Canonical Term Representations of Isomorphic Transitive DAGs for Efficient Knowledge-based Reasoning

Jutta Eusterbrock

GMD - German National Research Center for Information Technology
Rheinstr. 75, 64295 Darmstadt, Germany,
E-mail: eusterbr@darmstadt.gmd.de

Abstract. This paper presents an efficient term-based representation of transitive, directed, acyclic graphs (DAGs) and classes of isomorphic objects, their operations, concepts and subconcepts within an extended Horn clause logic with equational specifications. As a major new result, a well-founded term ordering, resulting from a decomposition theorem, is defined which proves the existence of unique term representatives for semantical identical objects and classes of isomorphic DAGs. Moreover, canonical forms for object classes are constructed from a single arbitrary term representative of the class by replacing constants with variables. Generally, the subsumption relation on non-ground graphterms models semantical projection. Hence, set theoretic decision, construction and transformation procedures on transitive DAGs and their classes are implemented efficiently by basic operations of logic programming: term rewriting, instantiation, generalization, subsumption, unification and anti-unification. Moreover, there is a one-to-one relation between semantically defined graph properties and syntactic term properties. Hence, knowledge-based deductive and inductive reasoning about transitive DAGs may be performed syntactically. Significant speed improvements have been achieved in the SEAMLESS synthesis system in using canonical forms for the storage and unification for the retrieval of cases.

1 Introduction

In numerous emerging applications of computing science as distributed systems [19], software architecture [12], automated analysis of partial-order sorting algorithms [11] or data bases [14], graphs play a key role for the representation of sets of linked objects. This paper addresses the question of how to implement particular as well as classes of partially ordered sets, ie. transitive, directed acyclic graphs, and graph-theoretic knowledge within a logic programming framework. The objective is to provide formal foundations for building knowledge-based systems which use graphs as representation means and efficiently support graph-theoretic reasoning. Efficient here means that the total amount of storage and the computational complexities to retrieve knowledge on graphs, to perfom graph constructions and transformations, and to prove properties of and relationships between graphs are optimized when dealing with a large body of knowledge. A

major emphasis is on aiding the syntax-guided abstraction from a given set of examples described by graphs.

Pure Logic Programming and its extensions constitute a powerful and flexible specification and programming framework for problems that involve reasoning with structured objects. Various uniform methods for these tasks – centered around some basic mechanisms as unification, term rewriting, inference and mathematical induction – have been devised. Logically, objects are modeled as terms which have certain properties expressed as formulas, eg. axioms or theorems. Term-indexing techniques (cf. [21]) yield significant speed-ups for knowledge retrieval. However, taking a logical view and programming environments or specific graph-theoretic provers in itself is not sufficient to determine the encoding of graphs and to take advantage of these results.

Interactive graph theoretic provers (cf. [5]) can be used only in restricted ways. There is no unique way to represent graphs as terms. Moreover, we are not merely interested in the representation of individual concrete objects as such, but regard the transformation thereof under given operations. Implementing *dynamic change* of objects by providing graph rewrite systems which may be used to evaluate term expressions (cf. [4]) requires extensions of the usual first order programming languages. The main deficit of current representations concerns complexity issues. Graph theory concerns various NP-hard problems. As a standard means, incidence matrices or adjacency lists have been extensively used as implementation format for single graphs within conventional programming environments (cf. [22]). The techniques proposed in [10] are designed to encode single taxonomies efficiently within a logic-based framework. Efficient means that the total storage is optimal with respect to fast answers for reachability queries. However, our main concern are techniques which allow to deal efficiently with a large amount of knowledge described by sets of graphs.

Reformulation of problems by *abstraction* is an approach in practical problem solving to achieve effective systems (cf. [18]). A formally well-defined abstraction function is imposed on domains by *behavioral equivalences*, especially *isomorphism*, which define classes of objects showing identical behavior with respect to the input-output relation of a program or function. Therefore, we would like to determine efficiently whether two graphs (G_1, G_2) are behaviorally equivalent. One way to approach this problem is by constructing a code or canonical form for each graph with the following properties: (i) if $code(G_1) = code(G_2)$, then G_1 is behaviorally equivalent to G_2; and (ii) if G_1 is behaviorally equivalent to G_2 then $code(G_1) = code(G_2)$. Codes representing *semantically identical* finite graphs with the properties (i) and (ii) are easily constructed by some form of alphabetical sorting from any string representation (ground term, incidence matrix). Until now, however, no method for producing a code satisfying (i) and (ii) together on the class of *isomorphic* graphs in general or especially for DAGs is known. There are examples of graph codes which satisfy (i), but not always (ii), and inversely. Codes with the property (i) are used as indexes for storage of assertions. The isomorphism test is performed on each stored assertion that matches the index of the current graph against a case base of the indexes of stored graphs.

This allows to detect many, but not all cases of graph isomorphism. Using this heuristic for deciding graph isomorphism may result in substantial reductions of used amount of storage and search complexities in automated reasoning (cf. [22]). Having a canonical representation satisfying property (ii) is essential when using efficient reasoning procedures whose correctness is based on the *unique-name-assumption*. For example, evaluating $\leftarrow p(t(a,b))$ with respect to the fact $p(t(b,a))$ and using SLD-resolution yields "no". The answer is incorrect, if both terms are intended to represent identical or equivalent objects. The problem of semantical equivalence of terms, representing graphs, arises also as an impediment in graph theoretic proving [4, 20] as there is no characterization of graph equivalence by canonical equations.

In this paper, new fundamental results, providing the foundations for efficient well-understood graph-theoretic reasoning – bridging the gap between effective semantical reasoning and efficient syntax-guided reasoning – will be presented. Transitive DAGs are represented by specific ground terms, called *graphterms*. Thus, operations may be specified correctly by conditional term equations. These equations will be used as rewrite rules in order to evaluate each term to a canonical form. Terms containing variables may be associated with classes of behaviorally equivalent objects. As a major result, one-to-one correspondences between DAGs and graphterms, classes of isomorphic objects and equivalent graphterms are proven. Moreover, canonical forms can be constructed from a single arbitrary member by replacing constants by variables, ie. term generalization, in a predetermined manner. Further abstractions may be constructed by replacing subterms by variables. The resulting subsumption ordering on non-ground terms reflects refinement of behavioral equivalences. As a further substantial new result, it is exemplified that semantically defined graph properties are expressible in a one-to-one way by syntactically defined elementary term properties. Formulas on graphterms allow comprehensive knowledge to be explicitly stated.

The remainder of this paper is organized as follows. Section 2 introduces the formal syntax and semantics for graphterms and their operations. Section 3 states a decomposition theorem for deciding isomorphism between transitive DAGs, defines graphterm equality and constructively proves the existence of canonical forms. In section 4, it is sketched how these results can be used for implementing knowledge-based systems, which perform efficient reasoning, storage and retrieval on transitive DAGs. Finally, these results are summarized.

2 Representation of DAGs and their Operations

The implementation of transitive DAGs and their operations will be developed in the framework of algebraic data structures and abstract data types (cf. [2]). Algebraic systems represent objects by identifying them with particular terms. Laws of computation are modeled as conditional equations which may be applied as rewrite rules when further constraints are satisfied. Concepts as correct representation, canonical form and behavioral equivalence are formalized. Abstract data type specifications can be embedded into Horn clause logic (cf. [13]).

The syntax of this combined logic enables the definition of behavioral equivalences and functions by conditional equations and allows function calls within the head of clauses. It is assumed that the reader is familiar with the basics of discrete mathematics and logic programming, especially the techniques incorporated from abstract data type theory, design of algebraic data structures (cf. [2]), results on reduction orderings (cf. [3]). Some commonly used notation is recalled. Directed acyclic graphs (DAGs) are considered as pairs (V, E), where V denotes a non-empty, finite base set of vertices and E the set of directed edges, edges written $[a, b]$ or $a > b, a, b \in V$. A transitive DAG is a DAG such that for every nodes $a, b, c \in V$: $[a, b] \in E, [b, c] \in E \Rightarrow [a, c] \in E$. Due to the equivalence of transitive DAGs and posets, graph theoretical and ordering concepts will be used synonymously. $a > b$ ($a >^{nt} b$) means a is (direct) predecessor of b. V' is minimal (maximal) subset of V if for all $b \in V'$ there exists no $a \in V \setminus V'$ such that $a < y$ ($a > y$).

For any minimal or maximal set $V' \subset V$, P/V' denotes the maximal subgraph of P whose vertex set is restricted to V'. $\{\{.\}\}$ is used to distinguish multisets from sets. $\mathcal{T}_{\mathcal{V},\mathcal{F}}$ denotes the set of well-formed terms (with variables $\mathcal{V}$ and operators $\mathcal{F}$). ε denotes the empty term. $t_{|\omega}$ refers to the subterm of t at occurrence ω. $arity(t)$ and $degree(t)$ denote the number of arguments of t. $occ(s, t)$ is the number of occurrences of s in t. $domt(t)$ is the tree domain of t, $domt(s, t)$ is the subset of the tree domain of t, which contains the occurrences of the subterm s in t. Identity of subterms defines the quotient set $domt(t)/_{\equiv}$ by $\omega_1 \equiv \omega_2 : t_{|\omega_1} = t_{|\omega_2}$ iff $\omega_1, \omega_2 \in domt(t)$. $\subset$ denotes the subset as well as the subterm property. The relation $\subset^{nt}$ on subterms is defined by $s_1 \subset^{nt} s$ iff $\neg\exists s_2 \subset t[s_1 \subset s_2, s_2 \subset s]$. $t[\omega \leftarrow s]$ denotes the term obtained from t by replacing s at occurrence ω. $t[\omega_1 \leftarrow s_1] \ldots [\omega_r \leftarrow s_r]$ denotes the term obtained from t by $t_1 = t[\omega_1 \leftarrow s_1], \ldots, t_r = t_{r-1}[\omega_r \leftarrow s_r]$. *root(t)* is the leftmost outermost element of t. $func(t)$ denotes the set of functor symbols occurring in t, $|t|$ is defined recursively by $|t| = 1$ if $t \in \mathcal{V} \cup \mathcal{F}$, $|t| = 1 + \sum_{i=1,\cdots,r} |t_i|$ if $t = f(t_1, \cdots, t_r)$. The standard list notation is used; [] denotes the empty list. $..[f, t_1, t_2, \ldots, t_r]$ is a binary function, which, given $t_1, \ldots, t_r$ and functor f, constructs the term $f(t_1, \ldots, t_r)$.

As an intermediate step in the development of term representatives for arbitrary DAGs, the algebra of *multiple transitive reductions* $\mathcal{D}^m$ is introduced. The algebra will be taken as semantic domain which is subject to formal specification. Multiple transitive reductions correspond to transitive DAGs, however the set of edges is considered as multiset. Considering finite base sets, multiple transitive reductions of DAGs are uniquely determined.

Definition 2.1 Let $P = (V, E)$ be a transitive and finite DAG.

a) The *transitive reduction* P^- of P is defined as the graph with the minimum number of edges whose transitive closure yields P ([24]).

b) Let P^- be the transitive reduction of P. The *multiple transitive reduction* P^* of P^- is defined as the graph obtained by duplicating subgraphs according to their number of predecessors.

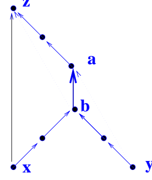
Proposition 2.1 *Let $P = (V, E)$. For each $a, b \in V$ with $(a > b) \notin E$, $(b > a) \notin E$ $\uplus$ is defined by*

$$P \uplus (a > b) = (V, E'), E' = E \cup (a > b) \ \setminus (E_1 \cup E_2 \cup E_3), \text{ if}$$

$$E_1 = \bigcup_{b>y, a>^{nt}y} a >^{nt} y, E_2 = \bigcup_{z>^{nt}b, z>a} z >^{nt} b, E_3 = \bigcup_{b>x, z>a} z >^{nt} x.$$

$\uplus$ is a transitive reduction-preserving operation, correctly corresponding with edge insertion.

The correctness-proof is based on set-theoretic arguments and exemplified by the diagram below.



Definition 2.2 defines a term algebra, called *graphterms* which can be interpreted as multiple transitive reducts. *Constraints* have to be defined which restrict an inductively constructed term algebra to semantically useful terms.

Definition 2.2 *(Graphterms).* Let $\mathcal{F}$ denote any countable set of unary function symbols. Let $\mathcal{T}_\mathcal{F}$ denote the smallest inductively generated set of expressions, such that $f(nil)$ and $f([t_1, \ldots, t_n]) \in \mathcal{T}_\mathcal{F}$, if $f \in F$ and $[t_1, \ldots, t_n]$ is a list of expressions $t_i \in \mathcal{T}_\mathcal{F}, i = 1, \ldots, n$. The set of graphterms $\mathcal{G}_\mathcal{F}$ is the smallest subset of $\mathcal{T}_\mathcal{F}$ for which the constraints (**G1**), (**G2**) and (**G3**) hold. For any $t \in \mathcal{G}_\mathcal{F}$, $s_1, s_2, s \subset t$

- **(G1)** $s_1 \neq s_2 \wedge s_1 \subset s_2 \Rightarrow s_2 \not\subset s_1$;
- **(G2)** $s_1 \subset s_2, s_2 \subset s \Rightarrow \forall s_3[s_3 \in domt(s_1, t) \Rightarrow s_3 \not\subset^{nt} s]$;
- **(G3)** $root(s_1) = root(s_2) \Rightarrow s_1 = s_2$.

The *interpretation* $\iota : \mathcal{G}_\mathcal{F} \to \mathcal{D}^m$, which maps graphterms onto multiple transitive reducts is defined as follows. Let $t \in \mathcal{G}_\mathcal{F}$, $\nu : func(t) \to V$ denote any bijective mapping. $\iota(t) = (\nu(func(t)), E)$ with $[\nu(f), \nu(g)] \in E$, iff there is a subterm with root f that directly precedes a subterm with root g in the corresponding graphterm. (**G1**) ensures that the relations associated with graphterms are anti-symmetric. (**G2**) prohibits relational links that are part of the transitive closure of the associated transitive DAGs from being coded in the graphterm and (**G3**) requires the number of copies of a subterm to comply with the number of their direct predecessors in the associated transitive DAGs. In the sequel, the

brackets for lists will be removed from graphterms to enhance readability.[1] For instance, assume a poset $P = \{z > b, b > b_x, b_x > x, b > b_y, b_y > y, z > z_a, z_a > a, a > x, u > u_a, u_a > a, u > y\}$ and its graphterm representation t as follows $t = \Lambda([z([b([b_x([x([])]), b_y([y([])])]), z_a([a([x([])])])]), u([u_a([a([x([])])]), y([])])])$. Then $t' = \Lambda(z(b(b_x(x), b_y(y)), z_a(a(x))), u(u_a(a(x)), y))$ will be used as graphterm representation of P, where Λ is a dummy root to connect trees together and with the intended meaning being an artificial maximum for all elements.

Having graphterms, set-theoretic operations on DAGs can be implemented by term rewriting. This will be exemplified by giving the term rewrite rules for the operation $add_edge : \mathcal{G}_F \times \mathcal{G}_F \rightarrow \mathcal{G}_F$ which implements adding of edges. Let t denotes any graphterm with subterms t_a, t_b.

$$add_edge(t, [t_a, t_b]) = \left\{ \begin{array}{l} t[\omega \leftarrow \varepsilon][t_b \leftarrow \varepsilon][t_a \leftarrow t_b] \\ \quad \text{iff } t_a \not\subset t_b \;\wedge\; t_b \not\subset t_a \;\wedge\; maximal(t_b) \\ t[\omega \leftarrow \varepsilon][\omega_b \leftarrow \varepsilon][t_a \leftarrow t_b] \\ \quad \text{iff } t_a \not\subset t_b \;\wedge\; t_b \not\subset t_a \;\wedge\; \neg maximal(t_b) \end{array} \right\} \quad (1)$$

with the auxiliary expressions

$$t_{ab} = ..[f_a, [t_{b_1}|L]] \;,\; ..[f_a|L] = t_a[\omega_a \leftarrow \varepsilon],\; \omega_a = \{\omega_1 \in domt(t_a) | t_a|\omega_1 \subset t_b\},$$
$$\omega_b = \{\omega \in domt(t_b, t) | \exists \omega_0 \in domt(t) : t_a \subset t_{|\omega_0} \;\wedge\; t_{|\omega} \subset^{nt} t_{a_{|\omega_0}}\},$$
$$\omega = \{\omega_1 \in domt(t) | t_{|\omega_1} \subset t_b \;\wedge\; \exists \omega_2 \in domt(t) : (t_{|\omega_1} \subset^{nt} t_{|\omega_2} \;\wedge\; t_a \subset t_{|\omega_2})\}.$$

Semantical correctness with respect to the interpretation function ι is shown by proving that the diagram below is commutative

$$\begin{array}{llcl} add_edge : & (P, (a, b)) & \rightarrow & P' \\ & \quad\uparrow\uparrow & & \uparrow \;\iota \\ \uplus : & (t, (f, g)) & \rightarrow & t' \end{array}$$

using the fact that the term rewriting processes correspond directly with the set theoretic operations on the associated graphs (cf. Proposition 2.1).

Example 1 Consider the term $t = \Lambda(z(b(b_x(x), b_y(y)), z_a(a(x))), u(u_a(a(x)), y))$, encoding the poset of proposition 2.1. Evaluating $add_edge(t, [a(x), b(b_x(x), b_y(y))])$ implements the insertion of the edge $[a, b]$ into the corresponding DAG. This yields the graphterm $\Lambda(z(z_a(a(b(b_x(x), b_y(y)))), u(u_a(a(b(b_x(x), b_y(y)))))))$.

Further operations and relations on transitive DAGs can be easily implemented by adding the corresponding term rewrite rules and using the lexicographic ordering on subterm positions. For example, consider the subgraph relation. Let s_1, s_2 denote subterms of graphterm t and ω_1, ω_2 such that $t_{|\omega_1} = s_1$, $t_{|\omega_2} = s_2$. Then $s_1 \supset s_2$ if ω_1 is a prefix of ω_2.

[1] However, the list notation for trees is merely a trick to avoid to deal with arbitrary branching factors, for example non-fixed arities of functors. In the chosen notation a tree is either a leaf or a node consisting of a list of trees.

3 Equivalence, Abstraction and Canonical Form

Objects are behaviorally equivalent, if they behave identically with respect to a defined semantic relation [18]. Behavioral equivalence allows to reformulate problems on higher levels by abstraction. An abstraction is regarded as a homomorphism of a term algebra to a quotient term algebra. The use of abstraction in problem solving is an effective way to reduce search complexities [18], because only representatives of classes have to be examined. A way to take advantage of behavioral equivalence in automated reasoning is to model semantic to perform reasoning with respect to equivalence classes. In order to have terminating proof procedures, the existence of canonical class representatives based on a well-founded ordering has to be proven (cf. [3]).
In the given problem solving context, isomorphism defines behavioral equivalence on graphs. As a major step towards the definition of a well-founded ordering on graphterms and term equations which model isomorphism, a decomposition theorem is established. The common feature of decomposition theorems is that any property for a large member of a family can be deduced from properties of smaller members of the same family and using joining conditions. Decomposition theorems are the key to efficient inductive proofs and the recursive construction of graph families (cf. [25, 16]). However, there is no simple characterization of DAG isomorphism. To see this, consider the following example.

Example 2 Let $P_1 = (V_1, E_1)$, $P_2 = (V_2, E_2)$ and $|E_1|, |E_2| > 1$. P_1 is isomorphic to P_2 ($P_1 \simeq P_2$), cannot be proven from the facts that $P_1 - \{a\} \simeq P_2 - \{b\}$ and some maximal elements $a \in V_1$ and $b \in V_2$ have the same outdegrees. Let $E_2 = \{a_0 < x, a_0 < a, a_1 < a\}$, $E_2 = \{b_0 < b, b_2 < b_1, b_1 < b\}$. Then $P_1 - \{a\} \simeq P_1 - \{b\}$ and $degree(a) = degree(b)$. However, $P_1 \not\simeq P_2$.

Now the decomposition theorem that characterizes isomorphism of transitive DAGs is stated. Let $P = (V, E)$ be any transitive reduction. For any maximal $a \in V$: $P = (P_a, \overline{P_a})$ denotes the pair of graphs induced by the edge partition with $P_a := (V, E_a)$, $E_a := \{x > y \in E | a > x \in E\}$, $\overline{P_a} := (V, \overline{E_a})$,$\overline{E_a} := E \setminus E_a$.

Theorem 3.1 *(Decomposition Theorem). Let $P_1 = (V_1, E_1)$, $P_2 = (V_2, E_2)$ be transitive reductions with $|V_1| = |V_2|$. There exists an isomorphism from P_1 to P_2 ($P_1 \simeq P_2$), iff*

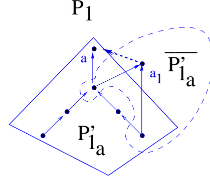
a) $|E_1| = |E_2| = 0$.

b) $|E_1|, |E_2| \geq 1$ *and* $\exists a_1, a \in V_1$, $b_1, b \in V_2$, *a, b being maximal, such that:*

(ISO1) $P_1' = (V_1, E_1 \setminus \{a > a_1\})$, $P_2' = (V_2, E_2 \setminus \{b > b_1\})$.

(ISO2) $P'_{1_a} \simeq P'_{2_b}$ *and* $\overline{P'_{1_a}} \simeq \overline{P'_{2_b}}$.

Proof. The used decomposition and the proof is sketched below.



(1) $\Rightarrow$. **ISO1**, **ISO2** follow from $P_1 \simeq P_2$ by contradiction.

(2) $\Leftarrow$. It has to be shown that b) induces an isomorphism from P_1 to P_2. Let $P_1 = (V_1, E_1) = (V_1, E_1' \cup \{a > a_1\})$ and $P_2 = (V_2, E_2) = (V_2, E_2' \cup \{b > b_1\})$ denote decompositions such that $P'_{1_a} \simeq P'_{2_b}$ and $\overline{P'_{1_a}} \simeq \overline{P'_{2_b}}$. Then, there exists a bijective order-preserving mapping $\phi : V_1 \rightarrow V_2$ with the properties (1) $x > y \in E_1' \Leftrightarrow \phi(x) > \phi(y) \in E_2'$ and (2) $\phi(a) = \phi(b)$ and $\phi(a_1) = \phi(b_1)$. Since a, b are maximal elements, $x > y \in E_1' \cup \{a > a_1\} \Leftrightarrow \phi(x) > \phi(y) \in E_2' \cup \{\phi(a) > \phi(a_1)\} = E_2$. Hence, ϕ defines an order-preserving isomorphism from P_1 to P_2.

Definition 3.1 introduces a graphterm ordering and graphterm equality, applying the decomposition theorem through the axioms (**GO3**), (**GO4**), (**SG2**) and (**SG3**). These equations may be refined by heuristics (cf. **(GO1)**, **(GO2)**) which yields for some graph types more efficient procedures to decide equality. The definition uses the fact that any ordering on a given well-founded set S can be extended to define an ordering on the finite multisets over S. It is well-known that the multiset ordering obtained from a well-founded ordering is again well-founded. Let $\gg_{gt}$ and $==_{gt}$ denote the extensions of the term ordering $>_{gt}$ and the $=_{gt}$ equality to multisets. Let s_1, s_2 be arbitrary graphterms, $|s_1| > 1$, $|s_2| > 1$, then they can be rewritten as $s_1 = ..[a, s_{1_1}, \ldots, s_{r_{1_1}}]$, $s_2 = ..[b, s_{1_2}, \ldots, s_{r_{2_2}}]$. For graphterms and lists of them, there will be defined mutually recursive orderings, which are composed of different orderings (alphabetical ordering of functors, length of terms and subterm properties).

Definition 3.1 *(Graphterm ordering).* Let t_1, t_2 denote any graphterms. The graphterm ordering $>_{gt}$ and equality $=_{gt}$ are defined as follows.

(G1) $t_1 =_{gt} t_2$ if $|t_1| = |t_2| = 1$.

(G2) $t_1 >_{gt} t_2 \Leftarrow t_1 >_{gt(t_1,t_2)} t_2$, if $|t_1|, |t_2| > 1$.

(G3) $t_1 =_{gt} t_2 \Leftarrow t_1 =_{gt(t_1,t_2)} t_2$, if $|t_1|, |t_2| > 1$.

The graphterm ordering and equality use the ordering $>_{gt(t_1,t_2)}$ and equality $=_{gt(t_1,t_2)}$ on subterms which compares subterms with respect to their embedding terms. As before, $\gg_{gt(t_1,t_2)}$ and $==_{gt(t_1,t_2)}$ denote the extensions to multisets.

(GO1) $s_1 >_{gt(t_1,t_2)} s_2 \Leftarrow |s_1| > |s_2|$.

(GO2) $s_1 >_{gt(t_1,t_2)} s_2 \Leftarrow |s_1| = |s_2| \wedge$
$\{\{s_{1_1}, \ldots, s_{r_{1_1}}\}\} \gg_{gt(t_1,t_2)} \{\{s_{1_2}, \ldots, s_{r_{2_2}}\}\}$.

(GO3) $s_1 >_{gt(t_1,t_2)} s_2 \Leftarrow |s_1| = |s_2| \wedge$
$\{\{s_{1_1}, \ldots s_{r_{1_1}}\}\} ==_{gt(t_1,t_2)} \{\{s_{1_2}, \ldots, s_{r_{2_2}}\}\} \wedge$
$t_1[s_1 \leftarrow \varepsilon] >_{gt} t_2[s_2 \leftarrow \varepsilon]$.

(GO4) $s_1 =_{gt(t_1,t_2)} s_2 \Leftarrow |s_1| = |s_2| \wedge$
$\{\{s_{1_1}, \ldots s_{r_{1_1}}\}\} ==_{gt(t_1,t_2)} \{\{s_{1_2}, \ldots, s_{r_{2_2}}\}\} \wedge$
$t_1[s_1 \leftarrow \varepsilon] =_{gt} t_2[s_2 \leftarrow \varepsilon]$.

Graphterm equality induces a partition on graphterms, such that classes correspond one-to-one with classes of isomorphic DAGs. The result will be given in Theorem 3.2. Let α be any alphabetical, total ordering on $\mathcal{F}$. Then $>_{gt}$ may be extended to an ordering $>_{gt(\alpha)}$ which sorts equivalent ground graphterms. The choice of the ordering permits the computation of canonical forms for object representatives as well as for sets of terms corresponding to isomorphic classes.

Definition 3.2 *(Extended graphterm ordering).*

a) Let t_1, t_2 denote graphterms. The graphterm ordering $>_{gt(\alpha)}$, equality $=_{gt(\alpha)}$ and the ordering relations on subterms with respect to their covering terms t_1, t_2 are defined as follows:

(TG1) $t_1 >_{gt(\alpha)} t_2 \Leftarrow t_1 >_{gt(t_1,t_2)(\alpha)} t_2$.

(TG2) $t_1 =_{gt(\alpha)} t_2 \Leftarrow t_1 =_{gt(t_1,t_2)(\alpha)} t_2$.

(SG1) $s_1 >_{gt(\alpha)(t_1,t_2)} s_2 \Leftarrow s_1 >_{gt(t_1,t_2)} s_2$.

(SG2) $s_1 >_{gt(\alpha)(t_1,t_2)} s_2 \Leftarrow s_1 =_{gt(t_1,t_2)} s_2 \wedge$
$\{\{s_{1_1}, \ldots, s_{r_{1_1}}\}\} ==_{gt(\alpha)(t_1,t_2)} \{\{s_{1_2}, \ldots, s_{r_{2_2}}\}\}$,
$root(s_1) >_\alpha root(s_2)$.

(SG3) $s_1 =_{gt(\alpha)(t_1,t_2)} s_2 \Leftarrow s_1 =_{gt(t,t_1)} s_2$,
$\{\{s_{1_1}, \ldots, s_{r_{1_1}}\}\} ==_{gt(\alpha)(t_1,t_2)} \{\{s_{1_2}, \ldots, s_{r_{2_2}}\}\} \wedge$
$root(s_1) =_\alpha root(s_2)$.

b) A graphterm t is called $gt(\alpha)$*-sorted* or $gt(\alpha)$*-normalized*, if

(Gs1) $|t| = 1$.

(Gs2) $|t| > 1$, $t = ..[f, s_1, s_2, \ldots, s_r]$.

(1) $s_1 \geq_{gt(\alpha)(t,t)} s_2 \geq_{gt(t,t)(\alpha)} \cdots \geq_{gt(\alpha)(t,t)} s_r$;

(2) $s_1, s_2, \ldots, s_r$ are $gt(\alpha)$-normalized.

Lemma 3.1

a) The ordering $\geq_{gt(\alpha)}$ is a simplification ordering, i.e. monotonous: $s_1 >_{gt(\alpha)(t_1,t_2)} s_2 \Rightarrow f(\ldots, s_1, \ldots) >_{gt(\alpha)(t_1,t_2)} f(\ldots, s_2, \ldots)$ and has the subterm property $f(\ldots, s, \ldots) >_{gt(\alpha)} s$.

b) $=_{gt}$ is stable under term substitutions:
$t_1 =_{gt} t_2 \Rightarrow \forall s'_1, s''_1 \subset t_1, \forall s'_2, s''_2 \subset t_2$:
$s'_1 =_{gt(t_1,t_2)} s'_2, s''_1 =_{gt(t_1,t_2)} s''_2 \Rightarrow t_1[s'_1 \leftarrow s''_1] =_{gt} t_2[s'_2 \leftarrow s''_2]$.

Since simplification orderings are well-founded, $>_{gt(\alpha)}$ is well-founded. This means, each graphterm can be rewritten into its normalized form.
The above definitions can be employed to establish the fundamental new result in Theorem 3.2 that $=_{gt}$ corresponds with isomorphism of posets and canonical forms exist. Moreover, canonical forms are easily constructable from normalized ground terms by replacing constants one-to-one with variables.

Theorem 3.2 *(Canonical Forms). Given any finite term t, $funct(t) = f_1, \ldots, f_r$ denotes the set of function symbols occurring in t. The non-ground term $\sigma^{-1}(t)$ which is constructed from t by replacing the function symbols with disjoint variables in a one-to-one way is called* abstraction *of t. Let $P_1 = (V_1, E_1)$, $P_2 = (V_2, E_2)$ denote transitive DAGs and t_1, t_2 be normalized graphterms, representing P_1, P_2. Then the following assertions are equivalent.*

a) Let $\sigma^{-1}(t_1)$, $\sigma^{-1}(t_2)$ be abstractions of t_1, t_2, then they are identical up to renaming of variables.

b) The quotient classes $domt(t_1)/_{\equiv}$, $domt(t_2)/_{\equiv}$ on tree domains, constructed with respect to identity of subterms, coincide.

c) $P_1 \simeq P_2$. That is, there is a bijection $\phi : V_1 \to V_2$ such that $[v, w] \in E_1$ iff $[\phi(v), \phi(w)] \in E_2$.

d) The terms t_1, t_2 are graphterm-equivalent, i.e. $t_1 =_{gt} t_2$.

Proof:

Due to a lack of space, the comprehensive proof has to be omitted.

In other words: $\sigma^{-1}[t]$ is a canonical representative of the isomorphism class of P. Each normalized representative t_1 of poset P_1, $P_1 \simeq P$, is subsumed by $\sigma^{-1}[t]$ or instantiates this non-ground term. These results lead to a constructive procedure for proving isomorphism. Isomorphism between P_1, P_2 is decided by abstracting the normalized graphterm t_1 and testing by term subsumption, whether t_2 is an instance of the abstraction of t_1. A further variant of this result is the following. Suppose the normalized terms t_1,t_2 implement isomorphic objects, the most specific generalization of t_1,t_2 represents the class representative. This technique is known as *anti-unification* (cf. [23, 15]).

Corollary 3.1 *Let t_1, t_2 denote graphterm representatives of P_1, P_2.*

a) $t_1 =_{gt(\alpha)} t_2 \Leftrightarrow P_1 \equiv P_2$.

b) $t_1 =_{gt(\alpha)} t_2$ iff their normalizations coincide.

Example 3 The specifications of the graphterm equality and ordering – which

yield canonical forms for equivalence classes of DAGs – seem to be rather complicated. One might argue that simpler forms could yield the same results. The following examples shall demonstrate that simplification may result in non-unique forms for isomorphic DAG classes. Disjoint variables are assumed.

a) Suppose a graphterm equality without alphabetical sorting. Then both of the non-unifiable terms $t_1 = \Lambda(A(B,C), D(C))$ and $t_2 = \Lambda(A(C,B), D(C))$ implement $P = \{a > b, a > c, d > c\}$.

b) Suppose a graphterm equality which does not consider the context of subterms. Then $P = \{a > b, a > c, d > b, f > c\}$ may be implemented alternatively by the two non-unifiable terms $\Lambda(A(B,C), D(B, F(C)))$ and $\Lambda(A(B,C), D(C, F(B))$.

4 Efficient Knowledge-based Reasoning and Retrieval

Canonical graphterms provide the basis for powerful mechanism to express and store graph-theoretic knowledge efficiently. Knowledge can be expressed by formulas containing graphterms, graphterm variables, quantifiers and (non-classical) logical operators. Efficient retrieval, construction, deductive and inductive reasoning can be performed by syntax-guided uniform procedures, exploiting unification and matching, without having to take care of equality.

Ordered Concepts, Theory Formation and Knowledge Compression. Due to the insufficient knowledge about the domain of interest, an essential part of effective mathematical reasoning consists of the construction of examples and counterexamples and abstraction thereof by means of concept formation, cf. [6]. A *concept* is a subset of the universe the elements of which have common properties. On the universe a lattice with the void ($-$) as the minimum and the universe ($\top$) itself as maximum is defined by the subset relation. The analogous set theoretic comprehension of the mathematical notion "concept" and the interpretation of "types" as sets of terms and further on as sets of objects suggests us to state the properties of an object by type expressions, ie. the subsets it belongs to. Types are declared as unary predicates together with sets of defining axioms and subtype relations *type* $\sqsupset$ *subtype*. Subtype relations describe logical entailment and map the taxonomical structure on sets. In our notation, for reasons of efficiency subtype relations have to be declared explicitely.

Most important, the graphterm ordering yields the following remarkable result: *Many common graph properties, relations and inductively defined concepts such as adjacency, vertex degree, minor subgraphs, forests are expressible in the given term description language intuitively by purely syntactical means just by recursively identifying term and subterm properties due to the existence of canonical forms.* This is exemplified by the syntax-guided definitions "forest" and "shrub":

Types $forest, shrub$

Subtypes $gt = \top \sqsupset forest \sqsupset shrub \sqsupset -$

Axioms $forest(t) \leftarrow \forall s \subset t : |domt(s,t)| = 1$
$shrub(t) \leftarrow |t| = 1$
$shrub(t) \leftarrow |t| > 1 \wedge t = ..[f|[t_1, \ldots, t_r]] \wedge$
$shrub(t_1) \wedge shrub(..[f|[t_2, \ldots, t_r]]) \wedge degree(t_1) \geq r - 1.$

The encoding of semantic graph properties as term properties in a one-to-one way allows the mechanical exploration of common characteristics, ie. *theory formation* or *conceptual analysis* of large object classes and the automated construction of graph concepts without considering the semantical interpretation of terms, only based on syntactic criteria. This is illustrated by the following example.

Example 4 Assume the following set of facts have been infered:

$$\{Predicate(\Lambda(a(b(c)), d, f), e(b(c))), \neg Predicate(\Lambda(a(b(c)), d), e(b(c)))\}.$$

First, our theory formation algorithm constructs the following rules

$$\forall t : ?Predicate[t] \leftarrow occ(t|_{[1,1]}) \geq 2$$
$$\exists t : \neg Predicate[t] \wedge occ(t|_{[1,1]}) \geq 2$$

whereas '?' is used to sign formulas with the truth value *unknown* in a 3-valued logic. In further steps, inconsistencies are discovered and refined rules are derived:

$$\forall t : ?Predicate[t] \leftarrow occ(t|_{[1,1]}) \geq 2 \wedge arity(t|_{[1,1]}) \geq 3$$
$$\exists t : \neg Predicate[t] \wedge occ(t|_{[1,1]}) \geq 2 \wedge arity(t|_{[1,1]}) \leq 2.$$

The sketched procedures are very useful for knowledge compression. In our experimental application, several thousand positive and negative atoms – derived by the automated evaluation of proof processes – are being transformed into a few consistent rules, covering all given atoms. Moreover, the derived unique forms enable us to generalize techniques for program synthesis originally developed for lists in the even wider range of problems concerning posets (cf. [8]).

Efficient Storing and Retrieval. Generally, terms have proven to be useful as a flexible logic programming implementation in which various techniques for efficient storing, retrieval and exploiting inheritance can be incorporated (cf. [1, 10]). Employing canonical graphterms to represent knowledge on solved cases, the amount of used storage and hence the retrieval complexity may be reduced enormously, as the following trivial example shows.[2] The 12! =479,001,600 facts $worst_case_sort(\Lambda(a_1(...(a_{12}))), 0), \ldots, worst_case_sort(\Lambda(a_{12}(a_{11}(...(a_1)))), 0)$

[2] Although this seems to be a rather artificial example – no good programmer would use such an encoding – it has to be considered that the knowledge bases are built up automatically and far more complicated situations occur during proof processes.

expressing for each permutation of 12 elements that it needs 0 further steps to sort the given poset, may be stored by the single fact
$worst_case_sort(\Lambda(A_1(A_2(...(A_{12})))), 0)$, using abstraction.

Proving properties of graphs requires the extensive costly computation of characteristic values by evaluating recursive definitions. *Finite Differencing* seeks to replace costly recomputations of values with cheaper, incremental updates. Thus, for example, graphterms may be expanded to embody conceptual information, which is represented by distinguished functors, called typefunctors, and supplementary information. This is done by creating new arguments whose values maintain the desired information. For example, consider $t = \Lambda(z(x), u(u_a(a(x))))$. $\tilde{t} = \top(\Lambda, 5, [ch(z, 2, [ch(1, x, [])]), ch(4, u, [ch(3, u_a, [ch(2, a, [ch(1, x, [])])])])])$ is an expanded graphterm, derived from t, in which the number of successors and type information is explicitly encoded. Elimination of z yields $\tilde{t'} =$
$\top(\Lambda, 4, [ch(4, u, [ch(3, u_a, [ch(2, a, [ch(1, x, [])])])])])$. This demonstrates the efficiency of the incremental technique. The characteristic values of subterms don't have to be recomputed each time. Information retrieval by selecting arguments instead of costly computations may result in significant speed-ups.

Implementation and Application. SEAMLESS is a prototypical system implemented in Prolog and designed to assist software and algorithm synthesis from formal specifications. Synthesis is assumed to be an incremental knowledge-acquisition process.[3] Embedded graphterms represent knowledge of linked objects on different abstraction layers. SEAMLESS communicates with a composite graph layout component for visualization tasks (cf. [9]). Some features have been made accessible through a World-Wide Web user-interface. As an application which allows to test the DAG encoding in a non-trivial scenario demanding highly efficient representation, manipulation and retrieval operations, the automatic synthesis of optimal algorithms for partial-order sorting problems was chosen. In this domain, exhaustive brute-force searches are combinatorially explosive and solutions of non-trivial problems may take years and decades. In [7] the author documented a correction to a theorem on the complexity of optimal partial-order sorting. The correctness is based on the construction of a previously unknown algorithm by means of a first prototype. Search complexities were reduced drastically by strategic knowledge and re-use of solved cases, which were implemented as annotated expert knowledge, automatically augmented at runtime by machine learning procedures and encoded by graphterms.

5 Conclusions

In this paper, as a major new result the existence of canonical forms which identify an infinite class of isomorphic transitive DAGs, based on a decomposition

[3] SEAMLESS is an acronym for an incremental process model: **S**pecification, **E**xample Computation, **A**bstraction, **M**odification, **L**earning, **E**xtraction for **S**oftware **S**ynthesis and is being demonstrated (depending on a running server) at the URL *http://www.darmstadt.gmd.de/~ eusterbr/seamless.html.*

theorem has been proven. A simple procedure to construct canonical forms, based on abstraction and term rewriting, was presented. This extends the result in [4] where only the existence of normal forms for isomorphic DAGs was shown in the context of graph grammars. It avoids having to incorporate elaborated procedures (cf. [20]) for deciding semantic and behavioral equivalence into reasoning systems. Using canonical forms for knowledge representation instead of storing all equivalence class representatives reduces the used amount of storage and consequently, total retrieval complexities significantly. Graphterms bear an enormous potential for automated syntax-guided efficient graph-theoretic reasoning. Graph concepts are now characterized – in a one-to-one way – by elementary term properties, which can be checked efficiently by string operations. This allows the automated exploration of common characteristics of large object classes and the automated discovery of new graph concepts without having in mind the semantic interpretation of terms. The recursive structure of graphterms allows the efficient verification of properties through the application of decomposition theorems. Conceptual knowledge can be represented by partially ordered sorts. Ordered-sorted proof procedures may be used to prune search processes considerably (cf. [26]). The subsumption ordering on non-ground graphterms directly coincides with projection. Hence, set-theoretic semantical reasoning on transitive DAGs is efficiently implemented by syntax-directed operations as substitution, unification, anti-unification and subsumption. These results were incorporated into a knowledge-based synthesis system which was applied for the discovery of a previously unknown, non-trivial algorithm for a combinatorially explosive problem.

This approach suggests a very general approach for the efficient and mechanical handling of further graph classes in knowledge-based systems. The expectation is that numerous domain-specific heuristics for automated graph theoretical investigations used in graphtheoretical provers, expert and discovery systems (cf. [5, 6, 17, 20]) can be substituted by some well-understood operations of logic programming in a unified way.

References

1. H. Ait-Kaci and R. Nasr. Login, a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
2. G. Ausiello and G.F. Mascari. On the design of algebraic data structures with the approach of abstract data types. In W. Ng, editor, *Symbolic and Algebraic Computation*. Lecture Notes in Computer Science 72, Springer, 1979.
3. L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proc. LICS'86*, pages 346–357. IEEE, 1986.
4. M. Bauderon and B. Courcelle. Graph expressions and graph rewritings. *Math. Systems Theory*, 20:84–126, 1987.
5. D. Cvetkovic and I. Pevac. Man-machine theorem proving in graph theory. *Artificial Intelligence*, 35:1–23, 1988.
6. S.L. Epstein and N.S. Sridharan. Knowledge representation for mathematical discovery: Three experiments in graph theory. *J. of Applied Intelligence*, 1(1):7–32,

1991.
7. J. Eusterbrock. Errata to "Selecting the top three elements" by M. Aigner: A Result of a computer assisted proof search. *Discrete Applied Mathematics*, 41:131–137, 1992.
8. J. Eusterbrock. Program synthesis from examples by theory formation. to appear in Proc. ISMIS'97, LNAI, 1997.
9. J. Eusterbrock and M. Nicolaides. The visualization of constructive proofs by compositional graph layout: A world-wide web interface. Proc. CADE Visual Reasoning Workshop, 1996.
10. A. Fall. Spanning tree representations of graphs and orders in conceptual structures. In *Conceptual Structures: Applications, Implementation and Theory*, pages 232–246. LNAI 954, 1995.
11. W. Gasarch, W. Kelly, and W. Pugh. Finding the ith largest of n for small i,n. *Sigact News*, 27(2):88–96, 1996.
12. F. Giunchiglia and T. Walsh. Research directions in software architecture. *ACM Computing Surveys*, 27(2), 1995.
13. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer Verlag, LNAI 353, 1989.
14. Z. Jiao and P. M D Gray. Optimisation of methods in a navigational query language. Technical Report AUCS/TR9108, University of Aberdeen, 1991.
15. J.L. Lassez and K. Marriot. Explicit representation of terms defined by counter examples. *J. of Automated Reasoning*, 3:301–317, 1987.
16. C. Lautenbach. Decomposition trees: Structured graph representations and efficient algorithms. In M. Dauchet and M. Nivat, editors, *13th Colloquium on Trees in Algebra and Programming*, pages 28–39. Springer Verlag, 1988.
17. M. Lipman and R. Sedlmeyer. Developments in an expert system for graph theory investigation. In *14th Annual Computer Science Conference*, pages 327–330. Association for Computing Machinery, 1986.
18. M.R. Lowry. *Algorithm Synthesis through Problem Reformulation.* PhD thesis, Stanford University, 1989.
19. F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Parallel and distributed algorithms*, pages 215–226, 1989.
20. L. Moser. A decision procedure for unquantified formulas of graph theory. In E. Lusk and R. Overbeek, editors, *Proc. 9th International Conference on Automated Deduction*, pages 344–357. LNCS 310, Springer, 1988.
21. H.-J. Ohlbach. Abstraction tree indexing for terms. In *Proc. of the 9th European Conference on Artificial Intelligence*, pages 1–6. Pitman Publ., 1990.
22. I. Parberry. A computer-assisted optimal depth lower bound for nine-input sorting networks. *Mathematical Systems Theory*, 24:101–116, 1991.
23. G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, 1970.
24. J.A. La Poutre and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Graph Theoretic Concepts in Computer Science 87*, pages 106–120. LNCS 314, Springer, 1988.
25. R.G. Parker R.B. Borie and C.A. Tovey. Automatic generation of linear-time algorithms form predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.
26. Chr. Walther. Many-sorted unification. *J. Assoc. Comput. Mach.*, 35(1):1–17, 1988.