# On Additivity in Transformation Languages

Soichiro Hidaka, Frédéric Jouault, Massimo Tisi

# On Additivity in Transformation Languages

Sochiro Hidaka
Hosei University
3-7-2 Kajino-cho, Koganei-shi
Tokyo, Japan
Email: hidaka@hosei.ac.jp

Frédéric Jouault
TRAME team, Groupe ESEO
10 Bd Jeanneteau
Angers, France
Email: frederic.jouault@eseo.fr

Massimo Tisi
Inria, IMT Atlantique, LS2N
4 Rue Alfred Kastler
Nantes, France
Email: massimo.tisi@inria.fr

***Abstract*—Some areas in computer science are characterized by a shared base structure for data artifacts (e.g., list, table, tree, graph, model), and dedicated languages for transforming this structure. We observe that in several of these languages it is possible to identify a clear correspondence between some elements in the transformation code and the output they generate. Conversely given an element in an output artifact it is often possible to immediately trace the transformation parts that are responsible for its creation.**

**In this paper we formalize this intuitive concept by defining a property that characterizes several transformation languages in different domains. We name this property *additivity*: for a given fixed input, the addition or removal of program elements results in a corresponding addition or removal of parts of the output. We provide a formal definition for additivity and argue that additivity enhances modularity and incrementality of transformation engineering activities, by enumerating a set of tasks that this property enables or facilitates. Then we describe how it is instantiated in some well-known transformation languages. We expect that the development of new formal results on languages with additivity will benefit from our definitions.**

## I. Introduction

Different areas in computer science study the representation of information by a shared base data structure. Example of well-known structures are trees in program transformation, tables in databases, models in model-driven engineering. Academia and industry have proposed, for each one of such structures, dedicated transformation languages, with the promise of facilitating development and maintenance of transformation code.

Since the main objective of transformation programs is the construction of the target artifact, the relation between program elements and produced target elements has primary importance. In some cases (e.g., in template-based transformation languages [1]) the language fosters an explicit correspondence between the structure of the code and the structure of the resulting target artifact. In other cases the transformation code organization mirrors the structure of the output schema. For instance some rule-based model transformation languages are characterized by target-oriented rule organization [1], with one rule for each target element type and rules nested according to the containment hierarchy in the target artifact (e.g., if the target metamodel has an A element in which B elements can be nested, then the rule for creating As will contain the rule for creating Bs). Finally, rule-based languages like ATL [2] exhibit a fine-grained correspondence between output elements and program expressions: given a property of the target model it is always possible to identify the source code expression (i.e., in an ATL binding) that is responsible for its computation.

While some correspondence between transformation elements and output elements can be observed across several other kinds of transformation languages, the exact meaning and granularity of this correspondence can greatly vary from one language to another. In this paper we provide the formal definition of a cross-language property, which we name *additivity*, that aims to give a formal underpinning to this intuitive correspondence. This forms a base which should simplify the development of some new formal results on languages with additivity. In defining additivity for a language we abstract from the full semantics of its constructs but we focus on the relation between code elements and output elements. The underlying idea is that if a program construct is fully responsible for the creation of an output element, then the program obtained by removing that construct will not produce that element.

In the following we first provide a formal and abstract definition of additivity (Sec. II), and we give an overview of its possible uses in practice (Sec. III). We exemplify additivity on a first model transformation language (Sec. IV). Then we elaborate the details and implications of our definition (Sec. V). We provide a full formal instantiation of these concepts on a second example (Sec. VI). In Sec. VII we briefly discuss examples of additivity in other languages. We conclude the paper by comparing with related work (Sec. VIII) and drawing final conclusions (Sec. IX).

Note that an application of additivity, that we illustrate in the paragraph entitled "Partial Translation" in Sec. III, is described in detail in Hidaka and Tisi [3] (currently under review). In [3] the additivity of a language is just assumed and not thoroughly reasoned about as we do in the present paper. The focus in [3] is on showing how to interoperate unidirectional transformation languages with bidirectional transformation languages in two different technical spaces and to formally show well-behavedness of the integrated system.

## II. Additivity in a Nutshell

Intuitively, we talk of *additivity* when, by "cutting-out" a certain part of a program, we obtain a subprogram that generates a suboutput of the original program. Of course the removed part cannot be arbitrary. In this section we introduce

the basic formal definitions for additivity, and delay to the rest of the paper a more detailed formal treatment.

We limit our discussion to programs that take an input of type $S$ and produce values of type $T$. We call this class of programs *transformations* and denote with $t \in \mathcal{L}$ that a transformation $t$ is written in a language $\mathcal{L}$. Then we denote the *interpretation* of a transformation as $[\![t]\!] : S \to T$, and *application* of the transformation as $[\![t]\!]x \in T$ for the input $x \in S$. For simplicity in this paper we consider only well-formed transformations and disregard the possibility of run-time exceptions (i.e., any execution error) that would make the result undefined.

We need also a formal definition of what suboutput means, but we keep it abstract to allow for wide applicability to different domains: we define the suboutput relation as a *partial order* between data of type $T$, denoted by $\preceq_T$. Depending on the domain, $\preceq_T$ would typically correspond to sublist, subtree, subgraph, etc.

Then we define a *semantic containment* partial order $\preceq_{\mathcal{L}}^{\text{Sem}}$ between transformations, analogous to the concept of query containment in the database community (see, e.g., Abiteboul et al. [4]). Formally, we state that a transformation $t'$ semantically contains a transformation $t$ if $t$ always produces a suboutput of $t'$ when applied to the same input:

$$t \preceq_{\mathcal{L}}^{\text{Sem}} t' \Leftrightarrow \forall x \in S, [\![t]\!]x \preceq_T [\![t']\!]x \qquad \text{(SC)}$$

Note that the study on query containment in the database community shows that it is generally undecidable whether an arbitrary given pair of transformations have this relationship. However in this paper, instead of such arbitrary pairs, we are interested in particular pairs of transformations, that have a clear syntactic relation: one transformation is obtained from the other, by adding or removing a syntactic element (e.g., statement, rule, expression) in a particular position. Formally, we assume to have a purely syntactic partial order between transformation programs that we name *syntactic containment* $\preceq_{\mathcal{L}}^{\text{Syn}}$. This syntactic relationship helps us reason about the semantic containment.

Finally, we can provide the first key definition of *additive containment*. A transformation is additively contained in another, if it is both syntactically and semantically contained in it. Formally:

**Definition 1** (Additive containment)**.** *A transformation $t$ is additively contained $\preceq_{\mathcal{L}}^{\text{Add}}$ in a transformation $t'$ if $t \preceq_{\mathcal{L}}^{\text{Sem}} t'$ and $t \preceq_{\mathcal{L}}^{\text{Syn}} t'$.*

In the rest of the paper we will extensively reason about the position in the program where we manipulate transformations. We will use the standard notion of *context* $C$ that appears in popular texts of programming language semantics like Winskel [5, Chapter 11]. Each context is associated to a corresponding *hole* $\square$ in the program on which the manipulation takes place. $C[e]$ denotes a context with its hole filled with a program fragment $e$. We denote with $C[\ ] : U$ that the context $C$ has type $U$, meaning that only elements of type $U$ can fill the corresponding hole. The position of context holes in the program is inductively defined according to the syntax of the language.

We will be interested only in *additive contexts*, i.e. positions in the transformation code where manipulations produce additively contained programs:

**Property 1** (Additive context)**.** *A context $C$ is additive with respect to program fragments (e.g., expressions) of type $U$ syntactically related by $\preceq_{\mathcal{L}}^{\text{Syn}}$ if for any program fragments $e_1, e_2$ of type $U$, $e_1 \preceq_{\mathcal{L}}^{\text{Syn}} e_2 \Rightarrow C[e_1] \preceq_{\mathcal{L}}^{\text{Sem}} C[e_2]$*[1].

Since we also assume $e_1 \preceq_{\mathcal{L}}^{\text{Syn}} e_2 \Rightarrow C[e_1] \preceq_{\mathcal{L}}^{\text{Syn}} C[e_2]$, we also have, as a corollary, additive containment $C[e_1] \preceq_{\mathcal{L}}^{\text{Add}} C[e_2]$ for such a context.

A context $C[\ ] : U'$ can be additive with respect to $U'$ (i.e., any program fragment that can fill that context hole) or only to a subtype $U$ of $U'$ (i.e., a subset of the possible hole contents). When possible, removing the content of an additive context (i.e., filling it with the minimum element $\mathcal{Z}_U$ of $\preceq_{\mathcal{L}}^{\text{Syn}}$ for that context) will always produce a program that is additively contained in the original one.

Note that most contexts are not additive, since adding syntactic elements to a program has generally complex effects on the output (i.e., not simply additions). Moreover, while additivity is a form of covariance of code and output, some contexts may even exhibit contravariance, i.e., $e_1 \preceq_{\mathcal{L}}^{\text{Syn}} e_2 \Rightarrow C[e_1] \succeq_{\mathcal{L}}^{\text{Sem}} C[e_2]$. We show an example of such contexts in Sec. VI.

## III. Making Use of Additivity

Some model transformation languages already have one or several constructs that generate additive contexts. In the next sections we will show that this is the case for declarative ATL and UnQL [7]. These languages have been designed without relying on the additivity concept, which had not been defined yet. The reason why they nonetheless exhibit some level of additivity seems to be that this concept is linked to other desirable properties such as modularity. This section presents some possible advantages and applications for additivity. Of course, all of these could be (or have already been) achieved with good engineering, and without relying on an explicit definition of additivity. However, having a precise formal framework describing this property should make them easier to achieve, to generalize, and to explain. We expect new applications to be discovered as additivity is further studied, and as new transformation languages are designed to be additive.

**Link to Modularity.** Modularity is a desirable property of transformation languages. It makes it easier to reuse parts of transformations. Modularity is increased when coupling between transformation *modules* (e.g., rules) is decreased. Additivity has a strong requirement on coupling between modules: any module may be removed while keeping the transformation executable. Such a low coupling implies that a given module

[1] It can be considered as a congruence (see, e.g., Pierce [6] for the definition of congruence rules).

(e.g., rule, pattern element, binding) of a transformation can be removed, added, or replaced without requiring changes to the rest of the transformation. Therefore, having additivity implies having a relatively high level of modularity. Modules may still depend on each other implicitly, but may not explicitly reference each other. In rule-based transformation languages, this is related to the notion of *implicit rule call* [8]. Such a level of modularity notably enables mechanisms like transformation superimposition [9, 10].

**Transformation Change Impact Analysis.** Additivity goes beyond low coupling: each module must be linked to a specific part of the target model. Thus, programmers can know what part of a target model are created (or connected) because of the presence of a given module. This makes it easier for them to understand the impact of transformation changes. Ultimately, tools that analyze the impact of transformation changes could rely on additivity. Such tools could create and use fine-grained traces between transformation constructs and target model elements.

**Transformation-level Incrementality.** Once the impact of a change to a transformation can be precisely identified, it becomes possible to act on it. For instance, given a model transformation change, a tool relying on additivity could derive a corresponding target model change. Leveraging the notation in Sec. II, let $t \in \mathcal{L}$ be a transformation, $x \in S$ a source model, and $y \in T$ the corresponding target model. Instead of denoting transformation interpretation with $[\![t]\!]$, let us further consider an explicit interpretation function $f : \mathcal{L} \to S \to T$ that takes a transformation, and a source model as input, and returns a target model as output. We can write: $y = ftx$ to denote the application of $f$ to both the transformation, and the source model. Typically, incremental transformation engines react to source model changes by performing corresponding target model changes. With explicit interpretation function $f$ the transformation becomes just another input on which incrementality can be performed.

**Partial Translation.** One of our main applications of additivity is partial translation of ATL to UnQL, to achieve a partial bidirectionalization of ATL [3]. ATL's additive contexts are leveraged to remove corresponding rules, output pattern elements and bindings that include language constructs for which no rule for compilation to UnQL is available. Additivity ensures that the rest of the ATL transformation produces submodels of the output of the original ATL transformation, and so does the UnQL transformation that is translated from the projected ATL transformation, so that the updates to that target submodel are reflected to the source model through backward UnQL transformation. Partiality allows users to make use of the unidirectional language with full expressive power in the forward direction of the transformation, while updates on part of the target model are automatically backpropagable.

**Language Alignment.** In Sec. V we will describe the lattice induced by additive containment on a transformation language. Given two transformation languages with appropriate additivity contexts, we can design a translator between them as a homomorphism between their additive containment lattices. This has the benefit of enhancing the *modularity of the translator*. E.g., the translator can be developed incrementally, by concentrating at each step on translating different additivity modules.

In the previous example of ATL-to-UnQL translation, although additivity of UnQL is not strictly required for this bidirectionalization, we can build a translator that actually preserves the partial order of the two languages. This means that we can further project the ATL transformation to reduce the part of the target model for which the updates are subject to backward propagation. This may open the opportunity for fine-grained write access control of target models.

**Divide and Conquer.** The level of modularity achievable with language constructs exhibiting additivity should also be useful in any *divide and conquer* execution strategy. For instance, parallelization or distribution of model transformation could leverage the fact that additivity links specific parts of a transformation to specific parts of a target model. In a map-reduce context, this would correspond to the first step: map. Additivity does not explicitly help with the reduce phase, but this should be achievable with appropriate trace information. Additionally, such a divide and conquer strategy could possibly be applied to transformation verification.

## IV. Example 1: ATL

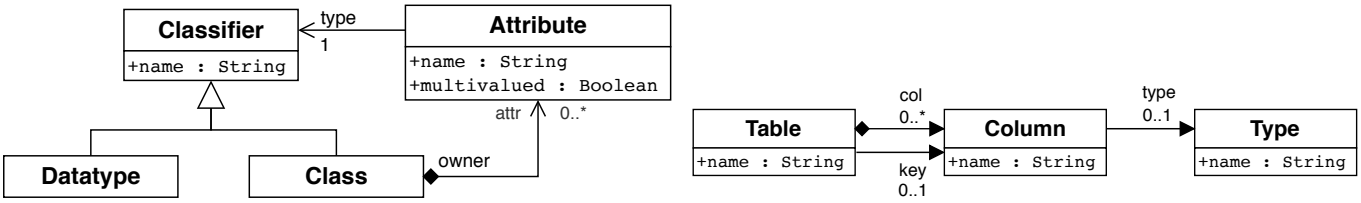
In this section we show how the concept of additivity can be instantiated on rule-based transformation languages, by applying the previous definitions to the ATL language. Language constructs in which additivity is not instantiated are also exemplified.

We refer to a simple example inspired from the Class2Relational case study [11][2], the de-facto standard example for model transformations, but simplified and adapted for illustrative purpose[3]. The transformation expresses the relation between the UML classes of the application's data model and corresponding relational tables. Figures 1a and 1b show the source and target metamodels of the transformation. The ATL transformation in Listing 1 transforms the Class model from Fig. 2a into the Relational model from Fig. 2b.

```
1 module Class2Relational;
2 create OUT : Relational from IN : ClassDiagram;
3
4 uses helperLib;
5
6  rule DataType2Type {
7   from d:ClassDiagram!DataType
8     (ClassDiagram!Class.allInstances()
9        ->select(c | c.name = d.name)->isEmpty())
10  to t:Relational!Type
11    (name<-d.name)    }
12
13 rule Class2Table {
14   from c:ClassDiagram!Class
15   to t:Relational!Table
16     (name<-c.name, col<-c.attr,
17        key<-c.attr->select(a | a.name.endsWith('Id'))   )}
18
19 rule SingleValuedAttribute2Column {
```

[2]http://web.archive.org/web/20100824053242/http://sosym.dcs.kcl.ac.uk/events/mtip05/

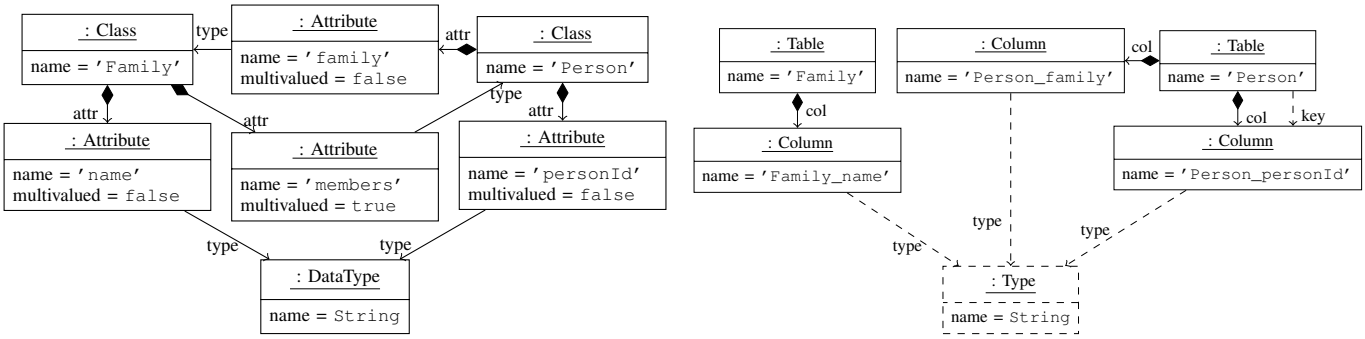
[3]Notably this version of the transformation does not support multivalued attributes.




(a) Class diagram metamodel

(b) Relational metamodel

Fig. 1: Metamodels of the case study




(a) Sample ClassDiagram model

(b) Sample Relational model

Fig. 2: Sample models (corresponding elements have the same position)

```
20 from a:ClassDiagram!Attribute (not a.multivalued)
21 to c:Relational!Column
22    (name<-a.owner.name+'_'+a.name,
23     owner<-a.owner,
24       type<-if a.type.oclIsTypeOf(ClassDiagram!DataType)
25         then a.type else thisModule.stringType endif   ) }
```

Listing 1: Class2Relational in ATL

An ATL transformation (called a *module*) consists of a set of rules that describe how elements of the target model are generated when patterns in the source model are matched. A flavor of the Object Constraint Language OCL [12] is used as expression language in the transformation rules. Rules are composed of an *input pattern* and an *output pattern*. Input patterns are associated with OCL *guards* that impose matching conditions on the input elements. Output patterns are associated with *bindings*: OCL expressions that define how to initialize properties of the elements created by the matched rule.

E.g., the rule *SingleValuedAttribute2Column* (lines 19–25) selects input model elements of type Attribute (line 20) and transforms them into output elements of type Column (line 21). A guard (line 20) imposes to match only attributes that are not multivalued (multivalued attributes would have a more complex representation in the relational schema). A first binding (line 22) computes the column name as the concatenation of class and attribute name; a second one (line 23) imposes that the owner of the Column corresponds to the owner of the Attribute; a third one (lines 24–25) states that if the attribute has a primitive type (*DataType*) then the type of the column should correspond to the type of the attribute (String otherwise). Note that the *stringType* helper is not defined here, but assumed to be imported (line 4) from a library of helpers.

Rule *Class2Table* (lines 13–17) creates a table for each class, initialises the list of columns with the respectively transformed attributes (line 16), and selects as table key the attribute whose name ends by the string *Id*, if any (line 17). Rule *DataType2Type* (lines 6–11) copies into relational types the class diagram types, but only if no class exists with the same name of the type, to avoid conflicts.

Note that ATL rules implicitly interact for generating cross-references among target model elements. E.g., at line 16 the binding `col<-c.attr` states that the columns (`col`) of the table to generate correspond to the attributes (`attr`) of the matched class (`c`). In practice, it is the result of the transformation of these attributes, by any rule (*SingleValuedAttribute2Column* here), which will be added as a column of the table.

Because of space constraints, in this article we discuss only the main features of the ATL language, exemplified in Listing 1. For an illustration of the complete language we refer the reader to [2].

**Additivity in ATL.** In ATL there is a clear correspondence between language elements and the part of the output they are responsible for. For instance the binding in line 17 is responsible for the creation of the *key* link in Fig. 2b: if we remove the binding and execute the resulting subtransformation, we obtain the same model in Fig. 2b, without the *key* link. This reasoning applies to all bindings, but also to elements in output patterns or to full rules. For instance the DataType2Type rule

(lines 6-11) is fully responsible for the generation of the Type element in Fig. 2b.

We formalize this concept by instantiating the definitions from Sec. II. Executing the transformation in Listing 1 without the grey part ($Class2Relational' \preceq_{\mathcal{L}}^{Syn} Class2Relational$) generates the submodel in Fig. 2b without the dotted part ($sampleRelational' \preceq_T sampleRelational$). For every input class diagram, $Class2Relational'$ generates a submodel of the $Class2Relational$ output, without *key* references and *Type* elements. Thus, $Class2Relational' \preceq_{\mathcal{L}}^{Sem} Class2Relational$ and, according to Def. 1:

$Class2Relational' \preceq_{\mathcal{L}}^{Add} Class2Relational$

We obtained an additive containment because all the syntactic elements we removed from $Class2Relational$ were contained in additive contexts.

In general, in ATL we identify three additive contexts:

- a module constitutes an additive context for rules,
- a rule is an additive context for output pattern elements,
- an output pattern element is an additive context for bindings.

Prop. 1 can be easily verified for each one of these three contexts. For instance a rule is an additive context for output pattern elements because if we replace its output pattern $e_2$ with a subpattern $e_1$ (i.e., $e_1 \preceq_{\mathcal{L}}^{Syn} e_2$), we obtain a transformation $C[e_1]$ that produces the same output of the original one $C[e_2]$, without the omitted output elements (i.e., $C[e_1] \preceq_{\mathcal{L}}^{\text{Sem}} C[e_2]$).

We inductively define the position of additive contexts in the ATL syntax by representing them as squares in the following rules:

$$\frac{}{\textbf{module}\ id;\ \textbf{create}\ id{:}id\ \textbf{from}\ id{:}id;\Box{:}RuleContext}$$

$$\frac{C{:}RuleContext}{C[rule\ \Box]{:}RuleContext}$$

$$\frac{C{:}RuleContext}{C[\textbf{rule}\ id\{\ \textbf{from}\ id{:}id!id(expression)\ \textbf{to}\ \Box\}]{:}OutPatternElementContext}$$

$$\frac{C{:}OutPatternElementContext}{C[outPatternElement,\Box]{:}OutPatternElementContext}$$

$$\frac{C{:}OutPatternElementContext}{C[id{:}id!id(\Box)]{:}BindingContext} \qquad \frac{C{:}BindingContext}{C[binding,\Box]{:}BindingContext}$$

The first rule states that there is always a top-level additive context of type $RuleContext$. The second states that in an additive context of type $RuleContext$ it is possible to add a rule and maintain an additive context of type $RuleContext$. The following four rules similarly define contexts of type $OutPatternElementContext$ and $BindingContext$.

Note that guards are not generally included in an additive context, e.g., removing the guard from a rule does not reduce the transformation output. Of course an additive context (e.g. for rules) may normally include non-additive contexts (e.g. for guards). However, there is no harm in the existence of such non-additive contexts, since they are kept unchanged when the surrounding additive contexts are manipulated. Moreover a deeper analysis of the OCL functional language may reveal other additive contexts within the guard expression (e.g. parts of the guard that can be removed guaranteeing the generation of a submodel[4]). While we limit the discussion of this section to the transformational part of the ATL language, we will present a full example of additivity in functional languages in Sec. VI.

Moreover, ATL includes a type of rules that are activated only if explicitly called (i.e. lazy rules). Lazy rules can not be simply removed from the transformation, since this would leave an incorrect program with dangling rule invocations. However lazy rules are additive as they can be replaced with empty rules (the minimum element $\mathcal{Z}_U$ of $\preceq_{\mathcal{L}}^{\text{Syn}}$ for their context), and the resulting transformation will be semantically contained in the original one. Finally, additivity requires that the imperative part of ATL is not used, but this does not reduce the expressive power of ATL because its declarative part is already computationally universal [13].

## V. Additivity and Compositionality

In this section we highlight mathematical structures in the semantics of transformation languages, related to additivity. Our main purpose is giving some insight into the relation between additivity and compositionality in the semantics. Moreover, detecting such structures in the transformation language semantics may practically help in the task of identifying additivity in existing languages.

In particular: 1) we discuss the special case of additivity at the top level of the transformation and its relation to transformation composition, 2) we describe the structure of additive contexts with some guidelines for their identification, 3) we introduce the notion of lattice of transformations as the global picture of the complete set of additively contained transformations.

*Additivity and Transformation Composition:* Our underlying general intuition behind additivity is that the interpretation of a composite transformation can be divided into those of component transformations followed by the combination of their results, i.e.,

$$\forall x \in S, [\![t_1 \oplus t_2]\!]x = [\![t_1]\!]x \otimes [\![t_2]\!]x \qquad \text{(DC)}$$

or analog, where $\oplus$ is the union-like language construct that combines two transformations, while $\otimes$ is an operator of the target data model to combine the results. Then, if $v_1 \preceq_T v_1 \otimes v_2$ for any values $v_1, v_2$ of type $T$, as we can expect in many data types like sets with set union, we can conclude that $t_1 \preceq_{\mathcal{L}}^{\text{Sem}} (t_1 \oplus t_2)$. Even without this rather strict "divide and conquer" property, if the "conquer" phase still "adds" to the results of individual transformations rather than having possibilities to "remove", then we still have $[\![t_1]\!]x \preceq_T [\![t_1 \oplus t_2]\!]x$ for all $x \in S$, to conclude $t_1 \preceq_{\mathcal{L}}^{\text{Sem}} (t_1 \oplus t_2)$, even if we allow the "conquer" phase to access beyond the individual results $[\![t_1]\!]x$ and $[\![t_2]\!]x$.

If a type $T$ constitutes a monoid $(M, \otimes)$, then, according to the order between the elements of the monoid, i.e., $v \preceq v'' \Leftrightarrow \exists \hat{v}.v \otimes \hat{v} = v''$ (see, for example Gondran and Minoux

[4]E.g., every $e_j$ in $e_1 \vee e_2 \vee \ldots \vee e_n$ in a guard is in an additive context, since removing any $e_j$ increases the chance of evaluating the guarded expression.

[14]), we indeed have $\mathcal{Z}_\otimes \preceq_T v \preceq_T v \otimes v'$ (and $v' \preceq_T v \otimes v'$ for commutative $\otimes$) for any two elements $v, v' \in M$, where $\mathcal{Z}_\otimes$ is the identity element ($\mathcal{Z}_\otimes \otimes v = v \otimes \mathcal{Z}_\otimes = v$ for all $v \in M$) of the monoid. Therefore, $[\![\mathcal{Z}_\oplus]\!]x \preceq_T [\![t_1]\!]x \preceq_T [\![t_1]\!]x \otimes [\![t_2]\!]x = [\![t_1 \oplus t_2]\!]x$, where $[\![\mathcal{Z}_\oplus]\!]x = \mathcal{Z}_\otimes$, for all $x \in S$, which means $\mathcal{Z}_\oplus \preceq^{\mathrm{Sem}}_{\mathcal{L}} t_1 \preceq^{\mathrm{Sem}}_{\mathcal{L}} t_1 \oplus t_2$, for any component transformations $t_1, t_2$ of type $T$, where $\mathcal{Z}_\oplus$ is the syntactic representation of $\mathcal{Z}_\otimes$. Since $t_1$, $t_2$ and $\oplus$ are also syntactic entities, we have $\mathcal{Z}_\oplus \preceq^{\mathrm{Add}}_{\mathcal{L}} t_1 \preceq^{\mathrm{Add}}_{\mathcal{L}} t_1 \oplus t_2$. Therefore, we can replace any composite transformation $t_1 \oplus t_2$ of monoid type by its component transformation $t_1$ (and $t_2$ if $\otimes$ is commutative) to produce suboutput, and further replace it by the unit (empty) transformation to produce empty output. In this way, property (DC) with monoid structure derives the additive containment relation $\preceq^{\mathrm{Add}}_{\mathcal{L}}$.

*Order-Preserving Contexts:* We have conducted this reasoning on the components at the top level, i.e., $\oplus$ is at the top level of the transformation. If compositionality of the semantics allows such reasoning on deeper subexpressions, i.e., within contexts, we can obtain $\preceq^{\mathrm{Add}}_{\mathcal{L}}$ in a bottom-up fashion. To do this, interpretation of subexpressions in the context should be possible, and the context should preserve the partial order, as in the following property.

**Property 2** (Order preserving context)**.** *A context $C$* preserves partial order $\preceq^{\mathrm{Sem}}_{\mathcal{L}}$ *if, for every program element (e.g., subexpresssion) $e_1, e_2$ of type $U$ in the context, there is an interpretation $[\![\_]\!]_U$ of such elements, and $[\![e_1]\!]_U \preceq_U [\![e_2]\!]_U$ (i.e., $e_1 \preceq^{\mathrm{Sem}}_{\mathcal{L}} e_2$) implies $C[e_1] \preceq^{\mathrm{Sem}}_{\mathcal{L}} C[e_2]$.*

When the interpretation of a language is defined in terms of interpretations of the subexpressions, like in property (DC) but in a deeper context, we can identify order preserving contexts to induce the additive containment from that in such subexpressions. It is not so uncommon to find such compositional semantics (e.g., UnCAL has such compositionality as we see in Sec. VI). Then this compositionality can be combined with monoid structures to build the relation $\preceq^{\mathrm{Add}}_{\mathcal{L}}$ from those in the subexpressions, since, as a corollary, for a partial order preserving context $C$ of type $U$ which constitutes monoid $(M, \oplus)$, we have $C[\mathcal{Z}_\oplus] \preceq^{\mathrm{Add}}_{\mathcal{L}} C[e] \preceq^{\mathrm{Add}}_{\mathcal{L}} C[e \oplus e']$ for any expression $e, e'$ of type $U$.

The remaining task is then to identify such contexts. Suppose a transformation language $\mathcal{L}$ is compositional in the sense that a transformation can be composed of two consecutive transformations $t_1$ and $t_2$ where the output of $t_1$ is the input of $t_2$, i.e., $\forall x \in S, [\![t_2 \circ t_1]\!]x = [\![t_2]\!]([\![t_1]\!]x)$, with $[\![t_2]\!] : U \to T$ and $[\![t_1]\!] : S \to U$ for some type $U$. Then, to make $t_2 \circ t_1$ additive at the context $t_2 \circ \Box$, it is sufficient to let the context satisfy Prop. 2. This is equivalent to making $t_2$ monotonically increasing, i.e., $\forall y_1, y_2 \in U, y_1 \preceq_U y_2 \Rightarrow [\![t_2]\!]y_1 \preceq_T [\![t_2]\!]y_2$. We reason about such property for UnCAL in Sec. VI.

*Lattice of Transformations:* Since by (SC) the interpretation of the transformation language preserves partial orders, we have a homomorphism from transformations to outputs. Further, we consider the lattice of transformations and the lattice of outputs, where edges in the former lattice represent $\preceq^{\mathrm{Add}}_{\mathcal{L}}$ and in the latter $\preceq_T$. Then, due to additive contexts, we have a homomorphism that preserves partial orders from the lattice of transformations to the lattice of outputs, where edges in the former lattice also represent syntactic operations on constructs in the additive contexts. In the former lattice, we fix a given running (correct) transformation to form the complete (convex) lattice for it with the given transformation at the top, and reason about removing part of the transformation to produce a suboutput.

Figure 3 shows the lattice derived from an ATL transformation consisting of two rules, each of which has one input pattern element and an output pattern element with two bindings (the headers of the transformations are omitted for simplicity). The top of the lattice is the complete transformation. Each edge represents partial order $\preceq^{\mathrm{Add}}_{\mathcal{L}}$ between transformations by adding/removing a rule or a binding. The bottom of the lattice represents (syntactically) the smallest transformation with no rule. It is an empty transformation that produces the empty target model for any input model because of the absence of any rule. The transformation with bold letters and surrounded by square is an example of subtransformation, where the bindings $b_{11}$ of rule $r_1$ and all the bindings in rule $r_2$ are removed. Such a subtransformation always forms a convex sublattice represented by the transformations with bold letters in the figure. Transformations surrounded by a square are upper bounds of this sublattice, with the aforementioned subtransformation being the least upper bound.

## VI. Example 2: UnQL

GRoundTram is an integrated framework for developing well-behaved bidirectional graph transformations expressed in the language UnQL [7]. Figure 4 shows the syntax of UnQL. It has templates ($T$) at the top level to produce rooted graphs with labeled ($L$) branches ($\{L : T, \ldots, L : T\}$), union of templates ($T \cup T$), graph variable reference ($\$g$), conditionals, SQL-like select (**select**) queries with bindings ($B$), and structural recursion (**sfun**). Bindings include graph pattern matching ($Gp$ **in** $\$g$) and Boolean conditions ($BC$). Conditions include the usual logical connectives, label equality and emptiness check $\mathbf{isEmpty}(T)$ which is false if there is at least one edge reachable from the root of the graph generated by $T$. Label expressions include label constants and label variables ($\$l$) bound by the label patterns in the graph patterns and the argument of structural recursion. Graph patterns bind the entire subgraph $\$g$ or labels on the branches and following subgraphs $\{Lp : Gp, \ldots, Lp : Gp\}$. Label patterns may include regular expressions of the labels ($Rp$) along the path traversing multiple edges.

Listing 2 shows the UnQL transformation that corresponds to the ATL transformation in Listing 1 except the part with gray background. It transforms the graph encoding of the Class model in Fig. 2a to the graph encoding of the Relational model in 2b, without the dotted part.

The transformation consists of structural recursion functions (**sfun**s). In the listing, the last definition (function

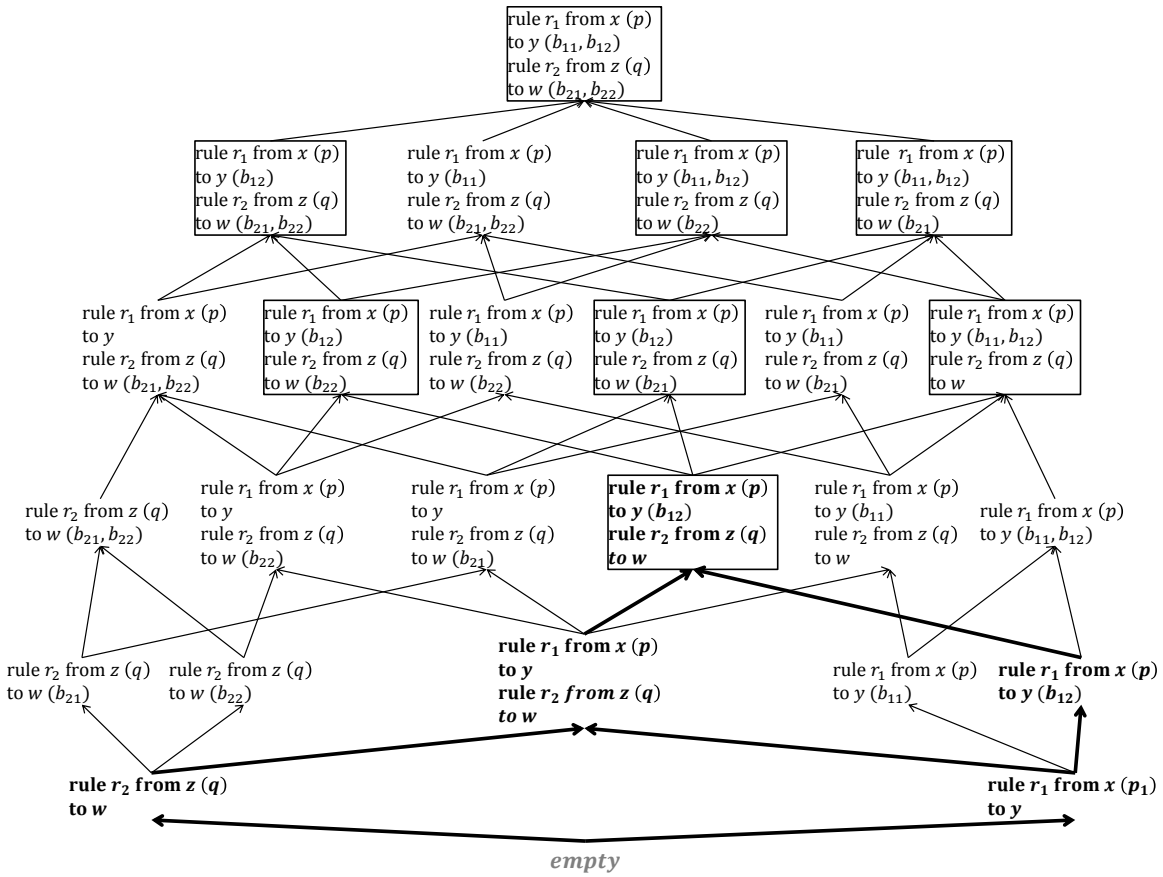



Fig. 3: An example of lattice for $\preceq_{\mathcal{L}}^{\mathrm{Add}}$ in ATL (for space reasons we just consider additivity in RuleContext and BindingContext)

$$
\begin{array}{lllll}
\text{(template)} & T & ::= & \{L:T,\ldots,L:T\} \mid T \cup T \mid \mathit{fname}(\$g) \mid \$g \mid \mathbf{if}\ BC\ \mathbf{then}\ T\ \mathbf{else}\ T \\
 & & \mid & \mathbf{select}\ T\ \mathbf{where}\ B,\ldots,B \\
 & & \mid & \mathbf{letrec}\ \mathbf{sfun}\ \mathit{fname}(\{L:\$g\})=T \mid \ldots \mid \mathit{fname}(\{L:\$g\})=T \\
 & & & \quad \mathbf{and}\ \mathbf{sfun}\ \mathit{fname}(\{L:\$g\})=T \mid \ldots \mid \mathit{fname}(\{L:\$g\})=T\ \mathbf{in}\ \mathit{fname}(T) \\
\text{(binding)} & B & ::= & Gp\ \mathbf{in}\ T \mid BC \\
\text{(condition)} & BC & ::= & \mathbf{not}\ BC \mid BC\ \mathbf{and}\ BC \mid BC\ \mathbf{or}\ BC \mid L=L \mid \mathbf{isEmpty}(T) \\
\text{(label\&its pattern)} & L & ::= & \$l \mid \mathtt{a} \qquad Lp ::= \$l \mid Rp \\
\text{(graph pattern)} & Gp & ::= & \$g \mid \{Lp:Gp,\ldots,Lp:Gp\} \\
\text{(regular path pat.)} & Rp & ::= & \mathtt{a} \mid \_ \mid Rp.Rp \mid (Rp|Rp) \mid Rp? \mid Rp* \mid Rp+
\end{array}
$$

Fig. 4: Syntax of UnQL

*Class2Relational* at lines 8–21) is the main function that corresponds to the ATL module *Class2Relational*. Its clauses generate elements according to different ATL rules — *Class2Table* (lines 17–19), while explanation for the clause for *SingleValuedAttribute2Column* is omitted here. References between target elements are represented by definition and call of sibling mutually-recursive functions: *col_Class2Table* called at line 19 for `col` in the rule *Class2Table* (lines 2–4). The SQL-like **select** expression (line 18) performs shallow copy of primitive data type values by capturing the subgraph under label `name` and creating the new edge with the same label and the same subgraph.

Graphs in UnQL are essentially trees with references (finite representation of infinite regular trees obtained by unfolding

$$
\begin{array}{lll}
e & ::= & \{\} \mid \{l:e\} \mid e \cup e \mid \&x := e \mid \&y \mid () \\
 & \mid & e \oplus e \mid e\ @\ e \mid \mathbf{cycle}(e) \qquad \{\text{ constructor }\} \\
 & \mid & \$g \qquad \{\text{ graph variable }\} \\
 & \mid & \mathbf{if}\ b\ \mathbf{then}\ e\ \mathbf{else}\ e \qquad \{\text{ conditional }\} \\
 & \mid & \mathbf{rec}(\lambda(\$l,\$g).e)(e) \qquad \{\text{ structural recursion application }\} \\
l & ::= & a \mid \$l \qquad \{\text{ label } (a \in \mathit{Label}) \text{ and label variable }\}
\end{array}
$$

Fig. 5: Syntax of UnCAL

cycles), and UnQL queries and constructs tree structures. So intuitively, more template means bigger outputs. Since UnQL is translated into its core language UnCAL, identification of additive contexts requires reasoning about those of UnCAL, so we do this in the following.

```
select
 letrec sfun col_Class2Table({attr:$g})
              = {col:Class2Relational($g)}
          | col_Class2Table({$l:$g})   = {}
    and sfun owner_SingleAttribute2Column({owner:$g})
              = {owner:Class2Relational($g)}
          | owner_SingleAttribute2Column({$l:$g}) = {}
    and sfun (* main function *)
       Class2Relational({Attribute:$g}) =
    if isEmpty(select {dummy:{}}
                 where {multivalued.Boolean:$g'} in $g,
                       {true:$d} in $g') then
    {Column:((select {name:{String:{$on ^ $n}}}
              where {owner._name.String:{$on:$d}} in $g,
                    {name.String:{$n:$d}} in $g)
             ∪ owner_SingleAttribute2Column($g))} else {}
     | Class2Relational({Class:$g}) =
      {Table:((select {name:$g} where {name:$g} in $g)
               ∪ col_Class2Table($g))}
     | Class2Relational({$l:$g}) = {}
in Class2Relational($db)
```

Listing 2: Class2Relational(Class2Table part) transformation in UnQL

*Additivity of UnCAL:* UnCAL is functional and compositional, having compositional semantics with direct interpretation of subexpressions. Therefore, as we discussed in Sec. V, we reason about Prop. 2 to identify additive contexts in UnCAL, through its *monotonically increasing* fragment.

We first introduce the syntax (Fig. 5) and its semantics. In the following, $DB_Y^X$ denotes graphs with the set $X$ and $Y$ of input and output markers, respectively, where input markers label roots and $X$ is omitted if it is a singleton of unique default $\&$ while output markers label nodes that are to be connected with other identically labeled roots, and $Y$ is omitted when empty.

There are nine constructors, three of which are nullary: $()$ constructs a graph without any nodes or edges ($[\![()]\!] \in DB^{\emptyset}$), $\{\}$ constructs a node with default input marker ($\&$) and no edges ($[\![\{\}]\!] \in DB$). $\&y$ is similar to $\{\}$ with additional output marker $\&y$ associated with the node ($[\![\&y]\!] \in DB_{\{\&y\}}$). The edge constructor $\{\_ : \_\}$ takes a label $l$ and a graph $g \in DB_Y$, constructs a new root with the default input marker with an edge labeled $l$ from the new root to the root of $g$; thus $\{l : g\} \in DB_Y$. The union $g_1 \cup g_2$ of graphs $g_1, g_2 \in DB_Y^X$ with the identical set of input markers $X = \{\&x_1, \ldots, \&x_m\}$, constructs $m$ new input nodes for each $\&x_i \in X$, where each node has two $\varepsilon$-edges to the root of $g_1$ labeled $\&x_i$ and the root of $g_2$ labeled $\&x_i$. Here, $\varepsilon$-edges are similar to $\varepsilon$-transitions in automata and used to connect nodes during the graph construction. Clearly, $g_1 \cup g_2 \in DB_Y^X$. The input node renaming operator $:=$ takes a marker $\&x$ and a graph $g \in DB_Y$ and relabels the root of $g$, i.e., $(\&x := g) \in DB_Y^{\{\&x\}}$. The disjoint union $g_1 \oplus g_2$ of two graphs $g_1 \in DB_{X'}^X$ and $g_2 \in DB_{X'}^Y$ with $X \cap Y = \emptyset$, the resultant graph inherits all the markers, edges and nodes from the operands, thus $g_1 \oplus g_2 \in DB_{X'}^{X \cup Y}$. The remaining two constructors connect output and input nodes with matching markers by $\varepsilon$-edges. $g_1 \mathbin{@} g_2$ appends $g_1 \in DB_{X' \cup Z}^X$ and $g_2 \in DB_Y^{X' \cup Z'}$ by connecting the output and input nodes with the matching subset of markers $X'$, and discards the rest of the markers, thus $g_1 \mathbin{@} g_2 \in DB_Y^X$. The cycle construction $\mathbf{cycle}(g)$ for $g \in DB_{X \cup Y}^X$ with $X \cap Y = \emptyset$ connects output and input nodes of $g$ with matching markers $X$, and constructs copies of input nodes of $g$, each connected with the original input node. The output markers in $Y$ are left as is. The semantics of conditionals is standard, with the syntax and semantics of $b$ identical to $BC$ of UnQL. Label and graph variables are introduced by the structural recursion operator $\mathbf{rec}$. For example, the following replaces every label `a` by `d` and contracts edges labeled `c`.

$$\begin{aligned}\mathbf{rec}(\lambda(\$l, \$g).\ &\mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{d} : \&\} \\ &\mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ \{\varepsilon : \&\} \\ &\mathbf{else}\qquad\qquad\ \ \{\$l : \&\})(\$db)\end{aligned}$$

We call the first and second operand of $\mathbf{rec}$ *body* and *argument*, respectively. We use $\$db$ as a special global variable to represent the input of the graph transformation.

Now we introduce a type system (Fig. 6) to statically reason about monotonicity, based on which we build additive contexts. Our reasoning about monotonicity with respect to graph is a generalization of Buneman et al. [7] in the sense that we also consider (1) decreasing fragment, (2) boolean expressions, and (3) a separate type system. The operator $\phi$ is defined by the table in the figure, where $+$, $-$, $0$ respectively means monotonically increasing, decreasing and no change with respect to the input. $?$ means undefined and unused in our identification of additive context. We extend the notion of monotonicity to the boolean expressions by setting $\mathsf{F} \prec \mathsf{T}$.

The environment $\Gamma$ stores the monotonicity of graph variables (M-VREF) and initialized with $\{\$db \mapsto 0\}$ at the top level, since we fix the graph while manipulating transformations. The monotonicities of the three nullary constructors are 0 (M-EMP,M-OMRK,M-GEMP) since they are independent from the input. The three binary constructors behave identically (M-UNI,M-DUNI,M-APND): monotonicities are preserved when those of the operands agree, or either of the two is 0; unknown ($?$) if disagree. Structural recursion $\mathbf{rec}$ as a binary expression behaves similarly (M-REC). Note that the graph variable $\$g$ inherits the monotonicity of the argument. For the unary operators, $\mathbf{cycle}$ and $:=$ preserve (M-CYC,M-IMRK) and $\mathbf{isEmpty}$ negates (M-ISEMP) monotonicities. For the conditionals (M-IF), monotonicity of the $\mathbf{then}$ branch is preserved if the condition has the same monotonicity and the $\mathbf{else}$ branch creates the empty graph.

Given these rules, order preserving (OP for short) contexts are provided in Fig. 7 where $C : DB_Y^X$ denotes that $C$ is OP with type $DB_Y^X$. OP contexts are systematically derived from (M-*) rules of expressions having subexpressions as follows. If the monotonicities of the subexpressions agree across the whole expression, then OP context of the subexpression is generated from the OP context of the expression. For multiple subexpressions, since the number of context hole is one, the monotonicities of others are 0. For example, (C-UNIR) is created by (M-UNI) with $m_1 = 0$. So we only show several of

$$\frac{}{\{\}:0}(\text{M-EMP}) \quad \frac{\Gamma \vdash e : m}{\Gamma \vdash \{l:e\} : m}(\text{M-EDG}) \quad \frac{\Gamma \vdash e_1 : m_1 \quad \Gamma \vdash e_2 : m_2}{\Gamma \vdash e_1 \cup e_2 : \phi(m_1, m_2)}(\text{M-UNI}) \quad \frac{\Gamma \vdash e : m}{\Gamma \vdash \&x := e : m}(\text{M-IMRK})$$

$$\frac{}{\&y : 0}(\text{M-OMRK}) \quad \frac{}{() : 0}(\text{M-GEMP}) \quad \frac{\Gamma \vdash e_1 : m_1 \quad \Gamma \vdash e_2 : m_2}{\Gamma \vdash e_1 \oplus e_2 : \phi(m_1, m_2)}(\text{M-DUNI}) \quad \frac{\Gamma \vdash e_1 : m_1 \quad \Gamma \vdash e_2 : m_2}{\Gamma \vdash e_1 \mathbin{@} e_2 : \phi(m_1, m_2)}(\text{M-APND})$$

| $\phi$ | $-$ | $0$ | $+$ |
|---|---|---|---|
| $-$ | $-$ | $-$ | $?$ |
| $0$ | $-$ | $0$ | $+$ |
| $+$ | $?$ | $+$ | $+$ |

$$\frac{\Gamma \vdash e : m}{\Gamma \vdash \mathbf{cycle}(e) : m}(\text{M-CYC}) \quad \frac{\Gamma \vdash e : m}{\Gamma \vdash \mathbf{isEmpty}(e) : -m}(\text{M-ISEMP}) \quad \frac{\Gamma \vdash e : m \quad \Gamma \vdash e_\mathsf{T} : m}{\Gamma \vdash \mathbf{if}\, e\, \mathbf{then}\, e_\mathsf{T}\, \mathbf{else}\, \{\} : m}(\text{M-IF})$$

$$\frac{\Gamma \vdash e_2 : m_2 \quad \Gamma \cup \{\$g \mapsto m_2\} \vdash e_1 : m_1}{\Gamma \vdash \mathbf{rec}(\lambda(\$l, \$g).e_1)(e_2) : \phi(m_2, m_1)}(\text{M-REC}) \quad \frac{}{\Gamma \vdash \$g : \Gamma(\$g)}(\text{M-VREF})$$

Fig. 6: Type system for the monotonicity of UnCAL

$$\frac{\Gamma \vdash C : DB_Y^X}{\Gamma \vdash C[e \cup \Box] : DB_Y^X}(\text{C-UNIR}) \quad \frac{\Gamma \vdash C : DB_Y^{\{\&x\}}}{\Gamma \vdash C[\&x := \Box] : DB_Y}(\text{C-IMRK}) \quad \frac{\Gamma \vdash C : DB_Y^{X_1 \uplus X_2}}{\Gamma \vdash C[e^{X_1} \oplus \Box] : DB_Y^{X_2}}(\text{C-DUNIR}) \quad \frac{\Gamma \vdash C : DB_Y^X}{\Gamma \vdash C[e_Z^X \mathbin{@} \Box] : DB_Y^Z}(\text{C-APND})$$

$$\frac{\Gamma \vdash C : \mathit{Bool}^-}{\Gamma \vdash C[\mathbf{isEmpty}(\Box)] : DB_Y^X}(\text{C-ISEMP}) \quad \frac{\Gamma \vdash C : DB_Y^X \; \Gamma \vdash e : m \; m \geq 0}{\Gamma \vdash C[\mathbf{if}\, e\, \mathbf{then}\, \Box\, \mathbf{else}\, \{\}] : DB_Y^X}(\text{C-IFT}) \quad \frac{\Gamma \vdash C : DB_Y^X \; \Gamma \vdash e : m \; m \geq 0}{\Gamma \vdash C[\mathbf{if}\, \Box\, \mathbf{then}\, e\, \mathbf{else}\, \{\}] : \mathit{Bool}}(\text{C-IFC})$$

$$\frac{\Gamma \vdash C : DB_{Y \cdot Z}^{X \cdot Z}}{\Gamma \vdash C[\mathbf{rec}(\Box)(e_Y^X)] : DB_Z^Z}(\text{C-RECB}) \quad \frac{\Gamma \vdash C : DB_{Y \cdot Z}^{X \cdot Z} \quad \Gamma \cup \{\$g \mapsto +\} \vdash e : m \quad m \geq 0}{\Gamma \vdash C[\mathbf{rec}(\lambda(\$l, \$g).e)(\Box)] : DB_Y^X}(\text{C-RECA})$$

Fig. 7: Rules for order preserving contexts of UnCAL

(C-*) rules. Since $\cup$ constitutes a commutative monoid, with $\{\}$ as the identity, and UnCAL has compositional semantics as mentioned in Sec. V, i.e., $[\![e_1 \cup e_2]\!]\rho = [\![e_1]\!]\rho \cup [\![e_2]\!]\rho$ where $\rho$ is the run-time variable environment ($\$db$ is bound to the input graph), top level context is additive as the base case. Superscript '–' of the context type means the context is contravariantly OP, i.e., $e_1 \preceq_{\mathcal{L}}^{\mathrm{Sem}} e_2 \Rightarrow C[e_2] \preceq_{\mathcal{L}}^{\mathrm{Sem}} C[e_1]$. Such contexts can safely constitute additive contexts, like the rule (C-ISEMP), since two consecutive contravariance constitute covariance.

Note that for **rec**, we have unconditional equivalence $\mathbf{rec}(e)(d_1 \cup d_2) = \mathbf{rec}(e)(d_1) \cup \mathbf{rec}(e)(d_2)$ [7] for any expression $e$ and graphs $d_1, d_2$. So $\mathbf{rec}(e)$ provides unconditional additive context in the argument position[5].

*Additivity of UnQL:* Since the data model of UnQL is the same as UnCAL, constituting commutative monoid with $\cup$, except we only have default input marker (i.e., every UnQL function/expression has type $DB$), the top level context of UnQL is also additive. Type of the additive context is always of this type, so we omit in the following. The additive context is also defined inductively. The rule for the same language constructs ((C-UNIR),(C-UNIL) and (C-EDG), (C-IF*),(C-ISEMP)) are the same. Since $\{L : T, \ldots, L : T\}$ is equivalent to $\{L : T\} \cup \ldots \cup \{L : T\}$, such a template under additive context constitutes additive contexts for each $T$. For compositions, since **select** translates to **rec**, $C[\mathbf{select}\, \Box\, \mathbf{where} \ldots]$ is additive for additive $C$, while $C[\mathbf{select}\, e\, \mathbf{where}\, P\, \mathbf{in}\, \Box]$ is so if $e$ and predicate $P$ are monotonically increasing. Sibling **sfun** definitions also constitute a partial order such that removal of a definition and calls of an **sfun** generates smaller transformation.

[5]It only applies to $\preceq_{\mathcal{L}}^{\mathrm{Add}}$ obtained by graph union $\cup$. However, as we argue by (C-RECA), we pose monotonicity restriction on the argument. Unconditional additive context can be formed only for $\preceq_{\mathcal{L}}^{\mathrm{Add}}$ by $\cup$ that extends graphs horizontally, while if we consider ':' (@) that extends graphs vertically ($\{\} \preceq_{\mathcal{L}}^{\mathrm{Add}} \{l : \{\}\} \preceq_{\mathcal{L}}^{\mathrm{Add}} \{l : \{l' : \{\}\}\} \preceq_{\mathcal{L}}^{\mathrm{Add}} \ldots$ or equivalently, $\& \preceq_{\mathcal{L}}^{\mathrm{Add}} \{l : \&\} \preceq_{\mathcal{L}}^{\mathrm{Add}} \{l : \&\} \mathbin{@} \{l' : \&\} \preceq_{\mathcal{L}}^{\mathrm{Add}} \ldots$), the condition as stated by (C-RECA) applies. The reason is that the body expression in **rec** may contain the reference to the graph variable bound by the **rec**. The graph bound to the variable is not affected by the horizontal extension of the graph in the argument position but is affected by vertical extension.

## VII. Additivity in Other Languages

Sec. IV and Sec. VI have presented detailed analyses of additivity in (respectively) ATL and UnQL. In the present section, we briefly discuss examples of additivity in other transformation languages. The purpose is to show that additivity is not limited to ATL and UnQL. However, detailed analyses of additivity in these languages is beyond the scope of this paper.

The most natural example of additivity in common use is provided by template-based transformation languages [1]. A template can be thought of as the combination of two parts: 1) a static part contains excerpts of the target artifact that are directly copied to output, 2) metacode is run at template instantiation time to compute the variable parts. The static part of the template is typically additive. E.g., in a model-to-text transformation language, by removing a static excerpt we obtain a template that always creates sub-strings of the original template.

It is harder to find additive contexts in QVT-Relation [15] than it is in ATL. First, QVT-Relation transformations may be multidirectional. Therefore the notions of source and target models only make sense once a choice of direction has been made, or if the transformation is unidirectional. Although QVT-Relation is declarative, it heavily relies on explicit rule

references. The only rules (called *relations*) that can be executed without an explicit reference from another rule are *top* relations. However, even top relations may be explicitly referenced from other relations. Top relations are additive but they need to be replaced with empty rules when they are referenced by other relations. As for property template items, although they are similar to ATL bindings, they can actually also appear in source patterns. Therefore, they may only be additive if they are part of a target pattern.

VTGG, first proposed by Jakob and Schürr [16] and formalized later [17], is a restriction of TGG [18] for declarative definition of updatable model views. The restrictions are: every class rule for each view metamodel are independent and thus can be applied in any order, and association rules are independent of any other association rules. This suggests that class rules are additive at the top level while they constitute additive contexts for their corresponding association rules.

We can look for more specific additive containments in several model transformation languages. For instance, in the UML-RSDS language [19], a rule $R_j$ is independent of preceding rules $R_1, ..., R_{j-1}$ in a transformation $T$, if the write frame of $R_j$ is disjoint from the read and write frames of each $R_i, i < j$. Provided that the rule does not delete target objects, there is a semantic containment of the transformation with rules $R_1$ to $R_{j-1}$ in the complete transformation. This constitutes an additive containment in the sense of this paper.

In general, it is not possible to find useful additive contexts in imperative languages since they do not impose restrictions preventing non-monotonic side effects on the output. For instance, a statement may undo what a previous statement did. However, additivity may appear when some constraints are added to the code. For instance, if the output of a program is limited to its standard output as a monotonically growing list of characters, then removing any *print* statement would result in a reduced output. This basically corresponds to what one can achieve in the case of template-based languages.

## VIII. Related Work

A recent study by Stevens [20] mentions partial transformation with a different meaning, since her partiality refers to the idea of tolerating imperfect consistency between source and target. Her notion of subspace is interesting; subspace is a subset of the domain or codomain of the transformation in which the user agreed to stay during update propagation. The subspace also characterizes the degree of consistency in that an element within a subspace is more/less compatible than the others outside the subspace. In our setting, the target subspace consists of the subset of the codomain covered by target models that are generated by the subtransformation.

Similarly to us, Terrell [21] studies the modularization of model transformations by a divide and conquer approach. However he focuses on *ordered model transformations*, i.e. the case in which source models and target models are ordered by a containment hierarchy, and sub-transformations can be defined at different levels of this hierarchy. W.r.t. his work, we put the focus on the relation among the transformation syntax and the output artefact.

Our notion of additivity is similar to the notion of query containment (e.g., Abiteboul et al. [4]) in the database community. Query $q'$ contains $q$ means for every database $d$ the set of tuples $q'(d)$ (result of applying $q'$ to $d$) includes $q(d)$. Whether two given arbitrary queries have containment relationship can be undecidable depending on the class of the queries, while we reason about syntactic manipulation on a given transformation by removing part of the transformation to obtain another one that is contained by the original one.

The notion of traceability we frequently mentioned in this paper (e.g., correspondence between program fragments and target model elements), has been extensively studied across different communities as the notion of *provenance* [22].

Kelsen et al. [23] provided their own notion of submodel and proposed an efficient decomposition algorithm. Their motivation was to comprehend complex models by decomposition. Our additivity could further enhance this comprehension at the transformation level. The major difference between our notion of submodel and theirs is that model elements in their submodel contain the same (collection of) non-reference features as those in the original model, while we also allow sub collection of features.

As we introduced in Sec. III, we leveraged the additivity contexts of ATL to partially bidirectionalize it [3]. A generic notion of partial order in transformation languages was proposed, and was instantiated for ATL. The notion of additive context was not defined, while the present paper makes it explicit, and explores additivity for several different languages, including their target language UnQL and its core language UnCAL.

## IX. Conclusion and Future Work

A formal characterization of the language features that make transformation languages effective is still missing. In this paper we try to shed some light on the correspondence between transformation code and resulting output, by introducing a simple but common property in transformation languages. This additivity property can be found in languages independently developed within different domains, making us believe it to be a fundamental property in transformation. In future work we want to study the relation of additivity with several independent results on modularization and parallelization of ATL and UnQL. We also want to study how additivity can be used to export such kind of results from one additive transformation language to another. Finally we want to extend our formalization to characterize not only sub-transformations but transformation partitioning and give a clear view of the divide and conquer approach fostered by transformation languages.

**Acknowledgments.** We thank Zheng Cheng and the members of IPL for their valuable comments and suggestions. We also thank the reviewers for their detailed and constructive reviews. This research is funded by JSPS KAKENHI Grant 26330096, 15KK0017 (in conjunction with Zaigai Kenshuin system of Hosei University).

## References


[1] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation Approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, Jul. 2006.

[2] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, Jun. 2008.

[3] S. Hidaka and M. Tisi, "Partial Bidirectionalization of Model Transformation Languages," under review for the journal of Science of Computer Programming, Jul. 2017.

[4] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley, 1995, ch. Static Analysis and Optimization, pp. 105–141.

[5] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[6] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[7] P. Buneman, M. F. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *VLDB Journal: Very Large Data Bases*, vol. 9, no. 1, pp. 76–110, 2000.

[8] I. Kurtev, K. van den Berg, and F. Jouault, "Rule-based modularization in model transformation languages illustrated with atl," *Science of Computer Programming*, vol. 68, no. 3, pp. 138 – 154, 2007, special Issue on Model Transformation. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642307001232

[9] D. Wagelaar, "Composition techniques for rule-based model transformation languages," in *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ser. ICMT '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 152–167. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69927-9_11

[10] D. Wagelaar, R. Van Der Straeten, and D. Deridder, "Module superimposition: a composition technique for rule-based model transformation languages," *Software & Systems Modeling*, vol. 9, no. 3, pp. 285–309, Jun 2010. [Online]. Available: https://doi.org/10.1007/s10270-009-0134-3

[11] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt, "Model Transformations in Practice Workshop," in *Satellite Events at the MoDELS 2005 Conference*. Springer-Verlag, 2006, pp. 120–127.

[12] J. B. Warmer and A. G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley Professional, 1998.

[13] A. S. Al-Sibahi, "On the computational expressiveness of model transformation languages," *ITU Technical Report Series*, no. TR-2015-184, 2015.

[14] M. Gondran and M. Minoux, *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*, 1st ed. Springer, 2008.

[15] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," Version 1.3, OMG Standard formal/2016-06-03, Jun. 2016, available from http://www.omg.org/spec/QVT/1.3.

[16] J. Jakob and A. Schürr, "View creation of meta models by using modified triple graph grammars," *Electron. Notes Theor. Comput. Sci.*, vol. 211, pp. 181–190, Apr. 2008.

[17] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr, "Efficient model synchronization with view triple graph grammars," in *Proceedings of the 10th European Conference on Modelling Foundations and Applications*, 2014, pp. 1–17.

[18] A. Schürr, "Specification of graph translators with triple graph grammars," in *WG '94*, ser. LNCS, vol. 903, Jun. 1995, pp. 151–163.

[19] K. Lano, *Agile Model-Based Development Using UML-RSDS*. Boca Raton, FL, USA: CRC Press, Inc., 2016.

[20] P. Stevens, "Bidirectionally tolerating inconsistency: Partial transformations," in *FASE*, New York, NY, USA, 2014, pp. 32–46.

[21] J. W. Terrell, "Ordered model transformations," Ph.D. dissertation, King's College London, 2014.

[22] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Found. Trends databases*, vol. 1, no. 4, pp. 379–474, Apr. 2009.

[23] P. Kelsen, Q. Ma, and C. Glodt, "Models within models: Taming model complexity using the sub-model lattice," in *FASE*, 2011, pp. 171–185.