Proofs-as-Model-Transformations

Iman Poernomo

Department of Computer Science,
King's College London
Strand, London, WC2R2LS
iman.poernomo@kcl.ac.uk

Abstract. This paper provides an overview of how to develop model transformations that are "provably correct" with respect to a given functional specification. The approach is based in a mathematical formalism called Constructive Type Theory (CTT) and a related synthesis formal method known as proofs-as-programs. We outline how CTT can be used to provide a uniform formal foundation for representing models, metamodels and model transformations as understood within the Object Management Group's Meta-Object Facility (MOF 2.0) and Model Driven Architecture (MDA) suite of standards [6, 8]. CTT was originally developed to provide a unifying foundation for logic, data and programs. It is higher-order, in the sense that it permits representation and reasoning about programs, types of programs and types of types. We argue that this higher-order aspect affords a natural formal definition of metamodel/model/model instantiation relationships within the MOF. We develop formal notions of models, metamodels and model transformation specifications by utilizing the logic that is built into CTT. In proofs-as-programs, a functional program specification is represented as a special kind of type. A program is provably correct with respect to a given specification if it can be typed by that specification. We develop an analogous approach, defining model transformation specifications as types and provably correct transformations as inhabitants of specification types.

1 Introduction

This paper outlines how a formal software verification and synthesis approach can be applied to the problem of developing model transformations in the Model Driven Architecture (MDA) strategy [6]. Our intent is to develop model transformations that are correct with respect to a given pre- and post-condition specification. A plethora of formal methods are available that might serve our purpose. We employ a theory previously developed as a unifying foundation of mathematics and programming, Constructive Type Theory (CTT), and a related synthesis formal method known as proofs-as-programs [2, 10].

In its simplest form, the MDA process involves a transformation between two models, of the form

$$\begin{array}{rcl} PIL & \xrightarrow{T} & PSL \\ T(PIM) & = & PSM \end{array} \tag{1}$$

A. Vallecillo, J. Gray, A. Pierantonio (Eds.): ICMT 2008, LNCS 5063, pp. 214–228, 2008.

A transformation T takes as input a model *PIM*, written in a source modelling language *PIL*, and outputs a new model *PSM*, written in a (possibly different) target modelling language *PSL*. The transformation might serve any number of purposes. It might describe how the *PIM* should be implemented for a particular middleware and platform, so that the resulting *PSM* contains specific implementation decisions that are to be realized by the system programmers. The transformation T should be applicable to any *PIM* written using the *PIL*. It is therefore defined as a general mapping from elements of the language *PIL* to elements of the language *PSL*.

The intention of MDA is to enable designers to focus most of their work on providing a robust, architecturally sound *PIM*. Then, given a particular platform and *PSL* choice, a designer applies a (possibly off-the-shelf) transformation to automatically obtain an appropriate *PSM*.

The methodology is powerful and useful. It can also be dangerous. There is already significant uptake of the strategy from within the enterprise software engineering sector. If a large community of developers agree to use the standard, then refinement-based development becomes a practical reality. Ideally, this will result in better software quality. However, because MDA is essentially an informal approach, it does not guarantee correct model transformations. Currently, given a specification, a transformation is developed by hand with little certification that the specification is met. Testing is still limited: an important research subarea concerns the development of an adequate range of tests [5], often involving a metamodel instance generation problem.

Using MDA without formal grounding can be dangerous. In fact, if model transformations are incorrect, the MDA process can result in software of a lower quality than that produced by traditional software development. This paper provides an overview of how higher-order constructive type theory (CTT) can be used to develop correct-by-construction MDA model transformations. CTT was originally developed to provide a unifying foundation for logic, data and programs. It is higher-order, in the sense that it permits representation and reasoning about ordinary programs, types of programs, types of types and programs that manipulate other programs and types.

We will consider transformations of the form (1) as higher-order typed functional programs. The input type will represent the *PIL* and the output type will represent the *PSL*. In MDA, the OMG defines these two languages as metamodels within the Meta-Object Facility (MOF 2.0). Consequently, we need to understand how the MOF 2.0 metamodelling hierarchy can be given a shallow embedding within CTT.

We believe that CTT is a "natural" formalism for representing the MDA and MOF. This is because the MDA and MOF are intrinsically higher-order. Metamodels are classifiers of classifers and so define types of types. Model transformations are programs that manipulate classifiers, and so, from a mathematical perspective, are functions that manipulate types. The nature of CTT also provides a convenient "built-in" logic, permitting types to include statements and constraints about instantiating terms, in a way that parallels the MOF and MDA

use of logical model constraints in the definition of metamodels and model transformations. Our approach will exploit the proofs-as-programs paradigm property of CTT [10]. At its simplest, the type theory of the lambda-calculus can be regarded as both a proof system and an executable pure functional programming language. A provably correct functional program can be obtained from the inhabiting term of its behavioural specification, treated as a type.

The paper proceeds as follows. Section 2 describes the MOF and explains how it is used to write metamodels within the MDA approach. Section 3 sketches the constructive type theory we use and the proofs-as-programs idea. Section 4 outlines our type theoretic encoding of the MOF and MDA. Conclusions and a discussion of future work is provided in Section 5.

This paper assumes the reader is familiar with the UML representation of classes, class relationships and class objects and has a partial familiarity with the MOF specification document [8]. A detailed study of constructive type theory can be found in [2] or [10] (we follow the formulation of the latter here). More details of our type theoretic treatment of the third and fourth level of the MOF 1.4 are given in [9], which might also serve as a detailed account of our type theory.

2 The MOF

The MOF 2.0 specification consists of two metamodelling languages, the EMOF and CMOF [8]. The former is a subset of, and may be defined reflexively within, the latter. For the purposes of illustrating our formalism, we will consider the subset of the CMOF that permits a UML class-style representation of metamodel grammars. We do not treat a number of other useful features present in the MOF 2.0, such as reflection.

Metamodelling in the MOF is commonly done according to a four level hierarchy, as depicted in [7, pp. 30–31] (the MOF 2.0 permits an any number of levels greater than two). The M_0 level consists of model instances. These might be data values, instantiated class objects, instantiated database tables, algorithms, XML code or function definitions. The M_1 level consists of models, which may also be considered as metamodel instances. This level includes elements such as UML diagrams, class, module and type declarations, database table declarations or XML schema. The M_2 level consists of metamodels, which may also be considered as MOF model instances. This level consists of metamodel descriptions, defining the syntax and semantics of M_1 elements. This level includes languages such as the UML, the XML, Java, the B specification language or *Casl* algebraic specification language. The M_3 level is the MOF language itself, used to define M_2 level elements.

UML-style classes, class associations or class object can be defined at any level in the MOF hierarchy, to serve different purposes. For instance, classes at the M_3 are used to type modelling languages, while classes at the M_2 level are used within modelling languages to type models. The levels are then related by an object-oriented-style class/object instantiation relationship. Class elements of level M_{i+1} provide type descriptions of level M_i objects. M_i objects instantiate M_{i+1} classes.

An important aspect of the MOF hierarchy is that M_1 and M_2 level information can be encoded in two separate ways: as model elements *or* object instances. This enables the MOF hierarchy to treat types as classifications and as forms of data. The principle works as follows. The MOF language is defined by a set of related model elements at the M_3 level. A metamodel is defined at the M_2 level by a set of MOF objects that instantiate the MOF model elements. This MOF object representation of a metamodel can also be rewritten as a M_2 metamodel that provides type descriptions via a set of model elements. A model at the M_1 level is understood as a set of elements that instantiate the classifiers of an M_2 level metamodel. Finally, these M_1 level elements can also be rewritten to form M_1 level model classifiers that specify the required form of an M_0 level model instantiation.

2.1 Object-Based Metamodels

The M_3 level MOF model consists in a set of associated M_3 level classes, "metametaclasses", hereafter referred to as MOF classes. The MOF classes classify the kinds of elements that make up a M_2 level metamodel. Metamodels are collections of associated M_2 instances of these MOF classes, in the same sense that, for example, a collections of M_0 UML objects represent an instance of a M_1 UML class diagram.

The MOF specification defines both the structure of MOF metamodels, consisting of roles and relationships, together with a structural semantics, consisting of constraints that must apply to any instances of the type structure. The MOF defines a set of associated M_3 level classes, the most important of which are as follows: `Classifier` (a general supertype of all metamodel classifiers), `Class` (typing all metamodel classifiers that are not basic data types), `Datatype` (a type of datatypes), `Property` (a type of attributes that may be associated with a metamodel classifier) and `Association` and `AssociationEnd` (typing associations that might hold between metamodel classifiers). The classes are related to each other in the obvious way and have a range of associated attributes treating, for instance, private and public accessibility and inheritance hierarchies. An important attribute of `Property` is the boolean `isComposite`. If the property is set to true, then the owning classifier contains the property and no cyclic dependencies via properties are permitted. If the property is false, then the property is a reference, and cyclic dependencies are permitted [8, pp. 36–37].

The MOF permits constraints to be associated with any of the elements of a metamodel. These can be written in an informal language, such as English, or a formal language, such as the Object Constraint Language (OCL). The MOF model employs constraints in two distinct ways. The MOF model itself has a set of constraints that are defined for each of its classes. These constraints define a static structural semantics of the model that specifies how M_2 metamodels should be formed. Also, the model contains a class called `Constraint` that is associated with all other classes of the model. Instances of this class are used to write a semantics for M_2 metamodels that, in turn, is used to specify how M_1 instantiating models must behave.

For the purposes of this paper, we may consider the simplified notion of a MOF metamodel as a collection of associated MOF class object instances. These instances are M_2 level objects.

Definition 1 (Metamodel). *A* metamodel M *is a set of* `Classifer`, `Class`, `Datatype`, `Attribute`, `Association`, `AssociationEnd` *and* `Constraint` M_2 *level objects. Objects within* M *may only refer to each other.*

2.2 Class-Based Description of Metamodels

A metamodel specification consists in a set of M_2 level objects. This is a data-centric view of a metamodel. When considering a metamodel as a model of models, we need to use this data to classify models. The MOF achieves this by means of an equivalent representation of a metamodel, as M_2 level *classes*, whose M_1 level object instances are *models*.

Given a metamodel MO represented as a set of M_2 level objects, we can build an equivalent M_2 level *class-based* representation MC as follows. Each `Class` M_2 object `o` in MO corresponds to a M_2 class $\mathsf{toClass}(\mathtt{o})$, whose class attributes each correspond to the M_2 level `Attribute` objects associated with `o`. Similarly, for each `Association` object `a` in MO that defines a relation between two `Class` objects $\mathtt{o}_1$ and $\mathtt{o}_2$, we add a class association in the metamodel MC between the classes that correspond to $\mathtt{o}_1$ and $\mathtt{o}_2$. Each `Constraint` object associated with an object o is mapped to a UML-style *note* that is associated with $\mathsf{toClass}(\mathtt{o})$. The contents of the note are the same as the contents of the constraint.

A class-based representation is important as it prescribes how the metamodel should be used as a typing structure for M_1 level models. It is important to note that, according to the MOF, not every collection of M_2 level classes defines a metamodel. To be valid, a metamodel must also have an object-based representation that instantiates the MOF model.

3 Constructive Type Theory

This section presents a brief summary of the constructive type theory (CTT) that shall be used to formalize. We define a version of Martin-Löf's predicative type theory with dependent sum and product types [4], and explain how the CTT provides an uniform framework for treating functions, types, proofs and programs.

We work with a lambda calculus whose core set of terms, P, are given over a set of variables, V:

$$\begin{aligned} P ::= V | \lambda\, V.\, P | (P\ P) | \langle P, P\rangle | \mathsf{fst}(P) | \mathsf{snd}(P) | \mathsf{inl}(P) | \mathsf{inr}(P) | \\ \mathsf{match}\ P\ \mathsf{with}\ \mathsf{inl}(V) \Rightarrow P \mid \mathsf{inr}(V) \Rightarrow P | \\ \mathsf{abort}(P) | \mathsf{show}(V, P) | \mathsf{select}\ (P)\ \mathsf{in}\ V.V.P \end{aligned}$$

The evaluation semantics of lambda abstraction and application are standard and widely used in functional programming languages such as *SML*: $\lambda\, x.\, P$ defines a function that takes x as input and will output $P[a/x]$ when applied to

a via an application $(\lambda\ x.\ P)a$. The calculus also includes pairs $\langle a, b\rangle$, where $\mathsf{fst}(\langle a, b\rangle)$ will evaluate to the first projection a (similarly for the second projection). Case matching provides form of conditional, so that $\mathsf{match}\ z\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow P \mid \mathsf{inr}(y) \Rightarrow Q$ will evaluate to $P[x/a]$ if z is $\mathsf{inl}(a)$ and to $Q[y/a]$ if z is $\mathsf{inr}(a)$. $\mathsf{show}(a, P)$ is a form of pairing data a with a term P. Terms leading to inconsistent state are represented $\mathsf{abort}(p)$. Evaluation is assumed to be *lazy* – that is, the operational semantics is applied to the outermost terms, working inwards until a neutral term is reached. We write $a \rhd b$ if a evaluates to b according to this semantics.

The lambda calculus is a programming language. We can compile terms and run them as programs. Like most modern programming languages, our calculus is typed, allowing us to specify, for example, the input and output types of lambda terms. The terms of our lambda calculus are associated with the following kinds of types: basic types from a set BT, functional types $(A \rightarrow B)$, product types $(A * B)$, disjoint unions $(A|B)$, dependent product types $(\prod x : t.a)$ where x is taken from V, and dependent sum types $(\Sigma x : t.b)$ where x is taken from V. The intuition behind the first four types should be clear. For example, if a term t has type $(A \rightarrow B)$, then t is a function that can accept as input any value of type A to produce a value of type B. A dependent product type expresses the dependence of a function's output types on its input term arguments. For example, if a function f has dependent product type $\prod x : T.F(x)$, then f can input any value of type T, producing an output value of type $F(arg)$. Thus, the final output type is parameterized by the input *value*. Typing rules provide a formal system for determining what the types of lambda terms should be. The core typing rules are displayed in Fig. 1. Further rules may be included in a straightforward way to accommodate recursion.

It is not permissible to define a totality of the collection of all types, as this results in an inconsistent theory. Instead, we employ a common solution, defining a predicative hierarchy of type universes of the form:

$$\textsc{Type}_0, \textsc{Type}_1, \textsc{Type}_2, \ldots$$

The typing rules for the universes, omitted for reasons of space, may be found in [9]. In these rules, the first universe $\textsc{Type}_0$ is the type of all types generated by the basic types and the typing constructors.

To encode objects and classes, we will require record types. These types have the usual definition – see, for example, [2] or [9]. A record type is of the form $\{a_1 : T_1; \ldots; a_n : T_n\}$, where $a_1, \ldots, a_n$ are labelling names. A record is a term $\{a_1 = d_1; \ldots; a_n = d_n\}$ of a record type $\{a_1 : T_1; \ldots; a_n : T_n\}$, where each term d_i is of type T_i. The term $\{a_1 = d_1; \ldots; a_n = d_n\}.a_i$ evaluates to the value d_i associated with the label a_i in the left hand side record.

To treat cyclic dependencies within metamodels, we require *co-inductive types*. Co-induction over record types essentially allows us to expand as many references to other records as we require, simulating navigation through a metamodel's cyclic reference structure. For the purposes of this paper, these types are abbreviated by means of mutually recursive definitions, of the form

$$\frac{}{x : A \vdash x : A}\ (\text{Ass-I})$$

$$\frac{\Delta, x : s \vdash p : A}{\Delta \vdash \lambda\, x : s.\ p : \prod x : s \bullet A}\ (\prod\text{-I})$$

$$\frac{\Delta_1 \vdash p : \prod x : s \bullet A \quad \Delta_2 \vdash c : s}{\Delta_1, \Delta_2 \vdash (p\ c) : A[c/x]}\ (\prod\text{-E})$$

$$\frac{\Delta, x : s \vdash p : A \quad x : s \text{ is not free in } A}{\Delta \vdash \lambda\, x : s.\ p : s \to A}\ (\to\text{-I})$$

$$\frac{\Delta_1 \vdash p : s \to A \quad \Delta_2 \vdash c : s}{\Delta_1, \Delta_2 \vdash (p\ c) : A}\ (\to\text{-E})$$

$$\frac{\Delta \vdash p : P[a/y]}{\Delta \vdash \mathsf{show}(a, p) : \Sigma y : s \bullet P}\ (\Sigma\text{-I})$$

$$\frac{\Delta_1 \vdash p : \Sigma y : s \bullet P \quad \Delta_2, x : P[z/y] \vdash q : C}{\Delta_1, \Delta_2 \vdash \mathsf{select}\ (p)\ \mathsf{in}\ z.x.q : C}\ (\Sigma\text{-E})$$

$$\frac{\Delta \vdash a : A \quad \Delta' \vdash b : B}{\Delta, \Delta' \vdash \langle a, b\rangle : (A * B)}\ (\text{prod-I})$$

$$\frac{\Delta \vdash p : (A_1 * A_2)}{\Delta \vdash \mathsf{fst}(p) : A_1}\ (\text{prod-E}_1) \qquad \frac{\Delta \vdash p : (A_1 * A_2)}{\Delta \vdash \mathsf{snd}(p) : A_2}\ (\text{prod-E}_2)$$

$$\frac{\Delta \vdash p : A_1}{\Delta \vdash \mathsf{inl}(p) : (A_1|A_2)}\ (\text{union-I}_1) \qquad \frac{\Delta \vdash p : A_2}{\Delta \vdash \mathsf{inr}(p) : (A_1|A_2)}\ (\text{union-I}_2)$$

$$\frac{\Delta \vdash p : A|B \quad \Delta_1, x : A \vdash a : C \quad \Delta_2, y : B \vdash b : C}{\Delta_1, \Delta_2, \Delta \vdash \mathsf{match}\ p\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow a \mid \mathsf{inr}(y) \Rightarrow b : C}\ (\text{union-E})$$

$$\frac{\Delta \vdash a : \bot}{\Delta \vdash \mathsf{abort}(a) : A}\ (\bot\text{-E})$$

Fig. 1. Typing rules for our lambda calculus

$$\mathrm{T} \equiv F(\mathrm{U}) : \textsc{Type}_i$$

$$\mathrm{U} \equiv G(\mathrm{T}) : \textsc{Type}_i$$

This is a notational convenience: the formal treatment of co-induction, and associated co-inductive recursion schemes, is given in [9]. This paper does not treat inheritance in metamodelling: we have equipped our type theory with a notion of subtyping to treat inheritance [9].

3.1 Proofs-as-Programs

The Curry-Howard isomorphism shows that constructive logic is naturally embedded within our type theory, where proofs correspond to terms, formulae to types, logical rules to typing rules, and proof normalization to term simplification. Consider a constructive logic whose formulae, WFF are built from exactly the same predicates that occur in our type theory. We can define an injection

A	$\mathsf{asType}(A)$
$Q(x)$, where Q is a predicate	$Q(x)$
$\forall x : T.P$	$\prod x : T.\mathsf{asType}(P)$
$\exists x : T.P$	$\Sigma x : T.\mathsf{asType}(P)$
$P \wedge Q$	$\mathsf{asType}(P) * \mathsf{asType}(Q)$
$P \vee Q$	$\mathsf{asType}(P) \mid \mathsf{asType}(Q)$
$P \Rightarrow Q$	$\mathsf{asType}(P) \rightarrow \mathsf{asType}(Q)$
$\bot$	$\bot$

Fig. 2. Definition of asType, an injection from WFF to types of the lambda calculus

asType, from well-formed formulae WFF to types of the lambda calculus as in Fig. 2.

The isomorphism tells us that logical statements and proofs correspond to types and terms:

Theorem 1 (Curry-Howard isomorphism). *Let $\Gamma = \{G_1, \ldots, G_n\}$ be a set of premises. Let $\Gamma' = \{x_1 : G_1, \ldots, x_n : G_n\}$ be a corresponding set of typed variables. Let A be a well-formed formula. Then the following is true. Given a proof of $\Gamma \vdash_{\mathsf{Int}} A$ we can use the typing rules to construct a well-typed proof-term $p : \mathsf{asType}(A)$ whose free proof-term variables are Γ'. Symmetrically, given a well-typed proof-term $p : \mathsf{asType}(A)$ whose free term variables are Γ', we can construct a proof in constructive logic $\Gamma \vdash A$.*

Theorem 2 (Program extraction). *Let $\Gamma = \{G_1, \ldots, G_n\}$ be a set of premises. Let $\Gamma' = \{x_1 : G_1, \ldots, x_n : G_n\}$ be a corresponding set of typed variables. Let $\forall x : T.\exists y : U.P(x, y)$ be a well-formed $\forall\exists$ formula.*

There is a mapping $\mathsf{extract}$ from terms to terms such that, if $\vdash p : \mathsf{asType}(\forall x : T.\exists y : U.P(x, y))$ is a well typed term, then $\vdash \forall x : T.P(x, \mathsf{extract}(p)x)$ is provable.

The proof of the theorem follows standard previous presentations, but requires an extension to deal with co-inductive types. The implication of this theorem is that, given a proof of a formula $\forall x : T.\exists y : U.P(x, y)$, we can automatically extract a function that f that, given input $x : T$ will produce an output fx that satisfies the constraint $P(x, fx)$.

Our notion of proofs-as-model-transformations essentially follows from this theorem. A model transformation of the form (1) can be specified as a constraint in the OCL over instances of an input PIM and an output PSM. Assuming we can develop types and to represent the PIM and PSM metamodels, and that the constraint can be written as a logical formula over a term for the metamodels, we can then specify the transformation as an $\forall\exists$ formula. Then, in order to synthesize a provably correct model transformation, we prove the formula's truth and apply the extraction mapping according to Theorem 2.

The main technical challenges posed by this approach are 1) the extract map is a non-trivial extension of the usual extraction map used in similar proofs-as-programs

approaches, modified to suit our more complicated type theory and 2) the way in which MOF-based metamodels can be formalized as types is not clear. Space does not permit us to describe the extraction mapping, but essentially it is developed using the generic machinery of [10]. The latter challenge is now addressed.

3.2 Metaclass Structures as Record Types

We first describe how the *structure* of classes and objects can represented within our type theory. Our encoding is standard (see, e.g., [13]). We define classes as recursive record types, with objects taken as terms of these types. We restricted our attention to classes with attributes but without operations, we will not deal with representing operations within class types. Our representation can be easily extended to this (again, see [13]).

First, recall that we shall treat the associations of a class in the same way as attributes. That is, if class M_1 is associated with another class M_2 with n the name of the end of the association at M_2, then we treat this as an attribute $n : M_2$ within M_1 if the multiplicity of n is 1, and $n : [M_2]$ otherwise.

Essentially, the idea is to map a class $\mathtt{C}$ with attributes and associations $\mathtt{a_1} : \mathtt{T_1}, \ldots, \mathtt{a_n} : \mathtt{T_n}$ to a record type definition

$$\mathrm{C} \equiv \{\mathtt{a}_1 : \mathrm{T}_1; \ldots; \mathtt{a}_n : \mathrm{T}_n\}$$

where each $\mathtt{a}_i$ is an element of *String* corresponding to the attribute name $\mathtt{a}_i$ and each T_i a type corresponding to the classifier $\mathtt{T_i}$. The class can reference another class or itself through the attribute types. The mapping therefore permits mutual recursion between class definitions. That is, each T_i could be C or could refer to other defined class types.

The encoding of classes is purely structural and does not involve a behavioural semantics. A semantics is instead associated with a structural class type through a logical specification in a way now described.

4 The MOF and MDA within CTT

If we can encode the MOF within our CTT, it is possible to apply proofs-as-programs to develop provably correct model transformations via extraction from proofs. Following previous work by the author [9], metamodel/model/model instantiation relationships of the MOF can treated using terms and types within the CTT's predicative type hierarchy. This framework enables us to define a higher order type for any metamodel ModelLang, so that $\vdash$ *model* : ModelLang is derivable if, and only if, the term *model* corresponds to a well formed model instance of the metamodel. Model transformations should then be representable as functions within the CTT that are typed by metamodel types.

The main concepts of the MOF have obvious formal counterparts within the CTT. Classes and objects are treated using recursive records. The four levels of the MOF are corresponding to the CTT's predicative hierarchy of type universes. The CTT's typing relation allows us to systematically treat MOF

model/metamodel/model/model instantiation relationships as follows. The M_3 level MOF classes are defined through $\textsc{Type}_2$ class types, M_2 level metamodel classifiers are given a dual representation as objects of the MOF class types and as $\textsc{Type}_1$ class types. M_1 level model entities are given a dual representation as terms of the metamodel types and as as $\textsc{Type}_0$ types, M_0 level implementations of models are instantiating terms of $\textsc{Type}_0$ types. This section outlines how to formalize the MOF classes and metamodels at levels M_3 and M_2.

4.1 Encoding of the MOF

The structure of MOF metamodels was defined as a set of M_3 level classes. It is possible to define a set of mutually recursive $\textsc{Type}_2$ level record types that encode these classes. A metamodel, considered as a set of M_2 level objects that instantiate the MOF classes, is then formally understood as a set of mututally recursive $\textsc{Type}_1$ level terms of these types.

For the purpose of illustration, the type of the MOF classifier class is as follows.

Definition 2 (MOF classifier type). *A MOF classifier is encoded by the following record type,* $\textsc{Classifer} \equiv \Sigma x : \textsc{ClassStruct}.\mathrm{MClassCst}(x)$ *where* ClassStruct *stands for the record*

$$\{\mathsf{name} : \mathit{String}; \mathsf{isAbstract} : \mathit{Bool}; \\ \mathsf{supertype} : \textsc{Classifer}; \mathsf{attributes} : [\textsc{Attribute}]\}$$

and $\mathrm{MClassCst}(x)$ *is a statement about* $x : \textsc{ClassStruct}$ *that formalize the constraints given in the OMG standard.*

A similar encoding is made for the other MOF elements: a record type, used to define the element's structure, is paired with constraints over the structure using a dependent sum, used to formally specify the element's semantics.

The type of all MOF-based metamodels, Metamodel, can be defined as a fixed point corresponding to mutually recursive set of MOF class instances. The definition follows from the MOF, where a metamodel is understood to consist of a set of associated metaclasses.

4.2 Metamodels as Types

Recall that metamodels have a dual representation, as M_2 level objects and as M_2 level classes. This dual representation is formalized by means of a transformation between instantiating Metamodel terms and $\textsc{Type}_1$ level types. The transformation is twofold: (1) A *reflection map* ϕ is applied to obtain a set of mutally recursive record types from a metamodel term. The map essentially obtains a type structure for the metaclasses and associations of the metamodel. (2) The constraints specified by the MOF metamodel description as `Constraint` objects are formalized as a specification over the type structure obtained from the reflection map. The transformation then uses this information to build a dependent sum type that represents the metamodel. The mapping is omitted for reasons of space – see [9] for details.

Definition 3 (Metamodel types). *Given a* METAMODEL *instance* a : METAMODEL*, the type* $\phi(a)$ *is called the metamodel structure type for* a*, and represents the structure of the metamodel, when considered as a collection of* M_2 *level classifiers. The general form of a metamodel type is* $\Sigma x : \phi(a).P(x)$ *for a generated predicate* P *and* a : METAMODEL*.*

Given a metamodel type $\Sigma x : \phi(a).P(x)$, the predicate P should be a formal specification of the `Constraints` objects that form part of the MOF metamodel for a. In general, when using our approach for formalizing MOF-based metamodels, it is not possible to automatically generate P, because the OMG specification permits `Constraints` to take any form. However, if we assume the constraints are always written in a subset of first order logic, such as the OCL, then it is possible to generate P in a consistent manner.

4.3 Metamodelling and Modelling Process

Given a typical, informal, MOF-based specification of a metamodel, consisting of metaclasses, meta-associations and constraints, it is quite straightforward to develop an instance of METAMODEL. The process is straightforward because MOF-based metamodel specifications, such as the OMG's definition of the UML, usually make use of OCL constraints. These can be readily translated into logical constraints in the METAMODEL Then, by application of ϕ, a TYPE_1 level type can be produced that defines the structure of the metamodel.

For example, given a MOF-compliant definition of the simple database metamodel of Fig. 3, it is possible to develop a metamodel term *rdb* that will yield a TYPE_2 dependent sum of the form $\Sigma x : \phi(rdb).P(x)$ where $\phi(rdb)$ is a coinductive record[1]

$$\text{RDB} \equiv \{\mathsf{nes} : [\text{NAMEDELEMENT}]; \mathsf{tables} : [\text{TABLE}]; \mathsf{keys} : [\text{KEY}]; \mathsf{columns} : [\text{COLUMN}]\}$$

built from the following types

$$\text{NAMEDELEMENT} \equiv \{name : String\}$$

$$\text{TABLE} \equiv \{name : String; tablecolumns : [\text{COLUMN}]; keys : [\text{KEY}]]\}$$

$$\text{KEY} \equiv \{name : String; keyColumns : [\text{COLUMN}]\}$$

$$\text{COLUMN} \equiv \{name : String\}$$

and P is a formula that is derived from `Constraint` metaobjects that were associated with the metamodel.

[1] We write this record informally here as a record: formally, it would involve a μ constructor, following [9]. The idea is that a term instance of this metamodel type will consist of a record of metaclass term instances that can mutually refer to each other. This also allows us to represent shared data, as in the case where the same column is referenced by a key and by a table.

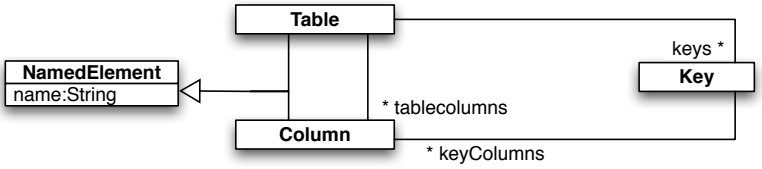

Fig. 3. Fragment of the DB metamodel

Similarly, given the OMG's definition of the UML, it is possible to develop a metamodel term uml that will yield a TYPE_2 dependent sum of the form $\Sigma x : \phi(uml).P(x)$ where $\phi(uml)$ is a fixed point, written in a recursive style as

$$\text{NAMEDELEMENT} \equiv \{name : String\}$$

$$\text{CLASS} \equiv \{name : String; isAbstract : Boolean; tag : String; super : Class; attrs : [\text{ATTRIBUTE}]\}$$

$$\text{PRIMITIVETYPE} \equiv \{name : String\}$$

$$\text{TYPE} \equiv \{name : String\}$$

$$\text{TYPEELEMENT} \equiv \{name : String; type : \text{TYPE}\}$$

$$\text{ATTRIBUTE} \equiv \{name : String; type : \text{TYPE}; multiplicity : \text{MULTIPLICITYKIND}\}$$

$$\text{MULTIPLICITYKIND} \equiv \{lowerBound : int; upperBound : int; ordered : Boolean; unique : Boolean\}$$

and P is a formula that is derived from `Constraint` metaobjects that were associated with the metamodel. As required, this corresponds to the standard equivalent M_2 level class-based metamodel diagram.

4.4 Proofs-as-Model-Transformations

Given a way of representing MOF metamodels and models within our type theory, we can then apply the proofs-as-programs approach entailed in Theorem 2 to extract correct model transformations. First, we specify transformations as $\forall\exists$ types of the form

$$\forall x : \text{PIL}.I(x) \rightarrow (\exists y : \text{PSL}.O(x, y))$$

where PIL and PSL are source and target metamodel types, $I(x)$ specifies a precondition on the input model x for the transformation to be applied, and $O(x, y)$ specifies required properties of the output model y. We use typing rules to obtain an inhabiting term for such a transformation type. Then, by virtue of Curry-Howard isomorphism and proofs-as-programs, this term can be transformed into a model transformation function f such that, given any PIM x satisfying the precondition $I(x)$, then the postcondition $O(x, fx)$ will be satisfied.

4.5 Example

Consider the following toy UML-to-relational-database model transformation specification: For each non abstract class of the UML model which is taged "CMP" persistent a table is created in the RDB model. This table has the same name as the class and holds one column per attributes of the class. Columns have the name of its corresponding attribute.

The transformation has UML models as its input and RDB models as output. The precondition of the transformation is class name uniqueness, sketched in OCL as

```
UMLModel->instanceof(Class)->forall(c1,c2 | c1.name = c2.name
   implies c1 = c2)
```

The postcondition of the transformation is

```
UMLModel->instanceoff(Class)->select(c |  c.isAbstract == false
 c.tag == 'CMP')->forall(c | RDBMModel->instanceof(Table)->one(t |
     t.name == c.name attributes->forall(a |
        t.columns->one( | col.name == a.name))))
```

The specification of the transformation is given as follows

$$\begin{aligned}\forall x : \textsc{Uml}.\forall c : \textsc{Class}.c \in x.\mathsf{classes} \wedge \\ c.\mathsf{isAbstract} = false \wedge c.\mathsf{tag} =' CMP' \rightarrow \\ \exists y : \textsc{Rdb}.!\exists t : \textsc{Table}.t \in y.\mathsf{tables} \wedge t.\mathsf{name} = c.\mathsf{name} \wedge \\ \forall a : \textsc{Attribute}.a \in c.\mathsf{attribs} \wedge \\ !\exists col : \textsc{Column}.col \in t.\mathsf{columns} \wedge col.\mathsf{name} = a.\mathsf{name} \end{aligned} \quad (2)$$

where

$$!\exists y : T.P(y) \Leftrightarrow \exists y : T.P(y) \wedge \forall z : T.P(z) \rightarrow z = y$$

This specification can then be proved using just over 100 applications of the typing rules of Fig. 1. Semi-automatic use of tactics in a theorem proving environment such as PVS should reduce the burden of proof on a human developer. Then, by application of the extraction mapping, we can obtain a lambda term whose input type is Uml and whose output is Rdb, $\lambda x : \textsc{Uml}.F(x.classes)$, where F is an extracted function of the form

```
F: [Class] -> [Tables]
F hd::tl ->
 if hd.tag = 'CMP'
  {name = hd.name; columns = v}::F(tl)
  where v = [{name = a.name}] a over all hd.attribs
  else F(tl)
 [a] -> if a.tag = 'CMP' then {name = a.name; columns = v}
  else []
```

This extracted function is the required model transformation.

5 Related Work and Conclusions

We have attempted to demonstrate that constructive type theory is a natural choice to formally encode the higher-order structure of the MOF. To the best of our knowledge, constructive type theory has not been used previously as a framework to treat metamodelling.

Favre [3] developed a methodology for writing correct transformations between UML-based metamodels. Transformations are understood formally in terms of the *Casl* algebraic specification language, so a notion of formal correctness is present and transformations are proved correct. The work has yet to be generalized to arbitrary MOF metamodels.

Akehurst et al. have used relational algebras to formalize metamodels and model transformations [1]. Thirioux et al. have a similar approach based on typed multigraphs [14]. Their framework forms an algebra with operations corresponding to the classification relationship between metamodels and models. From a type theoretic perspective, their formalisation is first order, and based in set theory. As a result, their model of the higher order nature of the MOF and model transformations is "flattened" into a single type universe (sets).

Structured algebraic specification languages that have been used for formalizing object-oriented specification should have the potential for formal metamodelling. We know of two approaches. Ruscio et al. have made some progress towards formalizing the KM3 metamodelling language using the Abstract State Machines [12]. Rivera and Vallecillo have exploited the class-based nature of the Maude specification language to formalize metamodels written in the KM3 metamodelling language [11]. Their treatment of the dual, object- and class-based, representation of metamodels is similar to ours, involving an equivalence mapping. The intention was to use Maude as a means of defining dynamic behaviour of models, something that our approach also lends itself to. Their work has the advantage of permitting simulation via rewriting rules.

Our experience with the small transformation described above is that, while the proof steps are often relatively trivial, there are a great many of them and the process of manual proof is laborious. We expect a tactic-based theorem prover will improve efficiency of the development process. A tactic is essentially a script that automates a sequence of proof steps. There are three robust tactic-based tools based on higher order lambda calculus: Nuprl, Coq and PVS. All three systems are equipped with advanced semi-automatic theorem provers and have been shown to be effective in the synthesis and verification of complex industrial software systems. A full implementation of our approach within Nuprl forms part of ongoing research by the author's group.

References

[1] Akehurst, D.H., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. Software and System Modeling 2(4), 215–239 (2003)

[2] Constable, R., Mendler, N., Howe, D.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Englewood Cliffs (1986) (Accessed May 2003), http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html
[3] Favre, L.: Foundations for mda-based forward engineering. Journal of Object Technology 4(1), 129–153 (2005)
[4] Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis (1984)
[5] Mottu, J.-M., Baudry, B., Le Traon, Y.: Mutation Analysis Testing for Model Transformations. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 376–390. Springer, Heidelberg (2006)
[6] Mukerji, J., Miller, J.: MDA Guide Version 1.0.1. Object Management Group (2003)
[7] OMG. Meta Object Facility (MOF) Specification. Object Management Group (2000)
[8] OMG. Meta Object Facility (MOF) Core Specification, Version 2.0. Object Management Group (January 2006)
[9] Poernomo, I.: A Type Theoretic Framework for Formal Metamodelling. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 262–298. Springer, Heidelberg (2006)
[10] Poernomo, I., Crossley, J., Wirsing, M.: Adapting Proofs-as-Programs: The Curry-Howard Protocol. Monographs in computer science. Springer, Heidelberg (2005)
[11] Rivera, J., Vallecillo, A.: Adding behavioural semantics to models. In: The 11th IEEE International EDOC Conference (EDOC 2007), Annapolis, Maryland, USA, pp. 169–180. IEEE Computer Society, Los Alamitos (2007)
[12] Ruscio, D.D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France (April 2006)
[13] Simons, A.J.H.: The theory of classification. part 3: Object encodings and recursion. Journal of Object Technology 1(4), 49–57 (2002)
[14] Thirioux, X., Combemale, B., Crégut, X., Garoche, P.-L.: A framework to formalise the mde foundations. In: Proceedings of TOWERS 2007, Zurich, June 25 2007, pp. 14–30 (2007)