SET-META-PRETTY-PRINT .
  pr VIEW-MAP-SET-APPL-ON-UNIT .
  pr DECL-EXT-SORT-TO-QID .

  op meta-pretty-print : Database View -> QidList .

    ...
endfm
```

4.8.3 Input Processing

The processing functions presented in the following modules are in charge of taking each term generated by the `meta-parse` function and, after transforming it into an element of the data types `Unit` or `View`, or generating some output, returning the database resulting from introducing in it such a term. We shall see in Section 4.9.1 how the appropriate function is called after having performed a first analysis of the term, in which it is detected whether the input corresponds to a unit, view, or command. In the cases of units and views the processing is quite similar. After a preprocessing of the term, the function `parseDecl` is called with each of the subterms representing declarations, resulting in units or views with the parsed declarations in it.

Unit Processing

The processing of a term resulting from the parsing of some input corresponding to a unit is accomplished by the `processUnit` function. This function takes as arguments a term of sort `Term`, which represents some preunit, and a database. The function then enters into the given database the unit obtained from the transformation of such term into a term of sort `Unit`.

```
fmod UNIT-PROCESSING is
  pr DATABASE .
  pr UNIT-DECL-PARSING .
  pr EVALUATION .
  pr VIEW-MAP-SET-APPL-ON-UNIT .

  vars F X : Qid .
  vars PU U U' : Unit .
  var DB : Database .
  vars T T' T'' : Term .
  var TL : TermList .
  vars PL PL' : ParameterList .
  var EIL : EImportList .
  var ME : ModExp .
```

The `parseParameterList` takes a term representing a list of parameters and returns the corresponding list.

```
  op parseParameterList : Term -> ParameterList .
  eq parseParameterList('_::_['token[T], T'])
    = (par downQid(T) :: parseModExp(T')) .
  eq parseParameterList('_`,_[T, T'])
    = parameterList(parseParameterList(T), parseParameterList(T')) .
```

The function `processParameters` takes a preunit and a database and returns the resulting premodule and database, after processing each of the parameters in the parameter list of the preunit. For each of the parameters an importation declaration is added to the module, and the module expression used as parameter theory is evaluated in the database. The function returns a pair composed by a preunit and a database (`UnitDatabasePair`) as its result.

The function `processParametersAux` proceeds recursively on each of the parameters in the list of parameters of the given module. For each parameter of the form `par X :: ME`, the module expression `ME` is evaluated using the `evalModExp` function, which was discussed in Section 4.6.5, and then a copy of such a theory is created using the `createCopy` function, which applies to the unit the appropriate transformation, as explained in Section 4.6.9.

```
  sort UnitDatabasePair .
  op <_;_> : Unit Database -> UnitDatabasePair .
  op preUnit : UnitDatabasePair -> Unit .
  op database : UnitDatabasePair -> Database .

  eq database(< PU ; DB >) = DB .
  eq preUnit(< PU ; DB >) = PU .

  op processParameters : Unit Database -> UnitDatabasePair .
  op processParametersAux : Unit ParameterList ParameterList EImportList
       Database -> UnitDatabasePair .
```

```
eq processParameters(PU, DB)
  = processParametersAux(
      PU, parameterList(PU), parameterList(PU), nil, DB) .

eq processParametersAux(
     PU, PL, parameterList((par X :: ME), PL'), EIL, DB)
  = processParametersAux(
      PU, PL, PL', ((including (par X :: ME) .) EIL),
      createCopy(ME, (par X :: ME), evalModExp(ME, PL, DB))) .
eq processParametersAux(PU, PL, nilParameterList, EIL, DB)
  = < addImportList(EIL, PU) ; DB > .
```

All the operators declared as constructors of sort `PreUnit` in the signature of Full Maude, given in Appendix A, are declared with two arguments, namely, the name, or name and interface, of the unit, and the list of declarations of such units. The function `processUnit2` is called with the term corresponding to the name, or name and interface, of the module as first argument, the term corresponding to the set of declarations as second argument, and an empty module of the appropriate type, in which the different declarations will be accumulated, as third argument.

The task of the function `processUnit3` is then to make a second level parsing of the input, building up, simultaneously, the preunit represented in the term passed as argument, and the unit resulting from the declarations without bubbles. This unit without bubbles will be used by the `evalPreUnit` function to build the signature with which to analyze the bubbles in the preunit (see Section 4.6.7).

The case of parameterized modules requires a special treatment of the parameters. These parameters are evaluated and are added as submodules in the appropriate way.

When the last declaration is parsed, the function `evalPreUnit` is called with the preunit (the top module with bubbles) as first argument, the empty copy of it as second argument, the top module without bubbles as third argument, and the database.

Note that the `processUnit` function adds a declaration importing the module `CONFIGURATION`, presented in Section 4.5.1, to the object-oriented modules, and that `processUnit3` adds a declaration importing the built-in module `BOOL` to all modules.

```
op processUnit : Term Database -> Database .
op processUnit2 : Term Term Unit Database -> Database .
op processUnit3 : Term Unit Unit Database -> Database .

eq processUnit('fmod_is_endfm[T, T'], DB)
  = processUnit2(T, T', emptyStrFModule, DB) .
eq processUnit('mod_is_endm[T, T'], DB)
  = processUnit2(T, T', emptyStrSModule, DB) .
eq processUnit('omod_is_endom[T, T'], DB)
  = processUnit2(T, T',
      addImportList((including 'CONFIGURATION .), emptyStrOModule), DB) .
eq processUnit('fth_is_endfth[T, T'], DB)
  = processUnit2(T, T', emptyStrFTheory, DB) .
eq processUnit('th_is_endth[T, T'], DB)
  = processUnit2(T, T', emptyStrSTheory, DB) .
eq processUnit('oth_is_endoth[T, T'], DB)
  = processUnit2(T, T',
      addImportList((including 'CONFIGURATION .), emptyStrOTheory), DB) .
```

```
  eq processUnit2('token[T], T', U, DB)
    = if downQid(T) inModNameSet builtIns
      then warning(DB,
             ('ERROR: 'built-in 'modules 'can 'not 'be 'redefined))
      else processUnit3(T',
             setName(U, downQid(T)),
             setName(U, modNameToQid(downQid(T))), DB)
      fi .
  eq processUnit2('_`[_`]['token[T], T'], T'', U, DB)
    = if downQid(T) inModNameSet builtIns
      then warning(DB,
             ('ERROR: 'built-in 'modules 'can 'not 'be 'redefined))
      else processUnit3(T'',
             preUnit(
               processParameters(
                 setParameterList(setName(U, downQid(T)),
                   parseParameterList(T')), DB)),
             setName(U, modNameToQid(downQid(T))),
             database(
               processParameters(
                 setParameterList(setName(U, downQid(T)),
                   parseParameterList(T')), DB)))
      fi .

  eq processUnit3('__[T, T'], PU, U, DB)
    = processUnit3(T', preUnit(parseDecl(T, PU, U)),
        unit(parseDecl(T, PU, U)), DB) .
  ceq processUnit3(F[TL], PU, U, DB)
    = evalPreUnit(
        preUnit(
          parseDecl(
            F[TL], addImportList((including 'BOOL .), PU), U)),
        unit(
          parseDecl(
            F[TL], addImportList((including 'BOOL .), PU), U)),
        DB)
      if F =/= '__ .

endfm
```

View Processing

A similar process is followed for views. Note that in case of operator maps going to derived terms we have bubbles, which will have to be treated using the signatures of the appropriate modules.

```
fmod VIEW-PROCESSING is
  pr UNIT-PROCESSING .
  pr VIEW-DECL-PARSING .
  pr VIEW-BUBBLE-PARSING .

  var F : Qid .
  vars T T' T'' T''' : Term .
  var M : Module .
  var VE : ViewExp .
  var PV : PreView .
```

```
vars ME ME' : ModExp .
var DB : Database .
var EODS : EOpDeclSet .
var MDS : MsgDeclSet .
var TL : TermList .
```

As the functions `getTheorySortSet` and `getTheoryClassSet` presented in Section 4.6.9, the functions `getTheoryOpDeclSet` and `getTheoryMsgDeclSet` return, respectively, the set of declarations of operators, and the set of declarations of messages in the theory part of the structure of the module given as argument.

```
op getTheoryOpDeclSet : ModName Database -> EOpDeclSet .
op getTheoryMsgDeclSet : ModName Database -> MsgDeclSet .
```

The processing of terms representing previews accomplished by the function `processView` is quite similar to the one accomplished by `processUnit` on terms representing preunits. The algorithms followed are also quite similar. Both proceed recursively on the list of declarations, accumulating them in a preunit or in a preview.

The solving of bubbles in views requires the signatures of the source and target units extended, respectively, with the declarations of variables in the view and with the mappings of these declarations. As we shall see in Section 4.5.2, the signatures of the built-in modules are not accesible at the metalevel, and thus built-in modules cannot be used directly as arguments of built-in functions. Thus, to be able to use them as targets of views, a 'dummy' module is created importing the corresponding predefined module. The source and target module expressions of the view are evaluated before the view processing itself starts.

As we saw in Section 4.8.1, parsing of terms representing operator and message maps requires the set of operator and message declarations in the theory part of the source theory.

```
op processView : Term Database -> Database .
op processView : Term ViewExp ModExp ModExp Database -> Database .
op processViewAux :
      Term PreView EOpDeclSet MsgDeclSet Module Database -> Database .

eq processView(
     'view_from_to_is_endv['viewToken[T], T', T'', T'''], DB)
  = processView(
      T''', downQid(T), parseModExp(T'), parseModExp(T''), DB) .

eq processView(T, VE, ME, ME', DB)
  = processViewAux(T,
      emptyPreView(VE, ME, ME'),
      getTheoryOpDeclSet(ME, evalModExp(ME, DB)),
      getTheoryMsgDeclSet(ME, evalModExp(ME, DB)),
      getSignature(ME, evalModExp(ME, DB)),
      evalModExp(ME', evalModExp(ME, DB))) .

eq processViewAux('__[T, T'], PV, EODS, MDS, M, DB)
*** - EODS and MDS are, respectively, the set of operation and
***   message declarations in the theory part of the source.
*** - M is the signature of the source theory.
  = processViewAux(
      T', parseDecl(T, PV, EODS, MDS, M), EODS, MDS, M, DB) .
```

```
  ceq processViewAux(F[TL], PV, EODS, MDS, M, DB)
    = insertView(
        view(name(parseDecl(F[TL], PV, EODS, MDS, M)),
          nilParameterList,
          source(parseDecl(F[TL], PV, EODS, MDS, M)),
          target(parseDecl(F[TL], PV, EODS, MDS, M)),
          solveBubbles(
            mapSet(parseDecl(F[TL], PV, EODS, MDS, M)),
            varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)),
            addVarDeclSet(
              varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)), M),
            (if target(parseDecl(F[TL], PV, EODS, MDS, M))
                  inModNameSet builtIns
             then (fmod 'DUMMY is
                     including target(parseDecl(F[TL], PV, EODS, MDS, M)) .
                     sorts none .
                     none
                     none
                     applyMapSetToVarDeclSet(
                       sortMapSet(
                         mapSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                           varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M)))
                     none
                     none
                   endfm)
             else addVarDeclSet(
                    applyMapSetToVarDeclSet(
                      sortMapSet(mapSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                      varDeclSet(parseDecl(F[TL], PV, EODS, MDS, M))),
                    getSignature(target(parseDecl(F[TL], PV, EODS, MDS, M)),
                      DB))
             fi))),
        DB)
      if F =/= '__ .

   ...

endfm
```

Command Processing

The function `processCommand` only handles the `reduce`, `rewrite`, and `down` commands. The other commands are directly evaluated by the rules for the top-level handling of the database (see Section 4.9.1). The `processCommand` function takes a term, which represents one of these commands, the name of the default module, and a database. The result is a list of quoted identifiers representing the result of the evaluation of the command that will be placed in the read-eval-print loop to be printed in the terminal.

The `reduce` and `rewrite` commands are basically evaluated calling the built-in functions `meta-reduce` and `meta-rewrite`, respectively. These functions are called with the appropriate modules. In the case of commands in which an explicit module is not specified the default module is used.

The preparation of the output for these functions becomes more complex when the `down` command is used. To deal with the `down` command, an auxiliary function `processCommand2` is introduced, returning the term resulting from the evaluation of the command.

```
fmod COMMAND-PROCESSING is
  pr UNIT-PROCESSING .
  pr UNIT-META-PRETTY-PRINT .

  vars T T' T'' : Term .
  var DB : Database .
  var M : Module .
  var MN : ModName .

  op processCommand : Term ModExp Database -> QidList .
  op processReduce : ModExp Module Term Database -> QidList .
  op processReduce : ModExp Unit Term Database -> QidList .
  op processRewrite : ModExp Module Term MachineInt Database -> QidList .
  op processRewrite : ModExp Unit Term MachineInt Database -> QidList .
  op processCommand2 : ModExp Term Module Database -> Term .
  op processReduce2 : ModExp Module Term Database -> Term .
  op processRewrite2 : ModExp Module Term MachineInt Database -> Term .
```

To illustrate the processing of commands, we give here the equations needed for the handling of the `red_.` and `down_:_.` commands.

```
  eq processCommand('down_:_[T, T'], MN, DB)
    = if processCommand2(MN, T', getFlatUnit(MN, DB), DB) =/= error*
      then ('\n 'Result
            meta-pretty-print(
              leastSort(getFlatUnit(parseModExp(T), DB),
                downTerm(processCommand2(MN, T', getFlatUnit(MN, DB), DB))))
            ':
            meta-pretty-print(getFlatUnit(parseModExp(T), DB),
              downTerm(processCommand2(MN, T', getFlatUnit(MN, DB), DB))))
      else ('\n 'ERROR: 'Incorrect 'input '.)
      fi .
  eq processCommand('red_.['bubble[T]], MN, DB)
    = processReduce(MN, getFlatUnit(MN, DB), 'bubble[T], DB) .

  eq processCommand2(MN, 'down_:_[T, T'], M, DB)
    = downTerm(processCommand2(MN, T', M, DB)) .

  eq processCommand2(MN, 'red_.['bubble[T]], M, DB)
    = processReduce2(MN, M, 'bubble[T], DB) .

  eq processReduce(MN, M, T, DB)
    = if meta-reduce(M, solveBubbles(T, M, true, DB)) =/= error*
      then ('\n 'Reduce 'in meta-pretty-print(MN) ':
            meta-pretty-print(M, solveBubbles(T, M, true, DB)) '.
            '\n 'Result
            meta-pretty-print(
              leastSort(M, meta-reduce(M, solveBubbles(T, M, true, DB)))) ':
            meta-pretty-print(M,
               meta-reduce(M, solveBubbles(T, M, true, DB))))
      else ('ERROR: 'Incorrect 'Command '.)
      fi .

  eq processReduce2(MN, M, T, DB)
    = meta-reduce(M, solveBubbles(T, M, true, DB)) .

   ...

endfm
```

4.9 Interaction with the Persistent Database

In the case of Full Maude, the persistent state of the system is given by a single object which maintains the database of the system. This object has an attribute `db`, to keep the actual database in which all the modules being entered are stored, an attribute `default`, to keep the identifier of the current module by default, and attributes `input` and `output` to simplify the communication of the read-eval-print loop given by the `LOOP-MODE` module with the database. Using the notation for classes in object-oriented modules (see Section 3.3) we can declare the class `database` as follows:

```
class database | db : Database, input : TermList,
                 output : QidList, default : ModId .
```

Since we assume that `database` is the only object class that has been defined—so that the only objects of sort `Object` will belong to the `database` class—to specify the admissible states in the persistent state of `LOOP-MODE` for Full Maude, it is enough to give the subsort declaration

```
subsort Object < State .
```

4.9.1 Top Level Handling of the Persistent Database

Note that, since the Full Maude specification is given as a system module in Core Maude, object-oriented declarations cannot be given directly. Instead, the equivalent declarations desugaring the desired object-oriented declarations have to be specified. We use also the same conventions discussed in Section 3.3.2 regarding the use of variables instead of class names in the objects and in the addition of variables of sort `AttributeSet` to range over the additional attributes. As we shall see in Chapter 5, this convention will allow us to extend the Full Maude system in a very simple and clean way.

To allow the use of the object-oriented notation the predefined module `CONFIGURATION`, presented in Section 3.3.2, is included in the following module `DATABASE-HANDLING`.

```
mod DATABASE-HANDLING is
  inc META-LEVEL .
  inc CONFIGURATION .
  pr VIEW-META-PRETTY-PRINT .
  pr VIEW-PROCESSING .
  pr COMMAND-PROCESSING .

  var F : Qid .
  var QIL : QidList .
  vars T T' T'' T''' : Term .
  var TL : TermList .
  var DB : Database .
  var MN : ModName .
```

We start introducing a subsort `DatabaseClass` of sort `Cid`, the operator declarations necessary for representing objects in class `DatabaseClass` as defined above, and variables to range over subclasses of class `DatabaseClass` and over attributes.

```
  sort DatabaseClass .
  subsort DatabaseClass < Cid .
```

```
op Database : -> DatabaseClass .
op db :_ : Database -> Attribute .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .
op default :_ : ModName -> Attribute .

var Atts : AttributeSet .
var X@Database : DatabaseClass .
var O : Oid .
```

Next, we introduce an auxiliary function `parseModName` to parse names of user-defined modules, and a constant `nilTermList` of sort `TermList`. Note that the name of a user-defined module must be a single identifier (a token) or, for parameterized modules, its name—a single identifier—and its interface.

```
op parseModName : Term -> Qid .
eq parseModName('token[T]) = downQid(T) .
eq parseModName('_`[_`]['token[T], T']) = downQid(T) .

op nilTermList : -> TermList .

eq (nilTermList, TL) = TL .
eq (TL, nilTermList) = TL .
```

Finally, we present the rules processing the inputs of the database. These rules define the behavior of the system for the different commands, modules, theories, and views entered into the system. For example, the first rule processes the different types of modules entered to the system. Note that the operators declared as constructors of sort `PreUnit` in the signature of Full Maude, given in Appendix A, are declared with two arguments, namely the name of the unit, or its name plus its interface, and the list of declarations of such a unit.

```
crl [module] :
   < O : X@Database | db : DB, input : (F[T, T']),
      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : processUnit(F[T, T'], DB), input : nilTermList,
         output :
           ('\n 'Introduced 'module: modNameToQid(parseModName(T))),
         default : parseModName(T), Atts >
      if (F == 'fmod_is_endfm) or (F == 'mod_is_endm)
           or (F == 'omod_is_endom) .
```

Notice the message placed in the output channel, and the change in the current module by default, which is now the new module just processed. Since the name of the module `T` can be complex—a parameterized module—some extra parsing has to be performed by the auxiliary function `parseModName`. Similar rules are given for the processing of theories and views.

```
crl [theory] :
   < O : X@Database | db : DB, input : (F[T, T']),
      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : processUnit(F[T, T'], DB), input : nilTermList,
         output :
           ('\n 'Introduced 'theory: modNameToQid(parseModName(T))),
         default : parseModName(T), Atts >
```

```
        if (F == 'fth_is_endfth) or (F == 'th_is_endth)
              or (F == 'oth_is_endoth) .

  rl [view] :
     < O : X@Database | db : DB,
        input : ('view_from_to_is_endv['viewToken[T], T', T'', T''']),
        output : nil, default : MN, Atts >
     => < O : X@Database |
           db : processView(
                  'view_from_to_is_endv['viewToken[T], T', T'', T'''], DB),
           input : nilTermList,
           output : ('\n 'Introduced 'view: downQid(T)),
           default : MN, Atts > .
```

Commands are handled by rules as well. For example, the `down`, `reduce`, and `rewrite` commands are handled by the following rules.

```
  rl [down] :
     < O : X@Database | db : DB, input : ('down_:_[T, T']),
        output : nil, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : processCommand('down_:_[T, T'], MN, DB),
           default : MN, Atts > .

  crl [red/rew] :
     < O : X@Database | db : DB, input : (F[TL]),
        output : QIL, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : processCommand(F[TL], MN, DB),
           default : MN, Atts >
        if (F == 'red_.) or (F == 'reduce_.)
            or (F == 'red-in_:_.) or (F == 'reduce-in_:_.)
            or (F == 'rew_.) or (F == 'rewrite_.)
            or (F == 'rew`[_`]_.) or (F == 'rewrite`[_`]_.)
            or (F == 'rew-in_:_.) or (F == 'rewrite-in_:_.)
            or (F == 'rew-in`[_`]_:_.)
            or (F == 'rewrite-in`[_`]_:_.) .
```

The `show module` command, which prints the specified module, or the current one if no module name is specified, is handled by the following rules.

```
  rl [show-module-1] :
     < O : X@Database | db : DB,
        input : ({'show`module`.}'PreCommand),
        output : nil, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : meta-pretty-print(getFlatUnit(MN, DB),
                      getTopUnit(MN, DB)),
           default : MN, Atts >  .
  rl [show-module-2] :
     < O : X@Database | db : DB, input : ('show`module_.[T]),
        output : nil, default : MN, Atts >
     => < O : X@Database | db : DB, input : nilTermList,
           output : meta-pretty-print(getFlatUnit(parseModExp(T), DB),
                      getTopUnit(parseModExp(T), DB)),
           default : MN, Atts > .

   ...
endm
```

4.9.2 The Full Maude Module

We now give the rules to initialize the loop, and to specify the communication between the loop—the input/output of the system—and the database. Depending on the kind of input that the database receives, its state will be changed, or some output will be generated.

```
mod FULL-MAUDE is
  pr META-FULL-MAUDE-SIGN .
  pr DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .
```

The state of the persistent system, which is supported by the built-in module LOOP-MODE, described in Section 1.3.4, is represented as a single object.

```
  subsort Object < State .

  op o : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
```

The rule specifying the initial value of the loop is given below. We initialize the database with the predefined modules introduced in Section 4.5.1.

```
  rl [init] :
     init
     => [nil,
         < o : Database |
            db : evalUnit(CONFIGURATION,
                   evalUnit(TRIV,
                     evalUnit(UP, emptyDatabase))),
            input : nilTermList,
            output : nil,
            default : nullModName >,
         nil] .
```

When some text has been introduced in the loop, the first argument of the operator `[_,_,_]` is different from `nil`, and we can use this fact to activate the following rule, that enters an input such as a module or a command from the user into the database. The constant `GRAMMAR` names the module containing the signature defining the top level syntax of Full Maude (see Section 4.2 and Appendix A). This signature is used by the `meta-parse` function to parse the input. If the input is syntactically valid[3], the parsed input is placed in the `input` attribute of the database object; otherwise, an error message is placed in the output channel of the loop.

[3]Of course, the input may be syntactically valid, but not semantically valid, since further processing—for example, of bubbles—may reveal a semantic inconsistency.

```
  crl [in] :
     [QIL,
      < O : X@Database | db : DB, input : nilTermList,
         output : nil, default : MN, Atts >,
      QIL']
     => if meta-parse(GRAMMAR, QIL) == error*
        then [nil,
              < O : X@Database | db : DB, input : nilTermList,
                 output : ('ERROR: 'incorrect 'input '.),
                 default : MN, Atts >,
              QIL']
        else [nil,
              < O : X@Database | db : DB,
                 input : meta-parse(GRAMMAR, QIL),
                 output : nil, default : MN, Atts >,
              QIL']
        fi
        if QIL =/= nil .
```

When the output attribute of the persistent object contains a nonempty list of quoted identifiers, the out rule moves it to the third argument of the loop. Then the Core Maude system displays it in the terminal.

```
  crl [out] :
     [QIL,
      < O : X@Database | db : DB, input : TL,
         output : QIL', default : MN, Atts >,
      QIL'']
     => [QIL,
         < O : X@Database | db : DB, input : TL,
            output : nil, default : MN, Atts >,
         (QIL' QIL'')]
        if QIL' =/= nil .

endm
```

4.10 Extending Full Maude

We illustrate in this section the way in which the module algebra of Full Maude can be extended by defining new module expressions and integrating them with the rest of the system. As an example of module-combining operations we present the classical sum, or union, of module expressions; and as an example showing the new possibilities for defining module operations we present a module expression such that, given any nonzero natural number, it generates a parameterized module specifying tuples of the corresponding size. The evaluation of these new module expressions follows the general scheme for the evaluation of module expressions discussed in Section 4.6.5, that is, a new module with such module expression as name must be created, giving to the expression the intended semantics.

The syntax used for the union of module expressions is

```
  op _+_ : ModExp ModExp -> ModExp [assoc prec 42] .
```

Its evaluation consists in generating a unit importing the two module expressions given as arguments [110, 60].

The syntax used for the n-tuple module expression is as follows:

```
  op TUPLE(_) : Token -> ModExp .
```

Its evaluation consists in the generation of a parameterized functional module with the number of TRIV parameters specified by the argument. A sort for tuples of such size, and the corresponding constructor and selector operators, are also defined. Note that the TRIV theory is predefined in Full Maude (see Sections 4.9.2 and 4.5.1). For example, the module expression TUPLE(3) produces the following module.

```
fmod TUPLE(3)[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV] is
   sorts 3Tuple .
   op (_,_,_) : Elt.C1 Elt.C2 Elt.C3 -> 3Tuple .
   op p1_ : 3Tuple -> Elt.C1 .
   op p2_ : 3Tuple -> Elt.C2 .
   op p3_ : 3Tuple -> Elt.C3 .
   var E1 : Elt.C1 .
   var E2 : Elt.C2 .
   var E3 : Elt.C3 .
   eq p1(E1, E2, E3) = E1 .
   eq p2(E1, E2, E3) = E2 .
   eq p3(E1, E2, E3) = E3 .
endfm
```

4.10.1 Extending the Signature

As mentioned in Section 4.2, in order to add any new module expressions the first thing we must do is to extend the signature of Full Maude. In the following module EXT-FULL-MAUDE-SIGN we extend the module FULL-MAUDE-SIGN, given in Appendix A, which contains the declarations of the signature of Full Maude, with the declarations for the syntax of the new module expressions.

```
fmod EXT-FULL-MAUDE-SIGN is
  including FULL-MAUDE-SIGN .

  op TUPLE(_) : Token -> ModExp .
  op _+_ : ModExp ModExp -> ModExp [assoc prec 42] .
endfm
```

In Section 4.2 we saw how a constant GRAMMAR of sort FModule was declared in the META-FULL-MAUDE-SIGN module. Such a constant was then identified with the metarepresentation of a module GRAMMAR, in which there was a declaration importing the FULL-MAUDE-SIGN module and declarations for the constructors of the bubble sorts. To extend the metarepresentation of the module to be used in the parsing process, we define a new constant EXT-GRAMMAR, of sort FModule as well, which extends the module GRAMMAR. We can do this using the addImportList function presented in Section 4.3.5 as in the following module.

```
fmod META-EXT-FULL-MAUDE-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .
  pr UNIT .

  op EXT-GRAMMAR : -> FModule .
  eq EXT-GRAMMAR
    = addImportList((including 'EXT-FULL-MAUDE-SIGN .), GRAMMAR) .
endfm
```

Now the constant EXT-GRAMMAR is identified with the metarepresentation of a module which contains the constructors of the bubble sorts and importation declarations for both FULL-MAUDE-SIGN and EXT-FULL-MAUDE-SIGN. Since EXT-FULL-MAUDE-SIGN imports FULL-MAUDE-SIGN, we get the desired result.

4.10.2 The Union Module Expression

As we explained in Sections 4.6.9 and 4.6.10 for the cases of the instantiation and the renaming module expressions, respectively, the declaration of any new kind of module expression must come together with the definition of the functions `evalModExp`, `modNameToQidList`, and `setUpModExpDependencies` on the new module operator. As discussed in Sections 4.6.9 and 4.8.1, equations for the `prepareModName` and `parseModExp` functions have to be given as well.

```
fmod UNION-EXPR is
  inc MOD-EXPR .
  pr EVALUATION .
  pr UNIT-DECL-PARSING .
```

First of all, we declare the new module operator of sort ModExp as follows.

```
  op _+_ : ModExp ModExp -> ModExp [assoc comm] .

  var PL : ParameterList .
  var DB : Database .
  vars T T' : Term .
  var EIL : EImportList .
  var QIVEPS : QidViewExpPairSet .
  vars ME ME' : ModExp .
  vars U U' U'' U''' : Unit .
  vars M M' M'' M''' : Module .
  vars MNS MNS' MNS'' : ModNameSet .
  vars VES VES' VES'' : ViewExpSet .
  var IS : InfoSet .
  var QIL : QidList .
```

As mentioned above, the evaluation of a union module expression consists in the creation of a new unit, with such a module expression as name, which imports the two module expressions being united. Note, however, that the unit being created has to be of the right type. The new unit will be generated having one type or another, depending on the types of the arguments of the union module expression.

The function `rightEmptyUnit` generates an empty unit of the lowest of the sorts of its two arguments. In case of having a nonstructured module as argument, the corresponding structured one is considered. If one of the two module expressions corresponds to a theory, then a theory is generated, and the lowest sort is taken between the sort of such a theory and the Unit sort immediately above the sort of the other unit; that is, sorts FUnit, SUnit, or OUnit are considered to do the comparison.

```
  ceq evalModExp(ME + ME', PL, DB)
    = evalUnit(
        addImportList((including ME .) (including ME' .),
          setName(
            rightEmptyUnit(
               getTopUnit(ME,
```

```
                evalModExp(ME, PL, evalModExp(ME', PL, DB))),
              getTopUnit(ME,
                evalModExp(ME', PL, evalModExp(ME', PL, DB)))),
           ME + ME')),
       evalModExp(ME, PL, evalModExp(ME', PL, DB)))
     if not unitInDatabase(ME + ME', DB) .

  op rightEmptyUnit : Unit Unit -> Unit [comm] .

  ceq rightEmptyUnit(U, U')
    = emptyStrFModule
     if U : StrFModule and U' : StrFModule .
  ceq rightEmptyUnit(U, U')
    = emptyStrFTheory
     if U : StrFTheory and U' : FUnit .
  ceq rightEmptyUnit(U, U')
    = emptyStrSModule
     if U : StrSModule and U' : StrSModule
        and not (U : StrFModule or U' : StrFModule) .
  ceq rightEmptyUnit(U, U')
    = emptyStrSTheory
     if U : StrSTheory and U' : SUnit
        and not (U : StrFTheory or U' : StrFTheory) .
  ceq rightEmptyUnit(U, U')
    = emptyStrOModule
     if U : StrOModule and U' : StrOModule
        and not (U : StrSModule or U' : StrSModule) .
  ceq rightEmptyUnit(U, U')
    = emptyStrOTheory
     if U : StrOTheory and U' : OUnit
        and not (U : StrSTheory or U' : StrSTheory) .
  eq rightEmptyUnit(unitError(QIL), U) = unitError(QIL) .
  eq rightEmptyUnit(U, unitError(QIL)) = unitError(QIL) .
```

As pointed out in Section 4.3.4, for each new module expression operator being introduced, we need to add equations for the `modNameToQid` function. For the union module expression we only need the following equation:

```
  eq modNameToQidList(ME + ME')
    = (modNameToQidList(ME) '+ modNameToQidList(ME')) .
```

Similarly, we need to give equations for the `setUpModExpDependencies` function, presented in Section 4.5.2, on the union module expression. Given a module expression of the form `ME + ME'` such that `ME` and `ME'` are in the database, we just need to add `ME + ME'` to the set of names of the modules depending on `ME` and `ME'`. In this way, if `ME` or `ME'` are redefined or removed from the database, `ME + ME'` will be removed as well.

```
  eq setUpModExpDependencies((ME + ME'),
       database(
         (< ME ; U ; U' ; M ; M' ; MNS ; VES >
          < ME' ; U'' ; U''' ; M'' ; M''' ; MNS' ; VES' > IS),
         MNS'', VES'', QIL))
    = database(
        (< ME ; U ; U' ; M ; M' ; MNS & (ME + ME') ; VES >
         < ME' ; U'' ; U''' ; M'' ; M''' ; MNS' & (ME + ME') ; VES' > IS),
       MNS'', VES'', QIL) .
```

```
  ceq setUpModExpDependencies((ME + ME'), DB)
    = warning(DB,
        ('\n 'ERROR: 'module modNameToQidList(ME) 'not 'in 'database '.))
      if not unitInDatabase(ME, DB) .
  ceq setUpModExpDependencies((ME + ME'), DB)
    = warning(DB,
        ('\n 'ERROR: 'module modNameToQidList(ME') 'not 'in 'database '.))
      if not unitInDatabase(ME', DB) .
```

The prepareModName function on a union module expression makes recursive calls with each of the module expressions given as arguments.

```
  eq prepareModName(ME + ME', QIVEPS)
    = prepareModName(ME, QIVEPS) + prepareModName(ME', QIVEPS) .
```

Finally, the equation for the parseModExp function is as follows:

```
  eq parseModExp('_+_[T, T'])
    = parseModExp(T) + parseModExp(T') .

endfm
```

4.10.3 The n-tuple Module Expression

Even though the n-tuple module expression is in principle of a completely different nature, the way of handling it is the same as the way of handling any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as name. New equations defining the semantics of functions evalModExp, modNameToQidList, setUpModExpDependencies, prepareModName, and parseModExp are given for this module expression.

```
fmod N-TUPLE-EXPR is
  inc MOD-EXPR .
  pr EVALUATION .
  pr UNIT-DECL-PARSING .
```

We assume that the parameter given to the expression is a nonzero natural number.

```
  sort NzNat .
  subsort NzNat < NzMachineInt .
  var I : NzMachineInt .
  cmb I : NzNat if I > 0 .
```

Then, the declaration of the operator for the n-tuple module expression is as follows:

```
  op TUPLE(_) : NzNat -> ModExp .

  vars N N' : NzNat .
  var PL : ParameterList .
  var DB : Database .
  var T : Term .
  var EIL : EImportList .
  var QIVEPS : QidViewExpPairSet .
```

The equation for the `evalModExp` is reduced to the creation of a module as indicated above. Some auxiliary functions are defined in order to generate the different declarations in the module.

```
op tupleParameterList : NzNat -> ParameterList .
op tupleImportList : NzNat -> ImportList .
op createCopyParameters : NzNat Database -> Database .
op tupleOpDeclSet : NzNat -> OpDeclSet .
op tupleOpDeclSetConstructor : NzNat -> OpDecl .
op tupleOpDeclSetConstructorName : NzNat -> Qid .
op tupleOpDeclSetConstructorArity : NzNat -> QidList .
op tupleOpDeclSetSelectors : NzNat NzNat -> OpDeclSet .
op tupleVarDeclSet : NzNat -> VarDeclSet .
op tupleEquationSet : NzNat -> EquationSet .
op tupleEquationSetAux : NzNat Term -> EquationSet .
op tupleTermArgs : NzNat -> TermList .

ceq evalModExp(TUPLE(N), PL, DB)
  = evalUnit(
       fmod TUPLE(N) is
          tupleParameterList(N)
          tupleImportList(N)
          sorts conc(index(', N), 'Tuple) .
          none
          tupleOpDeclSet(N)
          tupleVarDeclSet(N)
          none
          tupleEquationSet(N)
       endfm,
       createCopyParameters(N, DB))
    if not unitInDatabase(TUPLE(N), DB) .

eq createCopyParameters(N, DB)
  = if N == 1
    then createCopy('TRIV, (par index('C, N) :: 'TRIV), DB)
    else createCopyParameters(_-_(N, 1),
           createCopy('TRIV, (par index('C, N) :: 'TRIV), DB))
    fi .

eq tupleParameterList(N)
  = if N == 1
    then (par index('C, N) :: 'TRIV)
    else parameterList(tupleParameterList(_-_(N, 1)),
           (par index('C, N) :: 'TRIV))
    fi .

eq tupleImportList(N)
  = if N == 1
    then (including par index('C, N) :: 'TRIV .)
    else (tupleImportList(_-_(N, 1))
          (including par index('C, N) :: 'TRIV .))
    fi .

eq tupleOpDeclSet(N)
  = (tupleOpDeclSetConstructor(N)
     tupleOpDeclSetSelectors(N, N)) .
```

```
eq tupleOpDeclSetConstructor(N)
  = (op conc('‘(, conc(tupleOpDeclSetConstructorName(N), '‘))) :
          tupleOpDeclSetConstructorArity(N)
          -> conc(index(', N), 'Tuple)
          [none] .) .

eq tupleOpDeclSetConstructorName(N)
  = if N == 1
    then '_
    else conc('_‘,, tupleOpDeclSetConstructorName(_-_(N, 1)))
    fi .

eq tupleOpDeclSetConstructorArity(N)
  = if N == 1
    then index('Elt.C, N)
    else tupleOpDeclSetConstructorArity(_-_(N, 1)) index('Elt.C, N)
    fi .

eq tupleOpDeclSetSelectors(N, N')
  = if N == 1
    then (op conc(index('p, N), '_) :
               conc(index(', N'), 'Tuple) -> index('Elt.C, N)
               [none] .)
    else (tupleOpDeclSetSelectors(_-_(N, 1), N')
          (op conc(index('p, N), '_) :
                conc(index(', N'), 'Tuple) -> index('Elt.C, N)
                [none] .))
    fi .

eq tupleVarDeclSet(N)
  = if N == 1
    then (var index('E, N) : index('Elt.C, N) .)
    else (tupleVarDeclSet(_-_(N, 1))
          (var index('E, N) : index('Elt.C, N) .))
    fi .

eq tupleEquationSet(N)
  = tupleEquationSetAux(N,
      (conc('‘(, conc(tupleOpDeclSetConstructorName(N), '‘)))
         [tupleTermArgs(N)])) .

eq tupleTermArgs(N)
  = if N == 1
    then index('E, 1)
    else (tupleTermArgs(_-_(N, 1)), index('E, N))
    fi .

eq tupleEquationSetAux(N, T)
  = if N == 1
    then (eq conc(index('p, N), '_)[T] = index('E, N) .)
    else (tupleEquationSetAux(_-_(N, 1), T)
          (eq conc(index('p, N), '_)[T] = index('E, N) .))
    fi .
```

The equations for the `modNameToQidList`, `parseModExp`, `prepareModName`, and `setUpModExpDependencies` functions on the n-tuple module expression are as follows:

```
  eq modNameToQidList(TUPLE(N))
    = ('TUPLE '`( index(', N) '`)) .

  eq parseModExp('TUPLE`(_`)['token[T]])
    = TUPLE(parseMachineInt(T)) .

  eq prepareModName(TUPLE(N), QIVEPS) = TUPLE(N) .

  eq setUpModExpDependencies(TUPLE(N), DB) = DB .

endfm
```

4.10.4 Extended Full Maude

To integrate the specification of Full Maude with the specification dealing with these two new module expressions, we must redefine the module `FULL-MAUDE` given in Section 4.9.2. For the present purposes, we only need to redefine the `in` rule of that module. In general we may need to change any of the rules in it. We will come back to this question in Chapter 5, where we shall discuss further possibilities of extending Full Maude, adding new predefined modules in the initial database, defining new commands, or adding new attributes to keep some extra state in the persistent object. In the present case, however, we only need to change the module in which the inputs are parsed. This is done by changing the `in` rule such that the constant `EXT-GRAMMAR` is used instead of `GRAMMAR`. Notice that we also need to import the modules `UNION-EXPR` and `N-TUPLE-EXPR`. To facilitate incremental extensions of the system, we define a new module `EXT-DATABASE-HANDLING` importing these modules, and then another module `EXT-FULL-MAUDE` importing it. `EXT-FULL-MAUDE` also imports `PREDEF-UNITS`, `LOOP-MODE`, and `META-EXT-FULL-MAUDE-SIGN`.

```
mod EXT-DATABASE-HANDLING is
  pr DATABASE-HANDLING .
  pr UNION-EXPR .
  pr N-TUPLE-EXPR .
endm

mod EXT-FULL-MAUDE is
  pr META-EXT-FULL-MAUDE-SIGN .
  pr EXT-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .

  op o : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
```

We only give here the new `in` rule.

```
  crl [in] :
     [QIL,
      < O : X@Database | db : DB, input : nilTermList,
         output : nil, default : MN, Atts >,
      QIL']
     => if meta-parse(EXT-GRAMMAR, QIL) == error*
        then [nil,
              < O : X@Database | db : DB, input : nilTermList,
                 output : ('ERROR: 'incorrect 'input '.),
                 default : MN, Atts >,
              QIL']
        else [nil,
              < O : X@Database | db : DB,
                 input : meta-parse(EXT-GRAMMAR, QIL),
                 output : nil, default : MN, Atts >,
              QIL']
        fi
        if QIL =/= nil .

  ...

endm
```

4.11 Concluding Remarks

We have seen in the previous sections how, by using the metaprogramming capabilities of Maude, we have been able to define a Core Maude executable specification of Full Maude. With this specification we have endowed Full Maude with a module algebra in the Clear/OBJ tradition, including module hierarchies, parameterized modules, views, and module expressions. In addition, new module-combining and module-transforming operations have likewise been provided. We have seen, for example, how we have integrated the transformation of object-oriented modules into their corresponding system modules with the module algebra, and how new kinds of "parameterized" modules, such as parameterized extensions of `META-LEVEL` and the `TUPLE(`n`)` construction can be easily added.

The module algebra thus obtained becomes easily *extensible*, in the sense that a user may easily define new module transformations and module operations; that is, the user can not only do metaprogramming by using the module operations provided in the module algebra of the language, but can also do *meta-metaprogramming* by defining new ones. We have illustrated some of the possibilities for extending the module algebra by defining some new module expressions. Furthermore, given the reflective design of Maude, it is even possible to change both the module operations provided in the basic syntax of the language and that very syntax itself.

The Full Maude specification described here is part of an overall reflective and extensible design for the Maude system, in which key components of the system are written in Maude itself and can be modified or extended with relative ease. In particular, we plan a number of such extensions for the module algebra, including the addition of parameterized theories and views, and the development of other module composition and module transformation operations. Module operations to hide sorts and operators, and operations to redefine and remove messages in object-oriented modules are among the more interesting operations we plan to consider in extensions of our module algebra.

Furthermore, we plan to generalize the module algebra that we have developed for Maude into a *module algebra generator*, that could endow many other specification languages with powerful and extensible algebras for combining and transforming specifications. Module algebra generator uses can be supported by defining a *parameterized* algebraic data type, say $ModAlg[X]$, that, given a logic $\mathcal{L}$ having "good" metalogical properties, extends the data type $\texttt{Module}_{\mathcal{L}}$ of $\mathcal{L}$-theories to an algebra of theory-composition operations $ModAlg[Module_{\mathcal{L}}]$. The module algebra of Maude provided by the Full Maude specification should then be regarded as a particular instance of such a generic construction, namely, for the case in which the underlying logic $\mathcal{L}$ is rewriting logic.

Another research direction opened up by this approach that we plan to pursue in the future is formal reasoning about the properties of module algebra operations based upon their executable specifications.

Chapter 5

A Church-Rosser Checker

5.1 Introduction

This chapter explains the design and use of a Church-Rosser checker, a proving tool to check whether an order-sorted equational specification with an initial algebra semantics satisfies the Church-Rosser property. This tool, together with an inductive theorem prover—to prove theorems about such specifications—has been developed as part of the Cafe project [16]. It has been integrated within the Full Maude environment and also, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment [48]. Through that second integration, both tools can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [36]. They can also be used on their own to prove properties of equational specifications in Maude [27, 23, 21], that is, of Maude functional modules. An important feature of these tools is that they are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement. We explain in Section 5.4 how a complete execution environment for these tools has been built in Maude, and how they have been integrated with the Full Maude system.

Both tools treat equational specifications as *data* that is manipulated. For example, the Church-Rosser checker computes critical pairs and membership assertions by inspecting the equations in the original specification; and the inductive theorem prover may add an induction hypothesis as a new equational axiom. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic [19, 29], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in both tools [20, 26, 24, 25].

Rewriting logic reflection has one important additional advantage, namely, the modular and declarative treatment of *proof tactics* or *strategies* [29, 19]. Such tactics can be formally specified by rewrite rules at the metalevel of the theory expressing the tool's inference system, and can be easily changed at will without any modification whatsoever to the inference system itself. By contrast, most conventional theorem proving tools either deal with tactics as extralogical features outside the formal framework, or build them in deep inside the tools' code, or both.

The very high level of abstraction at which the tools have been developed has made it relatively easy for us to build them, makes understanding their implementation much easier, and will make their maintenance and future extensions much easier than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine [21], these important benefits in ease of development, understandability, maintainability, extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance. Even though they have not been optimized for performance, and in spite of using several levels of reflection and sophisticated rewriting modulo associativity and associativity-commutativity, the tools have quite competitive performance. For example, the inductive theorem prover is very responsive in interactive mode, with input-output time typically dominating over computation time. The Church-Rosser checker also offers reasonable performance. For example, generating all the critical pairs and membership assertions for the number hierarchy from the natural to the rational numbers takes 2,091,898 rewrites, which are performed in 12 seconds running on a 450MHz Pentium II.

5.1.1 Reflective Design of the Tools

The design of the present tools exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of each tool as a rewrite theory [71, 73, 72]. The fact that these tools reason about *equational theories* is not a restriction of the general framework capabilities: inference systems for reasoning about specifications in any other logic could be represented just as easily. However, the fact that rewriting logic contains equational logic as a sublanguage and that Maude has good support for its equational sublanguage is an added advantage.

The Church-Rosser checker tool has a simple design. Let T be a functional module, which has an initial algebra semantics. Such module, that we want to check whether it is Church-Rosser, is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses T as a data structure—that is, it actually uses its metarepresentation $\overline{T}$—and therefore is a rewrite theory at the metalevel. The checking process can be described in a purely functional way.

Although a bit more complicated, the reflective design of the inductive theorem prover is still simple. Its purpose is to prove inductive properties of a functional module T, which has an initial algebra semantics. The theory T, about which we want to prove inductive properties, is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T as a data structure the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ is already at the metalevel, we need *three levels of theories and two levels of reflection* to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{I}$.

5.1.2 The Church-Rosser Checker

The goal of *executable* equational specification languages is to make *computable* the abstract data types specified in them by initial algebra semantics. In practice this is accomplished by using specifications that are *ground*-Church-Rosser and terminating, so that the equations can be used from left to right as simplification rules; the result of evaluating an expression is then the canonical form that stands as a unique representative for the equivalence class of terms equal to the original term according to the equations. This approach is fully general; indeed, a well-known result of Bergstra and Tucker [5] shows that *any* computable algebraic data type can be specified by means of a finite set of ground-Church-Rosser and terminating equations, perhaps with the help of some auxiliary functions added to the original signature.

Therefore, for computational purposes it becomes very important to know whether a given specification is indeed ground-Church-Rosser and terminating. A nontrivial question is how to best support this with adequate tools. For proving termination one can use standard termination proof techniques that are well-supported in tools such as CiME [32]. A thornier issue is what to do for establishing the ground-Church-Rosser property for a terminating specification. The problem is that a specification with an initial algebra semantics can often be ground-Church-Rosser even though some of its critical pairs may not be joinable. That is, the specification can often be ground-Church-Rosser without being Church-Rosser for arbitrary terms with variables. In such a situation, blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both quite hopeless—the procedure diverges or gets stuck because of unorientable rules, and even with success may return a specification that is quite different from the original one—and unnecessary: the specification was ground-Church-Rosser!

Our Church-Rosser checker tool is oriented to checking specifications with an initial algebra semantics that have already been proved terminating using a termination tool and now need to be checked to be ground-Church-Rosser. Since, for the reasons mentioned above, user interaction will typically be quite essential, completion is not attempted. Instead, if the specification cannot be shown to be ground-Church-Rosser by the tool, proof obligations are generated and are given back to the user as a useful guide in the attempt to establish the ground-Church-Rosser property. Since this property is in fact inductive, in some cases the inductive theorem prover can be enlisted to prove some of these proof obligations (see Section 5.5.2). In other cases, the user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations. We give in Section 5.5 some methodological guidelines for the use of the tool, and illustrate the use with some examples; we also explain there that the issue of finding general inductive proof techniques for proving the ground-Church-Rosser property is at the moment an interesting open problem.

The present tool only accepts order-sorted conditional specifications where each of the operation symbols has either no equational attributes, or only the commutativity attribute[1]. Furthermore, it is assumed that such specifications do not contain any built-in sort or function, and that they have already been proved terminating using another tool. The tool attempts to establish the ground-Church-Rosser property *modulo* the commutativity of some of the operators by checking a sufficient condition. For order-sorted specifications, being Church-Rosser and terminating means not only confluence—so that a unique normal form will be reached—but also a *descent* property, namely that the normal form will have the least possible sort among those of all other equivalent terms. Therefore, the tool's output consists of a set of critical pairs and a set of membership assertions that must be shown, respectively, ground-joinable, and ground-rewritable to a term with the required sort.

After introducing some basic formal concepts and results underlying the tool's design, and after discussing several auxiliary functions used by the tool, including order-sorted commutative unification, we explain key parts of the tool and then give recommendations for its use and some examples. As in Chapter 4, we present the different modules with their main declarations, so that a general idea of the whole system can be gained, omitting declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions. We add dots "..." to indicate the missing equations and declarations.

[1] As we shall see in Section 5.3.1, in the current version of the tool the unification and matching algorithms are specified equationally, and only for signatures whose operators do not have equational attributes, or have just the commutativity attribute. There will be built-in unification and functions `meta-match` and `meta-unify` in Maude in the near future. Once they are available, the tool will be easily extensible to the case in which the different combinations of attributes supported in Maude are also supported by the tool.

5.2 Church-Rosser Order-Sorted Specifications

In this section we introduce the notion of Church-Rosser order-sorted specification. We assume specifications of the form $\mathcal{S} = (\Sigma, R \cup A)$ where Σ is a preregular order-sorted signature [56], and $R \cup A$ is a set of equations such that A is a set of *sort-preserving* equational axioms, that is, whenever $t =_A t'$ we have[2] $LS(t) = LS(t')$. The equations R will be used as rewrite rules modulo the axioms A. Furthermore, in what follows, and for the purposes of the present tool, the axioms A will only consist of *commutativity* axioms for certain binary operators in Σ. The problem, then, is to check whether our specification $\mathcal{S}$ has the Church-Rosser property.

After giving some auxiliary definitions, we introduce the notion of Church-Rosser order-sorted specifications, and describe the sufficient condition used by our tool to attempt checking the Church-Rosser property.

5.2.1 The Confluence Property

Let N_+^* be the set of sequences of positive integers, with Λ the empty sequence in N_+^*, and $\_ \cdot \_$ the concatenation operation on sequences. We call the members of N_+^* *positions*, and denote them by p, p', p'', and so on.

Given a signature Σ, a set of variables X, and a term $t \in \mathcal{T}_\Sigma(X)$ we can define its *set of positions* $\mathcal{P}(t) \subseteq N_+^*$ and *the subterm* $t|_p$ *of* t *at position* p (assuming $p \in \mathcal{P}(t)$) as follows:

1. if $t = x \in X$, then $\mathcal{P}(t) = \{\Lambda\}$ and $t|_\Lambda = t$.
2. if $t = f(t_1, \ldots, t_n)$, then $\mathcal{P}(t) = \{\Lambda\} \cup \{i \cdot p \mid 1 \leq i \leq n,\ p \in \mathcal{P}(t_i)\}$, $f(t_1, \ldots, t_n)|_\Lambda = f(t_1, \ldots, t_n)$, and $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$.

Finally, a term t with its subterm $t|_p$ replaced by the term t' is denoted $t[t']_p$.

A substitution is a replacement operation uniquely defined by a mapping from variables to terms, and is written out as $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. Given a set of axioms A, a substitution σ is an *A-unifier* of t and t' if $\sigma t =_A \sigma t'$, and it is an *A-match* from t to t' if $t' =_A t\sigma$.

The set of variables occurring in a term t is denoted $vars(t)$. Then, given a substitution $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, we denote by $D(\sigma)$ the set of variables $\{x_1, \ldots, x_n\}$, and by $I(\sigma)$ the set $\bigcup_{i=1}^n vars(t_i)$.

We say that substitution σ is *at least as general* as substitution ρ if there exists a substitution τ such that $\sigma\tau = \rho$. In this case we write $\rho \lessdot \sigma$. We say that substitution σ is *at least as general* as substitution ρ with respect to a subset X of $D(\sigma)$ if there exists a substitution τ such that $\sigma\tau = \rho$ when σ and ρ are restricted to X. In this case we write $\sigma\tau = \rho[X]$.

We say that a term t *A-overlaps* another term with distinct variables t' if there is a nonvariable subterm $t'|_p$ of t' for some position $p \in \mathcal{P}(t')$ such that the terms t and $t'|_p$ can be A-unified.

Definition 36 *Given an order-sorted equational specification $\mathcal{S} = (\Sigma, R \cup A)$ with the above assumptions, and given conditional rewrite rules $l \to r$ `if` $\bar{u} \downarrow \bar{v}$ and $l' \to r'$ `if` $\bar{u'} \downarrow \bar{v'}$ in R such that $vars(l) \cap vars(l') = \emptyset$ and $l|_p\sigma =_A l'\sigma$, for some nonvariable position $p \in \mathcal{P}(l)$ and A-unifier σ, then the triple*

$$ccp(l\sigma[r'\sigma]_p,\ r\sigma,\ \bar{u}\sigma \downarrow \bar{v}\sigma \wedge \bar{u'}\sigma \downarrow \bar{v'}\sigma)$$

is called a (conditional) critical pair.

[2] We denote the least sort [56] of a term t by $LS(t)$.

In the uses we will make of the above definition we will always assume that the unification and the comparison for equality have been performed *modulo* A, where A consists only of commutativity axioms for certain operators in an order-sorted signature; that is, we use order-sorted unification with commutative axioms and equality modulo such axioms. We consider all unifiers, not only the most general ones. Note also that the critical pairs accumulate the substitution instances of the conditions in the two rules, as in [11].

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a critical pair $ccp(t, t', c)$ is more general than another critical pair $ccp(u, u', c')$ if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$, and $c\sigma =_A c'$, where, as before, we assume A to consist of commutativity axioms for some operations or to be otherwise empty.

Then, given a specification $\mathcal{S}$, let $\mathrm{MCP}(\mathcal{S})$ denote the set of most general critical pairs between rules in $\mathcal{S}$ that, after simplifying both sides of the critical pair using the equations in $\mathcal{S}$, are not identical critical pairs modulo A of the form $ccp(t, t, c)$. To determine the overlaps of the lefthand sides of the rules the variables in them have to be renamed in order to get disjoint sets of variables appearing in each of them. Under the assumption that the order-sorted equational specification $\mathcal{S}$—with A involving only commutativity axioms—is terminating, then, if $\mathrm{MCP}(\mathcal{S}) = \emptyset$, we are guaranteed that the specification $\mathcal{S}$ is *confluent*—in the standard sense that if t can be rewritten modulo A to u and v using the rules in $\mathcal{S}$, then u and v can be rewritten modulo A to some w—and therefore, each term t has a unique canonical form $t\downarrow_{\mathcal{S}}$. Note that, due to the presence of conditional equations, we can have $\mathrm{MCP}(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ still confluent, because all the conditions in the critical pairs may be unsatisfiable, but establishing such unsatisfiability may require additional reasoning. More importantly for our purposes, even in the unconditional case we can have $\mathrm{MCP}(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ *ground*-confluent, that is, confluent for all ground terms. Therefore, assuming termination, $\mathrm{MCP}(\mathcal{S}) = \emptyset$ will ensure the confluence and, a fortiori, the ground-confluence of $\mathcal{S}$, but this is only a sufficient condition.

5.2.2 The Descent Property

For an order-sorted specification it is not enough to be confluent. The canonical form should also provide the most complete information possible about the sort of a term. This intuition is captured by our notion of Church-Rosser specifications.

Definition 37 *We call a confluent and terminating order-sorted specification* $\mathcal{S}$ Church-Rosser *iff it additionally satisfies the following* descent *property: for each term* t *we have* $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$. *Similarly, we call a ground-confluent and terminating specification* $\mathcal{S}$ ground-Church-Rosser *iff for each ground term* t *we have* $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$.

Note that these notions are more general and flexible than the requirement of confluence and *sort-decreasingness* [66, 49]. The issue is how to find checkable conditions for descent that, in addition to the computation of critical pairs, will ensure the Church-Rosser property. This leads us into the topic of specializations.

Given an order-sorted signature $(S, \leq, \Sigma)$, a sorted set of variables X can be viewed as a pair $(\bar{X}, \mu)$ where $\bar{X}$ is a set of variable names and μ is a sort assignment $\mu : \bar{X} \to S$. Thus, a *sort assignment* μ for X is a function mapping the names of the variables in $\bar{X}$ to their sorts. The ordering $\leq$ on S is extended to sort assignments by

$$\mu \leq \mu' \Leftrightarrow \forall x \in \bar{X}, \mu(x) \leq \mu'(x).$$

We then say that μ' *specializes* to μ, via the substitution

$$\rho : (x : \mu(x)) \leftarrow (x : \mu'(x))$$

called a *specialization* of $X = (\bar{X}, \mu')$ into $\rho(X) = (\bar{X}, \mu)$. Note that if the set of sorts is finite, or if each sort has only a finite number of sorts below it, then a finite sorted set of variables has a finite number of specializations.

The notion of specialization can be extended to axioms and rewrite rules. A specialization of an equation $(\forall X, l = r)$ is another equation $(\forall \rho(X), \rho(l) = \rho(r))$ where ρ is a specialization of X. A specialization of a rule $(\forall X, l \to r \texttt{ if } c)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r) \texttt{ if } \rho(c))$ where ρ is a specialization of X.

The checkable conditions that we have to add to the critical pairs to test for the descent property are called membership assertions.

Definition 38 *Let $\mathcal{S}$ be an order-sorted specification whose signature satisfies the assumptions already mentioned. Then, the set of* (conditional) membership assertions *for a conditional equation $t = t'$ `if` c is defined as*

$$\{\ t'\theta : LS(t\theta) \texttt{ if } c\theta \quad | \quad \theta \textit{ is a specialization of } vars(t) \textit{ and } LS(t'\theta \downarrow_{\mathcal{S}}) \not\leq LS(t\theta)\ \}$$

A membership assertion $t : s \texttt{ if } c$ is more general than another membership assertion $t' : s' \texttt{ if } c'$ if there exists a substitution σ such that $t\sigma =_A t'$, $s \leq s'$, and $c\sigma =_A c'$. Again, we assume A to consist only of commutativity axioms for some operators, or to be otherwise empty.

Let MMA($\mathcal{S}$) denote the set of most general membership assertions of all of the equations in the specification $\mathcal{S}$. Then, given a specification $\mathcal{S}$, the tool returns

$$checking(\mathcal{S}) = \langle\ \mathrm{MCP}(\mathcal{S}),\ \mathrm{MMA}(\mathcal{S})\ \rangle.$$

A fundamental result[3] underlying our tool is that the absence of critical pairs and of membership assertions in such an output is a sufficient condition for a terminating specification $\mathcal{S}$ to be *Church-Rosser*. In fact, for terminating unconditional specifications this check is a necessary and sufficient condition; however, for conditional such specifications, the check is only a sufficient condition because if the specification has conditional equations we can have unsatisfiable conditions in the critical pairs or in the membership assertions; that is, we can have $\langle\ \mathrm{MCP}(\mathcal{S}),\ \mathrm{MMA}(\mathcal{S})\ \rangle \neq \langle\ \emptyset, \emptyset\ \rangle$ with $\mathcal{S}$ still Church-Rosser. Furthermore, even if we assume that the specification is unconditional, since for specifications with an initial algebra semantics we only need to check that $\mathcal{S}$ is ground-Church-Rosser, we may often have specifications that satisfy this property, but for which the tool returns nonempty sets of critical pairs or of membership assertions as proof obligations.

Of course, in other cases, it may in fact be a matter of some error in the user's specification that the tool uncovers. In any case, the user has complete control on how to modify his/her specification, using the proof obligations in the output of the tool as a guide. In fact, several possibilities exist. The user can prove inductively that the critical pairs are ground-joinable, and that the

[3]A detailed proof of this result is beyond the scope of this work, and will be presented elsewhere. For related results in membership equational logic see [11].

membership assertions are ground-rewritable to a term with the required sort; however, at present the methods available for such proofs are quite limited. Alternatively, the user can instead modify the specification with the purpose of either correcting a found mistake, or modifying the already correct specification into a variant that the tool will hopefully certify Church-Rosser when resubmitted.

5.3 The Maude Specification of a Church-Rosser Checker

The Church-Rosser checker tool is written entirely in Maude. It is in fact an executable specification in rewriting logic of the inference system to check a sufficient condition for the Church-Rosser property.

The input to the Church-Rosser checker tool is given by the functional module we want to check, which, as already explained, must be an order-sorted conditional specification with operators involving only the commutativity attribute and, furthermore, it cannot contain any built-in sorts or functions. As we shall see in Section 5.4, the tool has been completely integrated within the Full Maude environment, so the Church-Rosser property can be checked on any specification (satisfying the conditions mentioned above) previously entered to Full Maude. Furthermore, the specification in question may be *structured*, with other specifications, perhaps defined by module expressions, as submodules.

The output of the tool consists of a set of critical pairs, a set of membership assertions (corresponding to descent proof obligations), and the set of variables appearing in them.

After introducing in Section 5.3.1 some auxiliary functionality for the manipulation of terms and order-sorted unification and matching, we explain in Sections 5.3.2 and 5.3.3 the specifications of the confluence and descent checks. The specification of the checking of the Church-Rosser property in then presented in Section 5.3.4.

5.3.1 Auxiliary Functionality

In this section we present some required auxiliary functions. We start by introducing some basic functions for the manipulation of terms, and then we present the unification and matching functions.

General Operations on Terms

We summarize in this section the definitions of several operations, in a module extending `META-LEVEL`, that manipulate terms: definitions for positions, as given in Section 5.2, and operations to get the subterm of a given term at a given position, to replace the subterm of a term at a given position by another term, to get all the nonvariable positions in a term, to apply a substitution to a term, and to get a copy of a term in which all the variables in it have been renamed.

```
fmod EXT-TERM is
  pr META-LEVEL .

  sorts Position PositionSet .
  subsort Position < PositionSet .

  op emptyPosSet : -> PositionSet .
```

```
op posSet : PositionSet PositionSet -> PositionSet
        [assoc comm id: emptyPosSet] .
```

The function `getSubterm` takes a term t and a position p and returns $t|_p$ if $p \in \mathcal{P}(t)$; otherwise, it returns `error*`.

```
op getSubterm : Term Position -> Term .
```

The function `replace` takes terms t and t' and a position p as input and returns the term $t[t']_p$ if $p \in \mathcal{P}(t)$, or `error*` otherwise.

```
op replace : Term Term Position -> Term .
```

The function `allNonVarPos` takes a module M and a term t and returns the set $\mathcal{P}(t)$ of all nonvariable positions of t: $\{p \in \mathcal{P}(t) \mid t|_p \notin vars(M)\}$.

```
op allNonVarPos : Module Term -> PositionSet .
```

The function `substitute` takes a term t and a substitution σ and returns the term $t\sigma$.

```
op substitute : Term Substitution -> Term .
```

Finally, we define the function `rename`, which takes a term and renames all the variables appearing in it. The renaming is done by replacing each of the variables in the term by another variable with its name concatenated with the character `@`, that is, each variable V is replaced by V`@`.

```
  op rename : Module Term -> Term .

  ...

endfm
```

Order-Sorted Commutative Unification and Matching

In this section we sketch the order-sorted unification algorithm implemented. It yields a complete set of unifiers for a unification problem in which it is assumed that the order-sorted signature of the specification in question can have some operators declared to be *commutative*—in which case, all the remaining subsort-overloaded versions of any such operator should also have been declared commutative—but no other equational axioms have been declared as attributes for the operators. Therefore, the order-sorted unification is performed *modulo* the commutativity of certain operators.

The basic idea is to turn each of the textbook-style inference rules for such a unification algorithm into corresponding rewrite rules in Maude. We give below the inference system; their Maude counterparts can be found in [25]. Here we only summarize the basic concepts and the top-level functionality.

Following [64], we unify using the sort information as soon as possible in order to quickly discard failures. Then, we complete the simplification process and push the assertions of the sorts on the solutions that have been found. A similar approach has been followed for order-sorted matching.

The order-sorted unification algorithm for solving equations (pairs of the form $t =^?_c t'$) can be described as the result of a simplification step followed by a solving step. The simplification step is a sequence of decomposition, merging, and mutation steps, transforming the initial unification problem into an equivalent disjunction of systems of fully decomposed equations of the form $x =^?_c t$, where x is a variable appearing only once in the system. The solving step consists in finding the finite set of solutions for the simplified system.

Thus, a unification problem is a set of equations, also called a system, denoted $(e_1, \ldots, e_n)$, or a set of systems, also called a disjunction of systems and written $S_1 \vee \ldots \vee S_n$. The set of variables occurring in a unification problem U is denoted $vars(U)$.

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a substitution σ is an $\mathcal{S}$-solution of the equation $t =^?_A t'$ if and only if $\sigma(t) =_A \sigma(t')$. The set of $\mathcal{S}$-solutions of an equation $t =^?_A t'$ is denoted $U(t, t', \mathcal{S})$.

A set of substitutions Φ is a complete set of $\mathcal{S}$-solutions of the equation $t =^?_A t'$ away from the set of variables W such that $vars(t) \cup vars(t') \subseteq W$ if and only if

- $\forall \sigma \in \Phi, D(\sigma) \subseteq vars(t) \cup vars(t')$ and $I(\sigma) \cap W = \emptyset$;
- $\forall \sigma \in \Phi, \sigma \in U(t, t', \mathcal{S})$; and
- $\forall \sigma \in U(t, t', \mathcal{S}), \exists \alpha \in \Phi$ such that $\alpha \lessdot_A \sigma[vars(t) \cup vars(t')]$.

Note that, as mentioned in Section 5.2, we do not compute a minimal set of unifiers. Also, regarding the subsort overloading of operators we assume that an operator symbol has always the same equational attributes—to rewrite modulo such attributes—in all its subsort-overloaded versions; and such equational attributes are restricted to be either the empty set, or the single equation of commutativity.

We give below the set of inference rules used for the unification algorithm. They operate on 3-tuples of the form $\langle V;\ E;\ \sigma\rangle$ and on 4-tuples of the form $[V;\ C;\ \sigma;\ \theta]$. A 3-tuple $\langle V;\ E;\ \sigma\rangle$ consists of a set of variables V, a set of equations E, and a substitution σ. A 4-tuple $[V;\ C;\ \sigma;\ \theta]$ consists of a set of variables V, a set of membership constraints C, and substitutions σ and θ. The rules operating on the first kind of tuples correspond to the first stage of the process, which is quite similar to syntactic unification. The main differences are in the rules **Check** and **Eliminate**, in which the sort information is used to try to quickly discard failure. In the second stage the constraints on the solutions are checked. We assume a well-founded order $\succ$ on equations, and we denote by $s \cap s'$ the set of maximal lower bounds of sorts s and s'.

Deletion of Trivial Equations

$$\frac{\langle\ V;\ \{t =^?_c t,\ E\};\ \sigma\ \rangle}{\langle\ V;\ E;\ \sigma\ \rangle}$$

Decomposition

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =^?_c f(t'_1,\ \ldots,\ t'_n),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =^?_c t'_1,\ \ldots,\ t_n =^?_c t'_n,\ E\};\ \sigma\ \rangle}$$
if $n \neq 2$ or f noncommutative

$$\frac{\langle\ V;\ \{f(t_1, t_2) =^?_c f(t'_1, t'_2),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =^?_c t'_1,\ t_2 =^?_c t'_2,\ E\};\ \sigma\ \rangle \vee \langle\ V;\ \{t_1 =^?_c t'_2,\ t_2 =^?_c t'_1,\ E\};\ \sigma\ \rangle}$$
if f commutative

Clash of Symbols

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =^?_c g(t'_1,\ \ldots,\ t'_m),\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$
if $n \neq m$ or $f \neq g$

Merging

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ x =_c^? t',\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ t =_c^? t',\ E\};\ \sigma\ \rangle}$$
if $x = t \succ t = t'$

Check

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$
if $x \neq t$ and (x occurs in t or $s \cap LS(t) = \emptyset$)

Eliminate

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ E\theta;\ \{\sigma\theta, \theta\}\ \rangle}$$
with $\theta = \{x \leftarrow t\}$
if x does not occur in t and $s \cap LS(t) \neq \emptyset$

Transition

$$\frac{\langle\ V;\ \emptyset;\ \sigma\ \rangle}{[\ V;\ \emptyset;\ \sigma;\ \sigma\ ]}$$

Solving $(x \leftarrow t)$

$$\frac{[\ \{x : s,\ V\};\ C;\ \{x \leftarrow t,\ \sigma\};\ \theta\ ]}{[\ V;\ \{(t : s),\ C\};\ \sigma;\ \theta\ ]}$$

Solving $(x : s)$

$$\frac{[\ \{x : s,\ V\};\ \{x : s',\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{s'' \in s \cap s'} [\ \{x : s'',\ V\};\ C;\ \emptyset;\ \theta\ ]}$$

Solving $(f(t_1, \ldots, t_n) : s)$

$$\frac{[\ V;\ \{f(t_1, \ldots, t_n) : s,\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{\substack{f : s_1 \ldots s_n \to s' \\ s' \leq s \\ s_1 \ldots s_n\, maximal}} [\ V;\ \{t_1 : s_1,\ \ldots,\ t_n : s_n,\ C\};\ \emptyset;\ \theta\ ]}$$

These inference rules are mapped into Maude specifications almost without modification. As already mentioned, the complete specification can be found in [25]. We give here just the following definitions.

```
fmod UNIFICATION is
  pr META-LEVEL .
  pr EXT-TERM .
  pr EXT-BOOL .
```

The solution for a unification problem will be given as a set of substitutions, and so, we need the following declarations:

```
  sort SubstitutionSet .
  subsort Substitution < SubstitutionSet .
  op emptySubstitutionSet : -> SubstitutionSet .
  op substitutionSet : SubstitutionSet SubstitutionSet -> SubstitutionSet
          [assoc comm id: emptySubstitutionSet] .

  op unify : Module Term Term -> SubstitutionSet .

   ...

endfm
```

The specification for order-sorted matching is quite similar to the specification for unification. The main differences come from the fact that now we have to consider noncommutative equations, since to find a match from a term t to another term t' is to find a substitution σ such that $t' =_c t\sigma$. The check for the correct sorting of an equation in solved form $x =_c^? t$ is reduced to comparing the sort of the variable x with the least sort of the term t being assigned.

```
fmod MATCHING is
  pr UNIFICATION .
```

For our tool we do not need the solution of the matching; it is enough to know whether there is a match or not. As for the unification case, we consider order-sorted matching with no axioms or only with the commutative axiom, and we assume that any subsort-overloaded function symbols have the same attributes.

```
  op match? : Module EquationSet -> Bool .

   ...

endfm
```

5.3.2 Confluence Check

We present in this section the specification for the confluence check, carried out by the `confluenceCheck` function. We assume that the module passed as argument to the `confluenceCheck` function has been extended with renamed copies of the variables in it. Specifically, we assume that for each variable V in the original module there is a variable V`@` which is not used[4]. Since to check the overlappings there cannot be common variables between the terms, we generate a renamed copy of one of the equations just by renaming each variable using the `rename` function presented in 5.3.1.

```
fmod CONFLUENCE-CHECK is
  pr UNIFICATION .
  pr MATCHING .
  pr EXT-TERM .

  var M : Module .
  vars T T' T1 T1' T2 T2' : Term .
  vars CP CP' : CritPair .
```

[4]It is also assumed that for each variable V of sort S in the original module there is a variable V`@@`S' for each subsort S' of S. These variables are used in the order-sorted unification algorithm.

```
vars CPS CPS' : CritPairSet .
vars Eq Eq' : Equation .
vars EqS EqS' EqS'' : EquationSet .
var P : Position .
var PS : PositionSet .
var Subst : Substitution .
var SubstS : SubstitutionSet .
```

We declare sorts for critical pairs (`CritPair`) and for sets of critical pairs (`CritPairSet`), and constructors for them. The constructors for critical pairs (`cp`) and for conditional critical pairs (`ccp`) have, respectively, two and four arguments. The two arguments of `cp` and the first two of `ccp` are the terms forming the critical pair. The two last arguments in a conditional critical pair correspond to the condition, which is given following the conventions for conditions in membership axioms, equations, and rules in the `META-LEVEL` module.

```
sort CritPair .
op cp : Term Term -> CritPair .
op ccp : Term Term Term Term -> CritPair .

sort CritPairSet .
subsort CritPair < CritPairSet .
op emptyCritPairSet : -> CritPairSet .
op critPairSet : CritPairSet CritPairSet -> CritPairSet
        [assoc comm id: emptyCritPairSet] .
```

Given a specification $\mathcal{S}$, the `critPairs` function finds all the critical pairs between the equations in $\mathcal{S}$ considered as rules, oriented from left to right.

One critical pair is generated for each unifier for each of the possible nonvariable overlappings of the lefthand sides of any two equations in the module. These critical pairs are calculated by finding all the possible such pairs for each of the equations in the module (`critPairs1`) with a renamed copy of each one of the other equations in the module including itself (`critPairs2`). For each pair of equations, their left sides are unified at any nonvariable position of the term of the first equation (`critPairs3`), and then a critical pair is constructed for each one of the solutions of the unification problem (`critPairs4`).

To simplify the exposition, we present only the equations for the unconditional case. As said above, the critical pair is formed by `critPairs4` for a pair of equations with an overlapping at some position with some substitution. In the cases when one or both of the equations involved are conditional, then the conjunction of the conditions with the substitution applied to them is placed as the condition of the critical pair.

```
op critPairs : Module -> CritPairSet .
op critPairs1 : Module EquationSet EquationSet -> CritPairSet .
op critPairs2 : Module Equation EquationSet -> CritPairSet .
op critPairs3 : Module Equation Equation PositionSet -> CritPairSet .
op critPairs4 :
     Module Equation Equation Position SubstitutionSet -> CritPairSet .

eq critPairs(M) = critPairs1(M, equationSet(M), equationSet(M)) .

eq critPairs1(M, ((eq T = T' .) EqS), EqS')
  = critPairSet(
      critPairs1(M, EqS, EqS'),
      critPairs2(M, (eq rename(M, T) = rename(M, T') .), EqS')) .
eq critPairs1(M, none, EqS) = emptyCritPairSet .
```

```
eq critPairs2(M, (eq T = T' .), (Eq EqS))
  = critPairSet(
      critPairs2(M, (eq T = T' .), EqS),
      critPairs3(M, (eq T = T' .), Eq, allNonVarPos(M, T))) .
eq critPairs2(M, Eq, none) = emptyCritPairSet .

eq critPairs3(M, (eq T1 = T1' .), (eq T2 = T2' .), posSet(P, PS))
  = critPairSet(
      critPairs3(M, (eq T1 = T1' .), (eq T2 = T2' .), PS),
      critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .),
        P, unify(M, getSubterm(T1, P), T2))) .
eq critPairs3(M, Eq, Eq', emptyPosSet) = emptyCritPairSet .

eq critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .), P,
     substitutionSet(Subst, SubstS))
  = critPairSet(
      cp(substitute(T1', Subst), replace(substitute(T1, Subst),
         substitute(T2', Subst), P)),
      critPairs4(M, (eq T1 = T1' .), (eq T2 = T2' .), P, SubstS)) .
eq critPairs4(M, Eq, Eq', P, emptySubstitutionSet) = emptyCritPairSet .
```

Once all the critical pairs have been generated, the trivial ones—those of the forms cp(t, t) or ccp(t, t, c), for some term t and condition c—are removed by the function delete, which is applied again to the set of critical pairs resulting from the simplification process. As above, we only give the equations for the unconditional case.

```
op delete : CritPairSet -> CritPairSet .

eq delete(critPairSet(cp(T, T'), CPS))
  = if T == T'
    then delete(CPS)
    else critPairSet(cp(T, T'), delete(CPS))
    fi .
eq delete(emptyCritPairSet) = emptyCritPairSet .
```

This simplification is achieved by the function simplify that reduces both sides of the critical pairs to their normal forms in the given specification.

```
op simplify : CritPairSet Module -> CritPairSet .

eq simplify(critPairSet(cp(T, T'), CPS), M)
  = critPairSet(cp(meta-reduce(M, T), meta-reduce(M, T')),
      simplify(CPS, M)) .
eq simplify(emptyCritPairSet, M) = emptyCritPairSet .
```

Finally, we apply the function maximalCPSet to get the set of most general critical pairs among those remaining. In order to get a minimal set of critical pairs we not only have to eliminate trivial ones, but we also need to take care of those that are repeated or that are less general than some other critical pair. The function maximalCPSet returns the set of most general critical pairs. It proceeds comparing each critical pair *cp* with all the others; if some critical pair is more general than *cp* then *cp* is removed. As mentioned in Section 5.2, a critical pair ccp(t, t', c) is more general than another one ccp(u, u', c') if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$, and $c\sigma =_A c'$, with A the commutativity attributes declared for some operators in the specification.

```
  op maximalCPSet : CritPairSet Module -> CritPairSet .
  op maximalCPSetAux :
       CritPair CritPairSet CritPairSet Module -> CritPairSet .
  op moreGeneralCP? : CritPair CritPair Module -> Bool .

  eq maximalCPSet(critPairSet(CP, CPS), M)
    = maximalCPSetAux(CP, CPS, emptyCritPairSet, M) .
  eq maximalCPSet(emptyCritPairSet, M) = emptyCritPairSet .

  eq maximalCPSetAux(CP, critPairSet(CP', CPS), CPS', M)
    = if moreGeneralCP?(CP, CP', M)
      then maximalCPSetAux(CP, CPS, CPS', M)
      else if moreGeneralCP?(CP', CP, M)
           then maximalCPSetAux(
                  CP', critPairSet(CPS, CPS'), emptyCritPairSet, M)
           else maximalCPSetAux(CP, CPS, critPairSet(CP', CPS'), M)
           fi
      fi .
  eq maximalCPSetAux(CP, emptyCritPairSet, critPairSet(CP', CPS), M)
    = critPairSet(CP, maximalCPSetAux(CP', CPS, emptyCritPairSet, M)) .
  eq maximalCPSetAux(CP, emptyCritPairSet, emptyCritPairSet, M) = CP .

  eq moreGeneralCP?(cp(T1, T1'), cp(T2, T2'), M)
    = match?(M, ((eq T1 = T2 .) (eq T1' = T2' .)))
      or-else
      match?(M, ((eq T1 = T2' .) (eq T1' = T2 .))) .
```

The `confluenceCheck` is then reduced to the following.

```
  op confluenceCheck : Module -> CritPairSet .

  eq confluenceCheck(M)
    = maximalCPSet(delete(simplify(delete(critPairs(M)), M)), M) .

   ...

endfm
```

Nonconfluence of a Natural Numbers Specification

Our first example illustrating the Church-Rosser checker is the very simple specification `NAT` of the natural numbers presented in Section 1.3.4. This specification is perfectly reasonable. Its initial model is the set of natural numbers $\mathbb{N}$, with the sum and product operators. However, although it is ground-confluent it is not confluent. One of the solutions for the unification in the overlapping of the last of the equations with itself at the top generates a nonconfluent critical pair.

The output given by the tool is as follows.

```
  Checking solution :
    var N : Nat .
    var N@ : Nat .
    cp s(N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
```

Note that all variables in the critical pairs, and in the membership assertions (none of the latter in this case) are collected and included as part of the output.

This critical pair was generated from one of the solutions of the unification corresponding to overlapping at the top the lefthand sides of the equation

```
eq s N * M = M + (N * M) .
```

and of its renamed copy

```
eq s N@ * M@ = M@ + (N@ * M@) .
```

Because of commutativity the unification of both lefthand sides at the top gives solutions

$$\{(\texttt{N} \leftarrow \texttt{N@});\ (\texttt{M} \leftarrow \texttt{M@})\} \text{ and } \{(\texttt{M} \leftarrow \texttt{s N@});\ (\texttt{M@} \leftarrow \texttt{s N})\}.$$

One critical pair is generated for each of these substitutions:

```
cp(M@ + (N@ * M@), M@ + (N@ * M@))
cp(s N + (N@ * s N), s N@ + (N * s N@))
```

The first one is trivial, and therefore is eliminated. Reducing each of the terms of the other one in the specification we obtain the critical pair

```
cp s(N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
```

Nonconfluent critical pairs generated by the tool can be used by the user to detect possible errors in the specification, or to try to complete it manually. This last possibility is similar to the process used in automatic completion tools, with the difference that the equations added in these tools are usually hard to understand, and that such automatic completion processes often diverge. Here, the user can decide if they are really needed, and then the way in which they are going to be added or can be replaced by other equations.

In this case the set of membership assertions is empty. This means that the equations are descending. We shall study descent checking in Section 5.3.3, and will return to this example, to make it confluent, in Section 5.5.1.

5.3.3 Descent Check

Given a specification $\mathcal{S}$, the function `descentCheck` returns MMA($\mathcal{S}$), that is, the set of most general membership assertions among those generated for all specializations of the equations not satisfying the descent condition. For each one of the possible specializations of each of the equations in the module (generated by the function `eqInstanceSet`) the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side, reduced to its normal form. Thus, for each equation $(\forall X, l = r)$ that considered as a rewrite rule fails to satisfy the descent condition, that is, $LS(r\downarrow) \not\leq LS(l)$, a membership assertion of the form `mb` r `:` $LS(l)$ `.` is generated. Similarly, for a conditional equation $(\forall X, l = r$ `if` $c)$ failing to satisfy the descent condition, a *conditional* membership assertion of the form `cmb` r `:` $LS(l)$ `if` c `.` is generated.

As in the discussion about the specification for the generation of critical pairs, we only present here the equations for the unconditional case.

We assume that for each variable V of sort S in the original module, there is a variable V`@`S' of sort S' for each subsort S' of S, and which is not used in the module.

```
fmod DESCENT-CHECK is
  pr MATCHING .
  pr EXT-BOOL .
```

Comparing the least sort for the sides of each equation is not enough. To check that an equation satisfies the descent condition we have to generate all the possible instances of it for each of the possible specializations of its variables, and we have to verify that in each of these instances the least sort of the term in the lefthand side is greater or equal than the least sort of the term in the righthand side. Given a set of equations, the following function `eqInstanceSet` generates all these instances.

```
  op eqInstanceSet : Module EquationSet -> EquationSet .
```

We do not want all these proof obligations. We only need the most general ones. As defined in Section 5.2, a membership assertion `mb` t `:` s `if` c `.` is more general than another membership assertion `mb` t' `:` s' `if` c' `.` if there exists a substitution σ such that $t\sigma = t'$, $s \leq s'$, and $c\sigma = c'$. The function `maximalMASet` returns the set of maximal elements in a set of membership assertions.

```
  op maximalMASet : MembAxSet Module -> MembAxSet .
  op maximalMASetAux : MembAx MembAxSet MembAxSet Module -> MembAxSet .
  op moreGeneralMA? : MembAx MembAx Module -> Bool .

  eq maximalMASet((MA MAS), M) = maximalMASetAux(MA, MAS, none, M) .
  eq maximalMASet(none, M) = none .

  eq maximalMASetAux(MA, (MA' MAS), MAS', M)
    = if moreGeneralMA?(MA, MA', M)
      then maximalMASetAux(MA, MAS, MAS', M)
      else if moreGeneralMA?(MA', MA, M)
           then maximalMASetAux(MA', (MAS MAS'), none, M)
           else maximalMASetAux(MA, MAS, (MA' MAS'), M)
           fi
      fi .
  eq maximalMASetAux(MA, none, (MA' MAS), M)
    = (MA maximalMASetAux(MA', MAS, none, M)) .
  eq maximalMASetAux(MA, none, none, M) = MA .

  eq moreGeneralMA?((mb T1 : S1 .), (mb T2 : S2 .), M)
    = match?(M, (eq T1 = T2 .)) and-then sortLeq(M, S1, S2) .
```

The `descentCheck` is then as follows.

```
  op descentCheck : Module -> MembAxSet .
  op descentCheck1 : Module EquationSet -> MembAxSet .

  eq descentCheck(M)
    = maximalMASet(
        descentCheck1(M, eqInstanceSet(M, equationSet(M))), M) .

  eq descentCheck1(M, ((eq T = T' .) EqS))
    = if sortLeq(M, leastSort(M, meta-reduce(M, T')), leastSort(M, T))
      then descentCheck1(M, EqS)
      else ((mb T' : leastSort(M, T) .) descentCheck1(M, EqS))
      fi .
  eq descentCheck1(M, none) = none .

   ...

endfm
```

The Integers Fail Confluence and Descent

Let us consider the following specification of the integers to illustrate the way in which the descent check is accomplished. Let `NAT` be the module defined in Section 5.3.2, which is imported by our module `INT`.

```
fmod INT is
   sorts Int Neg .
   protecting NAT .
   subsorts Zero < Neg < Int .
   subsort Nat < Int .
   ops s_ p_ : Int -> Int .
   op p_ : Neg -> Neg .
   ops _+_ _*_ : Int Int -> Int [comm] .
   op -_ : Int -> Int .
   op square : Int -> Nat .
   vars N M : Nat .
   vars I J : Int .
   vars P Q : Neg .
   eq s p I = I .
   eq p s I = I .
   eq I + 0 = I .
   eq I + s N = s (I + N) .
   eq I + p P = p (I + P) .
   eq I * 0 = 0 .
   eq I * s N = (I * N) + I .
   eq I * p P = (I * P) + - I .
   eq - 0 = 0 .
   eq - s N = p - N .
   eq - p P = s - P .
   eq square(I) = I * I .
endfm
```

As in the `NAT` example, this specification of `INT` is perfectly reasonable: it is ground-confluent, and its initial model is the ring of the integers with the usual operations, including also the successor, predecessor, and the square function. However:

- It is not confluent. For example, the equations

  ```
  eq I * s N = (I * N) + I .
  eq I * p P = (I * P) + - I .
  ```

  yield an unjoinable critical pair.

$$\begin{tikzcd}
& \texttt{s N * p P} \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\texttt{p (((N * P) + - N) + P)} & & \texttt{p (((N * P) + P) + - N)}
\end{tikzcd}$$

- Descent also fails. The equation

  ```
  eq square(I) = I * I .
  ```

  gives rise to a membership assertion, because the least sort of the term `square(I)` is `Nat`, but it is `Int` for the term in the righthand side.

To check descent, all the instances specializing the variables to smaller sorts are generated for each equation. Let us consider, for example, the instances for the equation

```
eq square(I) = I * I .
```

Note that for a variable `I` of sort `Int` the *preparation step* creates variables `I@Zero`, `I@Nat`, and `I@Neg`, of sorts `Zero`, `Nat`, and `Neg`, respectively. The following equations, specializing the original one, are generated:

```
eq square(I) = I * I .
eq square(I@Zero) = I@Zero * I@Zero .
eq square(I@Nat) = I@Nat * I@Nat .
eq square(I@Neg) = I@Neg * I@Neg .
```

For each of these instances the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side reduced to its normal form. The set of instances for which the descent condition is not satisfied is:

```
eq square(I) = I * I .
eq square(I@Neg) = I@Neg * I@Neg .
```

The proof obligations generated are:

```
mb I * I : Nat .
mb I@Neg * I@Neg : Nat .
```

It is easy to see that the first membership assertion is more general than the second one. Therefore, only the first is included in the output of the tool, together with three critical pairs.

```
Checking solution :
  var I : Int .
  var P : Neg .
  var N : Nat .
  var N@ : Nat .
  var P@ : Neg .
  cp s - P + (- P@ + (P * P@)) = s - P@ + (- P + (P * P@)) .
  cp p - N + (P@ + (N * P@)) = p (P@ + (- N + (N * P@))) .
  cp s ( N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
  mb I * I : Nat .
```

Regarding the critical pairs, the first one comes from the overlapping of the equation

```
eq I * s N = (I * N) + I .
```

with a renamed copy of itself at the top. The same critical pair is also generated in the overlapping of the equation

```
eq s N * M = M + (N * M) .
```

with itself at the top. Nevertheless, only one of them is taken; the other is eliminated in the last step of the process.

The second critical pair comes from the overlapping at the top of the equations

```
eq I * s N = (I * N) + I .
eq I * p Q = (I * Q) + - I .
```

And finally, the last of these critical pairs comes from the overlapping of the equation

```
eq I * p Q = (I * Q) + - I .
```

with a renamed copy of itself at the top.

We will revisit this example to find a confluent variant and to establish the ground-Church-Rosser property for it in Section 5.5.2.

5.3.4 The Church-Rosser Checker

In the present section we discuss the module `CRC`, which contains the definition of the function `checking`. This function checks the Church-Rosser property of an order-sorted equational specification satisfying the conditions mentioned in the previous sections. The checking for confluence and for descent is carried out by the functions `confluenceCheck` and `descentCheck` presented, respectively, in Sections 5.3.2 and 5.3.3.

```
fmod CRC is
  pr CONFLUENCE-CHECK .
  pr DESCENT-CHECK .

  var M : Module .
```

The output of the `checking` function is given as a 3-tuple consisting of a set of critical pairs, a set of membership assertions (corresponding to descent proof obligations) and the set of variables appearing in them. We define the following sort `CheckingSolution`.

```
  sorts CheckingSolution .
  op checkingSolution :
       VarDeclSet CritPairSet MembAxSet -> CheckingSolution .
```

As already mentioned in Sections 5.3.2 and 5.3.3, the modules passed to the `confluenceCheck` and `descentCheck` functions are previously *prepared* in order to get the variables needed in the respective processes. The function `prepareModule` adds to the module the variables needed in the checking. For each variable V of sort S in the original module a variable V`@`[5]—used in the generation of critical pairs—and a variable V`@`S' for each subsort S' of S—used in the descent checking—are created.

```
  op prepareModule : Module -> Module .
```

We define functions `varDeclSetInCritPairs` and `varDeclSetInMembAxs` to collect the set of variables in a set of critical pairs and in a set of membership assertions, respectively.

```
  op varDeclSetInCritPairs : Module CritPairSet -> VarDeclSet .
  op varDeclSetInMembAxs : Module MembAxSet -> VarDeclSet .
```

[5] As pointed out in Footnote 4, some other variables are added for the unification algorithm, namely, for each variable V of sort S in the original module a variable V`@@`S' for each subsort S' of S is also added.

Finally, the definition of the checking function is as follows.

```
  op checking : Module -> CheckingSolution .

  eq checking(M)
    = checkingSolution(
        (varDeclSetInCritPairs(prepareModule(M),
           confluenceCheck(prepareModule(M)))
         varDeclSetInMembAxs(prepareModule(M),
           descentCheck(prepareModule(M)))),
        confluenceCheck(prepareModule(M)),
        descentCheck(prepareModule(M))) .

  ...

endfm
```

5.4 Full Maude and the Church-Rosser Checker

We have presented in Section 5.3 the Maude specification of a Church-Rosser checker tool. Since this specification is executable, given the appropriate inputs, the different functions in it can be evaluated in Core Maude. However, that way of using the tool is quite inconvenient. Note that the argument of the `checking` function is a term of sort `Module`, that is, the metarepresentation of a module, and that the output returned is a term of sort `CheckingSolution`, which may contain critical pairs and membership assertions with terms at the metalevel. Just to illustrate what this means, we show below the interaction with the tool for the `NAT` specification presented in Section 5.3.2. Note that the output has been manually tabulated.

```
Maude> red checking(
             fmod 'NAT is
                including 'BOOL .
                sorts 'Zero ; 'Nat .
                subsort 'Zero < 'Nat .
                op '0 : nil -> 'Zero [none] .
                op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
                op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
                op 's_ : 'Nat -> 'Nat [none] .
                var 'M : 'Nat .
                var 'N : 'Nat .
                none
                eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
                eq '_+_[{'0}'Zero, 'N] = 'N .
                eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N,'M]] .
                eq '_*_[{'0}'Zero, 'N] = {'0}'Zero .
             endfm)
result CheckingSolution:
  checkingSolution(
    var 'N : 'Nat .
    var 'N@ : 'Nat .,
    cp('s_['_+_['N, '_+_['N@, '_*_['N, 'N@]]]],
       's_['_+_['N@, '_+_['N, '_*_['N, 'N@]]]]),
    none)
```

The interaction with the tool may become quite hard in this way. Notice that it is even worse in the case of the inductive theorem prover presented in [25, 24], because, as explained in Section 5.1.1, the design of the theorem prover requires two levels of reflection, that is, the interaction with the tool has to be made at the meta-metalevel of the modules on which the theorems are being proved.

In Section 4.10 we illustrated the flexibility of the Full Maude system by extending its module algebra with new module expressions. In the present section, we explain how the specification of the Church-Rosser checker, presented in the previous sections, can be integrated within the Full Maude environment, to allow checking of modules defined in Full Maude and to get a much more convenient user interface. Of course, it would have been possible to define an interface for the tool without integrating it with Full Maude. Since all the infrastructure built for Full Maude can be used by itself, just by selecting functions from that infrastructure in the needed modules, any of the two possibilities can give rise to an interface in a very short time. However, integrating the specifications of Full Maude and of the Church-Rosser checker we not only have such a needed infrastructure. With such an integration we can, for example, check the Church-Rosser property of any module in Full Maude's database. We can therefore use the tool on any module accepted by Full Maude, including structured modules, parameterized modules, etc. We still have, of course, the restrictions mentioned in the previous sections, that is, the module has to be functional and order-sorted, and each of the operation symbols in the specification has either no equational attributes or only the commutativity attribute, and the specification does not contain any built-in sort or function and has already been proved terminating using another tool. In the case of the inductive theorem prover, for example, we are not only able to prove theorems about any of the modules entered in Full Maude, satisfying certain conditions (see, i.e., [25]); in fact, we can keep the proof developed so far as part of the persistent state. We shall see in Section 5.4.2 how we can extend the persistent database object with some new attributes to keep the state of the proof.

5.4.1 The Signature

As explained in Section 4.2, and illustrated in Section 4.10.1, extensions to Full Maude often require extensions to the signature in which the input will be parsed. In the case of the Church-Rosser checker, we add declarations for two new commands, one of them to check a specific module in Full Maude's database, which is referred to by name, and another one to check the module by default, that is, the last module introduced in Full Maude.

Note that, as we did in Section 4.10.1, we introduce the operator declarations defining the new syntax in a module extending the module with the actual syntax. Thus, the following module `CRC-SIGN` does not extend the module `FULL-MAUDE-SIGN` as the module `EXT-FULL-MAUDE-SIGN`, presented in Section 4.10.1, but extends instead `EXT-FULL-MAUDE-SIGN` itself. In this way, all the extensions made in Section 4.10 are available as well.

```
fmod CRC-SIGN is
  including EXT-FULL-MAUDE-SIGN .

  op checking . : -> PreCommand .
  op checking_. : ModExp -> PreCommand .

endfm
```

As in Section 4.10.1, we define a new constant `CRC-GRAMMAR`, of sort `FModule`. This constant is identified with the metarepresentation of the module to be used in the parsing process, which extends the module identified with EXT-GRAMMAR. Note that EXT-GRAMMAR is the constant identified with the extension of GRAMMAR in Section 4.10.1, and GRAMMAR is the constant of sort FModule identified with the metarepresentation of module GRAMMAR, which imports the signature given in Appendix A and contains the declarations of the constructors for the bubble sorts. We have then the following module META-CRC-SIGN.

```
fmod META-CRC-SIGN is
  inc META-LEVEL .
  pr META-EXT-FULL-MAUDE-SIGN .

  op CRC-GRAMMAR : -> FModule .

  eq CRC-GRAMMAR = addImportList((including 'CRC-SIGN .), EXT-GRAMMAR) .

endfm
```

5.4.2 Extending the Database by Inheritance

In Section 4.9 we explained how the persistent state of the Full Maude system is given by a single object, which maintains the database of the system. Using object-oriented notation we can see this object as belonging to a class database defined as follows.

```
class database | db : Database, input : TermList,
                 output : QidList, default : ModId .
```

It is natural to think of extensions of the persistent database of Full Maude by defining subclasses of database inheriting its behavior and adding new functionalities to it, new attributes, etc. In the case of the Church-Rosser checker, we only want to add the new commands presented in the previous section. It is enough to add the following declarations, and the corresponding rules describing the behavior associated with the new commands.

```
class CRC .
subclass CRC < database .
```

In the case of the inductive theorem prover we want to keep the state of the proof as part of the persistent object, and then define the different commands acting on this proof. We do this by defining a subclass ITP of database with an attribute goal. The goal attribute is declared to be a value of sort Term, and keeps the state of the proof meta-metarepresented.

```
class ITP | goal : Term .
subclass ITP < database .
```

Notice that we can combine both tools by defining a new class, say CRC-ITP, inheriting from both CRC and ITP.

```
class CRC-ITP .
subclass CRC-ITP < CRC ITP .
```

In the following module, CRC-DATABASE-HANDLING, we extend the module EXT-DATABASE-HANDLING, presented in Section 4.10.4, with the declarations for the new class and with the rules describing the behavior associated with the new commands. Note that, as already pointed out in Section 4.9.1, the previous object-oriented declarations are given in their equivalent 'desugared' form.

```
mod CRC-DATABASE-HANDLING is
  pr EXT-DATABASE-HANDLING .
  pr CRC .

  var ME : ModExp .
  var DB : Database .
  var M : Module .
  var VDS : VarDeclSet .
  var CPS : CritPairSet .
  var MAS : MembAxSet .
  var MN : ModName .
  var QIL : QidList .
```

The 'desugared' object-oriented declarations presented above are as follows.

```
  sort CRC .
  subsort CRC < DatabaseClass .

  op CRC : -> CRC .

  var Atts : AttributeSet .
  var X@CRC : CRC .
  var O : Oid .
```

To be able to show the results of the checking, we need meta pretty printing functions. In Section 4.8.2 we saw how `meta-pretty-print` functions for sets of variable declarations and for sets of membership assertions are defined in module `DECL-META-PRETTY-PRINT`. We define here the following `meta-pretty-print` function for critical pairs.

```
  op meta-pretty-print : Module CritPairSet -> QidList .
```

The processing of the checking commands is handled by the following function `processCRChecking`. Its definition is very simple. Given a module expression and a database, it just calls the `checking` function presented in Section 5.3.4 with the flat form of the specified module in the database, and then gives the output as a list of quoted identifiers ready to be passed to the read-eval-print loop.

```
  op processCRChecking : ModExp Database -> QidList .
  op processCRCheckingAux : Module CheckingSolution -> QidList .

  eq processCRChecking(ME, DB)
    = ('\n 'Church-Rosser 'checking 'of meta-pretty-print(ME)
       processCRCheckingAux(getFlatUnit(ME, DB),
         checking(getFlatUnit(ME, DB)))) .

  eq processCRCheckingAux(M, checkingSolution(VDS, CPS, MAS))
    = if VDS == none and CPS == emptyCritPairSet and MAS == none
      then ('\n 'Checking 'solution ':
            '\n '\t 'The 'specification 'is 'Church-Rosser '.)
      else ('\n 'Checking 'solution ':
            meta-pretty-print(VDS)
            meta-pretty-print(addVarDeclSet(VDS, M), CPS)
            meta-pretty-print(addVarDeclSet(VDS, M), MAS))
      fi .
```

Finally, we give the rules describing the behavior of the CRC object associated to the new commands. In both cases the processing is carried out by the processCRChecking function, which is called with the specified module name, in the case of the checking_. command, or with the name of the default module, for the checking . command.

```
  rl [CRChecking] :
     < O : X@CRC | db : DB, input : ({'checking`.}'PreCommand),
        output : QIL, default : MN, Atts >
     => < O : X@CRC | db : DB, input : nilTermList,
           output : processCRChecking(MN, DB),
           default : MN, Atts > .
  rl [CRChecking] :
     < O : X@CRC | db : DB, input : ('checking_.[T]),
        output : QIL, default : MN, Atts >
     => < O : X@CRC | db : DB, input : nilTermList,
           output : processCRChecking(parseModExp(T), DB),
           default : MN, Atts > .

   ...

endm
```

5.4.3 The Full Maude Environment of the Checker

As we did in Section 4.10.4, we finish our specification redefining the FULL-MAUDE module presented in Section 4.9.2. The module EXT-FULL-MAUDE presented there only changed the in rule to make possible the use of the extended signature in the parsing of the inputs. In general, however, we do not only want to change this. We may want to add more predefined modules in the initial database, or to change the way in which the communication with the loop is accomplished. Once Maude has full support for objects and input/output, it will be possible to adapt the present specifications just by redefining these modules.

```
mod CRC-FULL-MAUDE is
  pr META-CRC-SIGN .
  pr CRC-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .
```

The constant crc is used as identifier for the object of class CRC.

```
  op crc : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@CRC : CRC .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
```

For the Church-Rosser checker and the theorem prover, we need to add a rule initializing the persistent object as an object of classes CRC and ITP, respectively. Notice that in the case of the inductive theorem prover, an initial value has to be specified also for the goal attribute.

```
rl [init] :
   init
   => [nil,
       < crc : CRC |
            db : evalUnit(CONFIGURATION,
                   evalUnit(TRIV,
                     evalUnit(UP, emptyDatabase))),
            input : nilTermList, output : nil,
            default : nullModName >,
       nil] .
```

The redefinition of the in rule uses the CRC-GRAMMAR to parse the inputs being entered to the integrated system.

```
crl [in] :
   [QIL,
    < O : X@CRC | db : DB, input : nilTermList,
         output : nil, default : MN, Atts >,
    QIL']
   => if meta-parse(CRC-GRAMMAR, QIL) == error*
      then [nil,
            < O : X@CRC | db : DB, input : nilTermList,
                 output : ('ERROR: 'incorrect 'input '.),
                 default : MN, Atts >,
            QIL']
      else [nil,
            < O : X@CRC | db : DB,
                 input : meta-parse(CRC-GRAMMAR, QIL),
                 output : nil, default : MN, Atts >,
            QIL']
      fi
      if QIL =/= nil .
```

The out rule remains as originally specified.

```
crl [out] :
   [QIL,
    < O : X@Database | db : DB, input : TL,
         output : QIL', default : MN, Atts >,
    QIL'']
   => [QIL,
       < O : X@Database | db : DB, input : TL,
            output : nil, default : MN, Atts >,
      (QIL' QIL'')]
      if QIL' =/= nil .
endm
```

5.5 How to Use the Church-Rosser Checker

This section illustrates with examples the use of the Church-Rosser checker tool, and suggests some methods that—using the feedback provided by the tool—can help the user establish that his/her specification is ground-Church-Rosser. For additional examples see [25].

We assume a context of use very different from the usual context for completing an equational theory. The starting point for completing a theory, say the theory of groups, is an equational theory that is *not* Church-Rosser. A Knuth-Bendix-like completion process then attempts to make it so by *automatically adding* new oriented equations.

In our case, however, we assume that the user has developed an *executable specification* of his/her intended system with an initial algebra semantics, and that this specification has already been *tested* with examples, so that the user is in fact confident that the specification is *ground-Church-Rosser*, and wants only to check this property with the tool.

Of course, the tool can only guarantee success when the user's specification is unconditional and Church-Rosser, and not just ground-Church-Rosser. That is, not generating any proof obligations is only a *sufficient* condition. But in many cases of interest—particularly for specifications with nontrivial sort orderings such as, for example, the specifications of the number hierarchy discussed in this section and in [25]—the specification will typically be *ground* Church-Rosser, but not Church-Rosser, so that a collection of critical pairs and of membership assertions will be returned by the tool as proof obligations.

An important methodological question is what to do, or not do, with these proof obligations. As the examples that we discuss illustrate, what should *not* be done is to let an automatic completion process add new equations to the user's specification in a mindless way. In some cases this is even impossible, because the critical pair in question cannot be oriented. In many cases it will certainly lead to a nonterminating process. In any case, it will modify the user's specification in ways that can make it difficult for the user to recognize the final result, if any, as intuitively equivalent to the original specification.

The feedback of the tool should instead be used as a guide for *careful thought* about one's specification. As several of the examples studied indicate, by analyzing the critical pairs returned, the user can understand why they could not be joined. It may be a mistake that must be corrected. More often, however, it is not a matter of a mistake, but of an axiom that is either *too general*—so that its very generality makes joining the critical pair impossible, because no more equations can apply to it—or *amenable to an equivalent formulation* that is unproblematic—for example, by reordering the parentheses for an operator that is ground-associative—or both. In any case, it is the user himself/herself who must study where the problem comes from, and how to fix it by modifying the specification. Interaction with the tool then provides a way of modifying the original specification and ascertaining whether the new version passes the test or is a good step towards that goal.

Since the user's specification has an *initial* algebra semantics and the property of interest is checking that it is *ground* Church-Rosser, the proof obligations returned by the tool are *inductive* proof obligations. Therefore, after having introduced some modifications that may already eliminate some of the critical pairs and membership assertions generated by the tool, the user may be left with proof obligations for which the best approach is not any further modification of the specification, but, instead, an inductive proof. This is illustrated in Section 5.5.2 with a specification of the integers with a function `square : Int -> Nat` for which a membership proof obligation

```
(∀I: Int) I * I : Nat
```

is generated by the tool. Provided that no equational lemmas are used in the proof, the inductive theorem prover presented in [25, 24] can be used for this purpose, and in fact succeeds in proving this proof obligation.

Inductive proof of the joinability of critical pairs is a thornier issue, for which we lack at present good methods. The problem is that, if we have a critical pair `cp(`t`, `t'`)` generated by the tool for a module M, proving inductively

$$M \vdash_{\text{ind}} t = t'$$

is *not* sufficient for ensuring joinability. This is because such a proof can add new equations as lemmas that may destroy the Church-Rosser property, so that joinability with those new equations does not guarantee joinability with the original equations. What must instead be proved inductively is

$$M \vdash_{\text{ind}} t \downarrow t'$$

but this requires new inductive methods that, as far as we know, have not yet been developed.

A related unresolved methodological issue is what to do with *conditional* critical pairs or membership assertions whose conditions are *unsatisfiable*. For example, the condition may be of the form

$$\mathtt{tt} = \mathtt{ff}$$

where `tt` and `ff` are constants in a user-defined Boolean sort. If we already *knew* that the specification was Church-Rosser, we could conclude from the fact that `tt` and `ff` are irreducible that they are different, so that the condition cannot be satisfied. But this is precisely what we need to prove. It is quite possible that a modular/hierarchical approach could be used, in conjunction with new inductive proof methods, to establish the unsatisfiability of such conditions and then discard the corresponding proof obligations. But such an approach has still to be developed.

5.5.1 Making the Natural Numbers Specification Confluent

In Section 5.3.2 we studied the confluence of a simple specification for the natural numbers. Let us reconsider it and analyze in more detail the checker's output, and how we can make the specification confluent. If we manage to make it confluent, since no membership assertions are generated by the tool, this would show that the specification is Church-Rosser and, a fortiori, ground-Church-Rosser.

The approach of classical completion systems consists in taking the critical pair generated and adding it to the specification after orienting it somehow. The user of the Church-Rosser checker tool can in some cases succeed by orienting a critical pair, adding it to the specification, checking termination, and resubmitting the modified specification to the checker. However, in this case it is clear that this critical pair cannot be oriented, since, if we were to orient it, its addition would turn the specification into a nonterminating one. The way to handle this problem in our approach consists in studying the specification, and, in particular, the equations generating the critical pairs, and trying to find a "smart" solution modifying the specification in a way as intuitive and as clear as possible, since it is the user who decides the modifications to apply. The key is then to give to the user the information needed to allow him/her to proceed as he/she thinks best.

As was seen in Section 5.3.2, only one critical pair is generated by the tool for the specification of the natural numbers presented there. We saw how this critical pair comes from the overlap of the equation

```
eq s(N) * M = M + (N * M) .
```

with a renamed copy of itself at the top, that is, its overlapping with this other one:

```
eq s N@ * M@ = M@ + (N@ * M@) .
```

The solution for the unification problem generating it is

$$\{(\texttt{M} \ \leftarrow\ \texttt{s N@});\ (\texttt{M@} \ \leftarrow\ \texttt{s N})\}.$$

This indicates that a term of the form `s N * s M` can be rewritten as follows.

$$\begin{tikzcd}[column sep=small] & \texttt{s N * s M} \arrow[dl, "*"'] \arrow[dr, "*"] & \\ \texttt{s (N + (M + (N * M)))} & & \texttt{s (M + (N + (N * M)))} \end{tikzcd}$$

Note that these terms cannot be further reduced, but their ground instances can be reduced; that is, the tool's output does not contradict the specification's ground confluence.

In many cases, if the specification is ground confluent, we can eliminate the critical pairs just by writing the equations in some other way, or perhaps by adding some new equations. As already mentioned, in other cases the best possibility may be to take the critical pair and add it as an equation after orienting it somehow, but, as we have explained, this critical pair cannot be oriented. In this case, what we can do is to rewrite the "problematic" equation in some other way. Looking to the unjoinable terms in the picture above, we can think of writing such equation as, for example,

```
eq s N * s M = s ((N + M) + (N * M)) .
```

Replacing the original equation by this one in the input to the Church-Rosser checker, the tool does not return any critical pair. This means that, once termination has been checked, we are guaranteed that the modified specification is confluent. Note the way in which we have associated the variables in the righthand side of the equation. If instead we were to modify the equation as, for example,

```
eq s N * s M = s (N + (M + (N * M))) .
```

we would be in exactly the same situation as before, and the same critical pair would again be generated. The point is that associativity of addition has not been declared as an attribute, and therefore the order of the parentheses is crucial for achieving confluence of the critical pair generated.

Since we do not have any proof obligation for descent we can conclude that the specification is Church-Rosser. The output of the tool is now the following:

```
Checking solution :
  The specification is Church-Rosser .
```

5.5.2 Proving the Integers Specification Ground-Church-Rosser

Let us continue now with the specification for the integers presented in Section 5.3.3. There, we discussed in detail from which equations each of the critical pairs were coming. Let us now apply our technique to eliminate them, so as to make the specification confluent. We will also see how the inductive theorem prover presented in [25, 24] can be used to take care of the membership proof obligation generated. After taking care in this way of all proof obligations we establish the ground-Church-Rosser property.

In Section 5.3.3 we saw that the critical pairs were produced by the equations

```
eq s N * M = M + (N * M) .
eq I * s N = (I * N) + I .
eq I * p Q = (I * Q) + - I .
```

in different ways. The first thing we should do is to apply the results in a modular way, that is, we should apply to the specifications the changes that have been applied on the specifications being imported. Therefore, we should incorporate the changes making the specification of the natural numbers confluent in the previous section. However, this does not necessarily mean that we are going to get fewer critical pairs or membership assertions. Notice that in the analysis of the critical pairs we saw how the first critical pair was generated from two different overlappings, one of them between one of the equations in `NAT` with itself, which was eliminated, and another one from one equation in `INT` also with a renamed copy of itself. Thus, in this example we get the same result from the tool. The solution generated is now as follows:

```
Checking solution :
  var I : Int .
  var P : Neg .
  var N : Nat .
  var P@ : Neg .
  var N@ : Nat .
  cp s - P + (- P@ + (P * P@)) = s - P@ + (- P + (P * P@)) .
  cp p - N + (P@ + (N * P@)) = p (P@ + (- N + (N * P@))) .
  cp s (N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
  mb I * I : Nat .
```

Observing the results and the considerations of Section 5.3.3, we realize that these critical pairs are generated for exactly the same reasons by which the critical pair in the specification for the natural numbers in Section 5.5.1 was generated. Therefore, we can make similar modifications in `INT`. Specifically, let us replace the two equations

```
eq I * s N = (I * N) + I .
eq I * p P = (I * P) + - I .
```

by the following three ones:

```
eq s N * s M = s ((N + M) + (N * M)) .
eq p P * s N = s ((P + - N) + (P * N)) .
eq p P * p Q = s ((- P + - Q) + (P * Q)) .
```

Calling the checker with these equations in the input module instead of the previous ones no critical pair is given in the output.

```
Checking solution :
  var I : Int .
  mb I * I : Nat .
```

Regarding the descent property, we need to prove inductively the membership assertion given. That is, we have to treat it as the proof obligation that has to be satisfied in order to be able to assert that the specification is ground-decreasing. In this case, we have to prove INT $\vdash_{\text{ind}}$ (∀I) I * I: Nat. This can be done using the theorem prover presented in [25, 24], as was shown in [25]. Therefore, we have successfully transformed our original INT specification into one that is confluent and ground-descending, and therefore ground-Church-Rosser.

5.5.3 Additional Examples

We present here several additional examples to illustrate the use of the tool. No attempt is made to study each of them in full detail, although we will make some comments about the results given by the tool and about how to make them Church-Rosser. Our intention is not to give a large collection of examples, but just a representative collection. Some more examples can be found in [24]. We give new specifications for natural numbers, integers, and rational numbers.

Natural Numbers (II)

We give now a specification for the natural numbers quite similar to the one presented in the manual of OBJ3 ([60], page 77), although with some modifications. In particular, the assoc attribute is removed from the declaration of the _+_ operator, and the declarations are changed so that no built-in operators are used. A new data type Boolean is defined in the following module BOOLEAN, the operator _>_ in NAT is defined of sort Boolean, and the equations in which the if_then_else_fi operator was used are replaced by equivalent sets of conditional equations.

```
(fmod BOOLEAN is
   sort Boolean .
   op true : -> Boolean .
   op false : -> Boolean .
 endfm)

(fmod NAT is
   pr BOOLEAN .
   sorts Nat NzNat Zero .
   subsorts Zero NzNat < Nat .
   op 0 : -> Zero .
   op s_ : Nat -> NzNat .
   op p_ : NzNat -> Nat .
   op _+_ : Nat Nat -> Nat [comm] .
   op _*_ : Nat Nat -> Nat .
   op _*_ : NzNat NzNat -> NzNat .
   op _>_ : Nat Nat -> Boolean .
   op d : Nat Nat -> Nat [comm] .
   op quot : Nat NzNat -> Nat .
   op gcd : NzNat NzNat -> NzNat [comm] .
   vars N M : Nat .
   vars N' M' : NzNat .
   eq p s N = N .
```

```
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq 0 * N = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0, N) = N .
  eq d(s N, s M) = d(N, M) .
  ceq quot(N, M') = s quot(d(N, M'), M') if N > M' == true .
  eq quot(M', M') = s 0 .
  ceq quot(N, M') = 0 if M' > N == true .
  eq gcd(N', N') = N' .
  ceq gcd(N', M') = gcd(d(N', M'), M') if N' > M' == true .
endfm)
```

We get the following result with the `checking .` command.

```
Maude> (checking .)

Church-Rosser checking of NAT
Checking solution :
  var M' : NzNat .
  var N' : NzNat .
  var M'@ : NzNat .
  var N@ : Nat .
  var N'@ : NzNat .
  var N@@NzNat : NzNat .
  ccp N'@ = gcd(N'@, d(N'@, N'@))
    if N'@ > N'@ == true .
  ccp s quot(d(M'@, N@), M'@) = 0
    if N@ > M'@ == true and M'@ > N@ == true .
  ccp s quot(d(N@@NzNat, N@@NzNat), N@@NzNat) = s 0
    if N@@NzNat > N@@NzNat == true .
  ccp s 0 = 0
    if M'@ > M'@ == true .
  ccp gcd(M'@, d(M'@, N'@)) = gcd(N'@, d(M'@, N'@))
    if N'@ > M'@ == true and M'@ > N'@ == true .
  cmb gcd(d(N', M'), M') : NzNat if N' > M' == true .
```

We have some critical pairs and one proof obligation for the descent property. Let us study them.

After a quick overview of the critical pairs generated we realize that we do not have to worry about most of them. We can see that the conditions in the critical pairs cannot be satisfied. Nevertheless, to eliminate conditional critical pairs with conditions that cannot be satisfied or even to reduce the conditions if possible is a tricky matter for which we do not yet have good methods at the moment. The difficult question is how to prove that two terms are not equal if we do not yet know whether our specification is confluent. Some research needs to be done in this direction before deciding how to handle these cases. For the time being the user will have to decide what to do with the critical pairs generated, and may have to consider a modified specification.

Let us look for example at the critical pair

```
  ccp s 0 = 0
    if M'@ > M'@ == true .
```

A natural number cannot be greater than itself, and the condition will never be satisfied. Therefore, in an intuitive sense we do not need to worry about this critical pair.

The same happens with the following two critical pairs:

```
ccp N'@ = gcd(N'@, d(N'@, N'@))
  if N'@ > N'@ == true .
ccp s quot(d(N@@NzNat, N@@NzNat), N@@NzNat) = s 0
  if N@@NzNat > N@@NzNat == true .
```

The conditions of the remaining two conditional critical pairs cannot be satisfied either, although in this case it is because the condition is given as a conjunction of two equalities that cannot be true simultaneously.

```
ccp s quot(d(M'@, N@), M'@) = 0
  if N@ > M'@ == true and M'@ > N@ == true .
ccp gcd(M'@, d(M'@, N'@)) = gcd(N'@, d(M'@, N'@))
  if N'@ > M'@ == true and M'@ > N'@ == true .
```

We have seen that none of the conditions in the critical pairs can be satisfied. Therefore, we can gain an intuitive assurance that our specification is ground confluent. But in order to have a full proof we still need to find proof methods to check the unsatisfiability of the condition in each of the critical pairs.

Let us study now the membership proof obligation

```
cmb gcd(d(N', M'), M') : NzNat if N' > M' == true .
```

It comes from the equation

```
ceq gcd(N', M') = gcd(d(N', M'), M') if N' > M' == true .
```

Since `gcd` has been declared as

```
op gcd : NzNat NzNat -> NzNat [comm] .
```

and the function `d` has been declared as

```
op d : Nat Nat -> Nat [comm] .
```

the term `gcd(d(N', M'), M')` does not have sort `NzNat`, which then gives rise to the proof obligation above. This membership assertion has to be proved to guarantee the descent of the specification. Unfortunately, the present inductive theorem prover does not handle proofs of conditional membership assertions, and, in addition, we also lack at present methods to handle the confluence assumption hidden in the equality of the condition.

Integers (II)

We give now a specification for the integers, also similar to that presented in the manual of OBJ ([60, page 77]). Let us suppose that the module `NAT` being imported is the one presented in the previous section. We apply to `INT` the same changes we made to `NAT`, namely, removing the `assoc` attribute for `_+_` and the occurrences of the `if_then_else_fi` operator.

```
(fmod INT is
    sorts Int NzInt .
    protecting NAT .
```

```
  subsort Nat < Int .
  subsorts NzNat < NzInt < Int .
  op -_ : Int -> Int .
  op -_ : NzInt -> NzInt .
  op _+_ : Int Int -> Int [comm] .
  op _*_ : Int Int -> Int .
  op _*_ : NzInt NzInt -> NzInt .
  op quot : Int NzInt -> Int .
  op gcd : NzInt NzInt -> NzNat [comm] .
  vars I J : Int .
  vars I' J' : NzInt .
  vars N' M' : NzNat .
  eq - - I = I .
  eq - 0 = 0 .
  eq I + 0 = I .
  eq M' + (- M') = 0 .
  ceq M' + (- N') = - d(N', M') if N' > M' == true .
  ceq M' + (- N') = d(N', M') if M' > N' == true .
  eq (- I) + (- J) = - (I + J) .
  eq I * 0 = 0 .
  eq 0 * I = 0 .
  eq I * (- J) = - (I * J) .
  eq (- J) * I = - (I * J) .
  eq quot(0,I') = 0 .
  eq quot(- I',J') = - quot(I',J') .
  eq quot(I',- J') = - quot(I',J') .
  eq gcd(- I',J') = gcd(I',J') .
 endfm)
```

The result generated by the tool is:

```
Church-Rosser checking of INT
Checking solution :
  var I : Int .
  var N'@NzNat : NzNat .
  var M'@NzNat : NzNat .
  var M'@ : NzNat .
  var N'@ : NzNat .
  var I@ : Int .
  var N'@NzNat@ : NzNat .
  var N@Nat@ : Nat .
  var M'@NzNat@ : NzNat .
  cp - (I@ + - I) = I + - I@ .
  cp I@ * I = I * I@ .
  ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
  ccp - d(M'@, N'@) = d(M'@, N'@)
    if N'@ > M'@ == true and M'@ > N'@ == true .
  ccp - d(N'@, N'@) = 0
    if N'@ > N'@ == true .
  ccp s s 0 = 0
    if 0 > M'@NzNat@ == true .
  ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
    if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
  ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
    if M'@NzNat@ > M'@NzNat@ == true .
  ccp s 0 = 0
    if M'@NzNat@ > M'@NzNat@ == true .
```

```
ccp d(N'@, N'@) = 0
  if N'@ > N'@ == true .
ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
   = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
  if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
  if N'@NzNat > M'@NzNat == true .
```

We have some critical pairs, among which only two are unconditional, and one proof obligation for descent. Let us study them.

None of the conditions in the conditional critical pairs can be satisfied—in an intuitive sense, but with the remaining proof obligation of showing that, indeed, those conditions are unsatisfiable—so, in some sense, we only need to worry about the unconditional critical pairs.

The critical pair

```
cp I@ * I = I * I@ .
```

is due to the fact that the `_*_` operator is not commutative. We can add this attribute, but taking care of adding it not only to the operators declared on sorts `Int` and `NzInt`, but also to the multiplication operators declared on `Nat` and `NzNat` in `NAT`. Note that this will cause the generation of new critical pairs as the ones discussed in Sections 5.3.2 and 5.3.3. We remove them by making the changes introduced there.

The second unconditional critical pair is the following.

```
cp - (I@ + - I) = I + - I@ .
```

A classical completion system would add the equation

```
eq - (I@ + - I) = I + - I@ .
```

to the specification, but this would introduce more critical pairs, and the process would have to be repeated several times without any guarantee of success or even of termination. However, looking at the critical pair we can easily see that the problem is that there is no way to reduce a sum where one of the arguments is a negative number. Moreover, if we try to proceed by adding the oriented critical pair, we obtain critical pairs in which the same situation appears. The question is that a term of the form `I + (- J)` cannot be reduced if `J` is of sort `Int`. But if `J` is a nonzero natural number then some equation could be applied, and if it is smaller than zero then it is of the form `- N'` for some `N'` of sort `NzNat`. But then `- (- N')` can be reduced to `N'`. The only thing we can say then is that probably the specification is confluent, although nothing can be concluded from these critical pairs. However, we can modify the specification not just to reduce the critical pairs, but in fact to avoid generating them. This can be easily done, just by changing the equation

```
eq (- I) + (- J) = - (I + J) .
```

by this other one:

```
eq (- N') + (- M') = - (N' + M') .
```

Assuming that we have checked the termination of the modified specification, after resubmitting it to the tool, we obtain the following result.

```
Church-Rosser checking of INT
Checking solution :
  var N'@NzNat : NzNat .
  var M'@NzNat : NzNat .
  var M'@ : NzNat .
  var N'@ : NzNat .
  var N'@NzNat@ : NzNat .
  var N@Nat@ : Nat .
  var M'@NzNat@ : NzNat .
  ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
  ccp - d(M'@, N'@) = d(M'@, N'@)
    if N'@ > M'@ == true and M'@ > N'@ == true .
  ccp - d(N'@, N'@) = 0
    if N'@ > N'@ == true .
  ccp s s 0 = 0
    if 0 > M'@NzNat@ == true .
  ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
    if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
  ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
    if M'@NzNat@ > M'@NzNat@ == true .
  ccp s 0 = 0
    if M'@NzNat@ > M'@NzNat@ == true .
  ccp d(N'@, N'@) = 0
    if N'@ > N'@ == true .
  ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
     = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
  cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
    if N'@NzNat > M'@NzNat == true .
```

All the critical pairs obtained are now conditional, with their conditions unsatisfiable. Therefore, modulo checking termination and the unsatisfiability of the conditions, we could conclude that it is confluent.

The conditional membership proof obligation is the same one we had for the specification of the natural numbers earlier in this section.

Rational Numbers

Finally we present a specification for the rational numbers, similar to that presented in the manual of OBJ ([60, page 78]). The specification imports the module `INT` with the modifications introduced in the previous section to make the specification Church-Rosser. The specification is modified, as `NAT` and `INT`, to make it satisfy the requirements of the tool, and the `comm` attribute is added to the declarations of the `_*_` operator.

```
(fmod RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [comm] .
  op _*_ : Rat Rat -> Rat [comm] .
```

```
  op _*_ : NzRat NzRat -> NzRat [comm] .
  vars I' J' : NzInt .
  vars R S : Rat .
  vars R' S' : NzRat .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  ceq J' / I' = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
     if gcd(J',I') =/= s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
  eq (S / R') * R = (R * S) / R' .
endfm)
```

The output of the tool is given in Appendix C.1. Studying in full detail this specification to make it Church-Rosser is a challenging problem. We are not going to do it here; we just present a general discussion about the methods that can be used for this purpose.

The output of the tool includes twenty seven critical pairs, out of which sixteen are conditional, and also eight membership assertions. Ten of the conditional critical pairs have nonsatisfiable conditions, and so, modulo checking the unsatisfiability of the conditions, we do not have to worry about them.

From the critical pairs

```
cp (0 * S'@) / R'@ = 0 .
cp (0 * R@) / R'@ = 0 * R@ .
```

we see that terms of the form `R * 0` cannot be further reduced. The same happens with terms of forms `s 0 * R` and `0 + R`, as can be observed in the critical pairs

```
cp S@ + (R@ * s 0) = S@ + R@ .
cp (0 + (R@ * R'@)) / R'@ = 0 + R@ .
```

We can eliminate these critical pairs by adding the following equations:

```
eq R * 0 = 0 .
eq R * s 0 = R .
eq R + 0 = R .
```

From the critical pairs

```
cp - - R@ / I@Int@NzInt = R@ / I@Int@NzInt .
cp - (S@ + (R@ * - R')) / R' = (- S@ + (R' * R@)) / R' .
```

we realize the need for the equations

```
eq - - R = R .
eq - R * S = - (R * S) .
```

After introducing these equations there are still twenty critical pairs and eight membership assertions left (see Appendix C.2). Ten of the sixteen conditional critical pairs have unsatisfiable conditions. Some of the critical pairs have been removed, but others may have been generated, or some of the critical pairs that already appeared may be simplified to a different form. Continuing with this process we can either show that some conditions are unsatisfiable or that some critical pairs are ground joinable by additional theorem proving methods, or we can gradually transform the specification into an equivalent one that is confluent, by making completely intuitive and clear changes, decided by the user himself/herself. Although it can be somewhat tedious we think that the advantages are clear. Regarding descent, the proof obligations have also to be either proved or avoided by an adequate transformation of the specification.

5.6 Concluding Remarks

Our experience in building and using the Church-Rosser checker described in this chapter, and the inductive theorem prover presented in [25], has been very encouraging. There are several natural extensions of these tools that we plan to pursue in the near future. The Church-Rosser checker should be extended in several directions. First, the set of equational attributes used to rewrite modulo should be extended to contain other axioms besides commutativity. Furthermore, using the results in [11, 12], the tool should be generalized so as to deal with membership equational logic specifications and not just order-sorted specifications. Completion should also be added as an additional facility, perhaps integrating CiME [32] as an auxiliary termination tool. Finally, this tool should also be extended from equational logic to rewriting logic, to check *coherence* [109] of rewrite theories, or to complete such theories so that they become coherent.

More broadly, the present experience suggests that the reflective approach that we have taken in building the present tools is a promising general methodology for building many other theorem proving and formal analysis tools based on formal systems for different logics. For example, a mechanization of linear logic in Maude has already been demonstrated [19]; and one could also build in this way a tool supporting temporal logic reasoning about nonexecutable properties of Maude modules or of systems defined in other languages.

An important added benefit of building a formal environment of tools this way is that, since each tool is a theory in rewriting logic, they are much easier to interoperate by just combining their corresponding rewrite theories. We have for example seen the usefulness of using the inductive theorem prover to prove proof obligations generated by the Church-Rosser checker. In general, using the reflective techniques and the flexible logical framework capabilities of rewriting logic, we hope to make good advances towards the goal of *formal interoperability* [81], that is, the capacity to move in a mathematically rigorous way across different formalizations, and to use in a rigorously integrated way the different tools supporting such formalizations.

Conclusions

We have proposed a logic-independent module algebra. Our proposal consists in defining module operations on structured theories that give structured theories as results. We have given with it a concrete semantics to structured or parameterized views, and to the instantiation of parameterized theories with parameterized views resulting in structured, and possibly parameterized, theories.

We have proposed the use of a reflective logic with good properties as a logical framework, such as rewriting logic, to specify such a module algebra. We have shown the good properties of our approach, among which we may highlight its flexibility, its extensibility, and that, since the module algebra is defined in the logic itself, this allows us to reason formally about it.

We have seen that rewriting logic not only allows us to define a module algebra as the one presented, but also, in a language with the characteristics of Maude, to also build whole execution environments for different languages and logics. In particular, we have shown in detail how the entire module algebra for Full Maude, as well as its user interface, can be specified in Maude, and how the specification can be easily extended with new module operations.

We have also presented a proving tool to check the Church-Rosser property of order-sorted equational specifications. We have seen how it has been easily integrated within the Full Maude environment, opening up the possibility of creating a theorem proving environment for Maude or for other languages for which different tools can be easily interoperated.

This thesis is, however, just the beginning. It opens up some interesting research directions that we would like to investigate:

- The module algebra of Chapter 2 should be further refined with theory inclusions and with diagrams corresponding to posets of such inclusions. Also, the module operations for Full Maude discussed in Chapters 3 and 4 are only fragments of a more general module algebra that could be defined in a logic-independent way for a wide range of logics, and could be specified in an executable way within the rewriting logic reflective framework to endow such logics with executable instances of the algebra.

  Some of the theory building operations defined in Chapter 2 have not been implemented yet. We plan to complete this setting to a general theory of structured module operations allowing both theories and modules, parameterization of theories and modules, parameterized views, etc.

- This unified setting will also be a natural candidate for a *generic* and logic-independent *module algebra*, applicable to a wide variety of logics. The idea is to use Maude as a formal metatool [26] to implement many other formal specification languages that can be executed in the framework logic. Then, such executable representations can be automatically extended with a module algebra by instantiating the generic module algebra for the logic in question.

  Also, since in the rewriting logic framework we can easily define mappings between logics reified as functions

  $$Module_{\mathcal{L}} \xrightarrow{\overline{\Phi}} Module_{\mathcal{L}'}$$

  these mappings will then become easily extensible in a homomorphic way to functions

  $$ModAlg[Module_{\mathcal{L}}] \xrightarrow{ModAlg[\overline{\Phi}]} ModAlg[Module_{\mathcal{L}'}]$$

that translate not only theories, but also *structured* theories.

- Besides this core generic module algebra of a logic-independent character, it seems very interesting to also simultaneously explore new module composition operations in the more concrete context of rewriting logic and Maude. If some new operations prove useful in practice and are of general applicability beyond rewriting logic, they could be expressed in a logic-independent way and could then be added to the generic module algebra, thus making the generic algebra itself, and not only Maude's module algebra, truly extensible.

- Another very important area is metalogical reasoning about module operations, proving metalevel properties, etc., and also exploiting theory structuring as a "divide and conquer" strategy in theorem proving. In this regard, there are two immediate open problems that can be suggested:

  - verifying properties of the Maude module algebra, and of the planned generic module algebra, using theorem proving tools and reflection, and
  - developing modular techniques for proving properties of structured and parameterized modules, both for rewriting logic and for other logics in a generic and logic-independent way.

- As pointed out in Section 5.6, there are very natural extensions of the Church-Rosser checker that we plan to accomplish, including: the extension to the case in which different combinations of axioms are supported, once we have unification modulo these axioms built-in in Maude; the extension to check coherence [109] between equations and rules in a rewrite theory; and also the addition of Knuth-Bendix and coherence completion as an additional facility.

Appendix A

The Signature of Full Maude

The Full Maude *system* is defined as a Core Maude module. That is, the entire semantics of Full Maude is defined and executed in Core Maude. The definition of the Full Maude system has been discussed in Chapters 3 and 4. Further details on the definition of the system, including the definition of all the internal functions implementing the system can be found in [42]. In particular, the *grammar* of the Full Maude *language*, that a user should follow to enter modules and commands, is itself a functional submodule of the overall Full Maude system specification. This allows giving the following specification of the Full Maude grammar in a form more perspicuous in certain ways than the corresponding BNF grammar form.

```
fmod SIGNS&VIEW-EXPRS is
  sorts Token NeTokenList Bubble
        SortToken Sort SortList SortDecl
        ViewToken ViewExp
        SubsortRel SubsortDecl
        OpDecl Attr AttrList Hook HookList .
  subsorts SortToken < Sort < SortList .
  subsort ViewToken < ViewExp .
  subsort Attr < AttrList .
  subsort Hook < HookList .

  op ((_)) : Token -> Token .

  *** extended sorts
  op _[_] : Sort ViewExp -> Sort [prec 40] .
  op __ : SortList SortList -> SortList [assoc] .
  op _,_ : ViewExp ViewExp -> ViewExp [assoc] .

  *** sort declaration
  op sorts_. : SortList -> SortDecl .
  op sort_. : SortList -> SortDecl .

  *** subsort declaration
  op subsort_. : SubsortRel -> SubsortDecl .
  op subsorts_. : SubsortRel -> SubsortDecl .
  op _<_ : SortList SortList -> SubsortRel .
  op _<_ : SortList SubsortRel -> SubsortRel .

  *** operator attributes
  op __ : AttrList AttrList -> AttrList [assoc] .
  op assoc : -> Attr .
  op associative : -> Attr .
  op comm : -> Attr .
  op commutative : -> Attr .
  op id:_ : Bubble -> Attr .
  op identity:_ : Bubble -> Attr .
  op left id:_ : Bubble -> Attr .
  op left identity:_ : Bubble -> Attr .
  op right id:_ : Bubble -> Attr .
  op right identity:_ : Bubble -> Attr .
  op strat(_) : NeTokenList -> AttrList .
  op strategy(_) : NeTokenList -> AttrList .
  op prec_ : Token -> Attr .
  op precedence_ : Token -> Attr .
  op gather(_) : NeTokenList -> Attr .
  op gathering(_) : NeTokenList -> Attr .
```

```
  op idem : -> Attr .
  op idempotent : -> Attr .

  op special(_) : HookList -> Attr .
  op __ : HookList HookList -> HookList [assoc] .
  op id-hook_ : Token -> Hook .
  op id-hook_(_) : Token NeTokenList -> Hook .
  op op-hook_(_:_->_) : Token Token NeTokenList Token -> Hook .
  op op-hook_(_: ->_) : Token Token Token -> Hook .
  op term-hook_(_) : Token Bubble -> Hook .

  *** operator declaration
  op op_: ->_. : Token Sort -> OpDecl .
  op op_: ->_[_]. : Token Sort AttrList -> OpDecl .
  op op_:_->_. : Token SortList Sort -> OpDecl .
  op op_:_->_[_]. : Token SortList Sort AttrList -> OpDecl .
  op ops_: ->_. : NeTokenList Sort -> OpDecl .
  op ops_: ->_[_]. : NeTokenList Sort AttrList -> OpDecl .
  op ops_:_->_. : NeTokenList SortList Sort -> OpDecl .
  op ops_:_->_[_]. : NeTokenList SortList Sort AttrList -> OpDecl .

endfm

fmod F&S-MODS&THS is
  including SIGNS&VIEW-EXPRS .

  sorts FDeclList SDeclList PreModule
        ImportDecl ModExp Parameter ParameterList
        ModuleName EquationDecl RuleDecl MembAxDecl
        VarDecl VarDeclList .
  subsort Parameter < ParameterList .
  subsorts Token < ModExp ModuleName .
  subsort VarDecl < VarDeclList .
  subsorts VarDecl ImportDecl SortDecl SubsortDecl
           OpDecl MembAxDecl EquationDecl VarDeclList < FDeclList .
  subsorts RuleDecl FDeclList < SDeclList .

  *** variable declaration
  op vars_:_. : NeTokenList Sort -> VarDecl .
  op var_:_. : NeTokenList Sort -> VarDecl .

  *** membership axiom declaration
  op mb_:_. : Bubble Sort -> MembAxDecl .
  op cmb_:_if_. : Bubble Sort Bubble -> MembAxDecl .

  *** equation declaration
  op eq_=_. : Bubble Bubble -> EquationDecl .
  op ceq_=_if_. : Bubble Bubble Bubble -> EquationDecl .

  *** rule declaration
  op rl[_] :_=>_. : Token Bubble Bubble -> RuleDecl .
  op crl[_]:_=>_if_. : Token Bubble Bubble Bubble -> RuleDecl .

  *** importation declaration
  op including_. : ModExp -> ImportDecl .
  op inc_. : ModExp -> ImportDecl .
  op protecting_. : ModExp -> ImportDecl .
  op pr_. : ModExp -> ImportDecl .
```

```
  *** parameterized module interface
  op _::_ : Token ModExp -> Parameter [prec 40 gather (e &)] .
  op _,_ : ParameterList ParameterList -> ParameterList [assoc] .

  op _[_] : Token ParameterList -> ModuleName .

  *** declaration list
  op __ : VarDeclList VarDeclList -> VarDeclList [assoc] .
  op __ : SDeclList SDeclList -> SDeclList [assoc] .
  op __ : FDeclList FDeclList -> FDeclList [assoc] .

  *** functional and system module and theory
  op fmod_is_endfm : ModuleName FDeclList -> PreModule .
  op mod_is_endm : ModuleName SDeclList -> PreModule .
  op fth_is_endfth : ModuleName FDeclList -> PreModule .
  op th_is_endth : ModuleName SDeclList -> PreModule .
endfm

fmod O-MODS&THS is
  including F&S-MODS&THS .

  sorts ClassDecl AttrDecl AttrDeclList
        SubclassDecl MsgDecl ODeclList .
  subsorts SDeclList MsgDecl SubclassDecl ClassDecl < ODeclList .
  subsort AttrDecl < AttrDeclList .

  *** object-oriented module and theory
  op omod_is_endom : ModuleName ODeclList -> PreModule .
  op oth_is_endoth : ModuleName ODeclList -> PreModule .

  *** class declaration
  op class_|_. : Sort AttrDeclList -> ClassDecl .
  op class_. : Sort -> ClassDecl .
  op _,_ : AttrDeclList AttrDeclList -> AttrDeclList [assoc] .
  op _:_ : Token Sort -> AttrDecl [prec 40] .

  *** subclass declaration
  op subclass_. : SubsortRel -> SubclassDecl .
  op subclasses_. : SubsortRel -> SubclassDecl .

  *** message declaration
  op msg_:_->_. : Token SortList Sort -> MsgDecl .
  op msgs_:_->_. : NeTokenList SortList Sort -> MsgDecl .
endfm

fmod MOD-EXPRS is
  including O-MODS&THS .

  sorts Map MapList .
  subsort Map < MapList .

  *** module expression
  op _*(_) : ModExp MapList -> ModExp .
  op _[_] : ModExp ViewExp -> ModExp .

  *** renaming maps
  op op_to_ : Token Token -> Map .
```

```
  op op_:_->_to_ : Token SortList Sort Token -> Map .
  op op_: ->_to_ : Token Sort Token -> Map .
  op op_to_[_] : Token Token AttrList -> Map .
  op op_:_->_to_[_] : Token SortList Sort Token AttrList -> Map .
  op op_: ->_to_[_] : Token Sort Token AttrList -> Map .
  op sort_to_ : Sort Sort -> Map .
  op label_to_ : Token Token -> Map .
  op class_to_ : Sort Sort -> Map .
  op attr_._to_ : Token Sort Token -> Map .
  op msg_to_ : Token Token -> Map .
  op msg_:_->_to_ : Token SortList Sort Token -> Map .
  op msg_: ->_to_ : Token Sort Token -> Map .

  op _,_ : MapList MapList -> MapList [assoc prec 42] .
endfm

fmod VIEWS is
  including O-MODS&THS .

  sorts ViewDecl ViewDeclList PreView .
  subsorts VarDecl < ViewDecl < ViewDeclList .
  subsort VarDeclList < ViewDeclList .

  *** view maps
  op op_to term_. : Bubble Bubble -> ViewDecl .
  op op_to_. : Token Token -> ViewDecl .
  op op_:_->_to_. : Token SortList Sort Token -> ViewDecl .
  op op_: ->_to_. : Token Sort Token -> ViewDecl .
  op sort_to_. : Sort Sort -> ViewDecl .
  op class_to_. : Sort Sort -> ViewDecl .
  op attr_._to_. : Token Sort Token -> ViewDecl .
  op msg_to_. : Token Token -> ViewDecl .
  op msg_:_->_to_. : Token SortList Sort Token -> ViewDecl .
  op msg_: ->_to_. : Token Sort Token -> ViewDecl .

  *** view
  op view_from_to_is_endv :
        ViewToken ModExp ModExp ViewDeclList -> PreView .
  op __ : ViewDeclList ViewDeclList -> ViewDeclList [assoc] .
endfm

fmod COMMANDS is
  including MOD-EXPRS .

  sorts PreCommand .

  *** down function
  op down_:_ : ModExp PreCommand -> PreCommand .

  *** reduce command
  op red_. : Bubble -> PreCommand .
  op red-in_:_. : ModExp Bubble -> PreCommand .
  op reduce_. : Bubble -> PreCommand .
  op reduce-in_:_. : ModExp Bubble -> PreCommand .

  *** rewrite command
  op rew_. : Bubble -> PreCommand .
  op rew[_]_. : Token Bubble -> PreCommand .
```

```
  op rew-in_:_. : ModExp Bubble -> PreCommand .
  op rew-in[_]_:_. : Token ModExp Bubble -> PreCommand .
  op rewrite_. : Bubble -> PreCommand .
  op rewrite[_]_. : Token Bubble -> PreCommand .
  op rewrite-in_:_. : ModExp Bubble -> PreCommand .
  op rewrite-in[_]_:_. : Token ModExp Bubble -> PreCommand .

  *** select command
  op select_. : ModExp -> PreCommand .

  *** show commands
  op show module . : -> PreCommand .
  op show module_. : ModExp -> PreCommand .
  op show all . : -> PreCommand .
  op show all_. : ModExp -> PreCommand .
  op show sorts . : -> PreCommand .
  op show sorts_. : ModExp -> PreCommand .
  op show ops . : -> PreCommand .
  op show ops_. : ModExp -> PreCommand .
  op show vars . : -> PreCommand .
  op show vars_. : ModExp -> PreCommand .
  op show mbs . : -> PreCommand .
  op show mbs_. : ModExp -> PreCommand .
  op show eqns . : -> PreCommand .
  op show eqns_. : ModExp -> PreCommand .
  op show rls . : -> PreCommand .
  op show rls_. : ModExp -> PreCommand .
  op show view_. : ViewExp -> PreCommand .
  op show modules . : -> PreCommand .
  op show views . : -> PreCommand .
endfm

fmod FULL-MAUDE-SIGN is
  including VIEWS .
  including COMMANDS .
endfm
```

Appendix B

Core Maude Predefined Modules

This Appendix contains the signature of the Maude 1.00 standard prelude. See [21] for a detailed discussion of each of the modules in this prelude.

```
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [special (...)] .
  op false : -> Bool [special (...)] .
endfm

fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
        [special (...)] .
  op _==_ : Universal Universal -> Bool
        [prec 51 special (...)] .
  op _=/=_ : Universal Universal -> Bool
        [prec 51 special (...)] .
endfm

fmod BOOL is
  protecting TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq false xor A = A .
  eq A xor A = false .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor true .
  eq A or B = A and B xor A xor B .
  eq A implies B = not(A xor A and B) .
endfm

set include BOOL on .

fmod IDENTICAL is
  op _===_ : Universal Universal -> Bool
        [prec 51 strat (0) special (...)] .
  op _=/==_ : Universal Universal -> Bool
        [prec 51 strat (0) special (...)] .
endfm

fmod MACHINE-INT is
  sorts MachineInt NzMachineInt .
  subsort NzMachineInt < MachineInt .
  op <MachineInts> : -> NzMachineInt [special (...)] .
  op <MachineInts> : -> MachineInt [special (...)] .
  op -_ : MachineInt -> MachineInt
      [prec 15 special (...)] .     *** minus
  op -_ : NzMachineInt -> NzMachineInt
      [prec 15 special (...)] .     *** minus
  op ~_ : MachineInt -> MachineInt
      [prec 15 special (...)] .     *** bitwise complement
```

```
  op _+_ : MachineInt MachineInt -> MachineInt
      [prec 33 gather (E e) special (...)] . *** addition
  op _-_ : MachineInt MachineInt -> MachineInt
      [prec 33 gather (E e) special (...)] . *** difference
  op _*_ : MachineInt MachineInt -> MachineInt
      [prec 31 gather (E e) special (...)] . *** multiplication
  op _*_ : NzMachineInt NzMachineInt -> NzMachineInt
      [prec 31 gather (E e) special (...)] . *** multiplication
  op _/_ : MachineInt NzMachineInt -> MachineInt
      [prec 31 gather (E e) special (...)] . *** division
  op _%_ : MachineInt NzMachineInt -> MachineInt
      [prec 31 gather (E e) special (...)] . *** remainder
  op _&_ : MachineInt MachineInt -> MachineInt
      [prec 53 gather (E e) special (...)] . *** bitwise and
  op _|_ : MachineInt MachineInt -> MachineInt
      [prec 57 gather (E e) special (...)] . *** bitwise or
  op _|_ : NzMachineInt NzMachineInt -> NzMachineInt
      [prec 57 gather (E e) special (...)] . *** bitwise or
  op _^_ : MachineInt MachineInt -> MachineInt
      [prec 55 gather (E e) special (...)] . *** bitwise xor
  op _>>_ : MachineInt MachineInt -> MachineInt
      [prec 35 gather (E e) special (...)] . *** left shift
  op _<<_ : MachineInt MachineInt -> MachineInt
      [prec 35 gather (E e) special (...)] . *** right shift
  op _<_ : MachineInt MachineInt -> Bool
      [prec 37 special (...)] .     *** less than
  op _<=_ : MachineInt MachineInt -> Bool
      [prec 37 special (...)] .     *** less or equal than
  op _>_ : MachineInt MachineInt -> Bool
      [prec 37 special (...)] .     *** greater than
  op _>=_ : MachineInt MachineInt -> Bool
      [prec 37 special (...)] .     *** greater or equal than
endfm

fmod QID is
  protecting MACHINE-INT .
  sort Qid .
  op <Qids> : -> Qid [special (...)] .
  op conc : Qid Qid -> Qid [special (...)] .
  op index : Qid MachineInt -> Qid [special (...)] .
  op strip : Qid -> Qid [special (...)] .
endfm

fmod QID-LIST is
  protecting QID .
  sort QidList .
  subsort Qid < QidList .
  op nil : -> QidList .
  op __ : QidList QidList -> QidList [assoc id: nil] .
endfm

fmod META-LEVEL is
  protecting QID-LIST .
  sorts FModule Module ModuleExpression Import ImportList
        QidSet MachineIntList Sort SortDecl SubsortDecl SubsortDeclSet Attr
        AttrSet OpDecl OpDeclSet VarDecl VarDeclSet Term TermList Equation
        EquationSet Rule RuleSet MembAx MembAxSet Assignment Substitution
```

```
      ResultPair Hook HookList .
  subsort FModule < Module .
  subsort Import < ImportList .
  subsort Qid < ModuleExpression .
  subsort Qid < QidSet .
  subsort MachineInt < MachineIntList .
  subsort Qid < Sort .
  subsort SubsortDecl < SubsortDeclSet .
  subsort Attr < AttrSet .
  subsort OpDecl < OpDeclSet .
  subsort VarDecl < VarDeclSet .
  subsort Qid < Term .
  subsort Term < TermList .
  subsort Equation < EquationSet .
  subsort Rule < RuleSet .
  subsort MembAx < MembAxSet .
  subsort Assignment < Substitution .
  subsort Hook < HookList .

  op _[_] : Qid TermList -> Term .
  op {_}_ : Qid Qid -> Term .
  op _,_ : TermList TermList -> TermList [assoc gather (e E) prec 120] .
  op _:_ : Term Qid -> Term .
  op _::_ : Term Qid -> Term .
  op none : -> QidSet .
  op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .
  op nil : -> MachineIntList .
  op __ : MachineIntList MachineIntList -> MachineIntList [assoc id: nil] .

  op fmod_is_______endfm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule
      [gather (& & & & & & & &)] .
  op mod_is________endm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet RuleSet -> Module
      [gather (& & & & & & & & &)] .
  op nil : -> ImportList .
  op __ : ImportList ImportList -> ImportList [assoc id: nil] .
  op including_. : ModuleExpression -> Import .

  op sorts_. : QidSet -> SortDecl .
  op subsort_<_. : Qid Qid -> SubsortDecl .
  op none : -> SubsortDeclSet .
  op __ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
      [assoc comm id: none] .

  op (op_:_->_[_].) : Qid QidList Qid AttrSet -> OpDecl .
  op none : -> OpDeclSet .
  op __ : OpDeclSet OpDeclSet -> OpDeclSet [assoc comm id: none] .

  op none : -> AttrSet .
  op __ : AttrSet AttrSet -> AttrSet [assoc comm id: none] .
  op assoc : -> Attr .
  op comm : -> Attr .
  op idem : -> Attr .
  op id : Term -> Attr .
  op left-id : Term -> Attr .
  op right-id : Term -> Attr .
```

```
  op strat : MachineIntList -> Attr .
  op memo : -> Attr .
  op prec : MachineInt -> Attr .
  op gather : QidList -> Attr .

  op special : HookList -> Attr .
  op __ : HookList HookList -> HookList [assoc] .
  op id-hook : Qid QidList -> Hook .
  op op-hook : Qid Qid QidList Qid -> Hook .
  op term-hook : Qid Term -> Hook .

  op var_:_. : Qid Qid -> VarDecl .
  op none : -> VarDeclSet .
  op __ : VarDeclSet VarDeclSet -> VarDeclSet [assoc comm id: none] .

  op mb_:_. : Term Qid -> MembAx .
  op cmb_:_if_=_. : Term Qid Term Term -> MembAx .
  op none : -> MembAxSet .
  op __ : MembAxSet MembAxSet -> MembAxSet [assoc comm id: none] .

  op eq_=_. : Term Term -> Equation .
  op ceq_=_if_=_. : Term Term Term Term -> Equation .
  op none : -> EquationSet .
  op __ : EquationSet EquationSet -> EquationSet [assoc comm id: none] .

  op rl[_]:_=>_. : Qid Term Term -> Rule .
  op crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule .
  op none : -> RuleSet .
  op __ : RuleSet RuleSet -> RuleSet [assoc comm id: none] .

  op _<-_ : Qid Term -> Assignment .
  op none : -> Substitution .
  op _;_ : Substitution Substitution -> Substitution [assoc comm id: none] .
  op {_,_} : Term Substitution -> ResultPair .

  op error* : -> Term .
  op errorSort : QidSet -> Sort .

  op meta-reduce : Module Term -> Term [special (...)] .
  op meta-rewrite : Module Term MachineInt -> Term [special (...)] .
  op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair
      [special (...)] .
  op meta-parse : Module QidList -> Term [special (...)] .
  op meta-pretty-print : Module Term -> QidList [special (...)] .
  op sortLeq : Module Sort Sort -> Bool [special (...)] .
  op sameComponent : Module Sort Sort -> Bool [special (...)] .
  op leastSort : Module Term -> Sort [special (...)] .
  op lesserSorts : Module Sort -> QidSet [special (...)] .
  op glbSorts : Module Sort Qid -> QidSet [special (...)] .
endfm

mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System [special (...)] .
endm
```

Appendix C

Church-Rosser checking of RAT

C.1 Church-Rosser Checking of the Original RAT Specification

The result given by the Church-Rosser checker tool for the original RAT specification presented in Section 5.5.3 is the following.

```
Church-Rosser checking of RAT
Checking solution :
  var S' : NzRat .
  var I' : NzInt .
  var N'@NzNat : NzNat .
  var M'@NzNat : NzNat .
  var J' : NzInt .
  var N'@NzNat@ : NzNat .
  var N@Nat@ : Nat .
  var M'@NzNat@ : NzNat .
  var J'@NzNat : NzNat .
  var I'@NzNat : NzNat .
  var R : Rat .
  var R' : NzRat .
  var S@ : Rat .
  var S'@ : NzRat .
  var R@ : Rat .
  var R'@ : NzRat .
  var R'@NzInt : NzInt .
  var R'@@NzInt : NzInt .
  var R@NzInt : NzInt .
  var R@@NzInt : NzInt .
  var I@Int@NzInt : NzInt .
  var S'@NzInt : NzInt .
  var S'@@NzInt : NzInt .
  var S@@NzInt : NzInt .
  cp S@ + (R@ * s 0) = S@ + R@ .
  cp R@ / (S'@ * - R') = - R@ / (R' * S'@) .
  cp - - R@ / I@Int@NzInt = R@ / I@Int@NzInt .
  cp - (S@ + (R@ * - R')) / R' = (- S@ + (R' * R@)) / R' .
  cp - (S@ * R@) / R' = (R@ * - S@) / R' .
  cp (R + (R' * (R@ * R'@))) / (R' * R'@)
     = (R + (R@ * (R' * R'@))) / (R' * R'@) .
  cp (0 + (R@ * R'@)) / R'@ = 0 + R@ .
  cp (S' * ((S' * S@) + (R' * R@))) / (S' * R')
     = ((S' * S@) + (R' * R@)) / R' .
  cp (S' * (S@ * R@)) / R' = (R@ * (S' * S@)) / R' .
  cp (0 * S'@) / R'@ = 0 .
  cp (0 * R@) / R'@ = 0 * R@ .
  ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
  ccp R@@NzInt
     = quot(R@@NzInt, gcd(R@@NzInt, s 0)) / quot(s 0, gcd(R@@NzInt, s 0))
    if gcd(R@@NzInt, s 0) =/= s 0 .
  ccp - d(N'@NzNat@, M'@NzNat@) = d(N'@NzNat@, M'@NzNat@)
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
  ccp - d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
  ccp R@@NzInt / (S'@ * R'@@NzInt)
     = (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))) / S'@
```

```
  if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
       / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
   = - (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
         / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt)))
  if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
       / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
   = quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / - quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  if gcd(R@@NzInt, - R'@@NzInt) =/= s 0 .
ccp (S@@NzInt + (R@ * R'@@NzInt)) / R'@@NzInt
   = R@ + (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
           / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
  if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp (R@ * S'@@NzInt) / R'@@NzInt
   = R@ / (quot(R'@@NzInt, gcd(R'@@NzInt, S'@@NzInt))
           / quot(S'@@NzInt, gcd(R'@@NzInt, S'@@NzInt)))
  if gcd(R'@@NzInt, S'@@NzInt) =/= s 0 .
ccp (R@ * S@@NzInt) / R'@@NzInt
   = R@ * (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
           / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
  if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp s s 0 = 0
  if 0 > M'@NzNat@ == true .
ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
  if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
  if M'@NzNat@ > M'@NzNat@ == true .
ccp s 0 = 0
  if M'@NzNat@ > M'@NzNat@ == true .
ccp d(M'@NzNat@, M'@NzNat@) = 0
  if M'@NzNat@ > M'@NzNat@ == true .
ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
   = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
  if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
mb R@NzInt / (R'@NzInt * S'@NzInt) : NzRat .
mb - R@NzInt / R'@NzInt : NzRat .
mb (R@NzInt * S'@NzInt) / R'@NzInt : NzRat .
cmb quot(J', gcd(J', I')) / quot(I', gcd(J', I')) : NzRat
   if gcd(J', I') =/= s 0 .
cmb quot(J', gcd(J', I'@NzNat))
      / quot(I'@NzNat, gcd(J', I'@NzNat)) : NzRat
   if gcd(J', I'@NzNat) =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I'))
      / quot(I', gcd(J'@NzNat, I')) : NzRat
   if gcd(J'@NzNat, I') =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I'@NzNat))
      / quot(I'@NzNat, gcd(J'@NzNat, I'@NzNat)) : NzRat
   if gcd(J'@NzNat, I'@NzNat) =/= s 0 .
cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
   if N'@NzNat > M'@NzNat == true .
```

C.2 Church-Rosser Checking of the Modified RAT Specification

The modified RAT specification discussed in Section 5.5.3, and the result given by the Church-Rosser checker tool for it are as follows.

```
(fmod RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [comm] .
  op _*_ : Rat Rat -> Rat [comm] .
  op _*_ : NzRat NzRat -> NzRat [comm] .
  vars I' J' : NzInt .
  vars R S : Rat .
  vars R' S' : NzRat .
  eq R * 0 = 0 .
  eq R * s 0 = R .
  eq R + 0 = R .
  eq - - R = R .
  eq - R * S = - (R * S) .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  ceq J' / I' = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
     if gcd(J',I') =/= s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
  eq (S / R') * R = (R * S) / R' .
 endfm)

Maude> (checking .)

  Church-Rosser checking of RAT
  Checking solution :
    var S' : NzRat .
    var I' : NzInt .
    var N'@NzNat : NzNat .
    var M'@NzNat : NzNat .
    var J' : NzInt .
    var N'@NzNat@ : NzNat .
    var N@Nat@ : Nat .
    var M'@NzNat@ : NzNat .
    var J'@NzNat : NzNat .
    var I'@NzNat : NzNat .
    var R : Rat .
    var R' : NzRat .
    var S@ : Rat .
    var S'@ : NzRat .
```

```
var R@ : Rat .
var R'@ : NzRat .
var R'@NzInt : NzInt .
var R'@@NzInt : NzInt .
var R@NzInt : NzInt .
var R@@NzInt : NzInt .
var S'@NzInt : NzInt .
var S'@@NzInt : NzInt .
var S@@NzInt : NzInt .
cp - (S@ + - (R' * R@)) / R' = (- S@ + (R' * R@)) / R' .
cp (R + (R' * (R@ * R'@))) / (R' * R'@)
  = (R + (R@ * (R' * R'@))) / (R' * R'@) .
cp (S' * ((S' * S@) + (R' * R@))) / (S' * R')
  = ((S' * S@) + (R' * R@)) / R' .
cp (S' * (S@ * R@)) / R' = (R@ * (S' * S@)) / R' .
ccp N'@NzNat@ = gcd(N'@NzNat@, d(N'@NzNat@, N'@NzNat@))
    if N'@NzNat@ > N'@NzNat@ == true .
ccp R@@NzInt
  = quot(R@@NzInt, gcd(R@@NzInt, s 0)) / quot(s 0, gcd(R@@NzInt, s 0))
    if gcd(R@@NzInt, s 0) =/= s 0 .
ccp - d(N'@NzNat@, M'@NzNat@) = d(N'@NzNat@, M'@NzNat@)
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
ccp - d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp R@@NzInt / (S'@ * R'@@NzInt)
  = (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))) / S'@
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
        / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  = - (quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
         / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt)))
    if gcd(R@@NzInt, R'@@NzInt) =/= s 0 .
ccp - quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
        / quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
  = quot(R@@NzInt, gcd(R'@@NzInt, R@@NzInt))
      / - quot(R'@@NzInt, gcd(R'@@NzInt, R@@NzInt))
    if gcd(R@@NzInt, - R'@@NzInt) =/= s 0 .
ccp (S@@NzInt + (R@ * R'@@NzInt)) / R'@@NzInt
  = R@ + (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp (R@ * S'@@NzInt) / R'@@NzInt
  = R@ / (quot(R'@@NzInt, gcd(R'@@NzInt, S'@@NzInt))
          / quot(S'@@NzInt, gcd(R'@@NzInt, S'@@NzInt)))
    if gcd(R'@@NzInt, S'@@NzInt) =/= s 0 .
ccp (R@ * S@@NzInt) / R'@@NzInt
  = R@ * (quot(S@@NzInt, gcd(R'@@NzInt, S@@NzInt))
          / quot(R'@@NzInt, gcd(R'@@NzInt, S@@NzInt)))
    if gcd(S@@NzInt, R'@@NzInt) =/= s 0 .
ccp s s 0 = 0
    if 0 > M'@NzNat@ == true .
ccp s quot(d(N@Nat@, M'@NzNat@), M'@NzNat@) = 0
    if N@Nat@ > M'@NzNat@ == true and M'@NzNat@ > N@Nat@ == true .
ccp s quot(d(M'@NzNat@, M'@NzNat@), M'@NzNat@) = s 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp s 0 = 0
```

```
    if M'@NzNat@ > M'@NzNat@ == true .
ccp d(M'@NzNat@, M'@NzNat@) = 0
    if M'@NzNat@ > M'@NzNat@ == true .
ccp gcd(M'@NzNat@, d(N'@NzNat@, M'@NzNat@))
  = gcd(N'@NzNat@, d(N'@NzNat@, M'@NzNat@))
    if N'@NzNat@ > M'@NzNat@ == true and M'@NzNat@ > N'@NzNat@ == true .
mb R@NzInt / (R'@NzInt * S'@NzInt) : NzRat .
mb - R@NzInt / R'@NzInt : NzRat .
mb (R@NzInt * S'@NzInt) / R'@NzInt : NzRat .
cmb quot(J', gcd(J', I')) / quot(I', gcd(J', I')) : NzRat
  if gcd(J', I') =/= s 0 .
cmb quot(J', gcd(J', I'@NzNat))
      / quot(I'@NzNat, gcd(J', I'@NzNat)) : NzRat
  if gcd(J', I'@NzNat) =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I'))
      / quot(I', gcd(J'@NzNat, I')) : NzRat
  if gcd(J'@NzNat, I') =/= s 0 .
cmb quot(J'@NzNat, gcd(J'@NzNat, I'@NzNat))
      / quot(I'@NzNat, gcd(J'@NzNat, I'@NzNat)) : NzRat
  if gcd(J'@NzNat, I'@NzNat) =/= s 0 .
cmb gcd(d(N'@NzNat, M'@NzNat), M'@NzNat) : NzNat
  if N'@NzNat > M'@NzNat == true .
```

Bibliography

[1] Michael Barr and Charles Wells. *Toposes, Triples and Theories.* Springer-Verlag, 1985.

[2] Hubert Baumeister. Unifying initial and loose semantics of parameterized specifications in an arbitrary institution. In Samson Abramsky and Tom Maibaum, editors, *Proc. Int. Conf. on Theory and Practice of Software Development, TAPSOFT'91, Brighton, UK, April 1991*, volume 493 of *Lecture Notes in Computer Science*, pages 103–120. Springer-Verlag, 1991.

[3] Jan Bergstra, Jan Heering, and Paul Klint. *Algebraic Specification.* ACM Press, 1989.

[4] Jan Bergstra, Jan Heering, and Paul Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.

[5] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Seventh Colloquium on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[6] Didier Bert and Rachid Echahed. Design and implementation of a generic, logic and functional programming language. In Bernard Robinet and Reinhard Wilhelm, editors, *Proceedings of European Symposium on Programming ESOP'86*, volume 213 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 1986.

[7] Michel Bidoit, Marie-Claude Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of ComputerProgramming*, 12(1):1–38, 1989.

[8] Michel Bidoit, Hans-Jörg Kreowski, Pierre Lescanne, Fernando Orejas, and Donald Sannella, editors. *Algebraic System Specification and Development. A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[9] Peter Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer, 1995.

[10] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[11] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science* Available at `http://maude.csl.sri.com`.

[12] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

[13] Roberto Bruni, José Meseguer, and Ugo Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.

[14] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[15] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjørner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.

[16] CafeOBJ-Project. *Proceedings of the CafeOBJ Symposium'98, Numazu, Japan.* April 1998.

[17] Carlos Castro. An approach to solving binary CSP using computational systems. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[18] Maura Cerioli and José Meseguer. May I borrow your logic? (Transporting logical structure along maps). *Theoretical Computer Science*, 173:311–347, 1997.

[19] Manuel Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, University of Navarre, 1998.

[20] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [65]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[21] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, March 1999. Available at `http://maude.csl.sri.com`.

[22] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude as a metalanguage. In Kirchner and Kirchner [65]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[23] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[24] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

[25] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Manuscript, SRI International, February 1998.

[26] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Maude as a formal meta-tool. Submitted for publication, February 1999.

[27] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[28] Manuel Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288, 1996.

[29] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[30] Manuel Clavel and José Meseguer. Internal strategies in a reflective logic. In B. Gramlich and Hélène Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

[31] CoFI Task Group on Language Design. CASL—The common algebraic specification language, version 1.0. Available at `http://www.brics.dk/Projects/CoFI`, October 1998.

[32] Evelyne Contejean and Claude Marché. The CiME system: tutorial and user's manual. Manuscript, Université Paris-Sud, Centre d'Orsay.

[33] Felix Cornelius, Michael Baldamus, Hartmut Ehrig, and Fernando Orejas. Abstract and behaviour module specifications. *Mathematical Structures in Computer Science*, 9(1):21–62, 1999.

[34] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 244–320. Elsevier, 1990.

[35] Răzvan Diaconescu. Extra theory morphisms in institutions: logical semantics for multi-paradigm languages. *Journal of Applied Categorical Structures*, 6(4):427–453, 1998.

[36] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[37] Răzvan Diaconescu. *Category-Based Semantics for Equational and Constraint Logic Programming*. PhD thesis, Oxford University, 1994.

[38] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet, Gordon Plotkin, and C. Jones, editors, Proceedings *of Workshop on Logical Frameworks (Edinburgh, United Kingdom, May 1991)*, pages 83–130. Cambridge University Press, 1991.

[39] Răzvan Diaconescu and Petros Stefaneas. Categorical foundations of modularization for multi-paradigm languages. Technical Report IS-RR-98-0014F, Japan Advanced Institute for Science and Technology, 1998.

[40] Theodosis Dimitrakos. Parameterising (algebraic) specifications on diagrams. In *Automated Software Engineering–ASE'98, 13th IEEE International Conference*, 1998.

[41] Francisco Durán and José Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [65]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[42] Francisco Durán and José Meseguer. The Maude specification of Full Maude. Manuscript, Computer Science Laboratory, SRI International, February 1999.

[43] Hartmut Ehrig, Michael Baldamus, and Fernando Orejas. New concepts of amalgamation and extension of a general theory of specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 199–221. Springer-Verlag, 1993.

[44] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer-Verlag, 1985.

[45] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints.* Springer-Verlag, 1990.

[46] Steven Eker. Term rewriting with operator evaluation strategy. In Kirchner and Kirchner [65]. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[47] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[48] Kokichi Futatsugi and Toshimi Sawada. Cafe as an extensible specification environment. In *Proceeding of the Kunming International CASE Symposium, Kunming, China*, November 1994.

[49] Isabelle Gnaedig, Claude Kirchner, and Hélène Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[50] Martin Gogolla and Maura Cerioli. What is an Abstract Data Type after all? (A bibliography on the workshops on abstract data types). In Egidio Astesiano, Gianna Reggio, and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*, pages 499–523. Springer Verlag, 1995.

[51] Joseph Goguen. Higher order functions considered unnecessary for higher order programming. Technical Report SRI-CSL-88-1R, Computer Science Laboratory, SRI International, January 1988.

[52] Joseph Goguen and Rod Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Proc. Logics of Programming Workshop*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1984.

[53] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[54] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.

[55] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987. Also as Technical Report SRI-CSL-87-7, July 1987.

[56] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. Also as Technical Report SRI-CSL-89-10, July, 1989.

[57] Joseph Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ: A meta-logical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.

[58] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, Eds., Addison-Wesley, 1985, pages 391-420.

[59] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV: Data Structuring*, pages 80–149. Prentice-Hall, 1978.

[60] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992. To appear in Joseph Goguen and Grant Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press.

[61] John Guttag and James Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[62] Jan Heering, Paul Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF. Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Revised version of December 1992 available at `http://www.cwi.nl/~gipe/`.

[63] Richard Jüllig and Yellamraju Srinivas. Diagrams for software synthesis. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 10–19. IEEE Computer Society Press, 1993.

[64] Claude Kirchner. Order-sorted equational unification. In *5th International Conference on Logic Programming*, Seattle, U.S.A., August 1988. Also as Technical Report INRIA 954, December 1988.

[65] Claude Kirchner and Hélène Kirchner, editors. *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, Nancy, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[66] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings, 15th Intl. Coll. on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1988.

[67] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.

[68] Hélène Kirchner and Pierre-Etienne Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.

[69] Paul Klint, editor. *The ASF+SDF Meta-Environment User's Guide*. Centrum voor Wiskunde en Informatica (CWI), 1995. Available at `http://www.cwi.nl/~gipe/`.

[70] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[71] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, August 1993. To appear in D.M. Gabbay, editor, *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[72] Narciso Martí-Oliet and José Meseguer. From abstract data types to logical frameworks. In Egidio Astesiano and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science, pages 48–80, 1994.

[73] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

[74] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[75] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[76] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[77] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[78] José Meseguer. Membership algebra. Lecture at the Dagstuhl Seminar on "Specification and Semantics", July 1996.

[79] José Meseguer, editor. *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, Asilomar, California, U.S.A*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[80] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Ugo Montanari and Vladimiro Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory, CONCUR'96, Pisa, Italy, August 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer-Verlag, 1996.

[81] José Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence, Fort Laurerdale, Florida, January 1998*, 1998.

[82] José Meseguer. Membership algebra as a semantic framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[83] José Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*. Springer-Verlag, 1999.

[84] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 104:114–158, 1993. Also as Technical Report SRI-CSL-90-06, June 1990.

[85] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[86] José Meseguer and Carolyn Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops, March and July 1997. Available at `http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html`.

[87] Peter Mosses. *Action Semantics*. Cambridge University Press, 1992.

[88] E. Najm and J-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 157–176. Chapman & Hall, 1997.

[89] S. Nakajima. Encoding mobility in CafeOBJ: an exercise of describing mobile code-based software architecture. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.

[90] Peter Ölveczky and José Meseguer. Specifying real-time systems in rewriting logic. In Meseguer [79], pages 65–89. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[91] Fernando Orejas, Elvira Pino, and Hartmut Ehrig. Institutions for logic programming. *Theoretical Computer Science*, 173:485–511, 1997.

[92] David Parnas. Information distribution aspects of design methodology. In *Proceedings of 1972 IFIP Congress*, volume 71 of *Information Processing*, pages 339–344. 1972.

[93] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, 15:1053–1058, 1972.

[94] José Quesada. The Maude parser: Parsing and meta-parsing β-extended context-free grammars. Manuscript. SRI International, 1999.

[95] José Quesada. Bidirectional and event-driven parsing with multi-virtual trees. In C. Martin-Vide, editor, *Mathematical and Computational Models in Linguistics*. John Benjamins, 1996.

[96] Horst Reichel. Initially restricting algebraic theories. In P. Dembínski, editor, *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 504–514. Springer-Verlag, 1980.

[97] Christophe Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In Hubert Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*. Springer, 1997.

[98] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, 1988.

[99] Donald Sannella and Martin Wirsing. Implementation of parameterised specifications. In M. Nielsen and E. Schmidt, editors, *9th International Colloquium on Automata, Languages and Programming ICALP'82*, volume 140 of *Lecture Notes in Computer Science*, pages 473–488, Aarhus, Denmark, July 1982, 1982. Springer-Verlag.

[100] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In Marek Karpinski, editor, *Proceedings of the 1983 International Conference on Foundations of Computation Theory. Borgholm, Sweden, August 21-27, 1983*, volume 158 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 1983.

[101] Horst Schubert. *Categories*. Springer-Verlag, 1972.

[102] Adolfo Socorro. *Desgin, Implementation, and Evaluation of a Declarative Object-Oriented Programming Language*. PhD thesis, University of Oxford, 1993.

[103] Yellamraju Srinivas and Richard Jüllig. SPECWARE: Formal support for composing software. In B. Moeller, editor, *Proceedings of the Conference on Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 399–422. Springer-Verlag, 1995.

[104] Carolyn Talcott. An actor rewrite theory. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[105] Andrzej Tarlecki. Towards heterogeneous specifications. In *Proc. Workshop on Frontiers of Combining Systems FroCoS'98*, Applied Logic Series. Kluwer Academic Publishers, 1998.

[106] James Thatcher, Eric Wagner, and Jesse Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, 4(4):711–732, 1982.

[107] Arie van Deursen. *Executable Language Definitions.* PhD thesis, University of Amsterdam, 1994.

[108] Patrick Viry. Rewriting modulo a rewrite system. TR-95-20, C.S. Department, University of Pisa, 1996.

[109] Patrick Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer-Verlag, 1994.

[110] Timothy Winkler. A short note on the OBJ3 module expression routines. Manuscript. SRI International, March 1991.

[111] Martin Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42(2):123–249, 1986.

[112] Martin Wirsing and Alexander Knapp. A formal approach to object-oriented software engineering. In Meseguer [79]. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.