Typing in Model Management

Andrés Vignaga[1], Frédéric Jouault[2], María Cecilia Bastarrica[1], and Hugo Brunelière[2]

[1] MaTE, Department of Computer Science, Universidad de Chile
{avignaga,cecilia}@dcc.uchile.cl

[2] AtlanMod, INRIA Rennes Center - Bretagne Atlantique, Ecole des Mines de Nantes
{frederic.jouault,hugo.bruneliere}@inria.fr

Abstract. Model management is essential for coping with the complexity introduced by the increasing number and varied nature of artifacts involved in MDE-based projects. Global Model Management (GMM) addresses this issue enabling the representation of artifacts, particularly transformation composition and execution, by a model called a megamodel. Typing information about artifacts can be used for preventing type errors during execution. In this work, we present a type system for GMM that improves its current typing approach and enables formal reasoning about the type of artifacts within a megamodel. This type system is able to capture non-trivial situations such as the use of higher order transformations.

1 Introduction

In the field of software development, the increasing use of Model-Driven Engineering (MDE) in the past years has lead to more and more complex situations. Indeed, MDE mainly suggests basing the software development and maintenance process on chains of model transformations. A single transformation is often quite easy to handle but, as soon as industrial use cases are tackled, we are faced with large sets of MDE artifacts (e.g., models, metamodels, transformations) from which a solution have to be assembled. Thus, in order to be able to use them, but without unintentionally increasing the complexity of MDE, we need to invent new ways of creating, storing, viewing, accessing, modifying, and using the metadata associated with all these modeling entities. This is the purpose of Global Model Management (GMM) [6].

As the managed modeling resources may be of varied natures, some support for efficiently organizing them is required. In order to cope with this heterogeneity, a GMM solution has to rely on an architecture which allows precisely typing all the involved entities and corresponding relationships. This should prevent type errors during execution, such as the attempted execution of a non-transformation, or the use of a transformation on arguments for which it is not defined.

Currently, our GMM approach assumes that all managed artifacts are models conforming to precise metamodels. Model typing is then simply based on the conformance relationship, and metamodels are used as types. Moreover, artifacts are also related by strong semantic links. For instance, a transformation refers to its source and target metamodels (i.e., its parameter and return types). Information based on this typing approach suffices for most common cases. However, this scheme notably fails when transformations explicitly depend on these semantic links like in the two following cases: *a*) when a metamodel (i.e., a type used as a value) is used as input to a transformation, and *b*) when a transformation is used as input to another transformation (i.e., a function used as a value). Under these circumstances, it may not be possible to automatically infer a complete type for some elements. For this reason, a more complex typing approach is required.

In this paper we present a first version of a static type system dedicated to GMM. Understanding transformations as functions on models, we introduce a predicative formal system with dependent types with infinite hierarchies of sorts. The type system builds on the existing solution and features dependent products. These types are powerful enough for overcoming the identified limitations. Expressing GMM elements as terms of our calculus enables to statically type check these elements in a mechanical fashion.

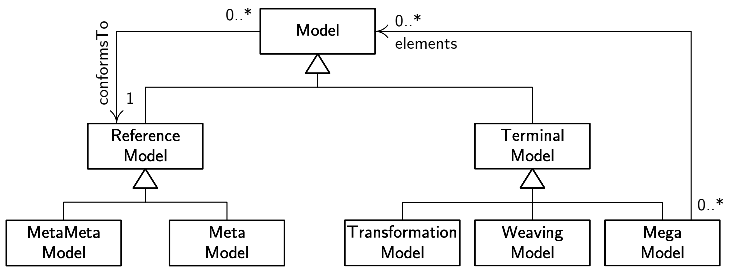
This paper is organized as follows. Section 2 describes the GMM approach to model management, characterizes the limitations of the current version, and introduces a simple example illustrating them. Section 3 details our formal system by providing the syntax of terms and types, type judgments, as well as the set of type rules that form the type system. Section 4 revisits the example in order to demonstrate the application of the type system, and discusses its prototypical implementation within a tool that realizes GMM. Section 5 discusses related work. Section 6 concludes.

2 Global Model Management

In this section we summarize the basic concepts of GMM that enable an understanding of the general context, we discuss how typing is currently addressed and its limitations. For illustrating these issues we also discuss a small example, which will be revisited later on after our solution is presented.

2.1 Global Model Management Conceptual Framework

A Global Model Management approach is based on several general concepts (see Fig. 1), which can be mapped to any concrete case. Most of these concepts, corresponding to a generic conceptual MDE framework, have already been presented in [5]. In addition to these, the concept of a *megamodel* is introduced here as a building block for *modeling in the large* [6]. The principle is the following: for each real-world complex system or process, there can be a *megamodel* [3] representing the different artifacts involved (i.e., models) and their relationships by specifying associated metadata. The type of an artifact or relationship between some artifacts, the identifier of a given artifact and its location, etc., are examples of such registered metadata.



```
context MetaMetaModel inv: self.conformsTo = self
context MetaModel inv: self.conformsTo.oclIsKindOf(MetaMetaModel)
context TerminalModel inv: self.conformsTo.oclIsKindOf(MetaModel)
```

Fig. 1. Global Model Management Conceptual Framework

MDE approaches generally introduce the following three different kinds of models, which occur in the conceptual framework of GMM illustrated in Fig. 1:

- *terminal models* (M1) conform to *metamodels* and are representations of real-world *systems*.
- *metamodels* (M2) conform to *metametamodels* and define domain-specific concepts.
- *metametamodels* (M3) conform to themselves and provide generic concepts for metamodel specification.

Several kinds of *terminal models* may be considered, for example, *weaving models*, and *transformation models*. A *megamodel* is also a specific kind of terminal model, whose elements represent models themselves, as well as relationships between them. As it is a terminal model, a megamodel conforms to a specific metamodel: the *metamodel of megamodel*. If represented as models, available tools, services and service parameters may also be managed by a megamodel. There are actually many events that may change a megamodel like the creation or deletion of a model or metamodel, or the execution of a given transformation.

In addition, the current GMM framework proposes different kinds of relationships between them. The *model transformation* relationship allows specifying source and target metamodels of a given model transformation, and can thus be regarded as its signature. From an execution point of view, the *transformation record* relationship offers a way of representing the metadata needed for any potential execution of a given transformation. This allows specifying its actual input and output models.

To summarize, a megamodel can be viewed as a metadata repository where precise representations of models and links between them are stored and made available to users for various and varied purposes. In particular, the framework should be able to represent type information for adequately typing each element in a megamodel, and provide precise directions on how to use that information.

2.2 Limitations of the Current Typing Approach

As mentioned in the introduction, the current typing solution in GMM follows a simple approach: in principle, all entities are models. Each model conforms to a concrete metamodel, which is its type. The has-type relation (denoted by a colon ':') is therefore defined as follows: *conformsTo*(m,M) iff $m{:}M$, for any model m and metamodel M. However, GMM involves other elements different from entities: relationships. Some elements have dual representations, for example a transformation may be regarded as a relationship (i.e., model transformation) but also as a model (i.e., transformation model) [4]. In the latter case, the type of the element is the metamodel it conforms to. For ATL (AtlanMod Transformation Language) transformations [11, 12], this type is plainly *ATL*, which does not carry information about source and target types. In the former case, the relationship is actually unidirectional and thus a model transformation is understood as a function on models. Type information associated to the element adequately refers to the type of source and target models. However, such models are typed as models whether they are transformations or not, which may cause a loss of type information.

In GMM, typing information plays an important role. The results of a transformation must be properly typed for a later use. This is especially critical when the result is another transformation. For transformations which do not handle dual elements the current typing approach is adequate.

As transformations operate on models, it is natural to type their inputs and outputs as models. The problem is that some of them may have a dual representation and thus the typing approach we initially described does not suffice. The most common case is that of a higher-order transformation (HOT). Consider a transformation h that produces another transformation t. Here h is considered as a transformation, but t is considered as a model. As a consequence of this situation they are typed differently. The type of h refers to the types of its source and target elements. In particular, the type of the target element is the type of t, which is the metamodel t conforms to (e.g., *ATL*). The type of t as a relationship does not fit into this scheme and thus t is only partially typed. We do know that it is a transformation, but we do not know the types of its source and target elements.

This typing approach presents an interesting benefit though. Some form of genericity is introduced: a HOT taking an ATL transformation as source accepts any model conforming to *ATL* (i.e., *any* ATL transformation), regardless of the number and type of its source and target elements. This capability is something we would like to preserve.

Another situation where typing by metamodel may lead to a loss of type information is when transformations operate on metamodels. In fact, the type of any metamodel received or generated by one of such transformations is a metametamodel (e.g., *KM3* [5]). Then, from the type of the transformation, it is possible to know that a metamodel is involved, but not which one. If this metamodel is used for typing other models involved in the same transformation, then it may not be possible to type that transformation.

When the two situations described so far meet, even harder problems arise. The *KM32ATLCopier* transformation [2] is one simple yet interesting case for illustrating these issues. This HOT receives a metamodel M and produces an identity transformation (called a copier), which is specifically applicable to models conforming to M. The type of the resulting copier transformation clearly depends on M. Type information about *KM32ATLCopier* as a relationship may be found in the header of its ATL definition:

```
create OUT : ATL from IN : KM3
```

This type information for *KM32ATLCopier* is insufficient. First, M is not associated to the type of model `IN`. Second, all we know about model `OUT` from its type is that it is a transformation. Third, it is not possible to specify that we know that both its source and target models conform to M.

By introducing other kinds of types such as function types and parametric types, we will be able to deal with these issues. In Sect. 4 this example will be revisited and a type for *KM32ATLCopier* carrying richer information will be discussed.

3 A Type System for GMM

Our solution is based on a typed calculus called cGMM. We define a mapping between GMM constructs and terms and types of that calculus, and then we define a type system for it. Expressing elements within a megamodel as terms enables statically typechecking those elements in a mechanical fashion. This is the main concern of our work, which does not aim to be considered as a complete formalization of GMM.

The cGMM calculus is a predicative typed λ-calculus with dependent types similar to the underlying language of Coq [17], the Predicative Calculus of (Co)Inductive Constructions (pCIC) [14, 19]. Although other approaches may be applicable, a functional one appropriately fits our needs. Like in the current implementation of GMM, we use typed variables for representing models conforming to a metamodel. Dependent products enables (dependent) function types for typing transformations, and universally quantified types for coping with genericity. In particular, higher-order functions naturally represent higher-order transformations. In addition, an infinite hierarchy of universes supports the notion of Type being a type (i.e., Type:Type), and enables a proper representation of the three kinds of models (M1 to M3) discussed in Sect. 2.1.

In order to formalize the type system we need to present some elements of our calculus first. We start by discussing its syntax and the mapping to GMM constructs, and then we address its typing.

3.1 Textual Syntax

All objects handled in cGMM have a type. Unlike most type theories, we do not make a syntactic distinction between types and terms because the type-theory itself forces terms and types to be defined in a mutually recursive way. We therefore define both types and terms in the same syntactical structure.

Sorts. Types are seen as terms and as such they should be typed. The type of a type is called a *sort*. In principle, we use types for typing models so we introduce the sort Type which intends to be the type of such types. Since sorts can be manipulated as terms they also should be given a type. Typing Type with itself leads to undecidable type systems [7]. As a consequence we need to introduce infinite sorts by means of a hierarchy of universes Type_i for any natural i. Thus our set of sorts $\mathcal{S}$ is defined by: $\mathcal{S} \equiv \{\mathsf{Type}_i | i \in \mathbb{N}\}$. These sorts satisfy the following property: Type_i:Type_{i+1}. In this way we understand Type_0 as the type of all metamodels (e.g., *Class* : Type_0), which turns Type_0 into a metametamodel. As in Coq, when referring to the universe Type_i the user will never mention the index i, which is managed by the system. Therefore from a user perspective Type:Type is safely assumed. Consequently, without indexes, Type is a metametamodel which conforms to itself, as required by the first constraint in Fig. 1. However, GMM is expected to support multiple metametamodels at the same time, for example KM3, Ecore, and so on. To that end, we define a separate hierarchy of universes for each of metametamodel, and a corresponding hierarchy should be included in $\mathcal{S}$ each time a new metametamodel is incorporated. In what follows we do not replicate our presentation for every possible metametamodel, rather, we refer to Type as an arbitrary metametamodel.

Terms. Terms are built from variables, several forms of dependent products, several forms of abstractions, applications, cartesian products, tuples, and projections. Assuming x is a variable and T, U are terms, cGMM terms are as follows:

Type	A sort, the type of all types
x	A variable
$\lambda x{:}T;U$	An abstraction (for type abstractions)
$\lambda x{:}T.U$	An abstraction (for classical λ-abstractions)
$\lambda x_1{:}T.x_2{:}U$	An abstraction (for functions with a constant result)
$\forall x{:}T.U$	A dependent product (for universal quantification)
$x{:}T\rightarrow U$	A dependent product (for dependent function types)
$T\rightarrow U$	A non dependent product (for function types)
$(T\ U)$	An application (for both functional application and type instantiation)
$U_1 \times \ldots \times U_n$	A cartesian product
$\langle T_1, \ldots, T_n\rangle$	A tuple
$\mathrm{T}\|_i$	A projection

Type is a metametamodel and as such belongs to M3. Variables map to models, either at M1, M2 or M3. If a variable is typed by Type it represents a reference model, which is an element of M2 or M3. If it is typed by a term typed by Type, then it denotes a terminal model, which is an element of M1.

A type abstraction is used for parameterizing types. Transformation models are represented as functions. GMM manages two kinds of transformations. On the one hand, transformation models can be externally defined in a suitable transformation language, such as ATL, and are thus seen as opaque operations on models where their internal definition is not accessible by the GMM environment. We call this kind of transformations *atomic* transformations. Currently, the only external transformation language supported by GMM is ATL through the GMM4ATL extension. In this work we assume that all atomic transformations are defined in ATL. On the other hand, transformations can be defined within a megamodel, using the language provided by the GMM4CT extension, as external compositions of other existing transformations, regardless of their kind. We call them *composite* transformations and they are model transformations (i.e., relationships) and not transformation models (i.e., entities). For representing first-order atomic transformations we use a special form of abstraction which returns a constant value. This variant was introduced for simplicity, because the only interesting information about the result of such transformations is its type. Therefore, the body of the corresponding function should be nothing but an arbitrary value of the right type. Since each function would require the inclusion of a suitable variable in the typing environment, which should be then accessed for retrieving its type, the environment would be bloated with this kind of definitions. With this abstraction, the variable representing the result is locally defined in the abstraction and therefore the term stays closed and no access to the environment is required. Other forms of transformations are represented by ordinary abstractions. For example, defining two atomic transformations like $\mathit{Class2Relational} \equiv \lambda y{:}\mathit{Class}.r{:}\mathit{Relational}$ and $\mathit{Relational2SQL} \equiv \lambda z{:}\mathit{Relational}.s{:}\mathit{SQL}$, it is possible to define a composite transformation $\mathit{Class2SQL} \equiv \lambda x{:}\mathit{Class}.(\mathit{Relational2SQL}\ (\mathit{Class2Relational}\ x))$.

A universal quantification represents the parametrization of a term with respect to a typed variable. This is usually used in conjunction with higher-order transformations for achieving genericity. Function types, which are just either dependent or non dependent functional views of products, are used for typing transformations. The non dependent case is the typical function type. In turn, in the dependent case, the target type depends on a value of the source type. A detailed example of this involving a HOT will be presented in the next section. An application on a parametric term allows its instantiation to a given type. An application on a function then maps to a transformation record and represents the execution of a transformation on a specific model.

Finally, a cartesian product enables a function type with multiple sources and multiple targets. A tuple is a sequence of typed terms, and projections extract a component from a tuple. As a remark, a HOT is a transformation that operates and/or produces other transformations. Thus a HOT is expressed as a function which either: *a*) has a parameter typed by a function type, or *b*) its body is a function. Free variables and substitution are defined as usual for λ-calculi. Substituting a term T to free occurrences of a variable x in a term U is denoted as $U\{x/T\}$.

3.2 Typing

A type system is a collection of type rules, however they are always formulated with respect to a static typing environment for the program fragment being checked. A *static typing environment* records the type of free variables during the processing of program fragments. For example, the has-type relation $a{:}A$ is associated with a static typing environment Γ that contains information about free variables of a and A.

Judgments. The description of a type system starts with the description of a collection of judgments of the form $\Gamma \vdash \mathcal{A}$ where Γ is a static typing environment, $\mathcal{A}$ is an assertion, and the free variables of $\mathcal{A}$ are declared in Γ. The static typing environment can be understood as a list of distinct typed variables such as $\varnothing$, $x_1{:}A_1, \ldots, x_n{:}A_n$. A static typing environment then maps to the notion of megamodel. The empty environment is denoted by $\varnothing$. The form of $\mathcal{A}$ determines the different judgments to be used within a type system. For our system, we need the following judgments:

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash T{:}U$	T is a well-formed term of type U in Γ

A judgment can be regarded as *valid* or *invalid*. Validity formalizes the notion of well typed programs and is based on type rules. Type rules are used to carry out step-by-step deductions, i.e., type derivations, which formally prove that judgments are valid.

Type Rules. Type rules may be organized according to their conclusion judgment. We distinguish environment well-formedness rules, whose names are of the form (Env ...), from term type rules. In turn, the latter group may be further organized in rules where terms are given the type Type (Type ...) and all the rest. Figure 2 shows some selected rules, where some of which are used further on the next section in our example.

The rule (Env $\varnothing$) is an axiom stating that an empty environment is a valid environment. This means that an empty megamodel is a valid megamodel. The rule (Env Var) extends an environment with a new variable provided that the variable is not defined in the environment and its type is a valid type. This corresponds to adding a new model to a megamodel. The rule (Type Ax) formalizes the property which holds for universes in the same hieararchy within $\mathcal{S}$. The rule (Var) extracts an assumption from an environment, that is, this allows us to use a model included in a valid megamodel.

Rules (Type DFun) and (Type Fun), construct dependent and non dependent function types respectively. In turn rules (Abs Par), (Abs DFun) and (Type Fun) type abstractions. These are type parametrization, dependent and non dependent functions respectively. Finally, rules (App TIns), (App DFun) and (App Fun) introduce applications. In the first case, it is a type instantiation, the others correspond to functional applications.

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env Var)
$$\frac{\Gamma \vdash T{:}s \quad s \in \mathcal{S} \quad x \notin \Gamma}{\Gamma, x{:}T \vdash \diamond}$$

(Type Ax)
$$\frac{\Gamma \vdash \diamond \quad i < j}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_j}$$

(Var)
$$\frac{\Gamma', x{:}T, \Gamma'' \vdash \diamond}{\Gamma', x{:}T, \Gamma'' \vdash x : T}$$

(Type DFun)
$$\frac{\Gamma \vdash T{:}\mathsf{Type}_i \quad i \leq k \quad \Gamma, x{:}T \vdash U : \mathsf{Type}_j \quad j \leq k}{\Gamma \vdash x{:}T \to U : \mathsf{Type}_k}$$

(Type Fun)
$$\frac{\Gamma \vdash T{:}\mathsf{Type}_i \quad i \leq k \quad \Gamma \vdash U : \mathsf{Type}_j \quad j \leq k}{\Gamma \vdash T \to U : \mathsf{Type}_k}$$

(Abs Par)
$$\frac{\Gamma \vdash \forall x{:}T.U : s \quad s \in \mathcal{S} \quad \Gamma, x{:}T \vdash t{:}U}{\Gamma \vdash \lambda x{:}T;t : \forall x{:}T.U}$$

(Abs DFun)
$$\frac{\Gamma \vdash x{:}T \to U : s \quad s \in \mathcal{S} \quad \Gamma, x{:}T \vdash t{:}U}{\Gamma \vdash \lambda x{:}T.t : x{:}T \to U}$$

(Abs Fun)
$$\frac{\Gamma \vdash x{:}T \to U : s \quad s \in \mathcal{S}}{\Gamma \vdash \lambda x{:}T.t{:}U : T \to U}$$

(App TIns)
$$\frac{\Gamma \vdash t : \forall x{:}U.T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T\{x/u\}}$$

(App DFun)
$$\frac{\Gamma \vdash t : x{:}U \to T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T\{x/u\}}$$

(App Fun)
$$\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T}$$

Fig. 2. Sample type rules of cGMM

In the next section we revisit the example of Sect. 2 in detail and present a type derivation which applies many of the rules discussed above.

Soundness. The purpose of a type system is to prevent programs from causing type errors during their execution. A type system is *sound* when only well typed programs execute without type errors [8]. This property of a type system is demonstrated by means of a soundness theorem. A proof of soundness rests upon the semantics of the underlying language, and other properties such as subject reduction and strong normalization. A full proof would deserve a paper on its own, as in [21]. Instead, we rely on the fact that cGMM is based on pCIC, which enjoys such properties [17]. In what follows we discuss some concepts related to those properties which lead to additional rule for our calculus.

Subject reduction, or type preservation, states that reductions preserve type and is formulated as follows. If $\varnothing \vdash T : U$ and $T \rhd T'$ then $\varnothing \vdash T' : U$ (which means that if T reduces then it does so to a value of type U). Note that this is not sufficient for type soundness because it does not rule out the case in which T has a type but it does not reduce. Additionally, the type system should not be able to type terms that cause type errors. In systems such as pCIC all reductions for all typable terms do terminate. Such terms are called strongly normalizing.

The fundamental rule that defines reduction $\triangleright$ identifies the application of a function to a given argument with its result. This is called β-reduction and the rule is: $\Gamma \vdash ((\lambda x{:}T.t)\ u) \triangleright t\{x/u\}$. In systems with dependent types, type conversion is additionally required. Type convertibility $T = U$ is achieved when terms T and U reduce to the same normal form. This enables a rule which says that two convertible well-formed types have the same inhabitants. In this way, terms of a type before a reduction are also typed by the type resulting from the reduction.

As a specific characteristic of cGMM, it is worth noting that an application of the form $((\lambda x_1{:}T.x_2{:}U)\ u)$ trivially reduces to $x_2{:}U$ in one step since no substitution is actually performed. In GMM, all transformations, even composite ones, are ultimately built in terms of atomic transformations like the one just discussed. This suffices for seeing that in cGMM well typed applications reduce to a value.

4 Application

In this section we demonstrate the application of the type system revisiting the *KM32ATLCopier* example, and discuss the prototypical implementation of the type system and its integration into the AM3 tool.

4.1 Example Revisited

Each element in a megamodel represented by an environment Γ is expressed as a term t of our calculus. Then, for a proper type T it should be possible to derive, using the type rules, a proof for $\Gamma \vdash t : T$. If this is possible, term t is well typed and T is its type. For the *KM32ATLCopier* transformation example we use KM3 as a concrete metametamodel instead of Type, and define a term as follows:

$$KM32ATLCopier \equiv \lambda M{:}\mathsf{KM3}.\lambda x{:}M.y{:}M$$

This definition can be interpreted as “a transformation for which, given a metamodel M we get a transformation that takes a model conforming to M as source produces a model conforming to M as a target”. At the outer level the term is an abstraction on variable M. The term at the inner level (i.e., the result of the outer term) is a function with a constant result which represents an atomic copier; its argument x is of type M and the result y is of type M as well. The purpose of the term is just to enable a proper typing for *KM32ATLCopier* and not to model its definition. If that was the case, then the copier would have been written as $\lambda x{:}M.x$, for emphasizing that the result is actually argument x, but in our context it is not necessary even though it would have been possible in this particular case. Now we can prove the following judgment which types *KM32ATLCopier*:

$$\varnothing \vdash \mathit{KM32ATLCopier} : M{:}\mathsf{KM3}{\rightarrow}M{\rightarrow}M$$

This type is a dependent function type on value M. Its co-domain is a non dependent function type where both the domain and the co-domain are M. The static typing environment is the empty environment, which means that no other elements are required within the megamodel for this definition to be meaningful. Next we show a type derivation that proves the well typing of the *KM32ATLCopier* definition.

We start by applying rule (Abs DFun) and as a result we have three subgoals: (1) $\varnothing \vdash M{:}\mathsf{KM3}{\rightarrow}M{\rightarrow}M : \mathsf{KM3}$, (2) $\mathsf{KM3} \in \mathcal{S}$, which naturally holds, and (3) $\varnothing, M{:}\mathsf{KM3} \vdash \lambda x{:}M.y{:}M : M{\rightarrow}M$. Proceeding with subgoal (1), we apply rule (Type DFun) for getting the following subgoals: (4) $\varnothing \vdash \mathsf{KM3}{:}\mathsf{KM3}$ which we can prove by applying rule (Type Ax) and rule (Env $\varnothing$), and then subgoal (5) $\varnothing, M{:}\mathsf{KM3} \vdash M{\rightarrow}M : \mathsf{KM3}$. We prove (5) by applying (Type Fun), which introduces the following subgoal twice: (6) $\varnothing, M{:}\mathsf{KM3} \vdash M{:}\mathsf{KM3}$. This is proved by succesively applying rules (Var), (Env Var), and (Type Ax). For proving (3) we apply rule (Abs Fun) which introduces subgoals: (5) and (2) again. This concludes the proof.

Type inference concerns algorithms that find types (if existing) for typing type-annotated terms, such as the term *KM32ATLCopier* defined before. However, if terms are constructed in a bottom-up fashion, that is, following a derivation from the leaves to the root, and the type information is properly preserved at each step, then the resulting term will be well typed by construction. As discussed next, this is the approach we follow in the implementation of the type system.

For concluding the example, we instantiate the *KM32ATLCopier* term for obtaining a specific copier transformation. To that end, we define the following term based on the *SQL* metamodel:

$$\mathit{SQLCopier} \equiv (\mathit{KM32ATLCopier}\ \mathit{SQL})$$

This term is simply a functional application. The result should be a transformation from *SQL* to *SQL* as we prove next for the following judgment:

$$\varnothing, \mathit{SQL}{:}\mathsf{KM3} \vdash \mathit{SQLCopier} : \mathit{SQL}{\rightarrow}\mathit{SQL}$$

Since substitution $M{\rightarrow}M\{M/\mathit{SQL}\}$ produces $\mathit{SQL}{\rightarrow}\mathit{SQL}$, the proof simply consist of applying rule (App DFun) which produces a subgoal which is exactly the same judgment we previously proved, and subgoal $\varnothing, \mathit{SQL}{:}\mathsf{KM3} \vdash \mathit{SQL}{:}\mathsf{KM3}$, which is proved as subgoal (6) before.

Next, we discuss the implementation of these mechanisms and their integration into a realization of GMM: the AM3 tool.

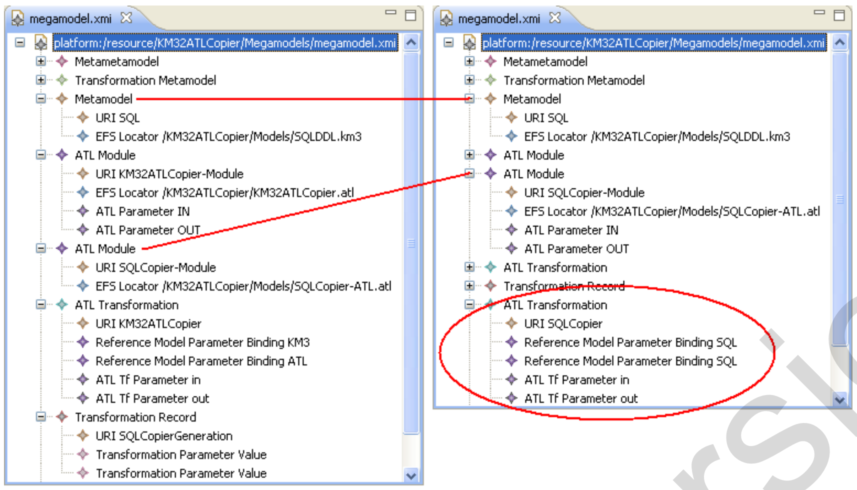
4.2 Implementation

Our type system was firstly emulated using System Coq, and then prototyped as a Java stand-alone system. In such an implementation terms are constructed in a bottom-up fashion, as discussed above, and types are inferred following the type rules. Our experiments were satisfactory, as the right types where found for well-typed terms, and also ill-typed terms correctly threw exceptions during their construction. However, our goal is to integrate such an implementation into the prototypical realization of GMM, which is provided by the Eclipse-GMT AM3 project [1]. Thus, some general information about its overall architecture and main features is first presented. Then, more details on the integration of the type system into the AM3 GMM prototype are given, still taking the same *KM32ATLCopier* transformation as a test example.

The Eclipse-GMT AM3 Global Model Management Solution. The current version of the Eclipse.org *AM3* solution implements the previously described conceptual framework and thus can be used as the GMM tool in the context of our experiments with the proposed type system. It is a project which is part of the *GMT* subproject, which is itself part of the top-level Eclipse *Modeling* project. As an Eclipse project, the AM3 prototype is fully open-source and thus all its source code is freely available from its Eclipse website and download server [1]. The generic and extensible AM3 Global Model Management solution, built on top of the Eclipse environment, provides not only the capabilities to explicitly specify the metadata associated with a given modeled system or MDE process, but also a standard *Megamodel Navigator* as well as generic and extensible editors for instantiating and editing the megamodel in a more user-friendly way. In addition, it offers several extension points allowing the definition of domain-specific extensions of the tool (i.e., extending both the metamodel of megamodel and the related UI components). Thus, AM3 is composed of two distinct sets of Eclipse plug-ins:

– The *core* plug-ins provide the basic metamodel of megamodel, the core runtime environment, the main APIs and an associated generic navigator and editors.
– The *extension* plug-ins provide extensions of the metamodel of megamodel, related specific APIs and corresponding extensions of the UI (for instance specific editor pages, contextual actions, etc).

With AM3, users can build their customized Megamodeling solution by extending either the core plug-ins or other already existing extension plug-ins. Indeed, a set of generic MDE extensions have already been developed: *GMM* for Global Model Management which implements the previously presented GMM conceptual framework, *GMM4ATL* for dealing with model transformations in ATL, *GMM4CT* for supporting Composite Transformations, etc.



```
megamodel.xmi
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
    URI KM32ATLCopier-Module
    EFS Locator /KM32ATLCopier/KM32ATLCopier.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
  ATL Transformation
    URI KM32ATLCopier
    Reference Model Parameter Binding KM3
    Reference Model Parameter Binding ATL
    ATL Tf Parameter in
    ATL Tf Parameter out
  Transformation Record
    URI SQLCopierGeneration
    Transformation Parameter Value
    Transformation Parameter Value
```

```
megamodel.xmi
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Transformation
  Transformation Record
  ATL Transformation
    URI SQLCopier
    Reference Model Parameter Binding SQL
    Reference Model Parameter Binding SQL
    ATL Tf Parameter in
    ATL Tf Parameter out
```

Fig. 3. Megamodel Samples for the *KM32ATLCopier* transformation (respectively before/after type derivation)

Integrating the type system into the AM3 Solution. In order to fully exploit the benefits of the type system in a concrete environment, it must be integrated into the current implementation of the AM3 tool. To that end, and based on the mapping between terms and GMM constructs presented in Sect. 3.1, possible implementation steps could be the following:

1. Develop the required interfaces allowing external tools to provide/retrieve information to/from the existing implementation of the type system;
2. Extend the current *GMM* extension of the AM3 tool so that all the information needed by the type system for a successful evaluation can be represented into a megamodel;
3. Modify the current transformation executor in the *GMM4ATL* extension so that, in the case of the execution of a HOT implemented in ATL, the required information is provided to the implementation of the type system. The result of its evaluation is then retrieved, by the AM3 tool, in order to automatically fill the megamodel with the complete type information.

Additionally, corresponding editors also need to be updated. In the current implementation, terms such as *Class2Relational* from Sect. 3.1, which is a first-order atomic transformation, can be created. For the *KM32ATLCopier* case which is a HOT, specifying *KM3* as the source metamodel must introduce a variable *M*. Then, specifying *ATL* as the target metamodel should allow the user to express that *M* will be both the source and the target of the resulting transformation. In this way, all the required information for deriving an appropriate type is available.

As an illustration, let us consider a megamodel registering the KM32ATLCopier model transformation (along with the KM32ATLCopier-Module transformation model) and the SQL metamodel. After KM32ATLCopier is applied to SQL according to the current AM3 implementation, the SQLCopierGeneration transformation record is created. Its target model is therefore the SQLCopier-Module transformation model. The state of the megamodel is shown in the left part of Fig. 3. Note that the source and target models of SQLCopier-Module were not created because the current typing approach does not provide enough information for doing so. Additionally, the corresponding model transformation (i.e., SQLCopier) was not created for the same reason. The megamodel resulting from deriving the type of the result of such an execution is shown in the right part of Fig. 3. According to the type derivations discussed before, it is possible to know that the result of the execution has type $SQL \rightarrow SQL$, and thus the information for properly completing SQLCopier-Module and creating the SQLCopier relationship is available.

To summarize, the implementation of the type system is integrated into the GMM prototype in such a way that it allows automatically inferring the previously lacking type information. Thus, the corresponding megamodel can now be automatically updated with the computed information.

5 Related Work

GMM is about managing models and other MDE-related resources which are defined elsewhere. So far the only exception to this is that composite transformations can in fact be defined in GMM. Typing becomes a critical issue when execution is considered, and can be studied both at intra-resource and inter-resource levels. In the former case, typing deals with elements within a resource, and the focus is on their internal properties. For example, a type system for a transformation language could ensure that produced models will satisfy some properties [9], such as good behavior. In the latter case, elements to be typed are the resources themselves. Typing in GMM mainly takes this second form. However, well typing of composite transformations is important to us as well.

Similarly to GMM, [10] presents a metamodel for describing MDE concepts and their relationships. Unlike GMM, only core concepts are considered and no tool support is reported. In particular, the typing of those concepts is not addressed or discussed, as we did for GMM.

Model typing is addressed in [16] for investigating transformation reuse. A form of subtyping for model types (i.e., metamodels) enables a sort of *subsumption* on models. Under some circumstances the same transformation may be applied to models of different types. A basic transformation language was introduced for discussing those circumstances, and a type system was defined for it. In that language, transformations are in-place procedures rather than functions, thus they may not be composed. In addition, they are not treated as models and HOTs are not addressed. Although it is related to inter-resource issues due to the subtyping relation, that type system, compared to ours, mainly deals with internal concerns of transformation definitions.

Constructive Type Theory was used in [15] for encoding the MOF layered metamodeling architecture. In particular, an infinite hierarchy of sorts was used for that purpose. However similar, the MOF hierarchy presents an extra level compared to GMM's. Additionally, the dual representation of elements at one level as types of that level and instances of types of the level above was represented, requiring reflection maps for establishing such a correspondence. Since MOF was the only metametamodel, no additional hierarchies were required as in our case. Such a formalism focuses on MOF and therefore is closed to the representation of MOF-based artifacts, which include metamodels, models, and so on, but excludes other MDE-based artifacts. In particular, model transformations and their execution was not considered in that framework.

Typechecking of compositions of transformations has been addressed in [20] and with more detail in [18]. Both approaches use different notions of model typing, and like ours, they require the same type for connecting two adjacent subtransformations. However, none of them provide explicit rules to that end. Additionally, HOTs as well as other cases discussed in this work are not handled.

6 Conclusions and Further Work

Typing in GMM enables transformation execution and is required for preventing type errors during that execution. We improved the current typing approach by proposing a type system that formally indicates how to reason about types in GMM. We showed how non-trivial situations, such as the use of HOTs, can now be handled. Although constructs like model weaving and model-to-text/text-to-model transformations are not yet supported, the issues they pose are similar to those we already dealt with in this work.

Our type system ensures good behavior but relying on the good behavior of atomic transformations. A stronger level of type safety would be achieved by integrating our type system with the type system of a transformation language. This is delicate since both type systems need to be aligned and the result of the integration should still be sound. ATL would be an appropriate case for this. In turn, our type system would benefit from including features from $F_{2<:}$ [8]. This would enable subtyping, not only for model types as in [16], but also for function types as well. Moreover, composite transformations are currently defined by the user, and type information can be used for supporting this manual process, and even for automating (parts of) it.

7 Acknowledgements

We would like to thank Alvaro Tasistro for his numerous contributions to this document. The present work has been jointly supported by the IST-FP6 MODELPLEX European project [13] and the Conicyt Chile-INRIA HOT project.

References

1. AM3 Project. Internet: `http://www.eclipse.org/gmt/am3/`, 2009.
2. ATL Transformations Zoo. Internet: `http://www.eclipse.org/m2m/atl/atlTransformations/`, 2009.

3. M. Barbero, F. Jouault, and J. Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *15th ECBS'08*. IEEE, 2008.
4. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model Transformations? Transformation Models! In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS'2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.
5. J. Bézivin and F. Jouault. KM3: a DSL for Metamodel Specification. In *8th IFIP*, pages 171–185, 2006.
6. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2004.
7. L. Cardelli. Typechecking Dependent Types and Subtypes. In M. Boscarol, L. C. Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer, 1986.
8. L. Cardelli. Type Systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.
9. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
10. J.-M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In *3rd Workshop in Software Model Engineering*, Lisbon, Portugal, 2004.
11. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
12. F. Jouault and I. Kurtev. Transforming Models with ATL. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.
13. MODELPLEX IST-FP6 European Project. Internet: `https://www.modelplex-ist.org/`, 2009.
14. C. Paulin-Mohring. *Le Système Coq.* Thèse d'habilitation, ENS Lyon, 1997.
15. I. Poernomo. A Type Theoretic Framework for Formal Metamodelling. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 262–298. Springer, 2006.
16. J. Steel and J.-M. Jézéquel. On Model Typing. *Software and System Modeling*, 6(4):401–413, 2007.
17. The Coq Proof Assistant Reference Manual. Internet: `http://coq.inria.fr/doc-eng.html`. Version 8.2, 2009.
18. B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers. UniTI: A Unified Transformation Infrastructure. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.
19. B. Werner. *Une Théorie des Constructions Inductives.* Thèse de doctorat, Université Paris 7, 1994.
20. E. D. Willink. OMELET: Exploiting Meta-Models as Type Systems. In D. H. Akehurst, editor, *2nd European Workshop on MDA*, pages 160–164. University of Kent, 2004.
21. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.