Rewriting Object Models With Cycles and Nested Collections

A Model-Based Metaprogramming Problem

Markus Lepper[1] and Baltasar Trancón y Widemann[1,2]

[1] `<semantics/>` GmbH, Berlin, DE
[2] Ilmenau University of Technology, Ilmenau, DE

post@markuslepper.eu

Abstract. Metaprogramming with classical compiler technology is similar to model-based code generation. We discuss a particular tool, `umod`, that generates complex data model implementations in Java, and a particular aspect of its support for declarative programming: The rewriting of data models in object-oriented style, based on the visitor pattern, with support for arbitrary reference graphs and nested collection-valued fields. We demonstrate that concerns of both compiler theory and model-based development apply, and that the distinction is overshadowed by a general commitment to semantic rigour.

1 Introduction: Compilation, Model-Based Code Generation, and Metaprogramming

The disciplines of classical compiler construction and model-based code generation are widely acknowledged, different community viewpoints aside, to address largely the same basic theoretical problems. One commonly cited characteristic difference in practice is the role assigned to the resulting code artifacts in the software development process:

On the one hand, a classical compiler is typically expected to take textual input code written in some more or less well-established programming language and legible for the programmer, and produce output binary code legible for some real or virtual machine, in the successful case without requiring or providing occasion for user interaction. The compiled program is then expected to run out of the box, or in the case of modular separate compilation, to integrate with other compiled modules (in traditional obscurity called "object code") by a comparatively simple cross-referencing procedure performed by a linker tool.

On the other hand, model-based code generators, in their pure form, feature input "languages" or model formats that are not programming languages in the classical sense, because they are defined by metamodelling rather than grammar, presented and edited visually rather than textually[3], and/or arbitrarily "domain-specific", that is specifically created for a small, or even singular, set of projects.

[3] Although tools that bridge classical grammar technology and modeling frameworks, such as Eclipse Xtext, are becoming increasingly popular.

The output is then expected to be fed into the project repository alongside hand-written code, and integration often requires nontrivial user intervention: for instance by completing stubs and skeletons, actual editing of generated code (frowned upon by software engineering purists, but pragmatically very useful), or using whatever complex interfacing mechanisms of physically disconnected code fragments the target language provides (such as inheritance in the object-oriented world).

In this paper, we present a particular effort in the construction of semantically sound programming language tools. It demonstrates that the distinction suggested by the above summary is blurred, and by far secondary to the distinction of rigorous and ad-hoc approaches. By sharing our experience, we intend to make a modest contribution to the exchange of ideas and techniques between the two disciplines, and general understanding of their relationship. To this end, we shall

1. discuss a subfield of classical compiler construction that predates model-based approaches (or at least their currently associated buzzwords) but has many requirements, problems and strategies in common, namely the implementation of *generative* or *meta*-programming;
2. present a case study from our own research on metaprogramming tools and techniques, which builds on a classical compiler constructors' viewpoint, but is related to mainstream model-based methodology closely enough to carry some illuminating analogies.

We shall start our discussion by refuting more precisely the distinction as outlined simplistically above, giving two particular arguments to the contrary.

The first concerns the often-cited distinction of model inputs having a distinctively higher level of abstraction than mere programs or being characteristically "non-executable". This is difficult to justify precisely:

> [Automated programming] *always has been a euphemism for programming with a higher-level language than was then available to the programmer. Research in AP is simply research in the implementation of higher-level programming languages.* [**?**]

The second, related argument concerns the level of abstraction of the back-end rather than the front-end. Delegating tedious and banal coding tasks to the computer has, evidently, always in the history of programming technology been a major goal. Examples of ad-hoc solutions abound, especially in self-application contexts (for instance see the GNU compiler collection [**?**]), but more generic tools (from LISP hygienic macros to parser generators) have also emerged early.

When such program fragment-generating tools, or metaprograms, are seen by the programmer as an *extension of his own productive capabilities beyond manually typing* code in his target language of choice, the essential distinction between programs and models, and the accidental one between compiler output being opaque and model-based generator output being transparent also vanish.

Tools functioning this way can benefit technically and semantically from the vast body of knowledge of classical compiler construction, but at the same time face many of the extra requirements and issues addressed by model-based code generation. We are confident that the investigation of particular problems and solutions in this area, such as the example technique discussed in the main part of this paper, can help to leverage the synergies of the complementary approaches.

1.1 The $^{\text{meta}}$-tools Approach

The example metaprogramming technique to be discussed below is part of the $^{\text{meta}}$-tools suite [?,?], an extensive collection of programming tools which, by its overall design strategy, is placed in the middle ground between classical compilers and model-based approaches.

The $^{\text{meta}}$-tools suite is designed to amplify the productivity of software development centered around the core technologies Java and XML, by leveraging high-level, declarative and semantically rigorous concepts, notations and styles. Technical implementations are provided through a pragmatic combination of libraries and style patterns (where the expressivity of the host platform suffices) and metaprogramming tools (where it must be transcended). Automatically generated code is human-readable throughout, and interfaced cleanly using the two modularity concepts provided by the Java host language: type parameterization and inheritance.

The $^{\text{meta}}$-tools have been validated mainly in self- and cross-application as well as the construction of other compilers, but also in the rapid prototyping of other medium-scale applications, in particular with emphasis on nontrivial algorithms and data structures.

2 The Data Model and Processor Generator `umod`

2.1 Models

A major component of the $^{\text{meta}}$-tools suite is the data model definition language and implementation generator `umod`. It provides a concise and expressive notation for specifying complex graph-like data structures. A `umod` model is a collection of model elements, represented as Java objects. The `umod` compiler generates Java code for the implementing classes from a model specification. Generated code comes with sophisticated support for many features: element subtyping, complex collection-valued attributes, pervasive early detection of spurious `null` references, inheritable constructor signatures, combinator-based pretty-printing, reified getters and setters for point-free programming (à la higher-order functional programming), stable deep equality, pattern matching, etc.

The code generated by `umod` thus emulates many desirable features of algebraic data types, but comes with the full power of object-oriented programming, in particular unrestricted access to low-level (imperative) programming constructs if necessary, and the ability to deal with arbitrary data graphs rather than trees. We have applied `umod` to generate abstract syntax representations for several other code generator tools, both classical compilers and other $^{\text{meta}}$-tools components.

Figure 1 shows a typical example from a real-word compiler construction project: The `umod` source text defines a model named `Sig`, to be translated into a Java `package`. The top level model element class `Statement` is realized as an `abstract` superclass of both `Assignment` and `Block`, etcetera. Both subclasses have fields (instance variables) named `left`, `right` and `stmts`, respectively. In the same text line follows an expression, giving the type of the field, and optionally, after the "`!`" character, a traversal indication (see next section).

```
MODEL Sig
VISITOR 0 Visitor
VISITOR 0 Rewriter IS REWRITER
// ...
TOPLEVEL CLASS
  Statement ABSTRACT
  | Assignment
          left   SEQ Variable                      ! V 0/0
          right  Expression                        ! V 0/1
  | Block
          stmts  SEQ Statement                     ! V 0/0
  Expression ABSTRACT
  | Reference
          var    Variable                          ! V 0/0
  Variable

  Statistics
          vars   Statement -> bool -> SET Variable  ! V 0/0 L RR
```

Fig. 1. Model Definition `umod` Source (Excerpt)

The syntax of the type declarations and the resulting carrier sets will be discussed in detail in Section 3.1. For practical programming it is important that the type constructors be fully compositional, as shown in the last field definition line of Fig. 1, and are covered by all features listed above.

2.2 Visitors

The `umod` system also provides code support for, and fine-grained control over, the *visitor* style pattern, the standard high-end control abstraction of traversal strategies for structured data in object-oriented programming [**?**,**?**]. The visitor pattern provides a concise, elegant, safe and robust style of associating data elements with effects: hence it is a prime example of *declarative* style (stating the "what" in user code and delegating the "how" to a lower level such as a library provider, or in our case, a generator). For data models that represent programs, the visitor machinery can be seen as the backbone of an *interpreter* (even if most actual visitors in a language processing tool will interpret only a small aspect each.)

Returning to the example, the annotations following the field types in Fig. 1 at the end of the source line, define *traversal plans* as a basis for the generation of visitor code. The slash "/" separates the numeric identifier of a plan and a number controlling the sequential order when visiting the fields on the same level of class definition. The second line in the example requests for the source code of a visitor called "`Visitor`", following the traversal plan "`0`". (Different visitors can be derived from one particular traversal plan. The requested flavour is indicated by a suffix in the declaration, see the next line.) The generated code is sketched in Fig. 2.

```
abstract class Visitor {                         // traversal plan 0/
  // ...
  void action(final Block b) {
    for (final Statement s : block.get_stmts())  // traversal order /0
      match(s);
  }
  void action(final Assignment a) {
    match(a.get_left());                         // traversal order /0
    match(a.get_right());                        // traversal order /1
  }
}
```

Fig. 2. Generated Visitor Code (Excerpt)

The generated code realizes the pure traversal, according to the selected traversal plan. Any desired effects are added by the user, by subclassing and method overriding. For our example, we consider the task of *copy propagation*, a typical basic compiler pass that recognizes assignments which redundantly copy values, and eliminates them by substitution. The anonymous class in Fig. 3 implements the recognition phase by descending transparently into the depth of a program model (regardless of the intervening path, for instance via nested `Block` elements), processing the model elements of `Assignment` class, and recording those that match a suitable pattern.[4]

The visitor pattern is rooted firmly in the imperative programming paradigm. That is, its semantics are based on sequential side effects, and rarely investigated globally and formally. True to the spirit of the meta-tools emphasis on semantic precision, we have addressed the problem, and demonstrated the benefits of a formal operational model with a nontrivial optimization strategy [**?**].

2.3 Rewriters

A further issue with visitor is, in their basic form, a strong asymmetry between declarative *input* (essentially type-directed node matching, encoded into method

[4] Pattern matching of subgraphs is performed by the meta-tools component Paisley, which is tightly integrated with `umod`. See [**?**].

```
Program copyPropagation(Program prog) {
  final Map<Variable, Variable> copies = new HashMap<>();
  new Visitor() {
    @Override void action(Assignment a) {
      // match pattern "Assignment({x}, Reference(y))" against "a"
      if (/* success */)
        copies.put(x, y);
      super.action(a);                           // top-down traversal
    }
  }.match(prog);
  // ... see below ...
}
```

Fig. 3. User-Defined Visitor Code

signatures) and imperative *reaction* (arbitrary method bodies). This resembles the relation between parser code generated from grammar declarations and interspersed "semantic actions". As a more declarative way of writing we have proposed a complementary extension to the visitor pattern that allows for similarly declarative specification of non-destructive data *transformation*, called the *rewriter* pattern [**?**]. In the language processing scenario, the rewriter machinery can be seen as the backbone of a *compiler* pass.

Returning again to the example, the third line in Fig. 1 requests rewriter code for the same traversal plan. The generated code is much more complex than in the simple visitor case, because its main purpose is not only to traverse the model, but also to propagate all changes consistently throughout. Different flavours are supported, the most sophisticated being the non-destructive type, combining copy-on-need with aggressive sharing and cycle detection.

Figure 4 shows the generated methods, their call graph, and the possible intervening overrides by the user. The method `rewrite(C2)` is the entry point for the rewriting process for all instances of class `C2`. It decides by means of a cache, whether the object has already been successfully rewritten or, otherwise, whether a clone already exists. The latter case indicates dynamically a cycle in the model reference graph. Otherwise it creates and memorizes a clone "on stock". All these operations employ library methods from the infrastructure.

Then it calls the generated method `rewriteFields(C2)`. This steps through the fields selected by the traversal plan on class definition level `C2`, calling `rewrite(F)` on each field value recursively. It updates the clone whenever a different value is returned. In this code, for every field of *collection type* one or more program loops are generated, which iterate this process for all values and create the resulting collection. Additionally `rewriteFields()` on the superclass `C1` is called, in order to process inherited fields.

On return, `rewrite()` code checks whether changes in field values have occured, and returns either the original or the modified clone accordingly. Change

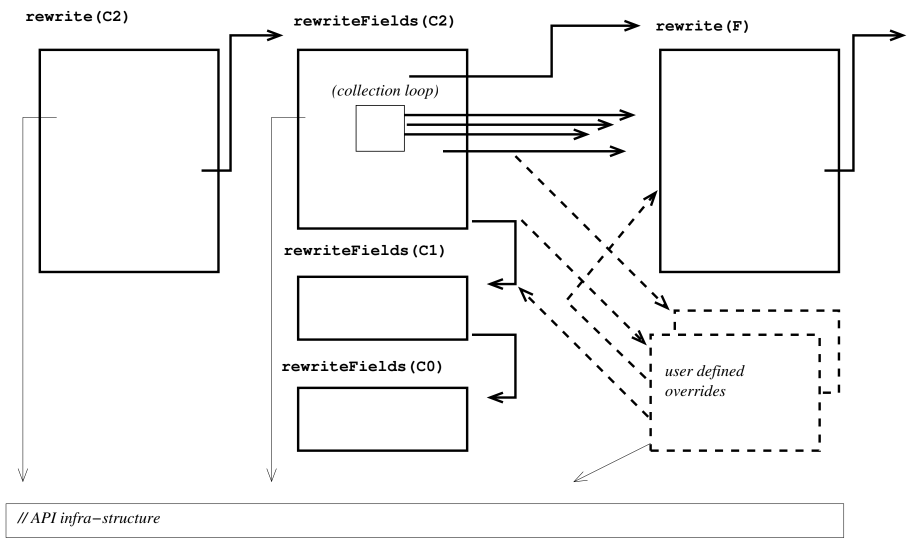

Fig. 4. Control flow in generated rewrite code and user override; cf. Fig. 2 and 3. *Arrow styles:* thick – generated control flow; thin – auxiliary API calls; dashed – interface to user code.

propagation is based not on single objects, but on *strongly connected components* (SCCs), as recognized by above-mentioned cycle detection.

Again, generated code behaves neutrally; the user defines the required transformation by subclassing and overriding, as indicated in Fig. 4. Both levels of generated methods can be overridden, and user code may re-use the generated methods. It also calls the infrastructure library for inquiring and modifying the state of caches and results. There the most important methods are

– `substitute(Object o)` – sets `o` as the result of rewriting the currently visited model element;
– `substitute_multi(Object... os)` – sets a list (of zero, one or more) model elements as the rewriting result.

The anonymous class in Fig. 5 uses the information collected by the code in Fig. 3 to implement the elimination phase of the copy propagation pass.

The rewriter approach to model transformation has many features of high-level declarative programming, notably: robustness against minor changes in the model definition, compositional organization of active code into fragments per model element class, and automated propagation of dynamic changes. But, in contrast to "pure" approaches such as attribute grammars, the declarative paradigm is broken deliberately at the level of user-defined code, where the full powers (and dangers) of the object-oriented host environment are exposed.

This decision, which greatly enlarges the class of expressible rewriting procedures, exposes some technical details. It gives the programmer control over, and responsibility for,

```
Program copyPropagation(Program prog) {
  final Map<Variable, Variable> copies = new HashMap<>();
  // ... see above ...
  return (Program)new Rewriter() {
    @Override void rewriteFields(Variable v) {
      if (copies.containsKey(v))
        substitute(copies.get(v));                // propagate
    }
    @Override void rewriteFields(Assignment a) {
      super.rewriteFields(a) ;                    // bottom-up rewriting
      // match pattern "Assignment({x}, Reference(y))" against "a"
      if (/* success */ && x.equals(y))           // now redundant?
        substitute_multi() ;                      // eliminate
    }
  }.rewrite(prog);
}
```

Fig. 5. User-Defined Rewriter Code

1. the execution order of rewriting actions in relation to the traversal effected by generated code (see [**?**] for a theoretical account);
2. the calling context, which may feature mutable state and stipulate restrictions on the type and multiplicity of local rewriting results.

The associated safety conditions can only be expressed partially in the static semantics of a host language such as Java. Our pragmatic solution maps as much to the type system as feasible, checks further conditions at runtime for fail-fast behavior, and leaves more difficult (or undecidable) issues to the user's caution.

So far, we have validated that rewriter-based programming competes favorably with more heavy-weight model transformation frameworks [**?**], and described very abstract and powerful denotational semantics for "well-behaved" object-oriented rewriters [**?**]. The overall goal can be described as allowing all kinds of user code in principle, but rewarding disciplined use with beneficial mathematical properties, for a flexible and conscious trade-off between rigor and agility. The following section explicates this strategy by discussing a novel problem, namely, how rewriting carries over to collection types.

3 Rewriting in the Presence of Nested Collections

3.1 Model Definitions in umod, Formally

The mathematical notation used in the following is fairly standard. It is inspired by Z notation [**?**], as it treats finite maps and sequences as relations with special properties, and thus allows the free application of set and relation operators and functions, as listed in Table 1.

$\mathbb{F}(A)$	Finite power set, the type of all *finite* subsets of the set A.
$A \rightarrow B$	The type of the *total* functions from A to B.
$A \nrightarrow B$	The type of the *partial* functions from A to B.
$A \twoheadrightarrow\!\!\!\!\!\!\not\;\; B$	The type of the *partial and finite* functions from A to B.
$A \leftrightarrow B$	The type of the relations between A and B.
$\mathsf{ran}\, a, \mathsf{dom}\, a$	Range and domain of a function or relation.
$S \lhd R$	$= R \cap (S \times \mathsf{ran}\, R)$, i.e. domain restriction of a relation.
$r^{\sim}$	The inverse of a relation
A^*	All possible finite sequences from elements of A, including the empty sequence.
$\mathbf{ID}_A$	$= \{a \in A \bullet (a \mapsto a)\}$, the identity relation.
$r \mathbin{;} s$	The composition of two relations: the smallest relation s.t. $a\ r\ b \wedge b\ s\ c \Rightarrow a\ (r \mathbin{;} s)\ c$

Table 1. Mathematical notation

Table 2 shows the components for defining the structure of a model, as far as needed for our problem: $\mathcal{C}_0$ is a finite set of predefined classes, e.g. imported from system libraries, and $\mathcal{T}_{\text{prim}}$ are some primitive data types. Any model declaration defines a finite set of classes C, the *model element classes*, i.e. the classes of the host language objects which together will make up one instance of the model. The total superclass function `extends` must be free of cycles, as usual, and well-founded in $\mathcal{C}_0$. F is the set of all field definitions, each related to one particular model class definition, indicated by *definingClass*; the collection of all fields of a certain class is given by *fields*.

Each field has a type, given by *fieldType*. Types are constructed in generationsas $T = \mathcal{T}_\tau$ for some arbitrary but fixed number τ. The zeroth generation $\mathcal{T}_0$ includes all predefined scalar types $\mathcal{T}_{\text{prim}}$, and non-null references to all external classes $\mathcal{C}_0$ and to all model element classes C. The further generations are made by applying the following *type constructors* in a freely compositional[5] way:

– `OPT` $\mathcal{T}_n$, – optional type, the special value `null` is allowed additionally.
– `SET` $\mathcal{T}_n$, – power set, contains all possible finite sets of values of T_n.
– `SEQ` $\mathcal{T}_n$. – sequence, contains all possible finite lists made of values from T_n.
– `MAP` $\mathcal{T}_{n,1}$ `TO` $\mathcal{T}_{n,2}$, abbreviated as $\mathcal{T}_{n,1}$ `->` $\mathcal{T}_{n,2}$, – all finite partial mappings from $\mathcal{T}_{n,1}$ to $\mathcal{T}_{n,2}$.
– `REL` $\mathcal{T}_{n,1}$`TO`$\mathcal{T}_{n,2}$, abbreviated as $\mathcal{T}_{n,1}$ `<->` $\mathcal{T}_{n,2}$, – all finite multi-maps/relations from $\mathcal{T}_{n,1}$ to $\mathcal{T}_{n,2}$.

For every type $t \in T$ there is an *extension* $[\![\, t \,]\!]$, which contains all permitted values for a field declared with type t. For $\mathcal{T}_{\text{prim}}$ and $\mathcal{C}_0$ these are inherited from the host language. For composite `umod` types the extensions are defined in Table 2, creating optional types and finite lists, sets, maps and multimaps as carrier types. The `OPT` types lead to relaxed getter and setter methods permitting the `null` value, which is otherwise rejected by throwing an exception. The

[5] Except for `OPT` which is idempotent.

$$\mathsf{disjoint}(\mathcal{C}_0, \mathcal{T}_{\text{prim}}, C, F, Q)$$
$$\texttt{extends} : C \to (C \cup \mathcal{C}_0)$$
$$\mathit{definingClass} : F \to C$$
$$\mathit{fields}(c) = \begin{cases} \{\} & \text{if } c \in \mathcal{C}_0 \\ \mathit{definingClass}^{\sim}(c) \cup \mathit{fields}(\texttt{extends}(c)) & \text{otherwise} \end{cases}$$
$$\mathit{fieldType} : F \to T$$
$$\exists \tau \bullet T = \mathcal{T}_\tau$$
$$\mathcal{T}_0 = \mathcal{T}_{\text{prim}} \cup \mathcal{C}_0 \cup C$$
$$\mathcal{T}_{n+1} ::= \mathcal{T}_n \mid \texttt{OPT}\, \mathcal{T}_n \mid \texttt{SEQ}\, \mathcal{T}_n \mid \texttt{SET}\, \mathcal{T}_n \mid \texttt{MAP}\, \mathcal{T}_n\ \texttt{TO}\ \mathcal{T}_n \mid \texttt{REL}\, \mathcal{T}_n\ \texttt{TO}\ \mathcal{T}_n$$
$$\texttt{.class} : Q \to C$$
$$\mathit{fields}(q) = \mathit{fields}(\texttt{.class}(q))$$
$$\begin{array}{ll}
[\![\, t : \mathcal{T}_{\text{prim}} \,]\!] & = \textit{// inherited from host language.} \\
[\![\, c : \mathcal{C}_0 \,]\!] & = \textit{// imported from libraries.} \\
[\![\, \texttt{OPT}\, t \,]\!] & = [\![\, t \,]\!] \cup \{\texttt{null}\} \\
[\![\, \texttt{SEQ}\, t \,]\!] & = \left([\![\, t \,]\!]\right)^* \\
[\![\, \texttt{SET}\, t \,]\!] & = \mathbb{F}[\![\, t \,]\!] \\
[\![\, \texttt{MAP}\, t_1\ \texttt{TO}\ t_2 \,]\!] & = [\![\, t_1 \,]\!] \twoheadrightarrow\!\!\!\!\!\!\!/\;\; [\![\, t_2 \,]\!] \\
[\![\, \texttt{REL}\, t_1\ \texttt{TO}\ t_2 \,]\!] & = \mathbb{F}\left([\![\, t_1 \,]\!] \times [\![\, t_2 \,]\!]\right) \\
[\![\, c : C \,]\!] & = \{q \in Q \mid \texttt{.class}(q) = c\}
\end{array}$$

Table 2. Model Element Classes, Fields and Types of a `umod` Model

other types are realized by specialized instances of the Java collection framework. The generational way of defining T ensures that every value of a type t is *free of cycles.*[6] It also ensures that all extension sets are disjoint, and we can assume some global "universe" $[\![\, T \,]\!]$. This will be used in few formulas for ease of notation.

Every model is a collection of model elements, i.e. Java objects of model element classes, i.e. values from $[\![\, C \,]\!]$. Each of these is identified by a reference or "pointer value" $q \in Q$, belongs to a certain model class $\texttt{.class}(q)$ and thus has certain fields $\mathit{fields}(q)$. The state of any model is always equivalent to a finite map from all fields of live objects to a permitted value for the respective field type. Field values may directly or indirectly refer to other model elements, and, in contrast to the well-founded collection-based field value world, the resulting graph may contain *arbitrary cycles.*

[6] This is necessary to impose any mathematical semantics on Java collection objects, whose behavior is theoretically undefined, and practically unreliable, in the presence of cyclic containment.

3.2 Rewriting Collections

The recursive calls to `rewrite()`, as explained informally[7] in Section 2.3, realize and define the rewriting process on the level of single elements and their classes.

The declarative approach mandates that this point-wise relation is *automatically* lifted to the rewriting of values $v \in [\![\, t \,]\!]$ of a collection type $t \in \mathcal{T}_{n>0}$. Here a potential clash of paradigms can occur: The programmer must rely on the consistency of the collections created by the rewriter. For this, precise semantics are required, based on the *mathematical* notions of sets, sequences, maps, etc.

The host language Java, just like many others, provides an object-oriented collection framework with interfaces called `Set`, `Map`, etc. But the mathematical metaphor is well-known to be lopsided: Their actual behavior relies implicitly on the immutability of contained elements, and explicitly on the order of interfering container mutations, such as `add`/`remove` or `put`, respectively. Hence the imperative implementation of transformation may be in conflict with the intended mathematics, in particular where control is shared between user-defined and generated code (solid vs. dashed arrows in Fig. 4). It is desirable that the implementation of $[\![\, \texttt{MAP}\,\_ \,]\!]$ should detect and signal error conditions whenever a conflict arises. Unfortunately, the corresponding tests are potentially very expensive, and there is no support from the standard libraries.

The following discussion makes two contributions:

1. Precise semantics for the rewriting of freely compositional collections are constructed by (**p1**) to (**p6**) in Table 3. Their implementation is in most cases straight-forward.
2. Second, for the critical case of rewriting maps inference rules are provided which can help to elide costly tests, and hence speed up a reliable implementation significantly.

In a first step we totally forget the structure of the model and the traversal order: We assume the execution of the whole rewriting process has succeeded and delivers k_v, the user defined point-wise map from model elements to (possibly empty) sequences of model elements as their rewriting results, see Table 3. Additionally, $\kappa = \bigcup_{(q,\overline{q}) \in k_v} \{q\} \times \mathsf{ran}\, \overline{q}$ is a relation which forgets the sequential order of these lists and flattens them into a multi-map. It is important that κ is not necessarily a map, nor total. It will be used in the rewriting of all collections which *directly* contain model elements, except sequences, which use k_v.

The semantics of rewriting collections can now be defined by constructing mappings, i.e. discrete functions $\rho_n : [\![\, \mathcal{T}_n \,]\!] \nrightarrow [\![\, \mathcal{T}_n \,]\!]$, which follow the generational structure of types. These mappings are defined as the family of the smallest relations which satisfy the properties (**p0**) to (**p6**) from Table 3.

The "basic trick" is to encode the collection field values and the rewriting functions themselves *both* as (special cases of) relations. The extension of the sequence type $[\![\, \texttt{SEQ}\, t \,]\!]$ can be encoded as $\mathbb{N} \leftrightarrow [\![\, t \,]\!]$; sets $[\![\, \texttt{SET}\, t \,]\!]$ are encoded by $\{\star\} \leftrightarrow [\![\, t \,]\!]$, and for $[\![\, \texttt{MAP}\, t\, \texttt{TO}\, u \,]\!] \subset [\![\, \texttt{REL}\, t\, \texttt{TO}\, u \,]\!]$ there is a canonical encoding

[7] For a formal definition see the forthcoming technical report.

$$
\begin{aligned}
k_v &: [\![C]\!] \rightarrow [\![C]\!]^* \\
\kappa &: [\![C]\!] \leftrightarrow [\![C]\!] \\
\rho_n &: [\![\mathcal{T}_n]\!] \nrightarrow [\![T_n]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{isMap}(r : A \leftrightarrow B) &\overset{def}{\iff} r^{\sim} ⨾ r \subseteq \mathbf{ID}_B \\
\mathsf{isInj}(r : A \leftrightarrow B) &\overset{def}{\iff} r ⨾ r^{\sim} \subseteq \mathbf{ID}_A \\
\mathsf{isTotal}(r : A \leftrightarrow B, s) &\overset{def}{\iff} s \subseteq \mathsf{dom}\, r
\end{aligned}
$$

$$
\begin{array}{lll}
s \in [\![\mathcal{T}_{\mathrm{prim}}]\!] \cup [\![\mathcal{C}_0]\!] \cup \{\star\} & \Longrightarrow \ (s \mapsto s) \in \rho_0 & (\mathbf{p0}) \\
c \in C \wedge s \in [\![\texttt{SEQ}\ c]\!] & \Longrightarrow \ (s \mapsto \mathsf{flatten}(s ⨾ k_v)) \in \rho_1 & (\mathbf{p1}) \\
t \in \mathcal{T}_n \neq C \wedge s \in [\![\texttt{SEQ}\ t]\!] & \Longrightarrow \ (s \mapsto s ⨾ \rho_n) \in \rho_{n+1} & (\mathbf{p2}) \\
t \in \mathcal{T}_n \wedge s \in [\![\texttt{SET}\ t]\!] & \Longrightarrow \ (s \mapsto s ⨾ (\rho_n \cup \kappa)) \in \rho_{n+1} & (\mathbf{p3}) \\
t_1, t_2 \in \mathcal{T}_n \wedge s \in [\![\texttt{REL}\, t_1\ \texttt{TO}\, t_2]\!] & \Longrightarrow \ (s \mapsto (\rho_n \cup \kappa)^{\sim} ⨾ s ⨾ (\rho_n \cup \kappa)) \in \rho_{n+1} & (\mathbf{p4})
\end{array}
$$

$$
\frac{\begin{array}{c} s \in [\![\texttt{MAP}\, t_1\ \texttt{TO}\, t_2]\!] \quad \wedge \quad t_1 \in \mathcal{T}_n \quad \wedge \quad t_2 \in C \quad \wedge \quad \mathsf{isInj}((\mathsf{dom}\, s) \lhd (\rho_n \cup \kappa)) \\ \wedge \quad \mathsf{isMap}((\mathsf{ran}\, s) \lhd \kappa) \end{array}}{(s \mapsto (\rho_n \cup \kappa)^{\sim} ⨾ s ⨾ \kappa) \in \rho_{n+1}} \qquad (\mathbf{p5})
$$

$$
\frac{s \in [\![\texttt{MAP}\, t_1\ \texttt{TO}\, t_2]\!] \quad \wedge \quad t_1, t_2 \in \mathcal{T}_n \quad \wedge \quad t_2 \notin C \quad \wedge \quad \mathsf{isInj}((\mathsf{dom}\, s) \lhd (\rho_n \cup \kappa))}{(s \mapsto (\rho_n \cup \kappa)^{\sim} ⨾ s ⨾ \rho_n) \in \rho_{n+1}} \qquad (\mathbf{p6})
$$

$$
\frac{\begin{array}{c} S \subset [\![T_n]\!] \quad \wedge \quad D = \bigcup_{r \in S} \mathsf{dom}\, r \quad \wedge \quad R = \bigcup_{r \in S} \mathsf{ran}\, r \\ \wedge \quad \mathsf{isInj}(D \lhd \rho_{n-1}) \quad \wedge \quad \mathsf{isInj}(R \lhd \rho_{n-1}) \quad \wedge \quad \mathsf{isTotal}(\rho_{n-1}, D \cup R) \end{array}}{\mathsf{isInj}(S \lhd \rho_n)} \qquad (\mathbf{pInj})
$$

Table 3. Rewriting Collections

for functions as relations anyhow. This allows the transparent application of the relational composition operator (⨾). This does not only lead to compact formulas, but also induces an intuition which can *easily be explained to programmers* of different skills, e.g. using diagrams.

The properties (**p0**) to (**p4**) suffice for the construction of ρ_n, as long the `MAP` constructor is not involved. These rewriting functions behave nicely: they are total, which means that no typing errors can occur. The cardinality of the user defined rewriting κ (empty, singleton or multiple result) is "automatically absorbed" by the containing collection. This allows to replace one model element by zero or more than one in a `SET` or on both sides of a `REL`.

A special case is the rewriting of $\texttt{SEQ}\, c$ for $c \in C$, i.e. rewriting $\overline{q} \in Q^*$. Here the sequential order of k_v is respected, and the list $k_v(q)$ is inserted "flattened" into the resulting list. This is described by (**p1**). All other sequences, all sets and all relations are rewritten by simply composing their encoding relation with the rewriting relation ρ_n of their elements' type generation, see rules (**p2**) to (**p4**).

Rewriting $\texttt{MAP}\, t_1\ \texttt{TO}\, t_2$ is the real issue: Consider the map of maps $\{\{a \mapsto b\} \mapsto c, \{a \mapsto d\} \mapsto e\}$, where the user's local rules specify d to be rewritten to b. The global rewriting procedure then faces a dilemma, with the following options:

– fail dynamically, e.g. by throwing an exception;
– weaken the type of the result from `MAP` to `REL`, possibly failing later due to violated context conditions;
– silently remove *both* offending pairs;
– silently obey the operational semantics of the underlying object-oriented implementation, thus creating a race condition where *either* of the offending pairs will be overwritten by the other, nondeterministically.

Obviously, the latter two options are unacceptable from a declarative viewpoint, and the possibility of dynamic failure implied by the two former should be contained as much as possible by means of precise diagnostics and static checks.

A given map $m \in [\![\mathcal{T}_{n+1}]\!] = [\![\texttt{MAP}\, t_1\ \texttt{TO}\, t_2]\!]$ is only *guaranteed* to be rewritten to a map m', as opposed to a general relation, if the underlying rewrite relation of all *range* elements is itself a map, and at the same time the rewrite relation of all *domain* elements is *injective*. This is illustrated by the diagram

$$\begin{tikzcd}
{} \arrow[r, "m"] & {} \arrow[d, "(\mathsf{ran}\, m) \lhd (\rho_n \cup \kappa)"] \\
{} \arrow[u, "((\mathsf{dom}\, m) \lhd (\rho_n \cup \kappa))^{\sim}"] \arrow[r, "m'"'] & {}
\end{tikzcd}$$

where the relational converse of the left-hand side must be a map.

The right-hand sides fall in two different cases: $t_2 \in C$ covered by rule (**p5**) and $t_2 \notin C$ by (**p6**). The data construction "under the line" is in both cases simple and basically the same as in (**p4**) for unrestricted relations, but he preconditions above the line are critical. For (**p6**) it is clear by construction that every ρ_n is a map. (For every type only one of (**p0**) to (**p6**) matches, and every rule adds exactly one maplet.) In (**p5**) mostly it must be checked dynamically whether $(\mathsf{ran}\, m) \lhd \kappa$ is a map, because this depends on the outcome of the user's code. But it *may* be known statically: A variant of generated rewriter code which does not offer the callback function `substitute_multi(Object...)` will produce only maps for κ, which are even total.

W.r.t. the left-hand side of the diagram, the check for injectivity (which is the map-ness of the inverse relation) must always be added explicitly, in (**p5**) and in (**p6**). In case of arbitrarily deep nested collection types on the left side of a map, the equality tests involved can be very expensive. Therefore it is necessary to *inherit and infer* the required properties as far as possible.

For injectivity it is clear that the rewriting function of a `SET`, `SEQ`, `REL` and `MAP` type is injective, if the rewriting function(s) of its element type(s) is (/are both) *injective and total.* Only then it can be guaranteed that different values of the collection type will be mapped to different values. This is formalized as (**pInj**).

For the start of the chain, when the element type is from $\mathcal{T}_{\text{prim}}$ or $\mathcal{C}_0$, it is clear that ρ_0 is total and injective, since it is the identity.

For C, i.e. reference values to model elements, map-ness and totality may be known statically, see remark above, or require explicit testing. Injectivity must always be tested explicitly. Again, $\mathtt{SEQ}\, c$ with $c \in C$ is an exception. In this special case totality and injectivity of the element level do *not* carry over to injectivity of the collection level: e.g., $\rho_0 = \{a \mapsto \langle x \rangle, b \mapsto \langle y, z \rangle, c \mapsto \langle x, y \rangle, d \mapsto \langle z \rangle\}$ leads to $\rho_1 \langle a, b \rangle = \langle x, y, z \rangle = \rho_1 \langle c, d \rangle$.

For the higher levels, the rewriting function $\rho_{n>0}$ is always *total* by construction, since (**p0**) to (**p6**) cover all types. The only "holes" come from the additional conditions in (**p5**) and (**p6**), which detect the typing errors when maps would be rewritten to non-maps. We may assume (w.l.o.g.) that in this case the inference process described here has been aborted anyhow. So the totality of the rewriting function ρ may be assumed and thus *injectivity is completely propagated upward* the type generations, as expressed in (**pInj**). In the optimum case it needs to be checked only once, for κ, and for $[\![\, \mathtt{SEQ}\, c \,]\!] \lhd \rho_1$ with $c \in C$.

For given concrete values, if ρ_n of the components is *not* injective, then ρ_{n+1} of the collections, restricted to those currently alive, still can be. Therefore explicit tests can become necessary also on higher levels, when the values appear on the left side of a map construction. These tests can be very expensive.

In the current implementation all these inferences cannot be leveraged, since the employed standard Java collection implementations do the `equals()` test for every "key" inserted into a map anyhow. But if an implementation were chosen with a "trusted back-door", which allowed to manipulate the underlying data structures with fewer tests, then these inference rules (which e.g. prove that there are never "identical" maps of maps of maps on the left side of a map) will become highly relevant.

Only rewriting algorithms (more or less similar to ours) which treat all values as immutable, can give precise semantics to a point-wise modification of a collection, which is referred to by the left side of a map. With these back-doors this would be possible in acceptable execution time.

If $(\mathsf{dom}\, m) \lhd \rho_n$ is *not* injective, then still a map may result when rewriting the map m, namely iff all values which point to the same value by ρ_n, also have the same value by m. (E.g. rewrite map $\{a \mapsto 2, b \mapsto 2\}$ when the user defines $\{a \mapsto a, b \mapsto a\}$.) For us is it not yet clear how these "accidentally correct" results should be handled. This is both an ergonomic and a philosophic design decision. More practical programming experience is required.

4 Conclusion

Rigorous application of mathematical principles to software design is not just an academic habit and end by itself; it can have very practical and profound impact on the reliability of software and the productivity of the development process. However, rigor comes only easy when a "pure" system can be designed from scratch. Programming tools and systems that build on legacy environments can only go so far. A particular danger to rigor lurks in the double standards of object-orientation with respect to models as transcendental mathematical entities and real, mutable data structures. We have illustrated this by a case study on mathematically sound, dynamic rewriting of complex object models. Where resorting to a pure "sandbox" system is not an option, programming discipline must be exercised carefully. This is made feasible by generating as much code as possible automatically. Using inheritance as the canonical object-oriented technique to interface generated and user code is a generally useful metaprogramming strategy that leverages both a great deal of logical independence of "moving parts", and the opportunity to investigate the remaining interferences systematically, and provide a pragmatic and effective combination of static and dynamic safety nets.

4.1 Related Work

Term rewriting has been one of the earliest subjects of basic research and of practical programming. The foundations have been laid in the early 20th century e.g. Church–Rosser theorem, etc. Since then a broad and thorough theory has evolved. With the upcome of compiler construction in the Nineteen-sixties, implementations became soon necessary, and a large *folklore tradition* began, where term rewriting algorithms were mapped in different ways to concrete programming techniques. Our own development is an attempt of systematizing these well-known programming patterns. Nevertheless, term rewriting in the narrow and classical sense does of course not touch the two problems treated in this paper, cycles in the data and freely compositional collection types.

One of the leading implementations and research tools in term rewriting is *Maude*. The foundation paper [**?**] is from 1993 and has hardly lost relevance.

Rho-graphs have been developed for combining pattern matching, as known from graph rewriting, and lambda calculus. Recently the treatment of cycles and optimal sharing has been added [**?**]. We are not aware of any implementation.

A widespread system, well-proven in practice, is Tom [**?**]. It realizes pattern matching and term rewriting by compiling a mixture of a dedicated control language and a high level hosting language, preferably Java. The problems of cycles and the semantics of maps are not addressed.

All these approaches are different from ours, since they are based on "rewrite-centered" languages specially designed for these techniques, and therefore can come to much stronger theoretical results. On the downside the programmer has to learn a further language and to cope with more or less hidden strategies.

Approaches like the visitor combinators from JJTraveler [**?**] are different to ours on the other side of the scale: They are more flexible and ad-hoc configurable since they involve no code generation at all, but only (slightly invasive) interface usage. Rewriting and visiting structures with sharing can be implemented on top, in many cases very elegantly, but is not supported initially. The problem of maps is not addressed.