GOLD : a graph oriented language for databases

Eindhoven University of Technology

Department of Mathematics and Computing Science

GOLD

a Graph Oriented Language for Databases

by

R. Post and P. De Bra

93/29

Computing Science Note 93/29
Eindhoven, September 1993

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

GOLD

a Graph Oriented Language for Databases

R. Post P. De Bra

Abstract

A simple and powerful graph transformation language for databases is defined whose operations are themselves graphs. It was designed to remedy some of the shortcomings of the GOOD language (see [20]), particularly, its restricted syntax for operations and its dependence on the rather awkward *abstraction* operator.

The new language, GOLD, has only one operation; operations of GOOD are included as special cases. Thanks to a new idea, the *core pattern*, many more forms of node addition can be specified with this one operation; abstraction becomes superfluous. Due to its capability to add multiple nodes at once, GOLD surpasses GOOD in expressive power: queries are always generic – *determinate* in the sense of [2] – but they are not restricted to be *constructive* in the sense of [10].

The simpler forms of the GOLD operation have straightforward interpretations. Instead of imposing syntactic restrictions, the GOLD definition extends the semantics to cover a very broad range of expressions.

1 Introduction

1.1 Purpose

There is an ever increasing interest in database systems that represent information as a network structure. In such systems, end-users, including novices, need to become comfortable with the network structure and actively employ that structure to find the information they need. It is very helpful to have a direct manipulation interface, in which network structure is displayed on-screen and users can move around by pointing and clicking. Many systems do provide such a graphical navigation facility, for instance, the influential NoteCards hypertext system [23]. Clearly, a wide range of information systems can benefit from this approach, in particular, hypertext systems and object oriented databases.

The language presented in this report serves as a basis for a similar graphical interface for *querying*. Queries are expressed as graph transformations, which has two major advantages. It allows us to perform querying for structure, as opposed to the content search that is customary in hypertext; and it solves the problem of representing query results in a manner suitable to the navigation mechanism. Besides ad-hoc user queries, the language can describe other mechanisms involving implicit or automatically generated nodes and links.

There is an elegant way of describing graph transformations in a manner suitable for a graphical interface: draw a prototypical graph and modifications to that graph. The graph is then embedded into the existing instance in every possible way, causing the modifications to take effect wherever a match is found. Combining such transformations into programs leads to a database programming language, standing out from other object oriented query languages in two ways: its database instances are plain graphs, and its operations are designed to be visualized as graphs themselves. Essentially, we have a graph grammar formalism, treated as a database programming language.

This idea was modeled in the GOOD database model and language described in [20, 21]. The language presented in this report, GOLD (Graph Oriented Language for Databases), is a close relative. While GOOD has five operations to perform different types of graph transformations, GOLD has only one operation, obtained by allowing a very liberal syntax and adding a new feature, the *core pattern*, to express grouping. (In section 3.6, the relationship will be exposed in more detail.)

The result is a very compact, elegant language that is more powerful than its predecessor. More expressions are meaningful; as a result, more queries can be expressed in a straightforward way, some of which cannot be expressed in GOOD at all.

Other query languages have been designed to operate on graphs, some with a diagrammatic representation of queries – see, for example, [4], [8], [9], and [13]. GOOD and GOLD are in a special position by the fact that their operations express graph transformations, whereas most other languages express some type of aggregating selection similar to the SQL `SELECT` clause. Very similar to our language is GraphLog [16], in which queries are also graphs that act as patterns over the existing database instance. However, GraphLog queries do not specify modifications to this graph; hence, there is no way to specify the addition of new nodes, and no way to specify deletion. As a consequence, it is a considerably weaker language: not only are new nodes inherently present in certain queries (e. g., the creation of annotations or indexes), but they also provide an elegant aid in composing complex queries – see the closing section for further discussion.

This report is structured as follows. After a brief informal introduction of the language, a formal definition is given and its correctness is established. Then, by syntactically restricting the GOLD operation and studying the expressive power of the languages thus obtained. The relationship with the GOOD language is exposed in detail.

This part of the report is focused on the fundamentals of the language; examples have been chosen with the purpose of introducing its basic features, not to expose its value in practical situations. Besides, we have intentionally omitted all sugarings and enhancements inessential to its operation.

In the closing section, we present our motivation for using GOLD as a hypertext query language, comparing its features to those of other languages. We also mention some of the additions and modifications necessary to turn the language into a practical tool.

1.2 Introductory examples

Let us first consider a toy[1] example database instance in GOLD: some persons with children and marriages indicated (figure 1). Note that a database instance is simply a directed graph; every node (a

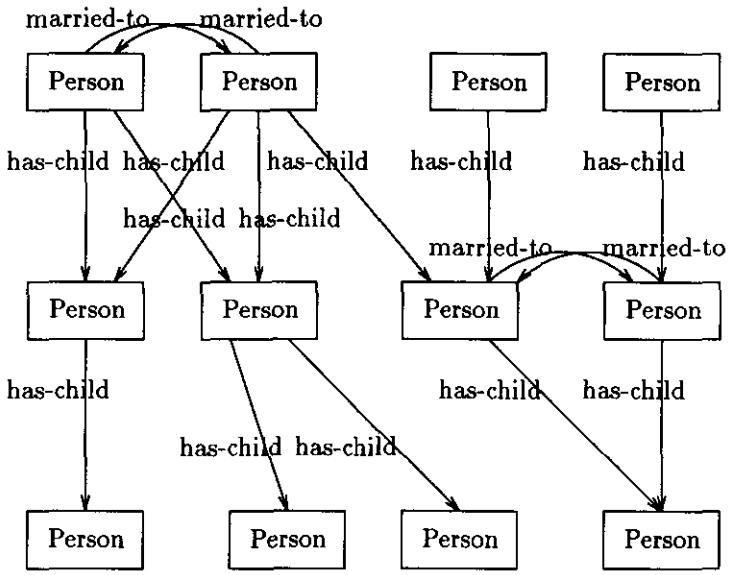
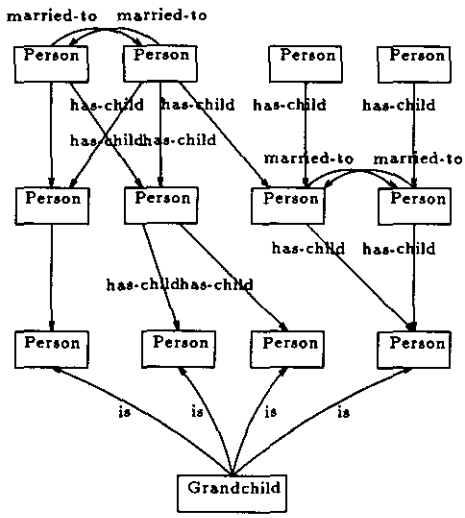


Figure 1: Example instance

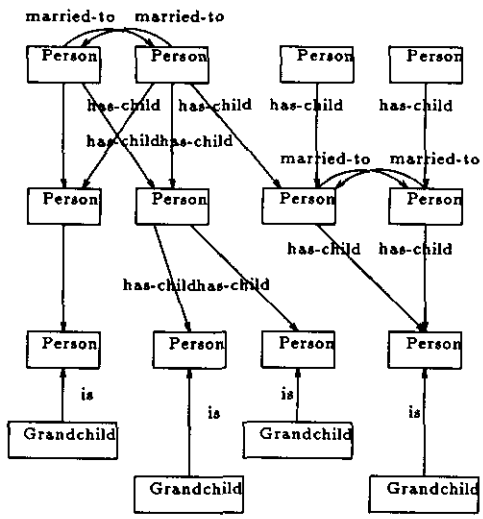
rectangle) has a label (indicating its type) and every edge edges (an arrow) also has a label (indicating the name of a node property, or a binary relationship between nodes).

As an example query, consider the request to find all persons with a grandparent. We must first decide how to represent the result. Suppose we indicate a person with a grandfather by making a node labeled 'Grandchild' point to that person with an arrow labeled 'is'. Three conceivable results of this request are given in figure 2.

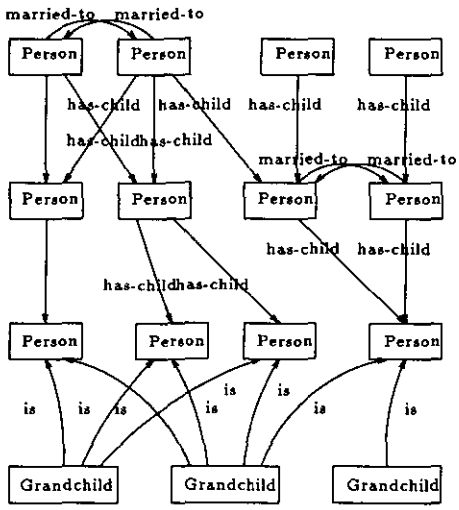
[1] In practice, instances will usually have more types of nodes and edges: for example, every person will have a name.



grandchildren (collectively)



grandchildren (individually)



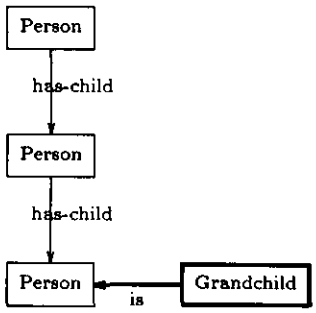
grandchildren (per grandparent)

Figure 2: Indicating grandchildren, three interpretations

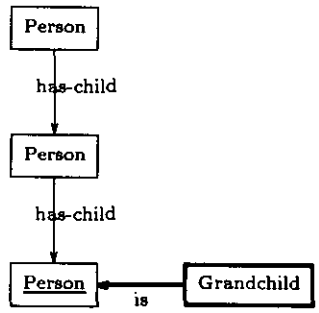
Note that the results differ only in the way the resulting nodes are grouped. In the first instance, one node is created for all grandchildren collectively; in the second, there is a node for each grandchild; in the third example, nodes are created per grandparent, with the proviso that for the two rightmost grandparents, who have the same grandchildren, only one node is created. The GOLD operation features a way for the user to specify this grouping.[2] In other query languages, the issue of how to represent the query result is usually not handled in the language itself; in GOLD, queries are represented as graph manipulations, and the issue is of direct importance. Still, for simple queries there will rarely be need to create more than one new node.

Figure 3 displays three GOLD operations that respectively produce the three results of figure 2 when applied to the example instance of figure 1.

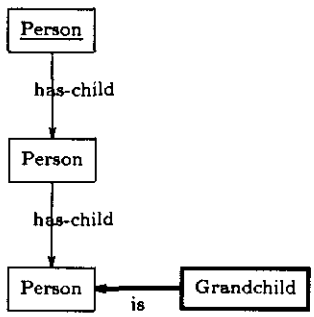
[2]This grouping feature is absent from GOOD, in which sequences of operations would be required to express most cases.



grandchildren (collectively)



grandchildren (individually)



grandchildren (per grandparent)

Figure 3: Three operations to yield the respective results in the previous figure

As apparent from figure 3, an operation is a pattern, in which the emboldened elements specify elements whose presence is required in the resulting instance. To specify grouping of new nodes, we distinguish a *core pattern* inside the non-bold part of the pattern. This is visualized by underlining its nodes and edges.[3]

GOLD is somewhat sophisticated in its way of deciding which nodes and edges to create. First of all, if for some embedding all emboldened elements in the pattern can already be matched in the existing instance, then nothing new is created for that embedding. Secondly, if for two embeddings, exactly the same instance extension must be made in order for the emboldened part to match, then this new part is created only once. This principle will be called *sharing* of new nodes.

For example, in the third instance of figure 2, where grandchildren are grouped by grandparent, two grandparents have the same set of children, and this set is created only once. It would be quite pointless to create two indistinguishable copies.

The mutual differences between the three example instances are due to the different *merging* of new nodes : respectively, no merging at all, merging by parent, or merging by grandparent. This principle is called 'merging' because the new parts for embeddings are combined regardless of whether they constitute identical copies. Merging is controlled by the user, by means of the core pattern.

Deletion of nodes and edges can be specified by drawing them with dashed lines. For example, the operation in figure 4 deletes all persons with at least one child, leaving 4 isolated persons in the example instance.

A more complex example is provided in figure 5.

[3] Initially, only color was used to mark core patterns, but this does not suffice for all purposes.

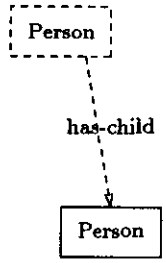


Figure 4: Deleting all parents

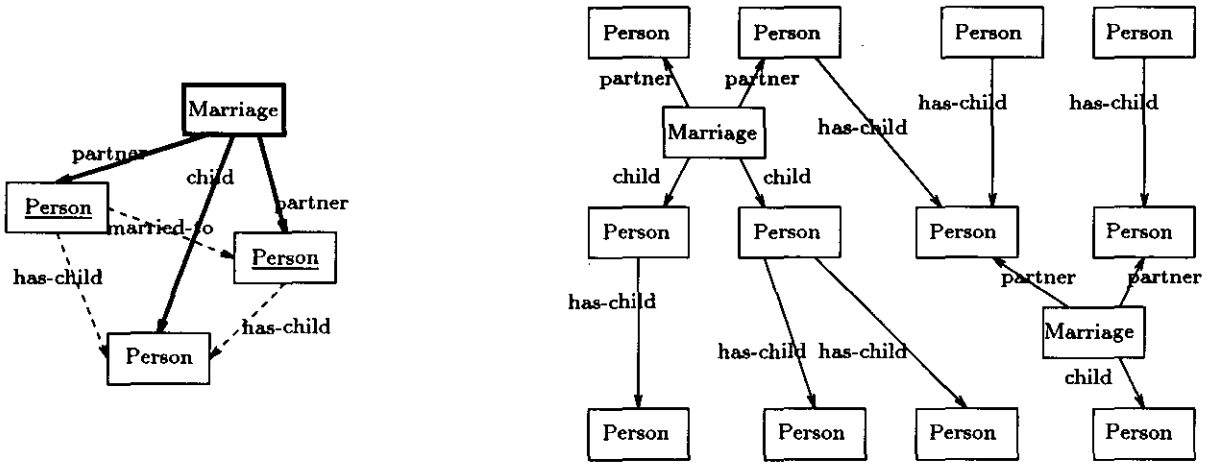


Figure 5: Restructuring the representation of marriages

The operation on the left restructures the representation of marriages, resulting in the instance on the right when applied to the example instance in figure 1. Assuming that a marriage is represented as a pair of edges between persons, these arrows are replaced with a 'Marriage' node from which two 'partner' edges leave. At the same time, any children that were placed under married persons directly are now placed under their marriages instead.

The arbitrary mixing of additions and deletions in one operation and the use of a core pattern are the main differences between GOLD and its predecessor, GOOD [7, 20, 21].

Operations can be composed into sequences. A special bracket can be put around sequences to indicate iteration; an iterated sequence is executed until its net effect is zero. For example, in figure 6, we have two operations, the second of which is iterated, together computing a transitive closure.

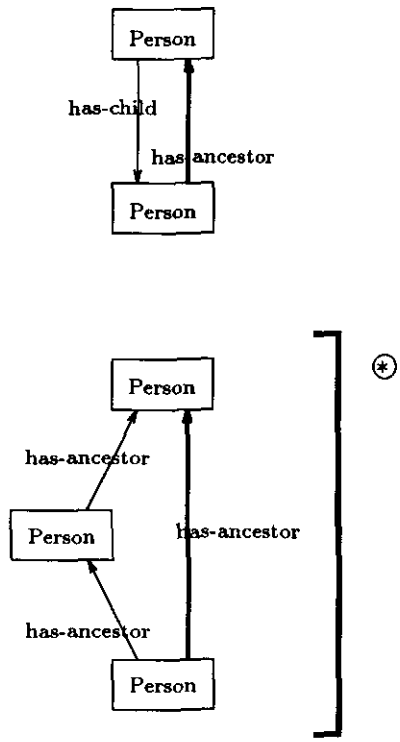


Figure 6: Transitive closure involves iteration

2 Language definition

2.1 Data model

Let NN, EL, NL be countably infinite sets, called the universe of *nodes*, *edge labels*, and *node labels*, respectively. Let $\lambda : NN \rightarrow NL$ be a function (the node labeling function).

A (database) *instance* is a directed, labeled graph $\langle N, E \rangle$, where N is a finite set of nodes and E is a finite set of labeled edges, each of the form $\langle n, \alpha, m \rangle$, where $n, m \in N$ and $\alpha \in EL$. For any instance $\mathcal{I} = \langle N, E \rangle$, we write $N(\mathcal{I})$ for N and $E(\mathcal{I})$ for E. *Pattern* is a synonym for instance.

Definition 2.1 An *embedding* of a pattern $\mathcal{J}$ into an instance $\mathcal{I}$ is a function f mapping the nodes of $\mathcal{J}$ to those of $\mathcal{I}$ such that labels and edges are preserved. A bijective embedding whose inverse is an embedding is called an *isomorphism*. An isomorphism of an instance onto itself is an *automorphism*; the set of automorphisms of an instance $\mathcal{I}$ will be denoted as $Aut(\mathcal{I})$.
□

Note that embeddings need not be injective.

2.2 Operation syntax and semantics

An *operation* is given by a *source pattern* and modifications to it. Within the source pattern, we distinguish a *core pattern*. Nodes and edges may be deleted from the source pattern, leaving a *deletion pattern*; they may be added, resulting in an *addition pattern*. In all, we have a four-tuple of subpatterns :

$$\langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$$

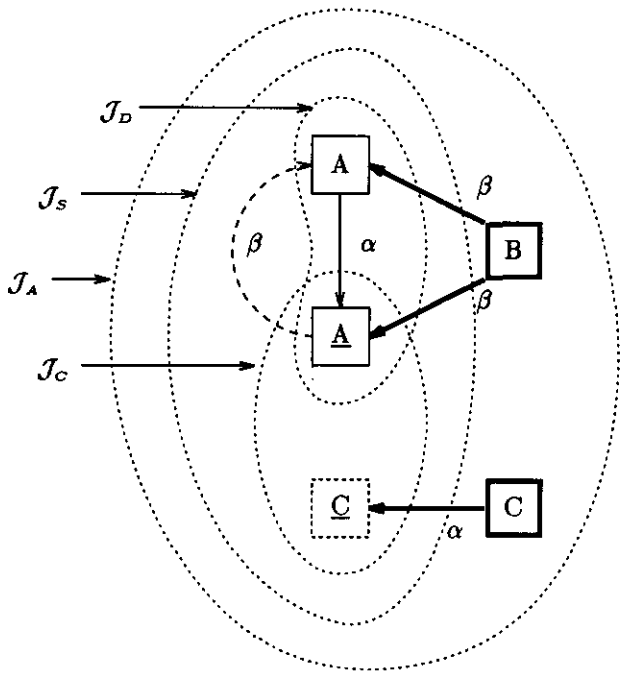


Figure 7: A legal GOLD operation

for the *deletion pattern*, the *core pattern*, the *source pattern*, and the *addition pattern*, respectively. The subpattern relations are:

$$\begin{aligned} \mathcal{J}_D &\leq \mathcal{J}_S \\ \mathcal{J}_C &\leq \mathcal{J}_S \\ \mathcal{J}_S &\leq \mathcal{J}_A \end{aligned}$$

An example is given in figure 7.

A *new* (or *added* element (edge or node) of an operation is an element in $\mathcal{J}_A$ that is not in $\mathcal{J}_S$; a *deleted* element (edge or node) is an element in $\mathcal{J}_S$ that is not in $\mathcal{J}_D$.

The idea is to find all embeddings of the source pattern in the source instance, and, for every embedding, delete all deleted elements and add all added elements. However, as shown in the grandchildren example (figure 2), we may wish to have a common added part for several embeddings.

To arrive at a deterministic interpretation in all cases, the following measures were taken:[4]

- the *core pattern* allows users to disambiguate queries by themselves: extensions for two embeddings equal on the core pattern are *merged*;
- additional rules of interpretation, amongst which is the *sharing* of extensions, remove all remaining ambiguity.

These rules are fairly simple, leading to an intuitive and computationally feasible semantics; furthermore, most of the operations of GOOD are included as special cases (cf. theorem 3.12), which allows us to apply most of the experience from studying and using GOOD, which has a working implementation (see [18].

If an operation t can transform an instance $\mathcal{I}$ into another instance $\mathcal{I}'$, this will be denoted as $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$. Operations can create new nodes, whose identities are chosen at random. It will be seen (lemma A.2)

[4] Another approach is to have a nondeterministic language, as was done in the G-Log specification language – see [27].

that operations are completely deterministic up to this choice; nevertheless, we must regard the effect of an operation as a relation on database instances.

Definition 2.2 For any operation $t = \langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$, and instances $\mathcal{I}, \mathcal{I}' \in \mathcal{U}$, $\mathcal{I} \overset{t}{\Rightarrow} \mathcal{I}'$ under the following conditions:

Write $\mathcal{I}'' \underset{D}{=} \langle N(\mathcal{I}) \cup N(\mathcal{I}'), E(\mathcal{I}) \cup E(\mathcal{I}') \rangle$. $\mathcal{I}$ is the *source instance*; $\mathcal{I}'$ is the *target instance*; $\mathcal{I}''$ is the *addition instance*.

Let F be the set of embeddings of the source pattern into the source instance that cannot be extended to embeddings of the addition pattern into the source instance.

An embedding g of the addition pattern into the addition instance is an *injective extension* of an $f \in F$, if g includes f and maps all new pattern nodes injectively to new instance nodes.

There must be a function η mapping all $f \in F$ to embeddings from $\mathcal{J}_A$ into $\mathcal{I}''$, such that

1. *(new nodes to new nodes)* for all $f \in F$, $\eta(f)$ is an injective extension of f;
2. *(merging)* for all $f_1, f_2 \in F$, if f_1 and f_2 are identical on $\mathcal{J}_C$, they are also identical on every new node of $\mathcal{J}_A$;
3. *(sharing)* for all $f_1, f_2 \in F$, if the ranges of $\eta(f_1)$ and $\eta(f_2)$ constitute isomorphic extensions of $\mathcal{I}$, i.e. there is an isomorphism on $\mathcal{I}''$ that fixes $\mathcal{I}$ and maps one range to the other, then $\eta(f_1)$ and $\eta(f_2)$ map the nodes of $\mathcal{J}_A \setminus \mathcal{J}_S$ to the same set of nodes in $\mathcal{I}''$;[5]
4. *(no unnecessary merging or sharing)* for all $f_1, f_2 \in F$, if neither merging nor sharing applies, the ranges of $\eta(f_1)$ and $\eta(f_2)$ are disjoint on the new nodes of $\mathcal{I}''$;
5. *(merging before sharing)* for all $f_1 \in F$, if $\langle n_1, \alpha, n_2 \rangle$ is an edge between two nodes in the range of $\eta(f_1)$, not both in $\mathcal{I}$, then there is an $f_2 \in F$, equal to f_1 on $\mathcal{J}_C$, such that for some edge $\langle m_1, \alpha, m_2 \rangle \in \mathcal{J}_A$, $\eta(f_2)(m_1) = n_1$ and $\eta(f_2)(m_2) = n_2$;
6. *(minimality)* every new node and edge of $\mathcal{I}''$ is in the range of some $\eta(f)$;
7. *(deletion after addition)* $\mathcal{I}'$ is the maximal subinstance of $\mathcal{I}''$ which does not contain any nodes or edges that some embedding from $\mathcal{J}_S$ to $\mathcal{I}$ maps a node or edge in $\mathcal{J}_S \setminus \mathcal{J}_D$ to.

□

GOLD is deterministic up to the choice of new node identities. Furthermore, the identities of existing nodes do not affect the outcome of operations. This property is known in literature as *determinacy*, and is universally regarded as a fundamental property of query languages. See appendix A.1 for details. It can be formally defined as follows.

Definition 2.3 Let R be a (two-place) relation on a set of instances. R is *determinate* if we have, for all instances $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4$, if $\langle \mathcal{I}_1, \mathcal{I}_2 \rangle$ is in R and $\mathcal{I}_1$ and $\mathcal{I}_3$ are isomorphic, then $\langle \mathcal{I}_3, \mathcal{I}_4 \rangle$ is in R if and only if $\mathcal{I}_2$ and $\mathcal{I}_4$ are isomorphic by the same isomorphism.
□

The ensuing definition of database transformations follows [2].

Definition 2.4 Let S be a finite set of node and edge labels. A *database transformation* over S is a relation on the instances over the labels in S that is

[5] I.e. the mappings are not necessarily pointwise equal on these nodes.

- determinate;
- *computable*: an effective procedure exists to compute a result on every possible input instance for which a result is defined.

□

We can now formally state our claim concerning GOLD:

Theorem 2.1 All GOLD programs express database transformations.
□

In addition, GOLD operations define a result on all instances. In this respect, GOLD is different from its close relative, GOOD (see [19, 20]), which leaves results undefined in some cases. For details, see appendix A.1.

2.3 Program syntax and semantics

Programs are defined recursively as follows. Every operation is a program. If p_1 and p_2 are programs, their sequential composition, denoted $p_1; p_2$, is also a program. The empty program is denoted Λ. If p is a program, the iteration of p, denoted $p^\star$, is also a program.

Nodes, once deleted, can never reappear as new nodes.[6] The auxiliary relation $\mathcal{I} \overset{p}{\underset{N}{\Rightarrow}} \mathcal{I}'$ will denote the property that p can transform $\mathcal{I}$ into $\mathcal{I}'$ without deleting any node in N.

Definition 2.5 If p is a program, and n is a number, then the notation p^n stands for a certain program determined as follows:

- p^0 denotes Λ;
- p^1 denotes p;
- p^{n+1} denotes $p^n; p$ for all $n > 0$.

□

Definition 2.6 For all sets of nodes N, operations t, programs p, p_1, p_2, and instances $\mathcal{I}, \mathcal{I}'$:

$$\mathcal{I} \overset{\Lambda}{\underset{N}{\Rightarrow}} \mathcal{I} \quad \underset{D}{\Longleftrightarrow} \quad N \subseteq N(\mathcal{I})$$

$$\mathcal{I} \overset{t}{\underset{N}{\Rightarrow}} \mathcal{I}' \quad \underset{D}{\Longleftrightarrow} \quad \mathcal{I} \overset{t}{\Rightarrow} \mathcal{I}' \wedge N \subseteq N(\mathcal{I}) \cap N(\mathcal{I}')$$

$$\mathcal{I} \overset{p_1;p_2}{\underset{N}{\Rightarrow}} \mathcal{I}' \quad \underset{D}{\Longleftrightarrow} \quad \exists \mathcal{I}'' \in \mathcal{U} : \mathcal{I} \overset{p_1}{\underset{N}{\Rightarrow}} \mathcal{I}'' \wedge \mathcal{I}'' \overset{p_2}{\underset{N}{\Rightarrow}} \mathcal{I}'$$

$$\mathcal{I} \overset{p^\star}{\underset{N}{\Rightarrow}} \mathcal{I}' \quad \underset{D}{\Longleftrightarrow} \quad \exists n \in \mathbb{N} : \mathcal{I} \overset{p^n}{\underset{N}{\Rightarrow}} \mathcal{I}' \wedge \mathcal{I}' \overset{p}{\underset{N}{\Rightarrow}} \mathcal{I}'$$

□

This four-place relation can now be used to define the effect of programs in GOLD: [7]

[6] Without this property, many of our observations on expressive power in section 3 become incorrect.

[7] For $\mathcal{I} \overset{t}{\Rightarrow} \mathcal{I}'$, t can now be read either as an operation or as a program; this is harmless, since in both readings, the same relation is expressed.

Definition 2.7 For all programs p and instances $\mathcal{I}, \mathcal{I}'$:

$$\mathcal{I} \overset{p}{\Rightarrow} \mathcal{I}' \underset{D}{\iff} \mathcal{I} \overset{p}{\underset{X}{\Rightarrow}} \mathcal{I}' \qquad \text{where } X = N(\mathcal{I}) \cap N(\mathcal{I}')$$

□

From this definition, the following laws of equivalence between programs almost immediately follow:

$$\begin{aligned} p_1;(p_2;p_3) &\equiv (p_1;p_2);p_3 \\ p_1 &\equiv \Lambda;p_1 \equiv p_1;\Lambda \\ \Lambda^\star &\equiv \Lambda \\ p_1^{\star\star} &\equiv p_1^\star \\ p_1^\star;p_1 &\equiv p_1^\star \equiv p_1;p_1^\star \equiv p_1^\star;p_1^\star \end{aligned}$$

Whenever a GOLD program does not define a result instance for some source instance, this is due to an iteration that fails to reach a fixpoint. Furthermore, GOLD programs, like operations, are determinate. For details, see appendix A.1.

3 Expressive power of GOLD

We have seen in theorem 2.1 that GOLD programs express only (determinate) database transformations. In the following we will study the classes of database transformations expressible in GOLD, and we will study the role of various features of the GOLD operation in detail. This is done by breaking up the operation in parts, and imposing further syntactic constraints; by exact characterizations of the expressive power of the sublanguages of GOLD thus obtained, we provide some insight in the contributions of the features that are dropped along the way. In particular, a detailed comparison will be made between GOLD and its relative, GOOD.

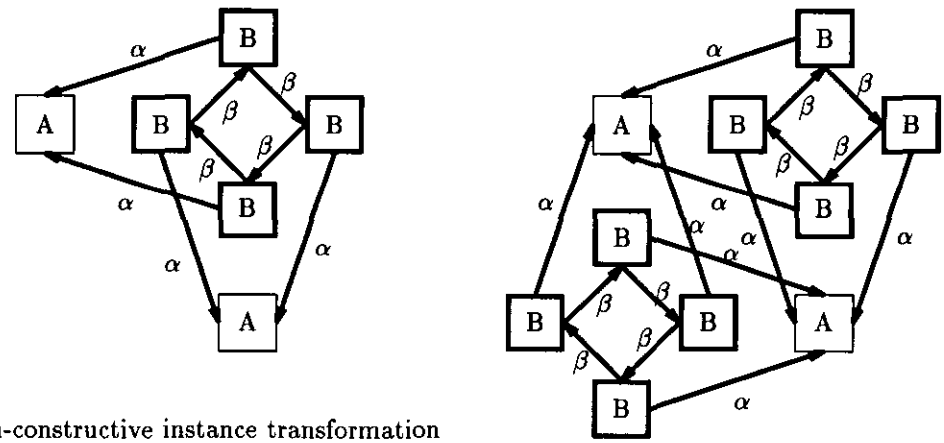
3.1 Constructiveness: a strong form of determinacy

The concept of *determinacy* was defined in definition 2.3; a stronger notion of determinacy was introduced in [7] and, in [10], was named *constructiveness*.

Definition 3.1 An instance transformation $(\mathcal{I}_1, \mathcal{I}_2)$ is *constructive* if there is a group homomorphism that maps every automorphism of $\mathcal{I}_1$ to an automorphism of $\mathcal{I}_2$ that is identical on their common nodes.

A database transformation is constructive if it contains only constructive instance transformations.
□

An example of a non-constructive transformation is given in figure 8. In [7, 10], it was shown that



a non-constructive instance transformation

(the new part is indicated in bold)

its constructive approximation

Figure 8: A non-constructive instance transformation

constructiveness essentially requires that new nodes are not mutually dependent (in our case, connected with edges) at the time they are created. In a constructive language, we can approximate non-constructive results by having multiple copies of the new part (as shown at the right of figure 8). Therefore, constructive languages are also characterized as languages unable to perform *copy elimination*.

This is particularly relevant here because GOLD, unlike, for example, GOOD, can perform non-constructive transformations.

3.2 Expressive power of full GOLD

The following definitions of expressive power are used in literature.

Definition 3.2 A *database language* is a set of expressions, each of which expresses, in the context of a given set of instances, a database transformation (over this set).

A database language L is *(determinate) complete* if for every finite set of labels S, every database transformation over S is expressible in L.

It is *(determinate) instance complete* if every instance transformation is expressible in L, i.e., is part of some database transformation expressible in L.

A database language L is *constructive complete* if for every finite set of labels S, every constructive database transformation over S is expressible in L.

It is *constructive instance complete* if every constructive instance transformation is expressible in L, i.e., is part of some database transformation expressible in L.
□

To study the expressive power of GOLD, we will curtail its operation and observe how expressive power is affected. General GOLD can express all constructive database transformations and some non-constructive ones :

Theorem 3.1 GOLD is not determinate complete, but is instance complete and constructive complete.

□

In essence, the fact that GOLD can contain arbitrarily large addition patterns makes it instance level complete; it is not fully determinate complete, because it can be shown that arbitrarily large addition patterns are also *required* to express any generic instance transformation in GOLD. Another way of stating this result is to say that GOLD can perform copy elimination of copies of predefined shape, or copies of bounded size.

To prove the theorem, we will split it up into the lemmas 3.2, 3.3, 3.4, of which it is an immediate consequence.

Lemma 3.2 There is a database transformation which is not expressible in GOLD.
□

The proof is given in appendix A.2.

Lemma 3.3 Every instance transformation can be expressed in GOLD.
□

For a proof, see appendix A.3.

Lemma 3.4 Every constructive database transformation can be expressed in GOLD.
□

Proof This immediately follows from theorem 3.1 below, which states that a sublanguage of GOLD expresses exactly all constructive database transformations.

□

3.3 GOLD with separated operators

We will now disallow the combination of node and edge operations, and of addition and deletion. This will be shown not to affect expressive power.

Definition 3.3 A GOLD operation $t = (\mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A)$ is an (n-edge) *edge addition* if it has n *added* edges, and no other added or deleted elements; it is an (n-edge) *edge deletion*) if it has n *deleted* edges, an no other added or deleted elements; it is an (n-node) *node addition* if it has n *added* nodes, possibly some added edges incident to these nodes, and no other added or deleted elements. it is an (n-node) *node deletion*) if it has n *deleted* nodes, possibly some deleted edges incident to these nodes, and no other added or deleted elements.
□

Note that these definitions refer to syntax; a 2-edge addition can very well add 3 edges to a particular instance.

Let GOLD_{sep} be the subset of GOLD in which only node additions, node deletions, edge additions, and edge deletions are allowed. Thus, operations are separated into four types; hence the name.

Lemma 3.5 GOLD_{sep} and GOLD have equal expressive power.
□

The proof is given in appendix A.4.

3.4 GOLD with only single addition or deletion

We can impose a further separation by requiring operations to have only one added or deleted node or edge. This restriction can be imposed independently on all four operations in GOLD_{sep}. We will denote the resulting sublanguages as GOLD subscripted with a subset of the symbols $\{1nd, 1na, 1ed, 1ea\}$. For instance, $\text{GOLD}_{\{1nd,\, 1ea\}}$ denotes GOLD in which only node additions, 1-node node deletions, 1-edge edge additions, and edge deletions are allowed. The following theorem says that expressive power is decreased if and only if this restriction is placed on node addition. This is in full agreement with the abovementioned observation, that constructiveness requires nodes to be dependent only on existing nodes at the moment of creation.

Theorem 3.6 If GOLD_X is a language as just described, then

- it expresses exactly all database transformations expressed by GOLD, if $1na$ is not in X;
- it expresses exactly all constructive database transformations, if $1na$ is in X.

□

Most cases are proved by suitable simulations of n-fold operations by means of 1-fold operations. These are easy to find. The technique is the same as that used in lemma 3.5: using new labels, the edges or nodes to be deleted or added are marked, then the actual deletion or addition takes place with a sequence of operations.

We consider the additional expressiveness of n-node addition over 1-node addition as an accidental property of the language. Non-constructive transformations, certainly the restricted ones expressible in GOLD, do not appear to be very important in practice. More interesting is the fact that when imposing the restriction to $1na$, we still have a constructive complete language. In particular, we will define GOLD_1 to be the language $\text{GOLD}_{\{1na\}}$. Then

Corollary 3.1 GOLD_1 is constructive complete.
□

This follows from theorems 3.11 and 3.12, given below.

3.5 GOLD without core patterns

Note that core patterns are effectively ignored except in node additions. To be precise, the following lemma holds.

Lemma 3.7 Let $t_1 = \langle \mathcal{J}_D, \mathcal{J}_{C1}, \mathcal{J}_S, \mathcal{J}_A \rangle$ and $t_2 = \langle \mathcal{J}_D, \mathcal{J}_{C2}, \mathcal{J}_S, \mathcal{J}_A \rangle$ be GOLD operations, such that $N(\mathcal{J}_{C1}) = N(\mathcal{J}_{C2})$ or $\mathcal{J}_A = \mathcal{J}_S$. Then t_1 and t_2 express the same database transformation.
□

However, in node additions, their presence is vital: removing core patterns from the language decreases expressive power.

A GOLD operation $t = \langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$ is a *plain* operation if $\mathcal{J}_C = \mathcal{J}_S$. (This effectively removes the core pattern feature; hence the name 'plain'.)

Let $\text{GOLD}_{\text{plain}}$ be the subset of GOLD_1 in which only plain operations are allowed.

Lemma 3.8 $\text{GOLD}_{\text{plain}}$ cannot express all constructive database transformations, although it can express all constructive instance transformations.
□

The proof can be obtained by analogy to the reasoning followed in [11]. Essentially, it can be shown that in $\text{GOLD}_{\text{plain}}$, arbitrarily many edges connecting a new node to the source pattern are required in order to express all constructive instance transformations. An example of a non-expressible database transformation is one that points out α-clusters of A-labeled nodes, where an 'α-cluster' is a maximal set of A-nodes connected with paths of labeled α-labeled edges, and clustering is done by creating, for each cluster, a new A-labeled node and α-labeled edges from this node to every existing A-labeled node in the cluster.

3.6 GOOD as GOLD

Let GOLD_{tup} be the subset of $\text{GOLD}_{\text{plain}}$ in which in all node additions, the new node has no incoming edges, and all outgoing edges have different labels. This is node addition as it appears in GOOD. The name stems from the fact that in this language, node addition effectively creates instantiations of tuples over existing nodes, provided that we view the outgoing edges of new nodes as tuple attributes.

Lemma 3.9 GOLD_{tup} cannot express all constructive instance transformations.
□

The reasoning required for this proof appears in [10] and [11], where node creation operators are used that can be shown to be equivalent to node addition in $\text{GOLD}_{\text{plain}}$. A non-expressible instance transformation is shown in figure 9.

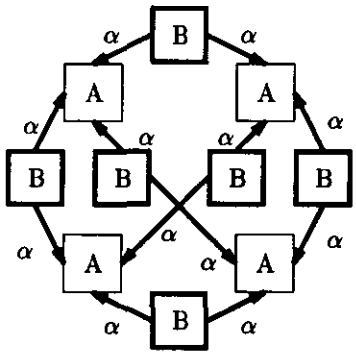


Figure 9: An instance transformation that cannot be expressed in GOLD_{tup}

3.6.1 GOOD operations are special cases of the GOLD operator

We can now substantiate that GOLD was obtained from GOOD by relaxing syntactic restrictions and adding the core pattern construct.

Theorem 3.10 The four operations of GOLD_{tup} are both syntactically and semantically identical to the operations with identical names in GOOD.
□

For more details, see appendix A.5.

3.6.2 Core patterns versus abstraction

Previous lemmas show that without an addition of some sort, GOLD_{tup} will not be constructive complete. In GOLD the core pattern feature fulfills this role. In GOOD, an extra operation is used to obtain constructive completeness : *abstraction*. We will use a definition similar to that in [20, 21].

Definition 3.4 An abstraction is given by a tuple $\langle \mathcal{J}, m, K, \alpha, \beta \rangle$ in which $\mathcal{J}$ is a pattern, m is a node of $\mathcal{J}$, K is a node label, and α, β are edge labels.
□

To understand the semantics of abstraction, it is necessary to view outgoing edges with identical labels as representing a grouping of the nodes that these edges point to :

Definition 3.5 If $\mathcal{I}$ is an instance, n is a node of $\mathcal{I}$, and α is an edge label, the *α-set of n* is the set of nodes m such that $\langle n, \alpha, m \rangle$ is an edge of $\mathcal{I}$.
□

The effect of abstractions is defined as follows.

Definition 3.6 Given two instances $\mathcal{I}, \mathcal{I}'$ and an abstraction $t = \langle \mathcal{J}, m, K, \alpha, \beta \rangle$, $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$ if $\mathcal{I}'$ is a minimal superinstance of $\mathcal{I}$ such that all new edges in $\mathcal{I}'$ leave a new node in $\mathcal{I}'$, and for all nodes $f(m)$ selected by the embeddings f of $\mathcal{J}$ into $\mathcal{I}$ there is a K-labeled node with β-edges to exactly those nodes of $\mathcal{I}$ with α-sets identical to the α-set of $f(m)$.
□

An example of abstraction can be found in figure 10. The edge α is drawn as a dashed arrow, leaving the node m; K and β are represented as an emboldened K-labeled node with a β-labeled edge to m.

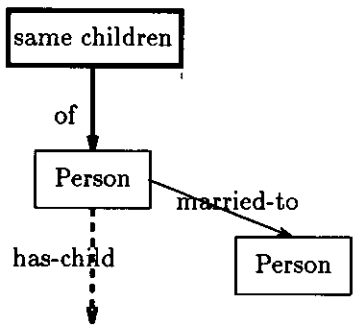


Figure 10: Abstraction : partitioning married persons according to children

Theorem 3.11 GOLD_{tup} with abstraction expresses exactly all constructive database transformations.

□

A proof can be found in, for example, [10].

The basic proposition of this report is the equivalence of abstraction and core patterns.

Theorem 3.12 GOLD_{tup} with abstraction and GOLD_1 have equal expressive power.
□

For a proof, it suffices to provide simulations for both directions. These are easy to find; details are in appendix A.6.

Abstraction is convenient in some cases, but in general we regard core patterns as a much more intuitive means to achieve the same level of expressive power. They provide a grouping facility which is useful in many cases; obviously, the minimal set of examples in this report does not suffice to bear out the practical applications of core patterns.

Abstraction was introduced partly because it provides the expressive power of a powerset operator while being computable in polynomial time (see [11]). From the language definition above, it will be clear that operations with core patterns share this advantage. For more details, see section A.1.2.

3.6.3 Another alternative: the contraction operation

The contraction operator is somewhat simpler to understand than abstraction. It 'merges' nodes by replacing them with a single new node that takes over all edges incident to any of these nodes.

We mention it here to show yet another way of achieving constructive completeness.

Definition 3.7 A contraction is given by a tuple $\langle \mathcal{J}, n, m \rangle$ where $\mathcal{J}$ is a pattern, of which n and m are nodes.
□

The effect of contractions can be defined as follows.

Definition 3.8 Given two instances $\mathcal{I}, \mathcal{I}'$ and an contraction $t = \langle \mathcal{J}, n, m \rangle$, $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I}'$ is a minimal instance such that for all embeddings f of $\mathcal{J}$ in $\mathcal{I}$, there is an embedding g_f of $\mathcal{J}$ in $\mathcal{I}'$, such that

- for all f, $f(n)$ is not in $\mathcal{I}'$, and $g_f(n)$ is not in $\mathcal{I}$;
- for all f, g_f and f are equal except on n;
- for all f_1, f_2, $g_{f_1}(n) = g_{f_2}(n)$ if and only if $f_1(m) = f_2(m)$.

□

Using the known GOOD operators (node addition, edge addition, node deletion, and edge deletion), we have algorithms to transform every abstraction operation into a program using contraction, and vice versa. To be precise, these programs produce programs equivalent for all instances except some that use some specific edge or node labels.[8]

It is easy to simulate contraction by means of abstraction. The idea behind a simulation of abstraction with contraction is that ordinary GOOD operations can already simulate abstraction 'up to copy': too many 'representants' of the new nodes may be created. It is possible to relate two nodes with an edge if and only if they are representants of the same abstraction-created node. Contraction can then be used to merge the resulting clusters of nodes into single nodes.

(A formal proof is omitted.)

It is interesting to note that, like addition, contraction can be generalized to work on more than one node. Instead of a single node, larger subinstances are collapsed, specified by a pattern. This leads to an language equally expressive to GOLD: it can perform copy elimination for copies of bounded size. It is also possible to devise a full copy elimination operation, which lifts the language to full completeness, but we have found no such operation that naturally fits in the pattern embedding paradigm. Details have been omitted from this report.

[8] This is unavoidable, as any simulation requires the use of (some small amount of) 'fresh' node or edge labels which are assumed not to occur in any of the instances the operation is applied to. When applied to instances that do contain such labels, the simulation and the original operation may have different effects.

4 Discussion and concluding remarks

We have introduced a graphical query language for databases and hypertext environments, that can be used to pose queries and updates to a network structure by means of direct manipulation.

This report was written to describe its visualization, using a basic set of examples; to provide a formal definition; and to expose the fundamental properties of the language regarding expressive power. Therefore, we have stripped down the language to its bare bones, and we have done very little to convince the reader of its practical usefulness. It will now be discussed briefly why we consider GOLD an improvement over existing database and hypertext query languages.

4.1 Purpose of GOLD

Querying in hypertext is often ill-supported, or is limited to querying for the content or attribute values of nodes. GOLD supports *structure-based* queries : it assumes that the hypertext forms a labeled graph, in which labels and structures recur in systematic patterns, making it useful to query for such patterns.[9] In other words, the hypertext structure must form a semantic network. (We can include content-based querying by allowing node and edge labels to be computed dynamically from content information.)

Many hypertexts, though not all, can be viewed as forming a semantic network in this manner. If they do, graphical browsing tools, that display the surroundings of the current node, are particularly useful, because the structure on display will be meaningful to the user. Such tools allow for a direct manipulation style of browsing; GOLD makes it possible to seamlessly integrate querying and browsing.

Because of its purpose, GOLD differs from many other database query languages in several respects :

- it is a visual language;
- it operates on labeled graphs;
- it transforms the database instance.

We will discuss these differences below.

4.2 GOLD is a visual language

At first sight, the fact that GOLD is a visual language is its most striking feature. However, there are many visual query languages, and GOLD does not claim to open a radical new approach to visual querying. It does differ from most visual languages by the fact that it represents queries as graphs acting as patterns on the instance. Some languages use the same principle (GraphLog [16], GOOD [20], the language of Catarci et al. [13]); many others are based on forms and condition boxes (for example, QBE [30] and VQL [28]).

Besides, the gap between textual and visual languages is not that big. Some textual query languages can easily be given visual counterparts, for example, the hypertext query languages of [4] and [9]. Generally speaking, it is not always possible to provide suitable visual expressions for all constructs; there is also a danger of cluttering the visual language with constructs and elements, which makes expressions large and hard to understand. A possible solution is a hybridic language; see, for example, the object SQL presented in [5]. GOLD was designed to be fully graphical, and hence, has a very simple syntax that is very easy to visualize. Still, many queries in textual languages are directly expressible GOLD queries. (See however the section on extensions, page 21.)

[9] A straightforward model of a hypertext as a GOLD instance does not capture *anything* of the content of hypertext nodes. Content can be captured with extra nodes, or we can extend GOLD for this purpose (cf. section 4.5).

Conversely, it is a trivial exercise to design textual versions of GOLD;[10] figure 11 gives example textual representations of the marriage query of figure 5.

CREATE Marriage m, m partner p_1, m partner p_1, m child p_3
DELETE p_1 married-to p_2, p_1 has-child p_3, p_2 has-child p_3
FROM Person p_1, p_2
WHERE p_1 married-to p_2, p_1 has-child p_3, p_2 has-child p_3
GROUP BY p_1, p_2

To aid legibility, deleted elements are listed twice.

Figure 11: The marriage restructuring query, presented textually

Observe that in the textual representation, the pattern formed by the nodes and edges is much harder to detect. It is for patterns like this one, with few nodes and a moderate number of edges between them, that the graph-based style of GOLD is most useful. If patterns consist only of paths, regular expressions are an exquisite formalism to specify them textually. If there are too many nodes or edges, the graph will be too large or too dense to be transparent for the user.

4.3 GOLD operates on graphs

GOLD uses labeled graphs as its data model. In a hypertext environment, nodes and links are modelled directly, capturing the end-user's view of the information in a simple and intuitive fashion.

This simplicity is deliberate : it makes the language easy to understand and easy to use. A simple graph-based model lends itself particularly well for a graphical interface. As mentioned in the introduction, graphical browsing aids are common in hypertext systems; graphical querying aids are similarly useful. GOLD can easily be adapted to operate on more complex data models. It remains to be seen to what extent the language will remain easy to understand and to use.[11]

As a result, much of the additional structure common in hypertext cannot be expressed directly; for example, we do not support a notion of composite nodes or context, contrary to some hypertext models [12, 17, 22, 26]; we have only one-to-one links; and we do not explicitly model node anchors, which more or less discards hypertexts in which nodes are large documents. For some applications, having direct support for such features may outweigh the added complexity it brings to the language. It is yet to be studied how to provide such support in GOLD.

In a general database context, the simplicity of the GOLD data model is even more striking. In comparison to NIAM [29], GOLD can model only binary relations between objects; compared to IFO [1] or the IQL data model [2], there is no notion of subclassing, nor is there a tuple or set construct to form complex values. As an example of a model containing all these features, consider PSM [25], which is basically NIAM extended with the features of IFO. By necessity, the visualization of database schemes in PSM employs far more graphical conventions than needed in GOLD. Drawing instances or queries in the same fashion would require even more conventions and use up far more screen real estate. Not surprisingly, the PSM query language proposed in [24] is textual.

We may need some extensions to GOLD (see section 4.5), but generally consider its simplicity a virtue, not a handicap. Features such as non-binary relations, composite nodes, and tuple or set aggregation can be represented by introducing extra nodes, so there is no real need to support them directly.

[10] I.e., apart from the literal mathematical notation, which is far too illegible to be of practical use.

[11] The same question can be asked for the models themselves. For example, end users can be trusted not to think of information in terms of sets and tuples.

4.4 GOLD transforms the database instance

In the relational database model, a database instance consists of tables (relations). A query, applied to an instance, will produce a relation.

Similarly, queries in object-based languages mostly yield sets of objects or sets of values. Gram [3, 4], a hypertext query language which operates on labeled graphs, yielding sets of paths. In [4], both an algebra version and an SQL version of Gram are presented that remain very close to their relational and object based counterparts.

By contrast, GOLD queries yield a transformed instance. (In this respect, GOLD is identical to GOOD [20].) By expressing queries explicitly as modifications to the instance, browsing and using query results no longer requires special representations or tools. Modifications made for queries will normally be 'virtual'; they won't permanently change the database, but will rather provide a temporary modification or view. Updates can be made by telling the system to record this temporary instance as the new one.

Compare this to Gram queries, which yield sets of paths in the instance. Sets of paths are not a very useful concept for the user, and it is unclear how to present them. In [4], the authors propose two ways of using Gram. Firstly, they let Gram queries define virtual links, one link for every pair of nodes in a path yielded by the query. It is more natural to let the creation of the links itself be specified as part of the query. This is what happens in a GOLD edge addition. Secondly, they use Gram queries to delimit the scope of navigation, by restricting navigation to the nodes and links present in a query result. This can be expressed more naturally with GOLD deletions. Not only is the GOLD approach more natural, it is also much more flexible, allowing the creation and deletion of virtual edges and nodes in ways that are hard if not impossible to mimic with Gram.

These objections hold for all languages of this type, for example, the hypertext query language of [9], and versions of object SQL.

Much closer to GOLD is the GraphLog language [14, 15, 16]. In fact, our edge additions are almost identical to the corresponding GraphLog operations. The basic difference remains that GraphLog queries express selections of nodes and edges, whereas GOLD expresses instance transformations, and as a result is more powerful. This is mainly due to the support of (virtual) node creation, which has important practical advantages.[12]

- First of all, the user is given more control over how to express query results. For example, the user may be interested in a certain set of paths, but, instead of creating virtual links between existing nodes, may wish to create one new node with links to the endpoints of those paths.
- Secondly, a much wider range of queries becomes available, because the structure of the result can be more complex. For example, in the persons example used in the introduction, we may wish to provide a 'tour of generations', creating one node for each generation, with 'next' links in between and 'has as member' links to persons. Queries such as these can easily be expressed in GOLD. Node creation allows the user to keep the original hypertext separated from the virtual structure defined on it for navigational purposes; this can do much to improve clarity.[13]
- Lastly, node creation can be essential in expressing intermediate results of a complex query. Some queries, even some that do not yield any new nodes, can only be expressed in GOLD using intermediate nodes. Apart from that, intermediate nodes are simply convenient in many situations. In effect, they bring the power of non-binary relations and nested values within a purely graph-based model. When working in such a model, this is an elegant way to obtain a powerful querying vehicle.

Like GraphLog and GOLD, the visual query language in [13] expresses queries as graphs that effectively

[12] GraphLog makes up for this to some extent by means of the *blob*, a one-to-many edge, and by the facility to contract existing nodes.

[13] Apart from that, if obeyed strictly, this rule has implementational advantages - it avoids having to put virtual link anchors inside existing nodes.

act as patterns on the instance. Its queries express selections, and result nodes are created to point out the results of queries. However, we prefer a direct specification of instance transformations as occurring in GOLD. Besides, the language has some limitations that make it weaker than GOLD.

4.5 Some sugaring is necessary

The power of GOLD is not obtained at the expense of simplicity. We have stressed that GOLD is a language with a very simple syntax and, for most expressions, a straightforward semantics. This does not necessarily mean that all GOLD programs are easy to understand. In fact, the language as presented here is a little oversimplified, something we did to bring out the fundamental properties without being distracted by non-essential features. The language will only be truly simple if queries in similar languages, such as GraphLog and Gram, look at least as easy when expressed in GOLD.

This is not always the case, basically for two reasons. One is the compactness of the GOLD operation, stemming from the fact that we allow non-injective embeddings, we do not create nodes that already exist, and we perform merging and sharing of new nodes, all in one operation. The interaction of these features is not very complicated for simple operations, but it is possible to write operations in which they do interact in a complicated way (cf. the marriage example, figure 5 on page 5. Especially the first two features are hardly used in practice. For this reason, we are considering to introduce injective embeddings and forced node addition, maybe by means of extra syntactic marking.

The second reason why GOLD as presented here can be awkward to use is that some fairly trivial manipulations take long sequences of operations, which makes it hard to catch the general intention of programs at first glance.[14] Typical tasks are forcing injective embeddings, forcing addition of nodes in the presence of similar ones, and simulating negation in patterns; these are all straightforward to simulate with few operations, but the resulting programs look pretty obscure. To a lesser extent, this also holds for path walking operations, which require iterations - unfortunately, not all iterations are as concise as the ancestor computation in figure 6, page 6. Some of this is remedied by marking nodes as suggested in the previous paragraph; negation and path walking are best supported by allowing in operations regular expressions with negation to be used in the place of plain edge labels. GraphLog and textual languages [4, 9] also have this feature.

Another insufficiency is in our lack of support for plain values, such as integers or strings. Practice demands at least the presence of a simple way to indicate (integer or string) attribute values to nodes, and to indicate common comparisons and computations on them in operations. A remedy is to allow the labeling of instance nodes with attribute fields, and the labeling of pattern nodes with expressions over attribute fields.

These extensions do not affect the observations on expressive power made in section 3. The suggestion to make node addition forced does not affect the range of manipulations that can be performed; neither do the other extensions, which all concern pattern matching, not the subsequent manipulation.

It is easy to come with new extensions and formally define them; the difficulty lies in finding visualizations that keep expressions easy to use. In particular, expressions that employ few features must remain simple enough to be immediately clear for the novice user.

4.6 GOLD as compared to GOOD

In the two previous sections, we compared GOLD to similar languages, and proposed some extensions to improve its practicality. The GOOD model and language [20] did not occur in that discussion because it is like GOLD in every aspect mentioned. We will now summarize the differences between the two languages.

[14] The proofs in section A.5 are ample evidence.

In GOLD, GOOD's operations are generalized to one general operation. Thanks to the greatly relaxed syntax that results, users can directly express many queries which take several operations in GOOD. In the previous section, other ways of achieving this were mentioned; similar extensions are actually implemented in the existing GOOD implementation [18], which is based on the syntax extensions and sugarings defined in the report [6].[15] The *core pattern*, which allows users to control the way in which nodes are added, further extends the range of directly expressible queries. We expect this will greatly improve ease of use. While this practical benefit was the main reason for their introduction, core patterns also remove the need for the abstraction operator. Furthermore, the possibility of adding multiple, interconnected nodes at once lifts expressive power to above that of GOOD and other constructive languages. More details are given in the appendix, section A.5.

Finally, there are minor differences in the data model. GOOD distinguishes between printable and nonprintable nodes, without providing any means of dealing with printable values. It also distinguishes between nonfunctional and functional edges; we may include the specification of this constraint and others in some version of GOLD.

Further, GOLD has no notion of database scheme;[16] new node and edge labels can be added on the fly.

In conclusion, GOLD is more compact and flexible than GOOD, while maintaining its benefits. We have taken care to design the language as a superset of GOOD. With the extensions mentioned above, it will largely be a superset of other languages as well, GraphLog in particular.

4.7 A system based on GOLD

Obviously, GOLD can only be useful with a smart graphical interface. The graphical representations of database instances and programs used in this report simply run off the screen for nontrivial examples. The interface will rely on techniques to selectively display instances and programs, and to aid in the composition of queries. Also important is a smooth integration between querying and navigation. The GOOD implementation (cf. [18]) proves that such an interface is practically feasible.

In conclusion, we regard GOLD as a solid basis for a graphical query interface. In comparison to similar languages, its power, compactness and flexibility are substantial improvements.

Acknowledgement

The work described in this report has greatly benefited from discussions with Jan Paredaens, Marc Andries, and Jan Hidders.

[15] Our approach differs from the approach taken there, in that we define our constructs not as *macros* (shorthands for GOOD programs), but instead provide a direct definition for their interpretation. Besides, our constructs are different - our single operation has many more syntactic possibilities, and the core pattern is completely new.

[16] A *scheme for an instance* $\mathcal{I}$ can be defined as an instance in which no two nodes carry the same label and upon which $\mathcal{I}$ has an embedding. A scheme is an important practical tool in working with a database, even more so if we extend it with graphical notations to express certain constraints on instances. Like the discussion of other tools, this is outside the scope of the present report.

A Proofs of theorems

In this appendix, we prove some of the theorems appearing in the report. In section A.1, the correctness of GOLD is established, that is, the property that GOLD programs express only determinate database transformations.

In further sections, several theorems are proved from section 3, concerning the expressive power of sublanguages of GOLD.

A.1 Correctness of GOLD

GOLD is designed as an imperative database programming language. We use a well-known correctness criterion for such languages, known as *determinacy*: the property that programs are deterministic up to the choice of new node identities.

This appendix discusses this notion in some detail, and proves that GOLD is indeed determinate (theorem 2.1). In the first section, we discuss some subtleties concerning the definition of determinacy; in the second section, we prove determinacy of single operations; in the third section, we generalize this result to programs.

A.1.1 Determinacy: defining correctness of operations

In GOLD, the identities of nodes are inaccessible to the user. In the pictorial representation of instances (for example, figure 1), node identities are not given explicitly; to be precise, the picture represents some instance out of an isomorphism class, without specifying which one exactly. Likewise, the language cannot use node identifiers directly; all it has is patterns, which means that node identities can be used only in equality tests.

This property of database languages is known as *genericity*; it arose from the desire to describe databases as structural descriptions of objects that may have an existence apart from this description. For instance, if GOLD is used as a hypertext language, and nodes are hypertext nodes, then these hypertext nodes exist apart from their status as labeled points in a GOLD database instance: they have contents, appearance, and possibly a certain behavior, not all of which is captured in the instance description, and they can be accessed and modified by other programs than the GOLD query processor alone.

Retaining genericity, we want database transformations to be deterministic, up to the choice of new node identities.

This is the property known as *determinacy*, defined in definition 2.3 on page 8.

Note that determinacy places an effective restriction on the possible instances resulting from a given instance. This can be seen by taking, in the definition, $\mathcal{I}_1 = \mathcal{I}_2$ and $\mathcal{I}_3 = \mathcal{I}_4$. Hence, the following definition is meaningful.

Definition A.1 An *instance transformation* is a pair of instances that is part of a database transformation.
□

For example, if $\mathcal{I}$ is an instance consisting of two nodes labeled A, $\mathcal{I}'$ is an instance consisting of one of these, and $\mathcal{I}''$ is an instance consisting of one isolated A-node not in $\mathcal{I}$, then $(\mathcal{I}, \mathcal{I}'')$ is an instance transformation, whereas $(\mathcal{I}, \mathcal{I}')$ is not.

These examples also serve to point out that determinacy is a stronger notion than determinism up to isomorphism, a notion which could be defined as follows.

Definition A.2 Let S be a set of labels, let U be the set of instances over S, and let R be a relation on U. Then $R^{\cong}$, pronounced R *modulo isomorphism*, is the relation on the isomorphism classes of U, given by

$$[\mathcal{I}]_{\cong}\ R^{\cong}\ [\mathcal{I}']_{\cong} \underset{D}{=} \mathcal{I}\ R\ \mathcal{I}'$$

for all instances $\mathcal{I}, \mathcal{I}' \in U$.

R is said to be *deterministic modulo isomorphism* if $R^{\cong}$ is a partial function.
□

We will state without proof the following fact.

Proposition A.1 All determinate relations are deterministic up to isomorphism. The reverse does not hold in general; it does hold, however, for all relations that relate only instances with disjoint sets of nodes.
□

A.1.2 Operations are determinate

Lemma A.2 Every GOLD operation, considered on instances over labels in a finite set S, expresses a database transformation over S.
□

As explained in the text, this means that we have an effective restriction on the range of possible instance transformations. To be precise, we have

Corollary A.1 With the semantics defined in the text, it holds that between any two instances that may result from the same operation applied to the same source instance, there is an isomorphism that fixes all nodes remaining from the original instance.
□

In this section, we will prove lemma A.2 in detail.

Definition 2.2 on page 8 assigns a semantics to all GOLD operations: for all operations t, it defines as its effect a relation on the universe of instances, by defining for all pair of instances $\mathcal{I}, \mathcal{I}'$ whether or not $\mathcal{I} \overset{t}{\Rightarrow} \mathcal{I}'$ holds. It must be proved that, for all t, when this relation is restricted to all instances over a finite set of labels S, it is computable and determinate.

As to computability, consider the following program. Its inputs are an arbitrary GOLD operation, $t = \langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$, and an arbitrary GOLD instance, $\mathcal{I}$; its output is an instance, $\mathcal{I}'$.

```
F := ∅; I' := I;
for all embeddings f of J_S in I
do
  if f cannot be extended to an embedding of J_A in I
  then add f to F
```

```
  fi
od
(* F is as in definition 2.2 *)
for all f in F
do
  if some previous f' in F is equal to f on J_C
  then
    N_f := N_f';
    f := f' on all nodes not in I
  else
    N_f := for all nodes m in N(J_A) \ N(J_S) a copy, c(m)
  fi;
  g := f∪c;
  for all nodes m in N_f
  do
    for all edges ⟨n_1, α, n_2⟩ where m = n_1 or m = n_2
    do
      if not present, add to I' the edge ⟨g(n_1), α, g(n_2)⟩
    od
od
for all embeddings f of J_S in I
  if for some previous f' in F, between the extensions spanned by
       N_f and N_f' with incident edges and nodes
       there is an isomorphism that fixes I
  then
    remove from I'' N_f with all incident edges
    set f equal to f' on all new nodes
  fi
od
  (* each embedding f has now been modified to η(f)
    for an η such that the definition is met *)
  (* I' now contains the addition instance, I'' *)
for all embeddings f of J_S in I
do
  remove all deleted nodes with incident edges and all deleted edges
od
```

The running time of this program is best split in two: the time required to find the embeddings, and the time spent on rest of the computation. The latter is linearly dependent on the number of embeddings found; the former, in a naive approach, requires $\mathcal{O}(n^m)$, where n is the number of instance nodes and m is the number of pattern nodes. Hence, the algorithm is polynomial in the instance size. Obviously, n^m is unacceptable in practice, but we will usually have dense instances, small and dense patterns, and a number of embeddings roughly proportional to the size of the instance, rather than a higher polynomial. With existing fast pattern matching techniques, we may be able to have linear running time or better. [17]

Lemma A.3 This program always yields an instance $\mathcal{I}'$; for that instance, $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$ holds.
□

Proof The lemma obviously holds if we may assume that before the last loop, $\mathcal{I}'$ always contains the addition instance $\mathcal{I}'' = \mathcal{I} \cup \mathcal{I}'$ for some $\mathcal{I}'$ for which $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$.

To prove that this is the case, it is sufficient to prove the claim that the embeddings f are modified to $\eta(f)$ for a certain choice of η such that the six criteria posed in the definition (def. 2.2, p. 8) are all met.

[17] Further study is in order before we can make any more detailed predictions about expected running time.

This will be left as an exercise to the reader.

□

The second step is to prove that operations are determinate. This will be done in two steps (lemmas A.4,A.5).

Lemma A.4 Given instances $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4$ with an isomorphism i from $\mathcal{I}_1$ to $\mathcal{I}_2$, a transformation t such that $\mathcal{I}_1 \stackrel{t}{\Rightarrow} \mathcal{I}_3$, and an isomorphism i' between $\mathcal{I}_3$ and $\mathcal{I}_4$ which agrees with i on the common nodes of $\mathcal{I}_1$ and $\mathcal{I}_3$ and the common nodes of $\mathcal{I}_2$ and $\mathcal{I}_4$, $\mathcal{I}_2 \stackrel{t}{\Rightarrow} \mathcal{I}_4$ holds.
□

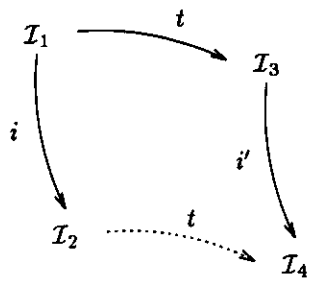


Figure 12: Going from isomorphism to transformation

Proof Assume such $\mathcal{I}_1\mathcal{I}_4$, t, i, and i'. Take F to be the set of embeddings of $\mathcal{J}_S$ in $\mathcal{I}_1$ that have no extension to an embedding of $\mathcal{J}_A$ in $\mathcal{I}_1$, and F' identically for $\mathcal{I}_2$. Take $\mathcal{I}''_1 = \mathcal{I}_1 \cup \mathcal{I}_3$ and $\mathcal{I}''_2 = \mathcal{I}_2 \cup \mathcal{I}_4$. Note that $F' = \{i \circ f \mid f \in F\}$, and that $i \cup i'$ is an isomorphism from $\mathcal{I}''_1$ to $\mathcal{I}''_2$.

Take an η that maps all f in F to embeddings of $\mathcal{J}_A$ to $\mathcal{I}''_1$ such that the six criteria of definition 2.2 are met. Take $\eta' = (i \cup i') \circ \eta \circ (i \cup i')^{-1}$. Then, η' maps all f in F to embeddings of $\mathcal{J}_A$ to $\mathcal{I}''_2$ such that the six criteria of the definition 2.2 are met. This is left for the reader to verify.

Since $\mathcal{I}''_2$ is the addition instance $\mathcal{I}_3 \cup \mathcal{I}_4$, where $\mathcal{I}_4$ is, as required, $\mathcal{I}''_2$ minus all deleted nodes and edges, $\mathcal{I}_2 \stackrel{t}{\Rightarrow} \mathcal{I}_4$ must hold.

□

Lemma A.5 Given instances $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4$ with an isomorphism i from $\mathcal{I}_1$ to $\mathcal{I}_2$, and a transformation t such that $\mathcal{I}_1 \stackrel{t}{\Rightarrow} \mathcal{I}_3$ and $\mathcal{I}_2 \stackrel{t}{\Rightarrow} \mathcal{I}_4$, there exists an isomorphism between $\mathcal{I}_3$ and $\mathcal{I}_4$ which agrees with i on the common nodes of $\mathcal{I}_1$ and $\mathcal{I}_3$ and the common nodes of $\mathcal{I}_2$ and $\mathcal{I}_4$.
□

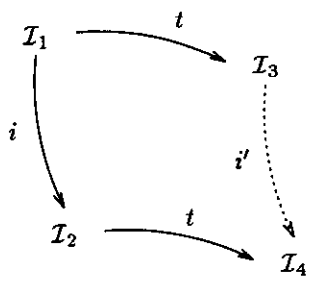


Figure 13: Going from transformation to isomorphism

This isomorphism can be found as follows:

- take i;
- delete all pairs involving nodes deleted by t;
- choose an embedding extension η from $\mathcal{I}_1$ to $\mathcal{I}_3$ and η' from $\mathcal{I}_2$ to $\mathcal{I}_4$;
- choose a set of source pattern embeddings f into $\mathcal{I}_1$ such that all new nodes of $\mathcal{I}_3$ are in the range of exactly one extension $\eta(f)$;
- for all these embeddings f and added pattern nodes m, add the pair $\langle \eta(f)(m), \eta'(i \circ f)(m) \rangle$ to the isomorphism.

It is quite straightforward to show that this construction is possible and does indeed yield an isomorphism i' which agrees with i on their common nodes.

This will now be described in more detail.

Assume four instances $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4$ and an operation t for which $\mathcal{I}_1 \stackrel{t}{\Rightarrow} \mathcal{I}_3$ and $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}_4$.

Take $\mathcal{I}''_1 = \mathcal{I}_1 \cup \mathcal{I}_3$ and $\mathcal{I}''_2 = \mathcal{I} \cup \mathcal{I}_4$.

Take F_1 and F_2 as the set of embeddings of $\mathcal{J}_S$ in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively, that can not be extended to embeddings of $\mathcal{J}_A$ in the same instance.

Let $\sim$ be defined on F_1 and on F_2 as the relation

$$f_1 \sim f_2 \underset{D}{\iff} f_1 \text{ restricted to } N(\mathcal{J}_C) = f_2 \text{ restricted to } N(\mathcal{J}_C) \tag{1}$$

Note that $F_2 = \{i \circ f \mid f \in F_1\}$ and that for all $f_1, f_2 \in F$, $f_1 \sim f_2$ iff $i \circ f_1 \sim i \circ f_2$.

Consider functions $\eta_1 : F \to Emb(\mathcal{J}_A, \mathcal{I}''_1)$ and $\eta_2 : F \to Emb(\mathcal{J}_A, \mathcal{I}''_2)$ for which, mutatis mutandis, the conditions posed in the definition hold.

For all $f \in F$, $\eta_1(f)$ and $\eta_2(i \circ f)$ are equal on the nodes of $N(\mathcal{J}_S)$, and injective on the nodes in $N(\mathcal{J}_A) \backslash N(\mathcal{J}_S)$; hence, there is a bijection from the range of $\eta_1(f)$ to that of $\eta_2(f)$ that maps, for all $m \in N(\mathcal{J}_A)$, $\eta_1(f)(m)$ to $\eta_1(f)(m)$. This bijection will be denoted as h_f.

Lemma A.6 For every $f \in F$, h_f is an embedding from $\mathcal{I}''_1$ restricted to the range of $\eta_1(f)$ to $\mathcal{I}''_2$ restricted to the range of $\eta_2(i \circ f)$.
□

Proof For all $f \in F$, the following holds.

Since $\eta_1(f)$ and $\eta_2(f)$ are both embeddings, all nodes n in the range of $\eta_1(f)$ have the same node label as $h_f(n)$.

For an arbitrary edge of $\mathcal{I}''_1$, $\langle n_1, \alpha, n_2 \rangle$ with n_1 and n_2 in the range of $\eta_1(f)$, if it is an edge of $\mathcal{I}_1$, then $\langle i(n_1), \alpha, i(n_2) \rangle$ is in $\mathcal{I}''_2$ which is a superinstance of $\mathcal{I}_2$; if it is not in $\mathcal{I}_1$, then, by condition 6, there must be some $f' \in F_1, m_1, m_2 \in N(\mathcal{J}_A)$ such that $\langle m_1, \alpha, m_2 \rangle \in E(\mathcal{J}_A)$ and $\eta_1(f')(m_1) = n_1, \eta_1(f')(m_2) = n_2$.

If $m_1, m_2 \in N(\mathcal{I})$, then, by condition 1, $\eta_1(f')(m_1) = n_1 = h_f(n_1)$ and $\eta_1(f')(m_2) = n_2 = h_f(n_2)$, so $\langle h_f(n_1), \alpha, h_f(n_2) \rangle$ is an edge of $\mathcal{I}''_2$; if not, then, by 5, w.l.o.g. it may be assumed that $f \sim f'$, and hence, for both n_1 and n_2, $\eta_2(i \circ f')(n_i) = h_f(n_i)$, which follows from 1 if $n_i \in N(\mathcal{J}_S)$, and from 2 otherwise.

Consequently, $\langle h_f(n_1), \alpha, h_f(n_2) \rangle = \langle \eta_2(i \circ f')(m_1), \alpha, \eta_2(i \circ f')(m_2) \rangle$ is an edge of $\mathcal{I}''_2$.

□

Lemma A.7 For all $f \in F$, h_f^{-1} is an embedding.
□

Proof Reversing the roles of $\mathcal{I}''_1$ and $\mathcal{I}''_2$ in the previous definitions, another bijection h'_f is obtained that is easily seen to be equal to h_f^{-1} for every $f \in F$. Apply lemma A.6, mutatis mutandis, to h'_f.

□

Consider a maximal set T of embeddings $f \in F$ that contains only embeddings with disjoint ranges:

$$\forall f \in F : \exists! f' \in T : \eta_1(f)(N(\mathcal{J}_A)) \cap \eta_1(f')(N(\mathcal{J}_A)) \not\subseteq N(\mathcal{J}_S) \tag{2}$$

Since $\sim$ is an equivalence relation, such a T exists.

Lemma A.8 For all different $f_1, f_2 \in T$, no new node is in the domain of both h_{f_1} and h_{f_2}.
□

Proof An immediate consequence of the definition.

□

Lemma A.9 For all different $f_1, f_2 \in T$, no new node is in the range of both h_{f_1} and h_{f_2}.
□

Proof Again, reverse the roles of $\mathcal{I}''_1$ and $\mathcal{I}''_2$, to show that the previous lemma applies to h_f^{-1}.

□
Combining these lemmas, and the fact that all h_f fix the old nodes, it follows that there exists a bijection h from a subset of $N(\mathcal{I}''_1)$ to a subset of $N(\mathcal{I}''_2)$, defined by

$$h \underset{D}{=} \bigcup_{f \in T} h_f$$

Let i' be the bijection defined as

$$i' \underset{D}{=} h \cup id_{N(\mathcal{I}_1)}$$

where $id_{N(\mathcal{I}_1)}$ is the identity function on $N(\mathcal{I}_1)$. Then:

Lemma A.10 i' is an isomorphism from $\mathcal{I}''_1$ to $\mathcal{I}''_2$ that on $N(\mathcal{I}_1)$ is equal to i, and whose inverse is equal to i^1 on $N(\mathcal{I}_2)$.
□

Proof By conditions 6, 2, and 3, all new nodes of $N(\mathcal{I}''_1)$ must be in the domain of h_f for some $f \in T$; similarly, all new nodes of $N(\mathcal{I}''_2)$ must be in the domain of h_f^{-1} for some $f \in T$. By 6, 2, 3, and 5, all new edges in $\mathcal{I}''_1$ must run between two nodes in the same h_f for some $f \in T$; similarly, all new edges in $\mathcal{I}''_2$ run between two nodes in the range of the same h_f for some $f \in T$. By lemmas A.6–A.9, h is a bijection which is an embedding in both directions. By construction of the h_f, h agrees with i on all nodes of $\mathcal{I}_1$ and $\mathcal{I}_2$; the lemma follows.

□

We can now prove lemma A.5.

Proof Consider $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4, t, \mathcal{I}''_1, \mathcal{I}''_2$, and h, as in lemma A.10. Take

$$D = \{f(m) \mid f \in Emb(\mathcal{J}_s, \mathcal{I}_2) \wedge m \in N(\mathcal{J}_s) \backslash N(\mathcal{J}_D)\}$$

Since h is the identity on D, from lemma A.10 it follows that h restricted to the nodes not in D is an isomorphism between $\mathcal{I}_3$ and $\mathcal{I}_4$ that fixes all original nodes.

□
From lemmas A.3, A.4, and A.5, the main lemma, A.2, immediately follows.

A.1.3 Correctness of programs

We must now generalize these results to programs, and obtain theorem A.13.

From lemma A.3, we know that all GOLD operations yield results on all instances. For programs in general, this is not the case. This is due to the unbounded iteration present in the language. It is well known that languages need this possibility of infinite looping in order to achieve Turing-completeness; since we want GOLD, a general graph manipulation language, to be Turing-complete, restricting the absence of results to the case of unbounded looping is in a sense the best we can do.

Definition A.3 Let p be a program and n be a number. The *expansion* of p up to n, denoted $ex_n(p)$, is inductively defined to be the following program:

- Λ, if $n = 0$ or p can be written as Λ, using the given laws of equivalence (cf. equations 1 on page 10) if necessary;
- $t; ex_{n-1}(q)$, if p can be written as $t; q$ again using the given laws of equivalence as necessary.

□

Note that all programs fall in exactly one category; consequently, every program has exactly one expansion up to n for each n. Now we can state, without proof, an obvious property of iteration:

Lemma A.11 For all instances $\mathcal{I}, \mathcal{I}'$, and programs p, if $\mathcal{I} \stackrel{p}{\Rightarrow} \mathcal{I}'$, then there is an $n \in \mathbb{N}$ such that $\mathcal{I} \stackrel{ex_n(p)}{\Rightarrow} \mathcal{I}'$.
□

Equally obvious is the following:

Lemma A.12 All GOLD programs express computable relations.
□

To prove theorem 2.1, we further need to establish

Lemma A.13 All GOLD programs express determinate relations.
□

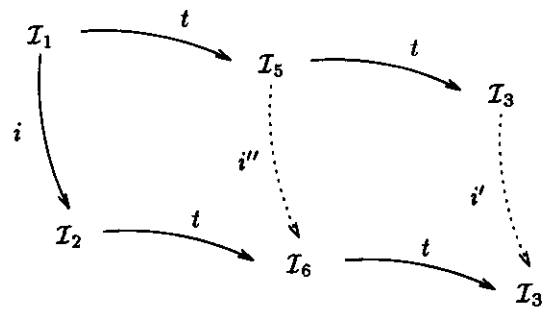


Figure 14: Going from program to isomorphism

Proof The single operation is determinate by lemma A.2. From lemma A.11, it follows that we only need to consider programs without iteration. All such programs are all determinate: trivially so if their length is 0 or 1. If their length is $n > 1$, they can be written as $p; q$, where p and q are both shorter than n.

Now consider instances $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4$ with an isomorphism i from $\mathcal{I}_1$ to $\mathcal{I}_2$ and such that $\mathcal{I}_1 \overset{p;q}{\Rightarrow} \mathcal{I}_3$. We are to prove that $\mathcal{I}_2 \overset{p;q}{\Rightarrow} \mathcal{I}_4$ iff there is an isomorphism i' from $\mathcal{I}_3$ to $\mathcal{I}_4$ which agrees with i on their common nodes. Let X be the set of nodes common to $\mathcal{I}_1$ and $\mathcal{I}_3$. From definition 2.7, we observe that $\mathcal{I}_1 \overset{p;q}{\Rightarrow} \mathcal{I}_3$ holds iff there is an instance $\mathcal{I}_5$ such that $\mathcal{I}_1 \overset{p}{\underset{X}{\Rightarrow}} \mathcal{I}_5$ and $\mathcal{I}_5 \overset{q}{\underset{X}{\Rightarrow}} \mathcal{I}_3$. For this $\mathcal{I}_5$, we have $\mathcal{I}_1 \overset{p}{\Rightarrow} \mathcal{I}_5$, $\mathcal{I}_5 \overset{q}{\Rightarrow} \mathcal{I}_3$, and $X \subseteq N(\mathcal{I}_5)$.

If $\mathcal{I}_2 \overset{p;q}{\Rightarrow} \mathcal{I}_4$, there is an analogous instance $\mathcal{I}_6$ between $\mathcal{I}_2$ and $\mathcal{I}_6$. Assuming that p and q are determinate, we can construct an isomorphism from $\mathcal{I}_5$ to $\mathcal{I}_6$ that agrees with i on their common nodes, and from that isomorphism, an isomorphism i' that agrees with it, and, thanks to the fact that $X \subseteq N(\mathcal{I}_5)$, agrees with i.

On the other hand, if there is an isomorphism i' from $\mathcal{I}_3$ to $\mathcal{I}_4$ which agrees with i on all common nodes, then it is easy to construct an $\mathcal{I}_6$ such that there is a suitable isomorphism from $\mathcal{I}_5$ to $\mathcal{I}_6$.

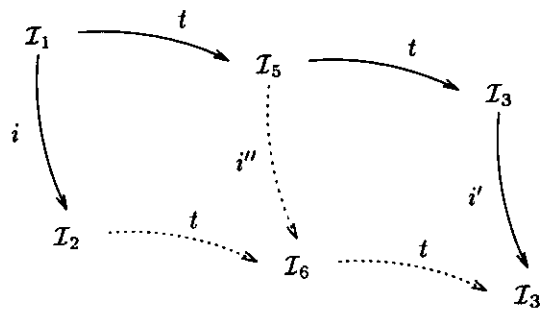


Figure 15: Going from isomorphism to program

□

From lemmas A.12, A.13, theorem 2.1 follows immediately.

A.2 A database transformation not expressible in GOLD

In this section, we prove lemma 3.2: there is a database transformation which is not expressible in GOLD.

Proof Consider an infinite sequence of nodes $a_1, a_2, \ldots$, all labeled A, two nodes b_1, b_2 labeled B, an infinite sequence of nodes $c_1, c_2, \ldots$, all labeled C, and an edge label α. For all $n \in \mathbb{N}$, define the instances

$$\begin{aligned}
\mathcal{I}_n \underset{D}{=} & \langle \{b_1, b_2, a_1, \ldots, a_n\}, \\
& \{\langle a_i, \alpha, a_{i+1}\rangle \mid 1 \leq i < 2n\} \\
& \rangle \\
\mathcal{I}'_n \underset{D}{=} & \langle \{b_1, b_2, c_1, \ldots, c_n\}, \\
& \{\langle c_i, \alpha, c_{(i \bmod 2n)+1}\rangle \mid 1 \leq i < 2n\} \cup \{\langle c_i, \alpha, b_{(i \bmod 2)+1}\rangle \mid 1 \leq i < 2n\} \\
& \rangle
\end{aligned}$$

For an illustration, see figure 16. Note that there exists a database transformation, say R, which contains

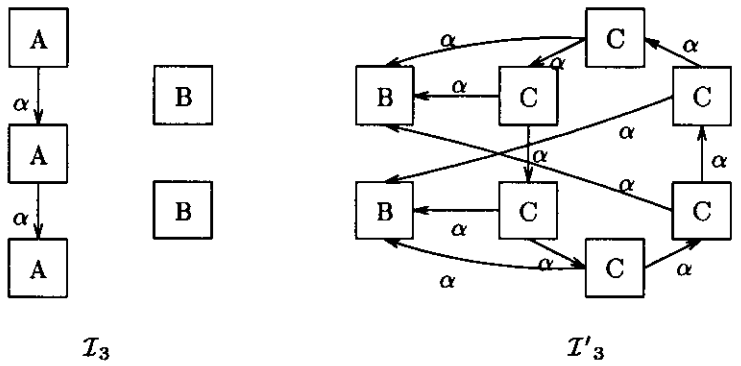


Figure 16: Instances $\mathcal{I}_3$ and $\mathcal{I}'_3$ constructed in the non-completeness proof.

$\langle \mathcal{I}_n, \mathcal{I}'_n \rangle$ for all $n \in \mathbb{N}$. Suppose that any such transformation is expressed by some GOLD program, p. Then let k be some prime number greater than the largest number of added nodes in all operations used in p. Supposedly, $\mathcal{I}_k \overset{p}{\Rightarrow} \mathcal{I}'_k$. Some operation t of p must add one of the new nodes in $\mathcal{I}'_k$; we will show that either t or some previous operation must contain at least $2k$ added nodes, thereby contradicting the existence of p. This proves that R is not expressible in GOLD.

If $\mathcal{I}_k \overset{p}{\Rightarrow} \mathcal{I}'_k$, then also $\mathcal{I}_k \overset{q}{\Rightarrow} \mathcal{I}'_k$, where q is the stepwise iteration expansion $ex_p(l)$ (cf. definition A.3, page 29, for some suitable $l \in \mathbb{N}$.

The automorphisms on instances also define a permutation group on the subsets of their nodes: if $\mathcal{I}''$ is an instance with some automorphism a, a acts on sets of nodes in $\mathcal{I}''$ as $a(X) = \{a(x) \mid x \in X\}$.

Consider the instance properties P_1, P_2 defined as follows: for any instance $\mathcal{I}''$, $P_1(\mathcal{I}'')$ holds if the nodes b_1, b_2 are in $\mathcal{I}''$, and some automorphism of $\mathcal{I}''$ swaps b_1, b_2; for any instance $\mathcal{I}''$, $P_2(\mathcal{I}'')$ holds if $P_1(\mathcal{I}'')$ and there is a subset X of nodes in $\mathcal{I}''$, such that, if

$$\begin{aligned}
Y &\underset{D}{=} \{a(X) \mid a \text{ any automorphism of } \mathcal{I}''\} \\
Y_1 &\underset{D}{=} \{Z \in Y \mid b_1 \in Z\} \\
Y_2 &\underset{D}{=} Y \setminus Y_1
\end{aligned}$$

then both Y_1 and Y_2 have k elements, and all b_1, b_2 swapping automorphisms of $\mathcal{I}''$ fully cycle the elements of Y. (Note that all $X \in Y_1$ contain b_1, while all $X \in Y_2$ contain b_2.)

We will prove that for all instances $\mathcal{I}_1, \mathcal{I}_2$ and transformations t, if we have $P_2(\mathcal{I}_2)$ but only $P_1(\mathcal{I}_1)$, and $\mathcal{I} \overset{t}{\Rightarrow} \mathcal{I}'$, then t must have at least $2k$ *added* nodes. There is at least one such t in q, since $P_2(\mathcal{I}_2)$ holds (take $X = \{b_1, c\}$, where c is a new node), but $P_2(\mathcal{I}_1)$ does not hold; further, $P_1(\mathcal{I}_1)$ and $P_1(\mathcal{I}_2)$ hold, hence P_1 also holds for all intermediate instances.

Consider instances $\mathcal{I}_1, \mathcal{I}_2$ and transformations t, such that $P_2(\mathcal{I}_2)$ and $P_1(\mathcal{I}_1)$ hold, but not $P_2(\mathcal{I}_1)$; further, $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$.

$P_2(\mathcal{I}_2)$ holds, so we may assume X, Y, Y_1, Y_2 as defined above.

Suppose that one of the sets in Y consists of nodes in $\mathcal{I}_1$. $P_2(\mathcal{I}_2)$ holds, so some b_1, b_2 swapping automorphism a_2 of $\mathcal{I}_2$ cycles all elements of Y; theorem 2.1 implies that there is an automorphism a_1 that is equal to a_2 on their common nodes. This implies that all elements of Y are subsets of nodes in $\mathcal{I}_1$, and $P_2(\mathcal{I}_1)$ holds, contradicting our initial assumption.

Therefore, some $X \in Y$ contains nodes not in $\mathcal{I}_1$; hence, t must be a node addition; hence, $\mathcal{I}_1$ must be a subinstance of $\mathcal{I}_2$, and all automorphisms of $\mathcal{I}_1$ are automorphisms of $\mathcal{I}_2$ restricted to the nodes of $\mathcal{I}_1$.

Let Y_{new} be the set consisting of all $X \in Y$ restricted to the new nodes in $\mathcal{I}_2$; let Y_{old} be the set consisting of all $X \in Y$ restricted to nodes of $\mathcal{I}_2$ also in $\mathcal{I}_1$.

Consider the set $\mathcal{A}$ of automorphisms of $\mathcal{I}_2$ that swap b_1, b_2. They all cycle Y with period $2k$, mapping new nodes only to new nodes, old nodes only to old nodes. Since, by assumption, $P_2(\mathcal{I}_1)$ does not hold, Y_{old} must be cycled with period 2. Hence, Y_{new} must be cycled with period $2k$. Hence, there must be at least $2k$ new nodes.

How does t add these nodes? Choose an embedding extension η which meets the criteria imposed in the definition of operation semantics (definition 2.2). All new nodes are in the range of some $\eta(f)$ for some, possibly the same, source pattern embedding f. Pick such an embedding f. The range of f (a set of nodes in $\mathcal{I}_1$) is cycled by the automorphisms in $\mathcal{A}$. None of these automorphisms can cycle any subset of the range of f with period $2k$, for this would allow us to prove that $P_2(\mathcal{I}_1)$ holds, contradicting the assumption that it doesn't. As a consequence, all new nodes must be in the ranges of the extension $\eta(f)$ for the same source pattern embedding f. Therefore, t must contain at least $2k$ *added* nodes.

□

A.3 Every instance transformation is expressible in GOLD

In this section, we prove lemma 3.3: every (determinate) instance transformation is expressible in GOLD.

Proof Consider an instance transformation $\langle \mathcal{I}, \mathcal{I}' \rangle$. Consider an edge label α not used in $\mathcal{I}$ or $\mathcal{I}'$. Let $\mathcal{I}_\alpha$ be the instance equal to $\mathcal{I}$ plus additional edges $\langle n, \alpha, m \rangle$ for all pairs of different nodes n, m in $\mathcal{I}$. Let ${\mathcal{I}'}_\alpha$ be the instance equal to $\mathcal{I}$ plus additional edges $\langle n, \alpha, n \rangle$ for all nodes n in $\mathcal{I}$. Then, if

$$t_1 \underset{D}{=} \langle \mathcal{I}, \mathcal{I}, \mathcal{I}, \mathcal{I}_\alpha \rangle$$
$$t_2 \underset{D}{=} \langle \mathcal{I}, {\mathcal{I}'}_\alpha, {\mathcal{I}'}_\alpha, {\mathcal{I}'}_\alpha \rangle$$

we obviously have

$$\mathcal{I} \stackrel{t_1;t_2}{\Rightarrow} \mathcal{I}_\alpha$$

Note that the embeddings of $\mathcal{I}_\alpha$ onto itself are exactly the automorphisms of $\mathcal{I}$; the identity in particular is an embedding. If all nodes of $\mathcal{I}'$ are in $\mathcal{I}$, then, with

$$t_3 = \langle \mathcal{I}' \setminus \mathcal{I}, \mathcal{I}_\alpha, \mathcal{I}_\alpha, \mathcal{I}' \cup \mathcal{I}_\alpha \rangle$$

we have

$$\mathcal{I}_\alpha \stackrel{t_3}{\Rightarrow} \mathcal{I}'$$

If not, take a node label A not used in $\mathcal{I}$ or $\mathcal{I}'$, take an A-labeled node n, take

$$\mathcal{I}_1 = \langle \{n\}, \emptyset \rangle$$

$$
\begin{aligned}
\mathcal{I}_A &= \mathcal{I}' \cup \mathcal{I}_1 \\
\mathcal{I}_0 &= \langle \emptyset, \emptyset \rangle \\
t_3 &= \langle \mathcal{I}' \setminus \mathcal{I}, \mathcal{I}_\alpha, \mathcal{I}_\alpha, \mathcal{I}_A \rangle \\
t_4 &= \langle \mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_1, \mathcal{I}_1 \rangle
\end{aligned}
$$

and we have

$$\mathcal{I}_\alpha \overset{t_3;t_4}{\Rightarrow} \mathcal{I}'$$

This is obvious from the fact that the identity is a source pattern embedding which, due to the A-labeled node, cannot be extended to an addition pattern embedding into $\mathcal{I}$; this identity clearly has the identity on $\mathcal{I}_A$ as an injective extension. Because all embeddings are automorphisms and $\langle \mathcal{I}, \mathcal{I}' \rangle$ is an instance transformation, no other nodes than those of $\mathcal{I}_A$ are present in the result of t_3. This also guarantees that the deletion step of t_3 removes exactly those nodes and edges from $\mathcal{I}$ that lack in $\mathcal{I}'$. The result is $\mathcal{I}'$, or, in the second case, $\mathcal{I}$ with an extra A-labeled node; this node exactly is removed by t_4.

□

A.4 Simulating combined operations with separate ones

In this section, we prove lemma 3.2: GOLD and GOLD_{sep} are equally expressive.

Proof As all GOLD_{sep} programs are GOLD programs, all we need to do is prove that all GOLD programs can be simulated with sequences of node additions, node deletions, edge additions, and edge additions. Replace all non-GOLD_{sep} operations $t = \langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$ with a sequence of GOLD_{sep} operations t' constructed as follows.

For every edge label α used in a deleted edge, choose a new edge label, $\alpha-$. Chose one more new special edge label, $-$.

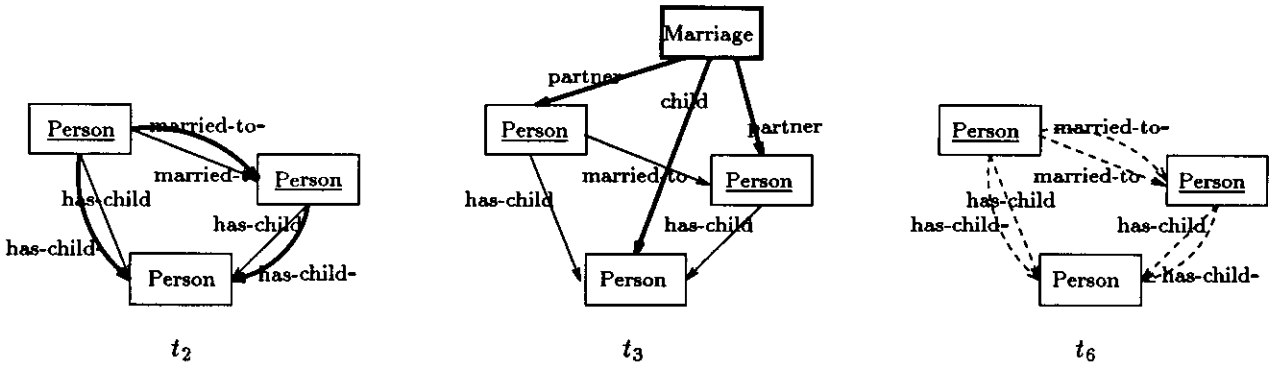
Let $\mathcal{J}_A{}'$ be the addition pattern restricted to the nodes of $\mathcal{J}_S$. For every edge label α used in an added edge of $\mathcal{J}_A{}'$, choose a new edge label, $\alpha+$. Let $\mathcal{J}_A{}''$ be $\mathcal{J}_A{}'$ where all added edges labeled α are replaced with edges labeled $\alpha+$. Let $\mathcal{J}_A{}'''$ be the union of $\mathcal{J}_A{}'$ and $\mathcal{J}_A{}''$, i. e., with both the α-edges and the $\alpha+$-edges. Let $\mathcal{J}_A{}''''$ be $\mathcal{J}_A$ without all added edges of $\mathcal{J}_A{}'$.

Let $\mathcal{J}_D{}'$ be $\mathcal{J}_S$ where all deleted nodes carry an extra edge to themselves, labeled $-$, and where for all deleted edges labeled α, a new edge is placed alongside labeled $\alpha-$. Let $\mathcal{J}_D{}''$ be the union of $\mathcal{J}_D{}'$ and $\mathcal{J}_A{}''$, i. e. it is $\mathcal{J}_S$ with all additional edges $\alpha+$, $\alpha-$, and $-$. Let $\mathcal{J}_D{}'''$ be $\mathcal{J}_S$ with all deleted edges removed that run between non-deleted nodes. Finally, let $\mathcal{J}_S{}'$ be $\mathcal{J}_D{}''$ from which all deleted nodes of $\mathcal{J}_S$ in t have been removed, and let $\mathcal{J}_D{}''''$ be $\mathcal{J}_S{}'$ from which all marking edges $\alpha+$, $\alpha-$, $-$ have been removed.

The sequence is $t' = t_1; t_2; t_3; t_4; t_5; t_6$, where

$$
\begin{aligned}
t_1 &= \langle \mathcal{J}_S, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A{}'' \rangle \\
t_2 &= \langle \mathcal{J}_S, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_D{}' \rangle \\
t_3 &= \langle \mathcal{J}_S, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A{}'''' \rangle \\
t_4 &= \langle \mathcal{J}_S, \mathcal{J}_C, \mathcal{J}_A{}'', \mathcal{J}_A{}''' \rangle \\
t_5 &= \langle \mathcal{J}_D{}''', \mathcal{J}_C, \mathcal{J}_D{}'', \mathcal{J}_D{}'' \rangle \\
t_6 &= \langle \mathcal{J}_D{}'''', \mathcal{J}_C, \mathcal{J}_S{}', \mathcal{J}_D{}'' \rangle
\end{aligned}
$$

Here, t_1 indicates the edges to be added; t_2 indicates the edges and nodes to be deleted; t_3 performs the node addition; t_4 adds the edges not added by t_3; t_5 deletes the nodes marked for deletion; t_6 deletes the edges marked for deletion, and removes all marking edges. An example is given in figure 17; in that example, the operations t_1, t_4 and t_5 are trivial and have been omitted. It is easy to see that t' behaves



t_2 t_3 t_6

Figure 17: Construction example: separated operations for the marriage example

identically to t on all instances in which none of the new labels $\alpha+$, $\alpha-$, $-$ occur. Therefore, every database transformation over some set of labels S can be simulated in this fashion, provided that the new labels are chosen outside of S.[18]

□

A.5 GOOD operations are subsumed in GOLD

In this section, we will show that, except abstraction, the operations of GOOD are those of GOLD_{tup}. This was stated in lemma 3.10.

Apart from notational differences between GOLD_{tup} and GOOD, there is one details that needs to be dealt with: the fact that edges in GOOD can be functional. To be precise, an edge label in GOOD is either functional or non-functional, and it is prohibited for instances to have a node from which two edges leave with the same functional edge label. As a result, edge additions in GOOD can produce results that are not instances because they violate this functionality criterion. In that case, the result is left undefined. GOLD_{tup} does define a result in such cases. A similar situation holds for node addition: in GOOD, it is required that the edges leaving the new node are functional edges. As a result, node addition in GOLD_{tup} applies to more cases than in GOOD: those cases in which an edge departing from the new node violates the functionality criterion in the instance to which the operation is applied.

When we say that the operations of GOLD_{tup} are identical to the operations in GOOD, we mean that they are identical on all cases in which the corresponding GOOD operation defines a result. The GOOD operations leave the result undefined exactly when the GOLD_{tup} operations produce an instance that violates the functionality criterion. This condition is easily checked with a GOLD_{tup} program.

An accessible source for the definition of GOOD is [20]. However, the definition given there lacks an iteration construct. More complete references are [6], [19]. We will not repeat these definitions, but substitute them in the following, more exact rewording of lemma 3.10.

Lemma A.14 $\mathcal{I}, \mathcal{I}'$ be instances and t an operation of GOLD_{tup}. Then

- if t is a node addition, let m be the added node, let K be the label of m, and let $\langle m, \alpha_j, m_j \rangle$ be its outgoing edges. Then, $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I}'$ is a minimal superinstance of $\mathcal{I}$ for which
  - for each embedding i of $\mathcal{J}_s$ in $\mathcal{I}$, there exists a K-labeled node in $\mathcal{I}'$ such that for all j, $\langle n, \alpha_j, i(m_j) \rangle$ is an edge in $\mathcal{I}'$;

[18] Unfortunately, it is impossible to have simulations without using at least one extra edge or node label.

- each edge in $\mathcal{I}'$ leaving a node of $\mathcal{I}$ is also an edge of $\mathcal{I}$;
- no two edges with identical labels leave the same new node in $\mathcal{I}'$.

- if t is an edge addition, let $\langle m_j, \alpha_j, m'_j \rangle$ be its added edges. Then, $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I}'$ is the minimal superinstance of $\mathcal{I}$ for which
  - for each embedding i of $\mathcal{J}_S$ in $\mathcal{I}$ and all j, $\langle i(m_j), \alpha_j, i(m'_j) \rangle$ are edges in $\mathcal{I}'$.
- if t is a node deletion, let m be the deleted node. Then, $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I}'$ is the maximal subinstance of $\mathcal{I}$ for which
  - for each embedding i of $\mathcal{J}_S$ in $\mathcal{I}$, $i(m)$ is not a node in $\mathcal{I}'$.
- if t is an edge deletion, let $\langle m_j, \alpha_j, m'_j \rangle$ be its deleted edges. Then, $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I}'$ is the maximal subinstance of $\mathcal{I}$ for which
  - for each embedding i of $\mathcal{J}_S$ in $\mathcal{I}$ and all j, $\langle i(m_j), \alpha_j, i(m'_j) \rangle$ are not edges in $\mathcal{I}'$.

□

These claims are straightforward to verify, and we will leave this to the reader. Note that in all cases except node additions, the embedding extension η required in definition 2.2 necessarily maps every embedding to itself.

A.6 Equivalence of core patterns and abstraction

In this section, we demonstrate the equal expressive power of the core pattern feature on one hand and the abstraction operation on the other, by providing simulations of one using the other.

A.6.1 Simulating core patterns with abstraction

We will now give a construction to replace (nontrivial usage of) core patterns with abstractions; an example of this construction is given in figure 18.

Let $t = \langle \mathcal{J}_D, \mathcal{J}_C, \mathcal{J}_S, \mathcal{J}_A \rangle$ be a node addition in GOLD_1. Let n be the added node of t; say it has label

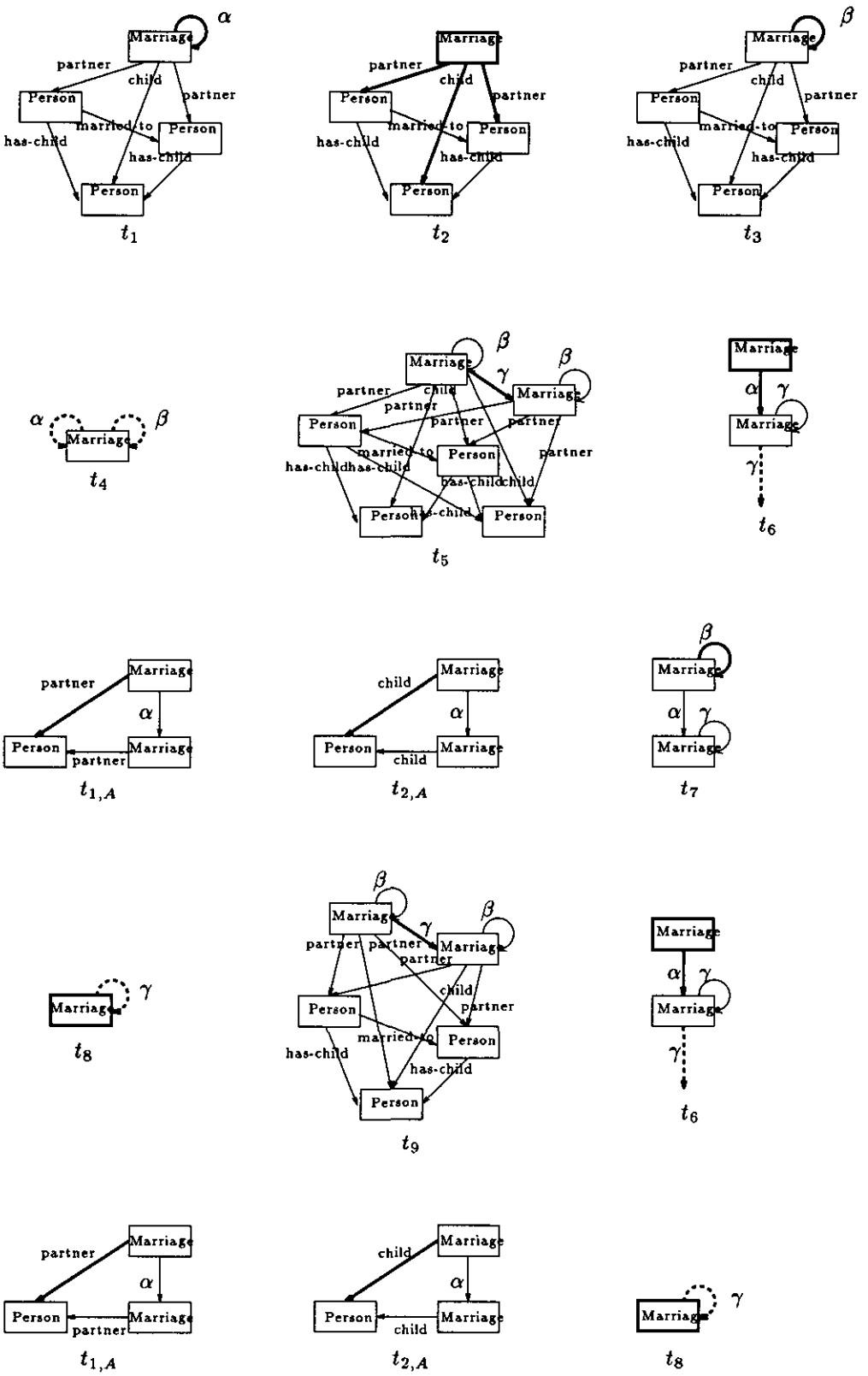


Figure 18: Construction example: simulating core patterns with abstraction for the marriage example

K. Let $\langle n, \alpha_i, m_i \rangle$ be the edges incident to n. Let $\alpha, \neg\alpha, \beta, \gamma$ be edge labels. Let

$$\begin{aligned}
\mathcal{J}_A' &= \mathcal{J}_A \text{ with an extra edge } \langle n, \alpha, n \rangle \\
\mathcal{J}_A'' &= \mathcal{J}_A \text{ with an extra edge } \langle n, \beta, n \rangle \\
\mathcal{J}_1 &= \langle \{n\}, \emptyset \rangle \\
\mathcal{J}_2 &= \langle \{n\}, \{\langle n, \alpha, n \rangle, \langle n, \beta, n \rangle\} \rangle \\
t_1 &= \langle \mathcal{J}_A, \mathcal{J}_A, \mathcal{J}_A, \mathcal{J}_A' \rangle \\
t_2 &= \langle \mathcal{J}_S, \mathcal{J}_S, \mathcal{J}_S, \mathcal{J}_A \rangle \\
t_3 &= \langle \mathcal{J}_A, \mathcal{J}_A, \mathcal{J}_A, \mathcal{J}_A'' \rangle \\
t_4 &= \langle \mathcal{J}_1, \mathcal{J}_2, \mathcal{J}_2, \mathcal{J}_2 \rangle
\end{aligned}$$

Let $\mathcal{J}_3$ be an instance of which both $\mathcal{J}_A''$ and some instance $\mathcal{J}_B$ are subinstances, such that some isomorphism from $\mathcal{J}_A'$ to $\mathcal{J}_B$ fixes all nodes of $\mathcal{J}_C$, $\mathcal{J}_A'$ and $\mathcal{J}_B$ have all different nodes except those in $\mathcal{J}_C$, and every edge of $\mathcal{J}_3$ is in either $\mathcal{J}_A'$ or $\mathcal{J}_B$. Let $\mathcal{J}_4$ be $\mathcal{J}_3$ with the additional edge $\langle n, \gamma, i(n) \rangle$. Let

$$\begin{aligned}
t_5 &= \langle \mathcal{J}_3, \mathcal{J}_3, \mathcal{J}_3, \mathcal{J}_4 \rangle \\
\mathcal{J}_5 &= \langle \{n\}, \{\langle n, \gamma, n \rangle\} \rangle
\end{aligned}$$

Let n' be some K-labeled node not equal to n. Define the abstraction

$$t_6 = \langle \mathcal{J}_5, n', K, \gamma, \alpha \rangle$$

For all edges $\langle n, \alpha_i, m_i \rangle$, define the patterns

$$\begin{aligned}
\mathcal{J}_{i,S} &= \langle \{n, m_i, n'\}, \{\langle n, \alpha_i, m_i \rangle\} \rangle \\
\mathcal{J}_{i,A} &= \mathcal{J}_{i,S} \text{ with the additional edge } \langle n', \alpha, m_i \rangle \\
t_{i,A} &= \langle \mathcal{J}_{i,S}, \mathcal{J}_{i,S}, \mathcal{J}_{i,S}, \mathcal{J}_{i,A} \rangle
\end{aligned}$$

Then let p_1 be a program consisting of all $t_{i,A}$, arbitrarily ordered into a sequence. Let n_1, n_2 be two K-labeled nodes, and

$$\begin{aligned}
\mathcal{J}_6 &= \langle \{n_1, n_2\}, \{\langle n_2, \alpha, n_1 \rangle, \langle n_1, \gamma, n_2 \rangle\} \rangle \\
\mathcal{J}_7 &= \mathcal{J}_6 \text{ with the additional edge } \langle n_2, \beta, n_2 \\
t_7 &= \langle \mathcal{J}_6, \mathcal{J}_7, \mathcal{J}_7, \mathcal{J}_7 \rangle \\
\mathcal{I}_0 &= \langle \emptyset, \emptyset \rangle \\
t_8 &= \langle \mathcal{J}_0, \mathcal{J}_5, \mathcal{J}_5, \mathcal{J}_5 \rangle
\end{aligned}$$

Let $\mathcal{J}_8$ be an instance of which both $\mathcal{J}_A''$ and some instance $\mathcal{J}_B$ are subinstances, such that some isomorphism i from $\mathcal{J}_A'$ to $\mathcal{J}_B$ fixes all nodes of $\mathcal{J}_A''$ except n, and no edges run between n and $i(n)$. Let

$$\begin{aligned}
\mathcal{J}_9 &= \mathcal{J}_8 \text{ with the additional edge } \langle n, \gamma, i(n) \rangle \\
t_9 &= \langle \mathcal{J}_8, \mathcal{J}_9, \mathcal{J}_9, \mathcal{J}_9 \rangle
\end{aligned}$$

If

$$p = t_1; t_2; t_3; t_4; t_5; t_6; p_1; t_7; t_8; t_9; t_6; p_1; t_8$$

then, for all instances $\mathcal{I}, \mathcal{I}'$ in which none of the edge labels $\alpha, \neg\alpha, \beta, \gamma$ occur, it holds that $\mathcal{I} \stackrel{t}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I} \stackrel{p}{\Rightarrow} \mathcal{I}'$. The proof is tedious and has been omitted.

A.6.2 Simulating abstraction with core patterns

An example of the following construction is provided in figure 19, which displays its results on the abstraction of figure 10.

Let $t = \langle \mathcal{J}, m, K, \alpha, \beta \rangle$ be an abstraction. Let X be the set of node labels C for which there is a node m' labeled C such that the edge $\langle m, \alpha, m' \rangle$ is in $\mathcal{J}$. Let B be the node label of m. Let $\alpha, \neg\alpha, \beta, \neg\beta, \gamma, \neg\gamma, \delta, \epsilon, \zeta$

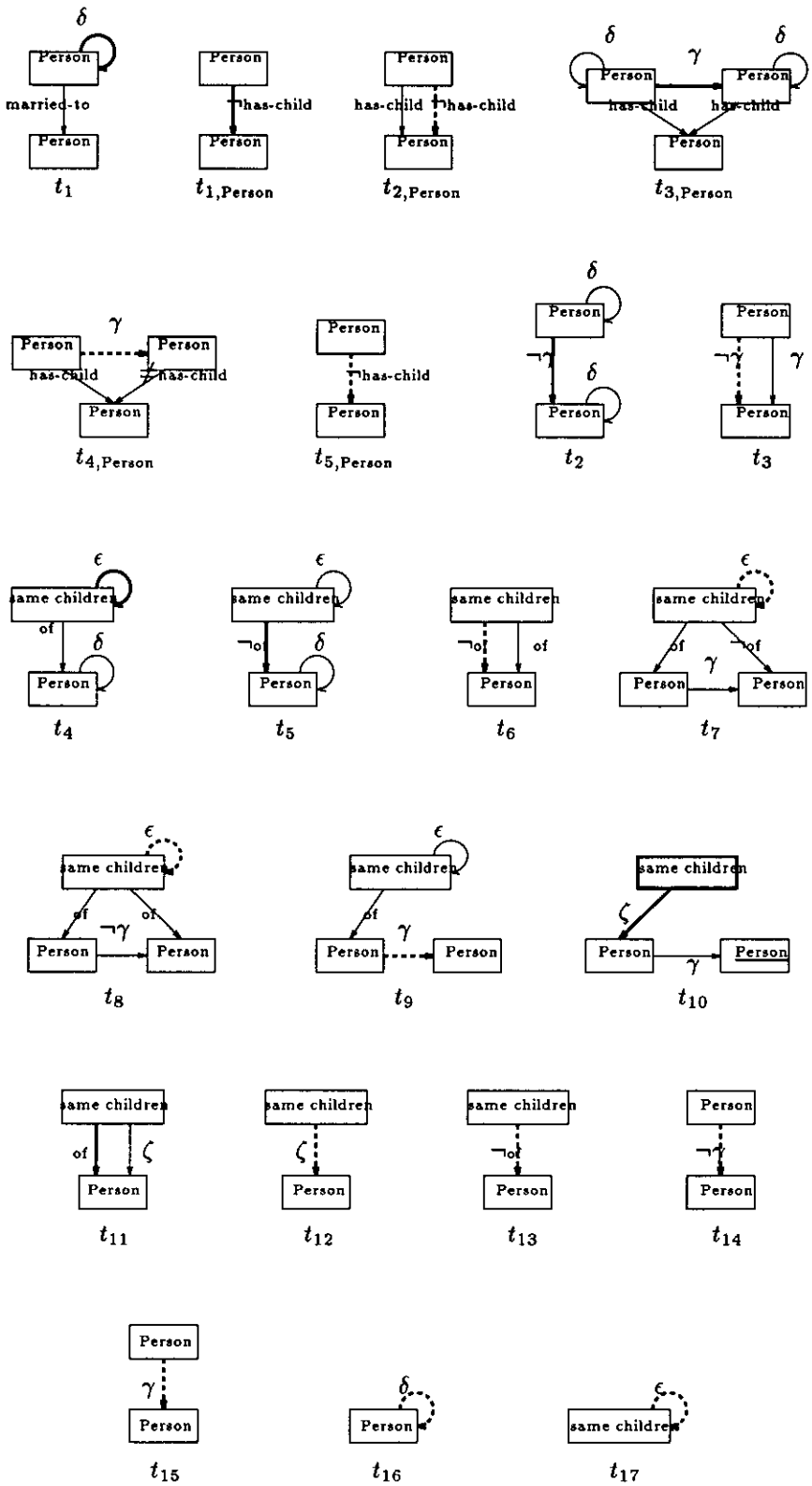


Figure 19: Example simulation of abstraction with core patterns

be edge labels. Let n be a new node labeled K.

$$\begin{aligned}
\mathcal{J}_1 &= \mathcal{J} \text{ with additional edge } \langle m, \delta, m\rangle \\
\mathcal{J}_2 &= \mathcal{J} \text{ with additional node } n \text{ and edge } \langle n, \beta, m\rangle \\
t_1 &= \langle \mathcal{J}, \mathcal{J}, \mathcal{J}, \mathcal{J}_1\rangle
\end{aligned}$$

Let b_1, b_2 be two nodes labeled B. For all C in X, let c be a node labeled C, and let p_C be a see $t_{1,C}; t_{2,C}; t_{3,C}; t_{4,C}$, where there transformations are as follows:

$$\begin{aligned}
\mathcal{J}_{1,C} &= \langle \{b_1, c\}, \emptyset\rangle \\
\mathcal{J}_{2,C} &= \langle \{b_1, c\}, \{\langle b_1, \alpha, c\rangle\}\rangle \\
\mathcal{J}_{3,C} &= \langle \{b_1, c\}, \{\langle b_1, \neg\alpha, c\rangle\}\rangle \\
\mathcal{J}_{4,C} &= \langle \{b_1, c\}, \{\langle b_1, \alpha, c\rangle, \langle b_1, \neg\alpha, c\rangle\}\rangle \\
t_{1,C} &= \langle \mathcal{J}_{1,C}, \mathcal{J}_{1,C}, \mathcal{J}_{1,C}, \mathcal{J}_{3,C}\rangle \\
t_{2,C} &= \langle \mathcal{J}_{2,C}, \mathcal{J}_{4,C}, \mathcal{J}_{4,C}, \mathcal{J}_{4,C}\rangle \\
\mathcal{J}_{5,C} &= \langle \{b_1, b_2, c\}, \{\langle b_1, \delta, b_1\rangle, \langle b_2, \delta, b_2\rangle, \langle b_1, \alpha, c\rangle, \langle b_2, \alpha, c\rangle\}\rangle \\
\mathcal{J}_{6,C} &= \mathcal{J}_5 \text{ with additional edge } \langle b_1, \gamma, b_2\rangle \\
t_{3,C} &= \langle \mathcal{J}_{5,C}, \mathcal{J}_{5,C}, \mathcal{J}_{5,C}, \mathcal{J}_{6,C}\rangle, \\
\mathcal{J}_{7,C} &= \langle \{b_1, b_2, c\}, \{\langle b_1, \alpha, c\rangle, \langle b_2, \neg\alpha, c\rangle\}\rangle \\
\mathcal{J}_{8,C} &= \mathcal{J}_5 \text{ with additional edge } \langle b_1, \gamma, b_2\rangle \\
t_{4,C} &= \langle \mathcal{J}_{7,C}, \mathcal{J}_{8,C}, \mathcal{J}_{8,C}, \mathcal{J}_{8,C}\rangle \\
t_{5,C} &= \langle \mathcal{J}_{1,C}, \mathcal{J}_{3,C}, \mathcal{J}_{3,C}, \mathcal{J}_{3,C}\rangle
\end{aligned}$$

Let p_2 be a program consisting of all p_C, arbitrarily ordered into a sequence.

$$\begin{aligned}
\mathcal{J}_3 &= \langle \{b_1, b_2\}\{\langle b_1, \delta, b_1\rangle, \langle b_2, \delta, b_2\rangle\}\rangle \\
\mathcal{J}_4 &= \mathcal{J}_3 \text{ with additional edge } \langle b_1, \neg\gamma, b_2\rangle \\
t_2 &= \langle \mathcal{J}_3, \mathcal{J}_3, \mathcal{J}_3, \mathcal{J}_4\rangle \\
\mathcal{J}_4 &= \mathcal{J}_3 \text{ with additional edge } \langle b_1, \neg\gamma, b_2\rangle \\
\mathcal{J}_5 &= \langle \{b_1, b_2\}\{\langle b_1, \gamma, b_2\rangle\}\rangle \\
\mathcal{J}_6 &= \mathcal{J}_5 \text{ with additional edge } \langle b_1, \neg\gamma, b_2\rangle \\
t_3 &= \langle \mathcal{J}_5, \mathcal{J}_5, \mathcal{J}_5, \mathcal{J}_6\rangle \\
\mathcal{J}_7 &= \langle \{m, n\}\{\langle n, \beta, m\rangle, \langle m, \delta, m\rangle\}\rangle \\
\mathcal{J}_8 &= \mathcal{J}_7 \text{ with additional edge } \langle n, \epsilon, n\rangle \\
t_4 &= \langle \mathcal{J}_7, \mathcal{J}_7, \mathcal{J}_7, \mathcal{J}_8\rangle \\
\mathcal{J}_9 &= \mathcal{J}_8 \text{ without the edge } \langle n, \beta, m\rangle \\
\mathcal{J}_{10} &= \mathcal{J}_9 \text{ with additional edge } \langle n, \neg\beta, m\rangle \\
t_5 &= \langle \mathcal{J}_9, \mathcal{J}_{10}, \mathcal{J}_{10}, \mathcal{J}_{10}\rangle \\
\mathcal{J}_{11} &= \langle \{m, n\}\{\langle n, \beta, m\rangle\}\rangle \\
\mathcal{J}_{12} &= \mathcal{J}_{11} \text{ with additional edge } \langle n, \neg\beta, m\rangle \\
t_6 &= \langle \mathcal{J}_{11}, \mathcal{J}_{12}, \mathcal{J}_{12}, \mathcal{J}_{12}\rangle \\
\mathcal{J}_{13} &= \langle \{n, b_1, b_2\}, \{\langle n, \beta, b_1\rangle, \langle n, \neg\beta, b_2\rangle, \langle b_1, \gamma, b_2\rangle\}\rangle \\
\mathcal{J}_{14} &= \mathcal{J}_{13} \text{ with additional edge } \langle n, \epsilon, n\rangle \\
t_7 &= \langle \mathcal{J}_{13}, \mathcal{J}_{14}, \mathcal{J}_{14}, \mathcal{J}_{14}\rangle \\
\mathcal{J}_{15} &= \langle \{n, b_1, b_2\}, \{\langle n, \beta, b_1\rangle, \langle n, \beta, b_2\rangle, \langle b_1, \neg\gamma, b_2\rangle\} \\
\mathcal{J}_{16} &= \mathcal{J}_{15} \text{ with additional edge } \langle n, \epsilon, n\rangle \\
t_8 &= \langle \mathcal{J}_{15}, \mathcal{J}_{16}, \mathcal{J}_{16}, \mathcal{J}_{16}\rangle \\
\mathcal{J}_{17} &= \langle \{n, b_1, b_2\}, \{\langle n, \beta, b_1\rangle, \langle n, \epsilon, n\rangle\}\rangle \\
\mathcal{J}_{18} &= \mathcal{J}_{17} \text{ with additional edge } \langle b_1, \gamma, b_2\rangle \\
t_9 &= \langle \mathcal{J}_{17}, \mathcal{J}_{18}, \mathcal{J}_{18}, \mathcal{J}_{18}\rangle
\end{aligned}$$

$$
\begin{aligned}
\mathcal{J}_{19} &= \langle \{n, b_1, b_2\}, \{\langle n, \zeta, b_1\rangle, \langle n, \epsilon, n\rangle, \langle b_1, \gamma, b_2\rangle\}\rangle \\
\mathcal{J}_{20} &= \mathcal{J}_{18} \text{ without } n \text{ and } \zeta\text{-edge} \\
t_{10} &= \langle \mathcal{J}_{20}, \langle \{n\}, \emptyset\rangle, \mathcal{J}_{18}, \mathcal{J}_{19}, \mathcal{J}_{19}\rangle \\
\mathcal{J}_{21} &= \langle \{n, b_1\}, \emptyset\rangle \\
\mathcal{J}_{22} &= \mathcal{J}_{21} \text{ with additional edge } \langle n, \zeta, b_1\rangle \\
\mathcal{J}_{23} &= \mathcal{J}_{22} \text{ with additional edge } \langle n, \gamma, b_1\rangle \\
t_{11} &= \langle \mathcal{J}_{22}, \mathcal{J}_{22}, \mathcal{J}_{22}, \mathcal{J}_{23}\rangle \\
t_{12} &= \langle \mathcal{J}_{21}, \mathcal{J}_{22}, \mathcal{J}_{22}, \mathcal{J}_{22}\rangle \\
\mathcal{J}_{24} &= \mathcal{J}_{21} \text{ with additional edge } \langle n, \neg\beta, b_1\rangle \\
t_{13} &= \langle \mathcal{J}_{22}, \mathcal{J}_{22}, \mathcal{J}_{22}, \mathcal{J}_{24}\rangle \\
\mathcal{J}_{25} &= \langle \{b_1, b_2\}, \emptyset\rangle \\
\mathcal{J}_{26} &= \mathcal{J}_{25} \text{ with additional edge } \langle b_1, \neg\gamma, b_2\rangle \\
t_{14} &= \langle \mathcal{J}_{25}, \mathcal{J}_{26}, \mathcal{J}_{26}, \mathcal{J}_{26}\rangle \\
\mathcal{J}_{27} &= \mathcal{J}_{25} \text{ with additional edge } \langle b_1, \gamma, b_2\rangle \\
t_{15} &= \langle \mathcal{J}_{25}, \mathcal{J}_{27}, \mathcal{J}_{27}, \mathcal{J}_{27}\rangle \\
\mathcal{J}_{28} &= \langle \{b_1\}, \emptyset\rangle \\
\mathcal{J}_{29} &= \mathcal{J}_{28} \text{ with additional edge } \langle b_1, \delta, b_1\rangle \\
t_{16} &= \langle \mathcal{J}_{28}, \mathcal{J}_{29}, \mathcal{J}_{29}, \mathcal{J}_{29}\rangle \\
\mathcal{J}_{30} &= \langle \{n\}, \emptyset\rangle \\
\mathcal{J}_{31} &= \mathcal{J}_{30} \text{ with additional edge } \langle n, \gamma, n\rangle \\
t_{17} &= \langle \mathcal{J}_{30}, \mathcal{J}_{30}, \mathcal{J}_{30}, \mathcal{J}_{30}\rangle
\end{aligned}
$$

If

$$p = t_1; p_2; t_2; t_3; t_4; t_5; t_6; t_7; t_8; t_9; t_{10}; t_{11}; t_{12}; t_{13}; t_{14}; t_{15}; t_{16}; t_{17}$$

then, for all instances $\mathcal{I}, \mathcal{I}'$ in which none of the edge labels $\alpha, \neg\alpha, \beta, \neg\beta, \gamma, \neg\gamma, \delta, \epsilon, \zeta$ occur, it holds that $\mathcal{I} \stackrel{i}{\Rightarrow} \mathcal{I}'$ iff $\mathcal{I} \stackrel{p}{\Rightarrow} \mathcal{I}'$.

The proof is tedious again, and has been omitted.

References

[1] ABITEBOUL, S., AND HULL, R. IFO: a formal semantic database model. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 525–565.

[2] ABITEBOUL, S., AND KANELLAKIS, P. Object identity as a query language primitive. In *ACM SIGMOD Intl. Conf. on Management of Data* (1989).

[3] AMANN, B., AND SCHOLL, M. Gram: A graph data model and query language. In *Hypertext '92* (Milano, Italy, 1992), D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, Eds., Association for Computing Machinery.

[4] AMANN, B., AND SCHOLL, M. Database query navigation. Submitted to EDBT'93, 1993.

[5] ANDRIES, M., AND ENGELS, G. A Hybrid Query Language for the Extended Entity Relationship Model. Tech. rep., Leiden University, Dept. of Comp. Science, 1993. (in preparation).

[6] ANDRIES, M., AND PAREDAENS, J. Macro's for the GOOD-transformation language. Tech. Rep. 91-20, University of Antwerp (UIA), Apr. 1991.

[7] ANDRIES, M., AND PAREDAENS, J. A language for generic graph-transformations. In *Graph-Theoretic Concepts in Computer Science, Int. Workshop WG 91* (1992), vol. 570 of *Lecture Notes in Computer Science*, Springer, pp. 63–74.

[8] ANDRIES, M., PAREDAENS, J., AND VAN DEN BUSSCHE, J. A graph- and object-oriented counterpart for SQL. In *Proceedings of The Second Far-East Workshop on Future Database Systems* (Singapore, Apr. 1992), Q. Chen, Y. Kambayashi, and R. Sacks-Davis, Eds., vol. 3 of *Advanced Database Research and Development Series*, World Scientific, pp. 276–285.

[9] BEERI, C., AND KORNATZKY, Y. A logical query language for hypertext systems. In *Proc. of the First European Conf. on Hypertext* (Paris, France, nov 1990).

[10] BUSSCHE, J. V., GUCHT, D. V., ANDRIES, M., AND GYSSENS, M. On the completeness of object-creating query languages. In *IEEE Symp. on Foundations of Computer Science* (1992).

[11] BUSSCHE, J. V., AND PAREDAENS, J. The expressive power of structured values in pure OODBs. In *ACM SIGMOD Intl. Conf. on Management of Data* (1991), pp. 291–299.

[12] CASANOVA, M. A., ET AL. The nested context model for hyperdocuments. In *Proceedings of the ACM Conference on Hypertext* (Dallas, Texas, 1991), Association for Computing Machinery.

[13] CATARCI, T., SANTUCCI, G., AND ANGELACCIO, M. Fundamental graphical primitives for visual query languages. *Information Systems 18*, 2 (1993), 75–98.

[14] CONSENS, M., AND MENDELZON, A. Expressing structural hypertext queries in GraphLog. In *Hypertext'89 Conf.* (nov 1989).

[15] CONSENS, M., AND MENDELZON, A. GraphLog: a visual formalism for real life recursion. In *Proc. of the ACM Symp. Principles of Database Systems* (1990).

[16] CRUZ, I., MENDELZON, A., AND WOOD, P. A graphical query language supporting recursion. In *ACM SIGMOD Intl. Conf. on Management of Data* (1987), pp. 323–330.

[17] FRANCA GARZOTTO, P. P. . D. S. HDM - a model for the design of hypertext applications. In *Proceedings of the ACM Conference on Hypertext* (1991).

[18] GEMIS, M., PAREDAENS, J., THYSSENS, I., AND VAN DEN BUSSCHE, J. GOOD : A graph-oriented object database system. In *SIGMOD Conference Proceedings* (1993).

[19] GYSSENS, M., PAREDAENS, J., BUSSCHE, J. V., AND GUCHT, D. V. A graph-oriented object database model. Tech. rep., University of Antwerp (UIA), 1990.

[20] GYSSENS, M., PAREDAENS, J., AND GUCHT, D. V. A graph-oriented object database model. In *Proc. of the ACM Symp. Principles of Database Systems* (Mar. 1990).

[21] GYSSENS, M., PAREDAENS, J., AND GUCHT, J. V. A graph-oriented object model for database end-user interfaces. *ACM SIGMOD Intl. Conf. on Management of Data 19*, 2 (May 1990).

[22] HALASZ, F., AND SCHWARTZ, M. The Dexter hypertext reference model. In *Proc. of the Hypertext Standardization Workshop* (jan 1990).

[23] HALASZ, F. G. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM 31*, 7 (1988), 836–851.

[24] HOFSTEDE, A. T., PROPER, H., AND WEIDE, T. V. D. Formal definition of a conceptual language for the description and manipulation of information models. Tech. Rep. 92-16, University of Nijmegen, Dept. of Informatics, Toernooiveld 1, Nijmegen, the Netherlands, July 1992.

[25] HOFSTEDE, A. T., AND WEIDE, T. V. D. Expressiveness in data modelling. Tech. Rep. 91/07, University of Nijmegen, Dept. of Informatics, Toernooiveld 1, Nijmegen, the Netherlands, July 1991.

[26] LANGE, D. A formal model of hypertext. In *Proc. of the Hypertext Standardization Workshop* (jan 1990).

[27] PAREDAENS, J., P.PEELMAN, AND TANCA, L. Deductive languages: A graph-based approach. In *Proceedings 92rd International Workshop on Foundations of Models and Languages for Data and Objects* (Aigen, Austria, Sept. 1991).

[28] VADAPARTY, K., ASLANDOGAN, Y., AND OZSOYOGLU, G. Towards a unified visual database access. In *ACM SIGMOD Intl. Conf. on Management of Data* (1993), pp. 357–366.

[29] WINTRAECKEN, J. *The NIAM Information Analysis Method: Theory and Practice.* Kluwer Academic Publishers, 1990.

[30] ZLOOF, M. Query-by-example: a data base language. *IBM Systems J. 16*, 4 (1977), 324–343.

Contents

1 Introduction **1**

1.1 Purpose 1

1.2 Introductory examples 2

2 Language definition **6**

2.1 Data model 6

2.2 Operation syntax and semantics 6

2.3 Program syntax and semantics 9

3 Expressive power of GOLD **11**

3.1 Constructiveness : a strong form of determinacy 11

3.2 Expressive power of full GOLD 11

3.3 GOLD with separated operators 13

3.4 GOLD with only single addition or deletion 13

3.5 GOLD without core patterns 14

3.6 GOOD as GOLD 14

3.6.1 GOOD operations are special cases of the GOLD operator 15

3.6.2 Core patterns versus abstraction 15

3.6.3 Another alternative : the contraction operation 16

4 Discussion and concluding remarks **18**

4.1 Purpose of GOLD 18

4.2 GOLD is a visual language 18

4.3 GOLD operates on graphs 19

4.4 GOLD transforms the database instance 20

4.5 Some sugaring is necessary 21

4.6 GOLD as compared to GOOD 21

4.7 A system based on GOLD 22

A Proofs of theorems **23**

A.1 Correctness of GOLD 23

A.1.1 Determinacy: defining correctness of operations 23
A.1.2 Operations are determinate 24
A.1.3 Correctness of programs 29
A.2 A database transformation not expressible in GOLD 30
A.3 Every instance transformation is expressible in GOLD 32
A.4 Simulating combined operations with separate ones 33
A.5 GOOD operations are subsumed in GOLD 34
A.6 Equivalence of core patterns and abstraction 35
A.6.1 Simulating core patterns with abstraction 35
A.6.2 Simulating abstraction with core patterns 37

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.

93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.