Model Transformation with Immutable Data

Paul Klint and Tijs van der Storm(✉)

CWI, Amsterdam, The Netherlands
storm@cwi.nl

Abstract. Mainstream model transformation tools operate on graph structured models which are described by class-based meta-models. In the traditional grammarware space, transformation tools consume and produce tree structured terms, which are described by some kind of algebraic datatype or grammar. In this paper we explore a functional style of model transformation using Rascal, a meta-programming language, that seamlessly integrates functional programming, flexible static typing, and syntax-based analysis and transformation. We represent meta-models as algebraic data types (ADTs), and models as immutable values conforming to those data types. Our main contributions are (a) REFS a simple encoding and API, to deal with cross references among model elements that are represented as ADTs; (b) a mapping from models to ADTs augmented with REFS; (c) evaluation of our encoding by implementing various well-known model transformations on state machines, meta-models, and activity diagrams. Our approach can be seen as a first step towards making existing techniques and tools from the modelware domain available for reuse within Rascal, and opening up Rascal's transformation capabilities for use in model driven engineering scenarios.

1 Model Transformation with Grammarware

There are strong analogies between modelware and grammarware, albeit that terminology is mostly disjoint. For example, in modelware, a state machine model can be described by a model described in Ecore and Ecore itself is described using the Ecore meta-model. In grammarware, a C program can be described by a grammar of the C language written in BNF notation and BNF notation itself is described by a BNF grammar. A key difference between these domains is how models are represented. In the modeling domain models and meta-models are represented and processed as mutable graphs while immutable, tree-based, representation prevails in the grammar domain. The focus of this paper is on analyzing and bridging the impedance mismatch between these graph-based and tree-based domains. This can bring various cross fertilization benefits:

- The ecosystem of models and modeling tools becomes available for the grammar-based approaches, e.g., EMF[1] (including Ecore[2] and

[1] See http://www.eclipse.org/modeling/emf/.
[2] See http://www.eclipse.org/ecoretools/.

P. Van Gorp and G. Engels (Eds.): ICMT 2016, LNCS 9765, pp. 19–35, 2016.
DOI: 10.1007/978-3-319-42064-6_2

EMFCompare[3]), GMF[4], and various model repositories. For example, these and other model-based tools could be used to explore, compare, and evolve the input and output of grammar-based tools.
- The analysis, transformation and development tools of the grammar-based approaches (e.g., parser generators, fact extractors, rewriting engines, refactoring tools, code generators, and language workbenches) become applicable to models. For example, mature techniques for refactoring, static analysis and program transformation become can be leveraged on model-based representations.

More specifically, we will explore whether and how Rascal—a meta-programming language that seamlessly integrates functional programming, flexible static typing, algebraic data types (ADTs) and grammar-based analysis and transformation—can be used as a bridge. A key question is then how to represent cross references in a tree-based setting that supports only immutable data. We present a simple framework, Refs, for representing graph-structured models as immutable values, based on unique identities and structure-shy traversal (Sect. 2). Refs is illustrated with simple transformations on state machines. We then show how general, class-based meta-models used in model-driven tools are mapped to Rascal's ADTs, augmented with Refs (Sect. 3). We have validated Refs by implementing a sample of well-known model transformations, ranging from the ubiquitous example of transforming families to persons, to executing UML Activity Diagrams (Sect. 4). We discuss results of our experiments and related work in Sect. 5. The results as presented should be seen as a proof-of-concept rather than a mature technology for model analysis and transformation in Rascal. All the code of Refs and the sample of model transformations can be found online at https://github.com/cwi-swat/refs.

2 Encoding References in Rascal

2.1 Essential Language Features

Rascal[5] is a functional programming language targeted at meta-programming tasks [14]. This includes source code transformation and analysis, code generation and prototyping of programming languages. Rascal can be considered a functional language, since all data is *immutable*: once values have been created they cannot be modified and the closest one can come to a mutable update is by creating a new value that consists of the original value with the desired change. Nevertheless, the language features mutable variables (see below) and in combination with higher-order functions, this enables representing mutable objects using *closures*: functions packaged with their variable environment. In addition to these latter features, the following Rascal features also play an important role in our proposal.

[3] See https://www.eclipse.org/emf/compare/.
[4] See http://www.eclipse.org/gmf-tooling/.
[5] See http://www.rascal-mpl.org.

Rascal features a static type system which provides (possibly parameterized) types and a type lattice with **void** at the bottom and **value** at the top. Two features are relevant for the current paper. First, types can be *reified*, i.e., types can be represented as and manipulated as values. Second, all user-defined ADTs[6] are a subtype of the standard type **node**. This makes it possible to write generic functions that are applicable for any ADT. We will use this to define type-safe, generic, functions for model manipulation.

When analyzing or transforming programs in real programming languages, many distinct cases have to be considered, one for each language construct, and the visit order of nested constructs has to be programmed explicitly leading to a lot of boiler-plate code. *Structure-shy matching* (only matching the cases of interest) and *traversal* (automating the visit of nested constructs) using Rascal's **visit** statement and deep match operator (/) enables matching and replacement of (deeply) nested subtrees without precisely specifying their surroundings. We will use this in the implementation of model transformation functions.

Very expressive variable assignments allow seemingly imperative coding style even though all data is immutable. As a first example, assume variable m is a map from strings to integers, its type is **map**[**str**,**int**]. The assignment `m["model"] = 3` will construct a new map value that reflects this modification and assigns it to m. Such assignments generalize over field lookup on tuples and constructor values.

Finally, functions and data constructors can be declared with optional *keyword parameters*.[7] When keyword parameters occur in function or ADT declarations, they should appear after ordinary parameters, and should be initialized with a default value. Keyword parameters are optional in function and constructor applications since a default value is always available from the corresponding declaration. The value of a keyword parameter is computed *on demand*, i.e., not when the function or constructor is called but at the moment that the parameter is retrieved during execution. In pattern matching, keyword parameters are ignored when left unspecified in a pattern, but matching a specific keyword parameter value can be done as well. We will exploit this by representing object identity by a keyword parameter that can be conveniently ignored by the model programmer and is only explicitly manipulated in our infrastructure.

2.2 Example: State Machines

Figure 1 shows ADTs capturing the structure of state machine models: Machine, State, and Trans. A machine has a name, and contains a list of states. The last argument of the machine constructor is a keyword parameter, representing the identity of the state machine. In this example the uid parameters are initialized with noId()—a function that throws an error, if the uid is accessed without being set explicitly. Recall from Sect. 2.1 that the default value is computed on demand.

States are then defined as a separate ADT, again with some arguments, and an identity. Finally, a transition is modeled as a value containing the triggering

[6] Recall that an Abstract Data Type is characterized by a set of values (created using *constructor* functions) and a set of functions that define operations on those values.

[7] Also known as *named parameters* or *keyword arguments* in other languages.

event name, and a reference to a state to. The generic type Ref[T] is used to model cross references (i.e., references which are not containment references). Its representation is not directly relevant for the user and is encapsulated by our framework.

Model values (e.g., values of the type Machine or State) need to have identity. Since it is cumbersome to deal with this manually, we introduce the concept of *realms*: these are spaces that manage sets of models of the same type. A realm is thus a technical/administrative mechanism that administrates all the identities of all the elements of all the models that have been created in that realm. All models are initialized via a realm, which ensures that newly created model values receive unique identities. For the realm concept, we use a record-of-closures representation—in this case a one element tuple consisting of the single closure named new—so that a realm statefully encapsulates unique id generation. A realm is created using the function newRealm(). A realm can then be used to initialize model values. For instance, a new Machine can be created as follows: Machine m = r.new #Machine, machine("someName", [])). The first argument represents a reified type (similar to Class<Machine> in Java) so that new creates a value of the right type. The second parameter represents a template for the model value. Note that the value for uid is not provided; it is precisely the responsibility of new to create a unique value for uid.

```
r = newRealm();
opened = r.new(#State, state("opened", []));
closed =  r.new(#State, state("closed", []));
opened.transitions = [trans("close", referTo(#State, closed))];
closed.transitions = [trans("open", referTo(#State, opened))];
doors = r.new(#Machine, machine("doors", [opened, closed]));
```

Fig. 1. Definition of state machine models using ADTs

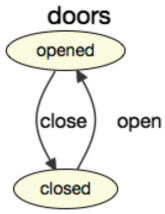

Fig. 2. Creation of a simple statemachine controlling doors in Rascal (left), and its automatically generated visualization (right)

An example snippet of Rascal code to manually create a simple state machine is shown in Fig. 2.[8] First, two states are created, initialized with empty lists of

[8] All visualizations are created automatically: for each meta-model we specify a transformation to a standard graph model. The latter is then visualized using Rascal's visualization library.

transitions. The transitions are added to the transitions field later, because they need to refer to the states themselves. Referring to another model value is done using the referTo function, which turns a Rascal value with an uid into an opaque reference value. Such reference values can be looked up using a generic lookup function given some root model that acts as the scope of the lookup. In the next section we show how referTo and lookup are used in model transformations.

2.3 Sample Model Transformations

Renaming Events. A very simple, endogenous, model-to-model transformation is the renaming of the event names in the transitions of a state machine. An example renaming could be achieved by (result shown in Fig. 3):

```
renameEvent(doors, "open", "OPEN")
```

This is expressed by the following function declaration:

```
Machine renameEvent(Machine m, str old, str new)
  = visit(m){ case t:trans(old, _) ⇒ t[event = new] };
```

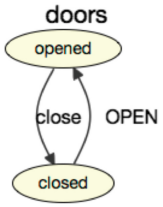

Fig. 3. State renaming

A visit takes an immutable value (in this case m), traverses it on a case-by-case basis, and returns a new value with possible local replacements when specific cases matched and defined a replacement. The single case matches all transitions with the name to be replaced (old), irrespective (_) of the state they go to. The matched transition is available as value of local variable t (bound using the colon :). The replacement for this case first assigns new to t's event field and inserts the new value of t in place of the originally matched transition. Every transition with an event equal to old will be replaced by a transition with a renamed event. As we already observed in Sect. 2.1, Rascal's pattern matching allows us to abstract from the uid keyword parameter, which is, however, automatically propagated to the replacement through t. Note also that Rascal's value semantics in combination with transitions having no uids (see the doors example in Fig. 2), works out extremely well here: the programmer does not have to worry about unintended sharing or aliasing.

Adding Reset Transitions. Another simple endogenous model transformation on state machines is the addition of reset transitions: when a reset event occurs, the machine should reset to its initial state. An example is (result shown in Fig. 4):

```
addResets(doors, {"reset"})
```

The following function addResets achieves this:[9]

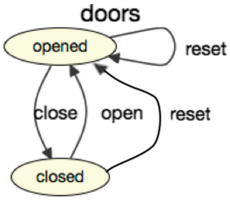

Fig. 4. Reset transition

[9] The concept of reset events is inspired by Fowler's example state machine DSL [5].

```
Machine addResets(Machine m, set[str] events) {
  resets = [ trans(e, referTo(#State, m.states[0])) | e ← events ];
  return visit (m) { case s:state(_, ts) ⇒ s[transitions=ts + resets] }
}
```

The comprehension in the first statement creates a list of transitions on each event `e` in `events` which has a reference to the initial state as target state. The **return**-statement creates the result using another **visit** construct: anywhere there's a state, it will be replaced with a state which has the additional transitions.

Flattening Inheritance. Flattening inheritance is a transformation which pushes down all inherited fields in class based meta-models. In this example the models are actually meta-models, conforming to the data type shown in Fig. 5. It is similar, but slightly simpler than meta-modeling formalisms such as Ecore [20] or KM3 [12]. This data type shows one more convenient feature of Rascal's keyword parameters: they can be declared on the ADT itself, which means that all constructors of that type will get them. So for type `Type`, both `class`, `prim` and `enum` value will have a `uid`. The following function implements flattening inheritance on meta-models conforming to the ADT of Fig. 5:

```
data MetaModel = metaModel(list[Type] types);

data Type(Id uid = noId())
  = class(str name, list[Ref[Type]] supers, list[Field] fields, bool abstract = false)
  | prim(str name)
  | enum(str name, list[str] values);

data Field(Id uid = noId())
  = field(str name, Ref[Type] typ, bool many = false, bool optional = false,
          bool containment = true,  Ref[Field] inverse = null());
```

Fig. 5. Rascal ADT describing MetaModels

```
MetaModel flattenInheritance(Realm realm, MetaModel mm) {
  Type flatten(Type t) {
    supers = [ flatten(lookup(mm, #Type, sup)) | sup ← t.supers ];
    t.fields = [ realm.new(#Field, f) | s ← supers, f ← s.fields ] + t.fields;
    return t;
  }
  return visit (mm) { case t:class(_, _, _) ⇒ flatten(t) }
}
```

This time, the transformation function receives a realm in addition to the meta-model to be transformed, since during the transformation new field objects need to be created: if a field of a super class is pushed down to two distinct subclasses, each of those new fields has to have its own identity. The local function `flatten` within `flattenInheritance` does the real work. It retrieves a list of supers from the

type t. Since this is a list of references, the lookup function is used to actually find the classes corresponding to those super types. These super classes are recursively flattened. For each of the flattened superclasses in supers a new field is created and added to the list of fields of t. Note that realm.new(#Field, f) actually clones the field f: only its identity will be different. Finally, the meta-model mm is transformed by replacing each class by its flattened version.

2.4 Implementing Refs

We have already shown how Refs can be used, and to eliminate all remaining mystery we now give a description how its constituents, the types Ref, Realm and Id, and the functions lookup and referTo can be implemented in Rascal.

Representing References Ref and Id are simply defined as (parameterized) ADTs:[10]

```
data Ref[&T] = ref(Id uid) | null();
data Id = id(int n);
```

Resolving References. The type-parametric function lookup requires more explanation:

```
&T lookup(node root, type[&T<:node] t, Ref[&T] r) = x
  when /&T x := root, x.uid == r.uid;
```

Its purpose is to resolve the reference r in a given model (represented as ADT, hence as a tree) root (root is of type **node** so all ADTs are acceptable). The second parameter t is a reified type denoting the type of the model element we are looking for, and is used to bind the type parameter &T at runtime: it allows &T to be used inside of pattern matches (see the **when**-clause). Note that the type parameter is constrained to be a **node**, so that we can access its arguments using the dot (.) notation. The third parameter r represents a reference to model elements of type &T. Since lookup resolves the reference, this &T is also the return type.

The actual return value x of lookup is then computed and bound in the **when** condition: /&T x := root performs a deep match on the root model and binds x to every model element of type &T, and then x.uid == r.uid checks that the uid of the matched element is equal to the uid of the given reference r.

Referring to Model Values. The function referTo turns model values of type &T into references of type Ref[&T]. To do this, it simply fetches the uid of the given model element x and creates a new reference to that uid:

```
Ref[&T] referTo(type[&T<:node] t, &T x)  = ref(x.uid);
```

The primary purpose of this function is to encapsulate the representation of references.

[10] The type parameter of Ref is technically not needed, but allows declarations of Refs document what they are referring to.

Realms: Scopes for Creating Model Values. The last component of our Refs implementation is the type Realm, which is an alias for a single element tuple:

```
alias Realm = tuple[&T(type[&T<:node], &T) new];
```

The tuple element is called new, a type-parameterized function with two arguments: a reified type, and an actual value of that same type that acts as a template for the new value that is created. The only way to construct a realm is by using the function newRealm:

```
Realm newRealm() {
  int n = −1;
  &T new(type[&T<:node] t, &T x) { n += 1; return x[uid=id(n)]; }
  return <new>;
}
```

It creates a closure that wraps a counter n for generating new uids, and a function new that increments the counter and assigns it as uid to its argument value x. The returned Realm value is a tuple with the new function as its single element. Since the local function new captures its environment containing n, every invocation of the new field of a realm will produce a model value with a new identity.

3 Mapping MetaModels to ADTs

In order to bridge the gap between model-based tools and Rascal, existing meta-modeling formalisms need to be mapped to ADTs. The previous section focused on how to encode references using unique ids and 4 helper functions. In this section we sketch how traditional meta-models can be represented using ADTs in Refs.

We assume that meta-models are structured similar to the meta-meta-model of Fig. 5. It defines classes, primitives, and enums. Classes can be abstract or not, contain a number of fields, and reference zero or more super classes. Each field has a type (class, primitive or enum), and defines a number of properties, such as whether it is a collection field, whether it is part of the containment hierarchy, or whether it is optional or not. Field definitions in meta-models often declare inverse relations (or "opposites") to support bidirectional navigation. For the remainder of this section, however, we leave this information implicit, since in Refs we have no way to maintain such relations automatically.

The first challenge is to deal with inheritance. ADTs do not support inheritance, so we preprocess a meta-model in two steps:

1. Flatten inheritance: push down all fields from super classes to all concrete classes (see Sect. 2.3 for an implementation).
2. Generalize type references: replace all references to a class C with a reference to the largest super class of C.

The first step allows us to represent all subclasses of some top class as a single ADT. The second step ensures that references to any of the subclasses can be

typed as (Refs of) that ADT. A consequence is that the meta-model becomes less restrictive. For instance, the "supers" field of class "Class" in a meta-meta-model typically is of type "Class". However, after generalizing type references, the type of the "supers" field will be "Type", since that is the largest super class of "Class". Although technically, this would allow use to create invalid models, we find it is an acceptable price to pay in exchange for a simple and direct encoding.

A preprocessed meta-model can then be translated to an ADT using the following procedure:

1. For each enum type E, define an ADT **data** E $= \mathsf{v}_1(), \ldots, \mathsf{v}_n()$, where each of the values v_i is mapped to a nillary constructor $\mathsf{v}_i()$.
2. For each top class C (a class which has no super classes), define the corresponding ADT with identity **data** C(Id uid = noId()) = ...
3. For each concrete class C' below a top class C define a constructor **data** C = c' (...).
4. For each field f of C', introduce a constructor parameter with the same name, and determine the type as follows:
   (a) If f has an enum type, use the corresponding ADT type.
   (b) If f has a class type, use the corresponding ADT type, and apply Table 1 to deal with multiplicity, containment, etc.
   (c) If f has a primitive type, use the corresponding Rascal primitive type and also apply Table 1, assuming that "Containment" is true.

Note that the ADT type for a class in step 4(b) always exists, because of generalizing type references during preprocessing. Note further that optionality for contained elements as per Table 1 require the use of the auxiliary Opt data type. Alternatively, however, for primitive fields, optionality could be represented using a keyword parameter with a sensible default value.

Table 1 shows how various combinations of field properties are encoded as Rascal types. An en dash in one of the property columns indicates "for either

Table 1. Deriving the type from meta-model field properties.

Containment	Optional	Many	Unique	Ordered	Encoding
–	–	True	True	True	–
True	–	True	True	False	set[T]
True	–	True	False	True	list[T]
True	–	True	False	False	map[T, int]
True	True	False	–	–	Opt[T]
True	False	False	–	–	T
False	–	True	True	False	set[Ref[T]]
False	–	True	False	True	list[Ref[T]]
False	–	True	False	False	map[Ref[T], int]
False	–	False	–	–	Ref[T]

true or false". The combination of the first row is unsupported, since Rascal has no data type for ordered sets. Contained references and primitives are ordinary child elements of constructors. If they are optional, they are wrapped in a generic Opt data type. Whenever a field is a non-containment reference, the Ref type is used. Since Refs contain only unique identities, uniqueness in sets will be based on reference equality. The same is true for references in multi-sets which are encoded using maps from value to int. Note that Refs are optional by default, via the null() constructor (see Sect. 2.4). The mapping above assigns uid fields to all ADTs. A possible refinement is to omit the uid field whenever there is no cross referencing field defined anywhere in the meta-model.

4 Exploring Refs for Model Transformation

4.1 Sampling Model Transformations

To validate our Refs solution we have made a selection of model transformations based on three criteria. First, the transformation should involve cross references. Without cross references, models are plain trees, which are well supported by the functional programming paradigm. Second, the examples should exercise various kinds of model transformations. This includes, Model-to-model (M2M), model-to-text (M2T) and text-to-model (T2M). But also, endogenous (type preserving) transformations and exogenous (type transforming) transformations [17]. Finally, the set of transformations should cover different transformation purposes, such as analysis, refactoring, translation, or execution.

Table 2 gives an overview of the defined meta-models with their description and size in SLOC.[11] Table 3 gives an overview of name, category (endogenous/exogenous), kind, description and size in SLOC of the implemented transformations. Transformations are grouped according to the source meta-model: Family and Persons, state machines (Fig. 1), meta-meta-models (Fig. 5), and UML Activity Diagrams [16].

Table 2. Overview of meta-models defined in Rascal

Meta-model	Description	SLOC
Family and persons	Families of named, male or female members	11
Statemachine	Statemachines with states and transitions (Fig. 1)	6
ADT	Algebraic Data Types with named constructors	14
Regexp	Regular expressions with choice, sequence and repetition	6
Model	Simple meta-models with classes, fields, primitives, and enumerations (Fig. 5)	13
Graph	Graphs with nodes and edges	9
Activity	UML Activity diagrams	56

[11] Source lines of code, not counting empty lines or comments.

Table 3. Overview of implemented transformations

Name	Cat	Kind	Description	SLOC
family2persons	Exo	M2M	Extract persons from a family	5
family2graph	Exo	M2M	Convert family to graph	34
renameEvent	Endo	M2M	Rename events in SM (Sect. 2.3)	2
addResetTransitions	Endo	M2M	Add transitions to SM (Sect. 2.3)	4
regexp2Statemachine	Exo	M2M	Convert regular expression to SM	44
statemachine2DFA	Endo	M2M	Determinize state machine	35
parallelMerge	Endo	M2M	Merge states in SM	23
statemachine2Graph	Exo	M2M	Convert SM to graph	5
flattenInheritance	Endo	M2M	Push down fields (Sect. 2.3)	9
generalizeTypeRefs	Endo	M2M	Change field types to largest super class	6
metaModel2Relational	Exo	M2M	MM to relational schema	57
metaModel2Java	-	M2T	From MM to Java code (text)	78
metaModel2Graph	Exo	M2M	Convert MM to graph	11
metaModel2ADT	Exo	M2M	Convert MM to ADT	38
source2Activity	-	T2M	Textual activity model to Activity Model	142
activity2Graph	Exo	M2M	Activity model to graph	7
executeActivity	Endo	M2M	Execute Activity Model	258

Although we have not performed a thorough comparison with existing solutions, it can be seen in Tables 2 and 3, that our solutions are very concise. For instance, family2persons is 41 SLOC in ATL[12] versus 5 lines in Rascal. We claim (but have not validated) that our solutions are at least as readable as the solutions we compare with. The largest model transformation is execution of Activity Diagrams, which we will discuss in more detail below.

4.2 Case Study: Executing Activity Diagrams

The transformation executeActivity involves executing activities by transforming a run-time model. The run-time state is expressed as part of the model itself, and steps in the execution consequently represent small modifications of the complete model. The total code size of this implementation is 56 SLOC for the meta-model, and 258 SLOC for the transformation code. This latter number includes 12 SLOC consisting of the modular extension of the meta-model ADT to represent runtime state. Compare this to the Java classes (partially generated from an Ecore meta-model) of the reference implementation for activity execution, which consists of 3704 SLOC.[13]

Activity Execution can be considered a pathological case from the perspective of typical model transformation use cases. The actual run-time state is represented as an extension of the static activity meta-model. For instance, the chal-

[12] See http://www.eclipse.org/atl/atlTransformations/.
[13] Can be found on Github at http://bit.ly/1puz0tC.

```
data Activity(Opt[Trace] trace = none());
data ActivityNode(bool running = false, list[Token] heldTokens = []);
data ActivityEdge(list[Offer] offers = []);
data Variable(Opt[Value] currentValue = none());
```

Fig. 6. Modular extension of the Activity ADT to represent runtime state

lenge described in [16] states that variables get an additional currentValue field, activities maintain a trace of executed nodes, activity nodes are either running or not, and hold a list of tokens. Finally, activity edges own a list of offered tokens. In Rascal this extension could be modularly defined through the use of keyword parameters, as shown in Fig. 6. The existing data types for Activity, ActivityNode, ActivityEdge and Variable, are simply extended with additional parameters, which will be available to all constructors of each respective data type. For brevity, we have omitted the new (run-time only) data types Trace, Token, and Offer.

Every step in the computation "transforms" the model, by changing values and relations within the augmented model. In the reference implementation (and many of the solutions submitted to the Transformation Tool Contest [19]), this is realized by mutating the relevant fields of the model objects in the methods that implement the interpreter. In our case however, each function really performs a transformation in that every modification results in a new activity model! Unfortunately, this also means that for use cases which heavily depend on frequent mutation of a model, our Refs-based framework is impractically slow. We have been able to run the tests provided of the TTC'15 case, but for the performance tests our implementation of activity execution performs extremely badly. The obvious reason being that every lookup or update of the model, requires traversing it. These results, however, must be qualified: executing a model by modifying it directly is very atypical in functional programming style, where runtime state of an interpreter is typically managed separately (e.g., in environments and stores).

Nevertheless, our performance experiments suggest two mechanisms for improvement. First of all, lookup can be memoized.[14] In fact, Rascal supports a @memo attribute which can be attached to any function declaration to enable memoization. As a result, looking up the same reference on the same model multiple times avoid traversing the model. Second, another way to avoid traversing the model upon lookup, is to make sure that a mapping of identities to model values is always available at the root model. This could be implemented by attaching (immutable) "companion" maps to constructor values (e.g., as a keyword parameter). Such a map links object identities of contained subterms to the actual subterms. Whenever a constructor is modified (e.g., a child element is replaced), the reference maps of the children are propagated to the parent automatically. As a result, the companion map of the root model can be used to lookup all defined entities that are contained by it. Although this seems a

[14] An optimization technique that caches the result of expensive computations.

structural solution to the problem of lookup, it would require modifying the Rascal runtime system to implement the propagation.

5 Discussion and Related Work

Discussion. Relative to the taxonomy of model transformation approaches of [3], Refs offers an operational, functional, term-based, statically-typed, modular, non-incremental, model-to-model approach. Refs can also easily accommodate template-based model-to-text transformations. Rule application is fully deterministic and explicit, but the **visit** construct can be used to automatically schedule declarative rules as well. As of now, Refs does not support transparant traceability, as is provided by model transformation languages like ATL [11] or ETL [15]. Keyword parameters could however be used to transparantly represent trace information. Inserting such trace links should be performed explicitly. The generality of Rascal as a programming language would make automated support quite challenging: traceability follows from data flow dependencies which can have arbitrary structure. Further work is needed to generalize existing approaches to origin tracking [10,23].

Looking at the sample of model transformations and case-study, the first thing to observe is Rascal's pattern matching facility is a clear win. This enables structure-shy traversal and transformation using **visit**, and is very useful for model transformations. As an added bonus, the unique id field can be ignored during matching, eliminating some boilerplate code. Furthermore, model values are what-you-see-is-what-you-get (WYSIWYG): they are fully self contained, can be written to file, or printed on the console during debugging.

The fact that model values are immutable also implies that mutations always produce new versions of the model. As a result intermediate stages of the model during a transformation process can be easily captured, inspected and stored. Features like "undo", tracing a transformation, comparing successive states of model states using difference algorithms, are trivial to realize. In a mutable world these features would be much harder to achieve.

Finally, an added benefit of immutable models is that model elements that are never the target of cross references do not need identity. This means that such model elements behave like proper immutable values. As a consequence there is no ambiguity regarding equality, or what it means when such an element is put in a set. For those truly immutable sub parts of a model, the developer of a model transformation can switch off thinking about references entirely, if so desired. This is most valuable in models that, for instance, represent expression languages.

An important difference between existing model transformation languages and Refs is that in the latter object creation, reference lookup and mutation are explicitly scoped. For instance, object creation is scoped by a realm. Both lookup and mutation are scoped by a root model. As a result, these scope "objects" need to be available whenever objects are created, looked up, or updated. The model transformations listed in Table 3 explicitly pass these objects through the functions that make up the transformation, or define a (module-level) global variable.

Even though our experience in writing model transformations in Rascal using Refs has been largely positive, the interaction between copying assignment and referential integrity can be subtle. For instance, when an element (with identity) is removed from a model, this might cause a dangling reference elsewhere, since the references are essentially symbolic. The programmer needs to manually ensure that existing references to the elements are nulled or cascade deleted. Of course, we could provide a generic library function to achieve this.

Another effect of automatic copying of immutable values is the creation of two different values with the same identity. When both such values are inserted into a model, then this model accidentally contains two different nodes with the same identity, which should never happen. It is the responsibility of the programmer to ensure that two such nodes only exist temporarily, if ever. A strategy to cope with this problem that is applied quite often in our sample of model transformations is maintaining tables, indexed on identity, so that the "modifications" on the map entries are always performed on the same element.

The biggest drawback of our current implementation of Refs is that lookup requires search. As discussed above, for larger models, with lots of in place mutation, searching through the model for every reference—even when using memoization—leads to impractical performance. Note however, that in-place mutation of models can be considered an anti-pattern in functional programming. For static model transformations which typically traverse the source model only once, the performance penalty of lookup is much less severe.

Related Work. Bridging grammarware and modelware has received a lot of attention, especially in how to map grammar based formalisms to meta-modeling frameworks, see for instance in [8,13,22]. The use of textual representations of models is generally recognized as being beneficial for productivity and tool development [8]. Most of this work concerns *front-end mappings*, i.e., providing mappings between models represented in worlds based on different modeling concepts. Work on *back-end mappings* that consider model transformations across different modeling worlds are scarce. The subject of model *transformation* using grammar-based tooling has also been relatively unexplored and this is where we make a contribution. We can completely focus on the problem of representing references and model transformations, since the Rascal language workbench takes care of all other bridging aspects like grammars, parsing, storage, symbol tables, semantic processing, and IDE support.

For various languages, embedded DSLs exist aiming at model analysis and transformation. For instance, FunnyQT [9] is a Clojure library providing model querying and transformation services based on in-place (mutable) transformations. Another approach is based on embedded DSLs for model transformation in Scala [6]. All these efforts use some form of mutability to achieve their goals, while we depend on a strictly immutable representation of models.

Representing References with Immutable Data. A first approach is to see a model as a graph in the mathematical sense: model elements represent nodes, non-primitive fields are edges. Such graphs can be easily represented as a (binary)

relation. Binary relations have the advantage that all operators from relational algebra like, join, intersection, projection and transitive closure are available for querying models. The disadvantage, however, is that transformations on the graph representation are hard to express in a functional style.

A *path* is a well-known method to describe a connection between two nodes in a graph or between the root of a tree and one of its (grand)children. In term-graph rewriting [21] a path is a simple list of integers denoting the indices of edges to be taken along the path. Paths in the context of model transformation could be represented as sequences of field accesses and collection indices. A reference to a model element could be encoded as such a path, starting at the root of the model. Unfortunately, these paths become out of date as soon as the model itself is transformed.

Assigning an identity to a graph node is a non-issue in an imperative or object-oriented setting, where a pointer or an object identity are readily available. In a database context an automatic *primary key* can be associated with records for later reference. Primary keys, however are local to an entity or class type, so to interpret a foreign key one needs information about the schema. Refs simulates these models using unique identities, which are scoped relative to a realm, instead of globally, or locally.

Representing graph structured data in functional programming is a well-researched problem. Erwig [4] introduces an inductive approach for defining generic graphs and graph algorithms in Haskell. Claessen and Sands [2] introduce a simple extension to Haskell based on non-updateable reference cells, together with an equality test in order to make sharing observable. Gill [7] presents an alternative solution based on generic reification of values with unobservable sharing to graphs with observable sharing. A more recent approach is based on structured graphs [18], which uses recursive binders inspired by parametric higher-order syntax [1] to represent cycles. It is, however, as of yet unclear, how to express non-trivial model transformations in these styles.

6 Conclusions

A lot of research on bridging modelware and grammarware has been focused on how to map textual concrete syntax to model-based abstract syntax. In this paper, have explored a similar bridge from the dual perspective of model transformation. We have presented a simple encoding of cross references in Rascal, a functional meta-programming language, featuring immutable data. The experience of implementing a sample of well-known model transformations has been largely positive: transformations are very concise, and can fully exploit Rascal's powerful pattern matching and traversal primitives. However, some directions for improvements are clearly visible. Performance seems to be adequate for model transformations in general, but starts to degrade quickly when models are traversed and updated frequently (as happens in the case of model execution). Further research is also needed into language extensions of Rascal to make model transformation even more elegant and efficient. The next step is to investigate mappings from and to meta-model formalisms, so that existing modeling

technology can be leveraged from within Rascal, as well as the other round, that Rascal can be applied in model-driven engineering scenarios.

References

1. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: ICFP 2008, pp. 143–156 (2008)
2. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Thiagarajan, P.S., Yap, R.H.C. (eds.) ASIAN 1999. LNCS, vol. 1742, pp. 62–73. Springer, Heidelberg (1999)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
4. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. **11**(05), 467–492 (2001)
5. Fowler, M.: Domain Specific Languages, 1st edn. Addison-Wesley Professional, Boston (2010)
6. George, L., Wider, A., Scheidgen, M.: Type-safe model transformation languages as internal DSLs in scala. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 160–175. Springer, Heidelberg (2012)
7. Gill, A.: Type-safe observable sharing in Haskell. In: Haskell 2009, pp. 117–128. ACM (2009)
8. Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 169–184. Springer, Heidelberg (2008)
9. Horn, T.: Model querying with FunnyQT - (extended abstract). In: ICMT 2013, pp. 56–57 (2013)
10. Inostroza, P., van der Storm, T., Erdweg, S.: Tracing program transformations with string origins. In: Di Ruscio, D., Varró, D. (eds.) ICMT 2014. LNCS, vol. 8568, pp. 154–169. Springer, Heidelberg (2014)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1), 31–39 (2008)
12. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
13. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3), 331–380 (2005)
14. Klint, P., van der Storm, T., Vinju, J.: EASY meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
15. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
16. Mayerhofer, T., Wimmer, M.: The TTC 2015 model execution case. In: TTC 2015, pp. 2–18 (2015)
17. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
18. Oliveira, B.C., Cook, W.R.: Functional programming with structured graphs. ICFP **47**(9), 77–88 (2012)

19. Rose, L.M., Horn, T., Krikava, F. (eds.): TTC 2015, CEUR Workshop Proceedings, vol. 1524. CEUR-WS.org (2015)
20. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, Upper Saddle River (2008)
21. Terese. Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)
22. van der Storm, T., Cook, W.R., Loh, A.: The design and implementation of object grammars. Sci. Comput. Program. **96**(P4), 460–487 (2014)
23. Van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symbol. Comput. **15**(5), 523–545 (1993)