An Algebraic Foundation for Adaptive Programming

Peter Thiemann

Institut für Informatik
Universität Freiburg, Germany
Email: thiemann@informatik.uni-freiburg.de

Abstract. An adaptive program is an object-oriented program which is abstracted over the particular class structure. This abstraction fosters software reuse, because programmers can concentrate on specifying how to process the objects which are essential to their application. The compiler of an adaptive program takes care of actually locating the objects. The adaptive programmer merely writes a traversal specification decorated with actions. The compiler instantiates the specification with the actual class structure and generates code that traverses a collection of objects, performing visits and actions according to the specification.
Previous approaches to compiling adaptive programs rely on standard methods from automata theory and graph theory to achieve their goal. We introduce a new foundation for the compilation of adaptive programs, based on the algebraic properties of traversal specifications. Exploiting these properties, we develop the underlying theory for an efficient compilation algorithm. A key result is the derivation of a normal form for traversal specifications. This normal form is the basis for directly generating a traversal automaton with a uniformly minimal number of states.

Key words: object-oriented programming, semantics, finite automata, compilation

1 Introduction

An adaptive program [14, 16, 13, 17] is an object-oriented program which is abstracted over the particular class structure. Adaptive programming moves the burden of navigating through a linked structure of objects of many different classes from the programmer to the compiler. The key idea is to only specify the landmarks for navigation and the actions to be taken at the landmarks, and leave to the compiler the task of generating traversal code to locate the "landmark" classes and to perform the actions.

This abstraction fosters software reuse in two dimensions. First, the same adaptive program applies unchanged to many similar problems. For example, consider the adaptive program *Average* that visits objects of class *Item* and computes the average of the field *amount* therein. This program can be compiled with respect to a class structure for a company, instantiating *Item* to *Employee* and *amount* to *salary*, to compute the average salary of the employees. But the same program can also be compiled by instantiating *Item* to *InventoryItem* and *amount* to *price*. This instance computes the average price of all items in stock.

Second, adaptive programming is attractive for programming in an evolving environment. Here, "evolving" means that classes, instance variables, and methods are added, deleted, and renamed, as customary in refactoring [15, 5, 8]. In this situation, many adaptive programs need merely be recompiled without change, thus alleviating the tedious work of refactoring considerably.

An adaptive program consists of two parts: a traversal specification and wrapper (action) specifications. The traversal specification mentions classes whose objects must (or must not) be visited in a certain order and the instance variables that must (or must not) be traversed. A wrapper specification links a class to an action that has to be performed when the traversal encounters an object of that class. Following Palsberg et al [17, 16], our semantics only considers actions to be performed on the first encounter with an object.

Although a traversal specification only mentions names of classes and instance variables that are relevant for the programming task at hand, the actual class structure, for which the adaptive program is compiled, may contain intermediate classes and additional instance variables. The compiler automatically generates all the code to traverse or ignore these objects. Likewise, wrapper specifications need only be present for classes whose objects require special treatment. Hence, the programmer writes the important parts of the program and the compiler fills in the boring rest.

1.1 Related work

Due to the high-level programming style of adaptive programs, their compilation is an interesting problem. Palsberg et al [17] define a formal semantics for adaptive programs, formalizing Lieberherr's original approach to compilation [13], and identify a number of restrictions. A subsequent paper [16] removes the restrictions and simplifies the semantics, but leads to a compilation algorithm which runs in exponential time in the worst case. Both papers rely on the theory of finite automata and employ standard constructions, like minimization and the powerset construction (which leads to the exponential worst case behavior). In addition, these works employ a more restrictive notion of traversal specification than the present paper.

Finally, Lieberherr and Patt-Shamir [14] introduce further generalizations and simplifications which lead to a polynomial-time compilation algorithm. However, whereas the earlier algorithms perform "static compilation", which processes all compile-time information at compile time, their polynomial-time algorithm performs "dynamic compilation", which means that a certain amount of compile-time information is kept until run time and hence compile-time work is spread over the code implementing the traversal. They employ yet another notion of traversal specification. While this is more general than their earlier work, the relation to our specifications is not clear.

The algebraic approach is based on a notion of derivatives which is closely related to quotients of formal languages [9] and to derivatives of regular expressions [6, 7, 4]. However, traversal specifications differ from standard regular expressions, so our derivatives are novel to this work.

1.2 Contribution of this work

The algebraic foundations of adaptive programming are based on the algebraic properties of traversal specifications. Exploiting the algebraic laws, we define a normal form for traversal specifications. If the specification contains alternative paths (like + in a regular expression) then the size of the normal form can be exponential in the size of the original specification, so that its computation takes exponential time, too. We show that the exponential bound is tight by exhibiting a suitable specification. For a specification without alternatives (coined "multiplicative specification" [16]) this step takes linear time.

Starting from a traversal specification in normal form our algorithm computes the state skeleton of the uniformly minimal traversal automaton, using a notion of derivatives for traversal specifications. Uniform minimality means that the number of states is minimal over all automatons that implement the traversal for all possible class structures. This step takes linear time for multiplicative specifications and exponential time in the worst case for general specifications. We show that the exponential bound is tight.

Only the final compilation step requires the actual class structure. It constructs the actual traversal automaton from the state skeleton and the class structure. It takes time proportional to the product of the sizes of both. We prove that the resulting automaton implements the semantics of a traversal specification. Hence the automaton is equivalent to the one constructed with "static compilation" by Palsberg et al [16].

The main technical contribution of this work is the exploration of the algebra of traversal specifications, in particular Theorems 1 and 2, which demonstrate that the formally constructed automaton is indeed uniformly minimal. These theorems can also be viewed as normal form results for a certain class of regular expressions.

Overview Section 2 establishes some formal preliminaries and defines a semantics of adaptive programs. Section 3 explores traversal specifications and their algebraic properties; culminating in the first compilation step. Section 4 deals with the second compilation step, the construction of the uniformly minimal traversal automaton. Section 5 considers extensions and further work, and Section 6 concludes.

This submission contains an appendix with all proofs, which will be removed from the final paper.

There is a companion paper dealing with the practical aspects of compiling adaptive programs by partial evaluation. It is available from the author's Website http://www.informatik.uni-freiburg.de/ thiemann/papers.

2 Semantics of Adaptive Programs

This section first recalls the basic concepts of class graphs and object graphs used to define the semantics of adaptive programs. Then, we define a generalized (with respect to previous work [16, 17]) notion of traversal specifications and use it to define a semantics of adaptive programs.

2.1 Graphs

A *labeled directed graph* is a triple (V, E, L) where V is a set of nodes, L is a set of labels, and $E \subseteq V \times L \times V$ is the set of edges. Write $u \xrightarrow{l} v$ for the edge $(u, l, v) \in E$; then u is the source, l the label, and v the target of the edge.

Let $G = (V, E, L)$ be a labeled directed graph. A *path from* v_0 *to* v_n is a sequence $(v_0, l_1, v_1, l_2, \ldots, l_n, v_n)$ where $n \geq 0$, $v_0, \ldots, v_n \in V$, $l_1, \ldots, l_n \in L$, and, for all $1 \leq i \leq n$, there is an edge $v_{i-1} \xrightarrow{l_i} v_i \in E$. The set of all paths in G is $\mathsf{Paths}(G)$.

If $p = (v_0, l_1, \ldots, v_n)$ and $p' = (v'_0, l'_1, \ldots, v'_m)$ are paths with $v_n = v'_0$ then define the concatenation $p \cdot p' = (v_0, l_1, \ldots, v_n, l'_1, \ldots, v'_m)$. For sets of paths P and P' let $P \cdot P' = \{p \cdot p' \mid p \in P, p' \in P', p \cdot p' \text{ is defined}\}$.

2.2 Class graphs and object graphs

Let $\mathcal{C}$ be a set of class names and $\mathcal{N}$ be a set of instance names, totally ordered by $\leq$. A *class graph* is a finite labeled directed graph $\mathcal{G}_C = (\mathcal{C}, \mathcal{E}_C, \mathcal{N} \cup \{\diamond\})$.

There are two kinds of edges in the class graph. A *construction edge* has the form $u \xrightarrow{l} v$ where $l \in \mathcal{N}$ $(l \neq \diamond)$. It indicates that objects of class u have an instance variable l containing objects of class v. There is at most one construction edge with source u and label l. Each cycle in $\mathcal{G}_C$ involves at least one construction edge.

An edge $u \xrightarrow{\diamond} v$ is a *subclass edge*, indicating that v is a subclass of u. Without lack of generality [3, 17, 14] we assume that class graphs are *simple*, i.e., every class is either *abstract* (all outgoing edges are subclass edges) or *concrete* (all outgoing edges are construction edges). In addition, if $u \xrightarrow{\diamond} v \in \mathcal{E}_C$ then v is concrete.

Figure 1 shows an example class graph with an abstract class A and three concrete classes B, C, and D. Dashed arrows indicate subclass edges, solid arrow indicate construction edges. Class A has subclasses B and C. Class B has one instance variable a of class D. Class C has an instance variable b of class D and another c of class A.

Let Ω be a set of objects. An *object graph* is a finite labeled graph $(\Omega, \mathcal{E}_O, \mathcal{N})$ such that there is at most one edge with source u and label l. The edge $u \xrightarrow{l} v$ means that the instance variable l in object u holds the object v.

Figure 2 shows an example object graph corresponding to the class structure in Fig. 1. The objects $C1$ and $C2$ have class C, $B1$ class B, and $D1$, $D2$, and $D3$ are object identities of class D.

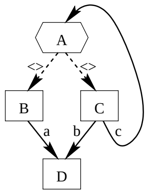

Fig. 1. Example class graph

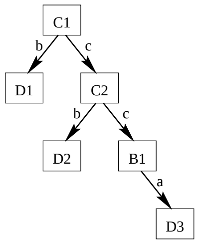

Fig. 2. Example object graph

A *class map* is a mapping $\mathsf{Class} : \Omega \to \mathcal{C}$ from objects to class names of concrete classes. The *subclass map* $\mathsf{Subclasses} : \mathcal{C} \to \mathcal{P}(\mathcal{C})$ maps a class name to the set of class names of all its subclasses, including itself. $\mathsf{Subclasses}(A)$ is the set of all $B \in \mathcal{C}$ such that there is a path $(A, \diamond, \ldots, \diamond, B)$ in the class graph.

2.3 Traversal specifications

A traversal specification answers the question "where do we go from here?" at some point of a traversal of an object or class graph. Hence, a traversal specification is either B (denoting a path to an object of class B), a concatenation of specifications, or an alternative of specifications. Figure 3 shows the formal syntax.

The semantics of a traversal specification is specified relative to a starting node A. It is a set of paths in a class graph.

$$
\begin{aligned}
\mathsf{RPathSet}(A, B) &= \{(A, l_1, A_1, \ldots, l_n, A_n) \in \mathsf{Paths}(\mathcal{G}_C) \mid A_n \in \mathsf{Subclasses}(B)\} \\
\mathsf{RPathSet}(A, \rho_1 \cdot \rho_2) &= \textstyle\bigcup_{B \in \mathsf{Target}(\rho_1)} \mathsf{RPathSet}(A, \rho_1) \cdot \mathsf{RPathSet}(B, \rho_2) \\
\mathsf{RPathSet}(A, \rho_1 + \rho_2) &= \mathsf{RPathSet}(A, \rho_1) \cup \mathsf{RPathSet}(A, \rho_2)
\end{aligned}
$$

The function Target yields the set of possible target classes of a traversal.

$$
\begin{aligned}
\mathsf{Target}(B) &= \{B\} \\
\mathsf{Target}(\rho_1 \cdot \rho_2) &= \mathsf{Target}(\rho_2) \\
\mathsf{Target}(\rho_1 + \rho_2) &= \mathsf{Target}(\rho_1) \cup \mathsf{Target}(\rho_2)
\end{aligned}
$$

$$\begin{array}{rll} \rho ::= & B & \text{simple path to } B \\ | & \rho \cdot \rho & \text{concatenation} \\ | & \rho + \rho & \text{alternative} \end{array}$$

Fig. 3. Traversal specifications

The definition of the semantics naturally adapts to object graphs, by replacing occurrences of class names with objects of the respective classes:

$$\begin{aligned} \mathsf{RPathSet}_{\mathcal{G}_O}(A, B) = \{\, & (o_0, l_1, o_1, \ldots, l_n, o_n) \in \mathsf{Paths}(\mathcal{G}_O) \mid \\ & \mathsf{Class}(o_0) \in \mathsf{Subclasses}(A), \\ & \mathsf{Class}(o_n) \in \mathsf{Subclasses}(B) \} \end{aligned}$$

2.4 Semantics

An adaptive program is a pair (ρ, W) of a traversal specification ρ and a *wrapper map* W. The map W maps a class name $A \in \mathcal{C}$ to an action to be executed when visiting an object of class A. Given an object graph $\mathcal{G}_O$, the semantics of (ρ, W) with respect to some initial object o is completely determined by listing the objects in the order in which they are traversed. Formally,

$$\mathsf{Trav}(\rho, o) = \mathsf{Seq}(\mathsf{RPathSet}_{\mathcal{G}_O}(o, \rho))$$

where

$$\begin{array}{ll} \mathsf{Seq}(\Pi) & = o_0 \mathsf{Seq}(\Pi_1) \ldots \mathsf{Seq}(\Pi_n) \\ \text{where} & \\ \{o_0\} & = \{o \in \Omega \mid o \ldots \in \Pi\} \\ \{l_1, \ldots, l_n\} & = \{l \in \mathcal{N} \mid o_0 l \ldots \in \Pi\} \quad l_i < l_{i+1} \\ \Pi_i & = \{w \in \Omega(\mathcal{N}\Omega)^* \mid o_0 l_i w \in \Pi\} \end{array}$$

To see that $\mathsf{Trav}(\rho, o)$ is well-defined, observe that

1. o_0 is uniquely determined in the first expansion of $\mathsf{Seq}()$ because each path in $\mathsf{RPathSet}_{\mathcal{G}_O}(o, \rho)$ starts with the inital object o;
2. o_0 is uniquely determined in every recursive expansion of $\mathsf{Seq}()$ because the initial segment $o_0 l_i$ of a path in an *object graph* completely determines the next object (there are no inheritance edges in an object graph).

To run the adaptive program on o means to execute the wrappers specified by W in the sequence prescribed by $\mathsf{Trav}(\rho, o)$.

3 Traversal algebra

In this section, we investigate some algebraic laws that hold for traversal specifications and define a normal form for them. Working towards a compilation

$$
\begin{aligned}
(\rho_1 \cdot \rho_2) \cdot \rho_3 &= \rho_1 \cdot (\rho_2 \cdot \rho_3) \\
(\rho_1 + \rho_2) + \rho_3 &= \rho_1 + (\rho_2 + \rho_3) \\
\rho_1 + \rho_2 &= \rho_2 + \rho_1 \\
\rho + \rho &= \rho \\
\rho_1 \cdot (\rho_2 + \rho_3) &= (\rho_1 \cdot \rho_2) + (\rho_1 \cdot \rho_3) \\
(\rho_1 + \rho_2) \cdot \rho_3 &= (\rho_1 \cdot \rho_3) + (\rho_2 \cdot \rho_3)
\end{aligned}
$$

Fig. 4. Laws for traversal specifications

algorithm, we define a notion of derivative for traversal specifications, where taking the derivative of a specification corresponds to visiting a certain node during a traversal. Finally, we consider the complexity of the resulting compilation algorithm.

3.1 Algebraic laws

Traversal specifications obey some algebraic laws. The concatenation of specifications $\cdot$ is associative. The alternative of specifications $+$ is associative, commutative, and idempotent. Furthermore, concatenation $\cdot$ distributes over $+$. Figure 4 shows the resulting laws.

Lemma 1. *The algebraic laws given in Fig. 4 are correct, in the sense that if $\rho_1 = \rho_2$ is a law then, for all A, $\mathsf{RPathSet}(A, \rho_1) = \mathsf{RPathSet}(A, \rho_2)$.*

A further law compresses a concatenation of simple paths even further.

Lemma 2. $A \cdot A = A$

Given an arbitrary total ordering on the set of class names, a traversal specification is in *normal form* if it has the form $w_1 + w_2 + \ldots + w_n$ where each w_i is a *traversal word* (that is, it is generated by the grammar $w ::= B \mid B \cdot w$) and $w_i < w_{i+1}$ in the induced lexicographic ordering. The function norm maps a traversal specification to its normal form. In the algorithm, $\square$ denotes the empty traversal specification.

$$
\begin{aligned}
&\text{norm}(\rho) = && \text{sort}(\text{norm'}(\rho, \square)) \\
&\text{norm'}(B, w_1 + \ldots + w_n) = && \text{prefix}(B, w_1) + \ldots + \text{prefix}(B, w_n) \\
&\text{norm'}(\rho_1 \cdot \rho_2, L) = && \text{norm'}(\rho_1, \text{norm'}(\rho_2, L)) \\
&\text{norm'}(\rho_1 + \rho_2, L) = && \text{norm'}(\rho_1, L) + \text{norm'}(\rho_2, L)
\end{aligned}
$$

$$
\text{prefix}(B, w) = \begin{cases} B & \text{if } w = \square \\ w & \text{if } w = B \cdot w' \\ B \cdot w & \text{otherwise} \end{cases}
$$

The function sort merges traversal specifications by sorting an alternative of words and removing duplicates. The function norm has exponential complexity in the worst case. To see that, consider the family of specifications $\rho_n = (A_1 + B_1)\cdots(A_n + B_n)$ for distinct A_i and B_i. Each ρ_n has size linear in n, but $\mathrm{norm}(\rho_n) = A_1 \cdot A_2 \cdots A_n + \ldots + B_1 \cdot B_2 \cdots B_n$ has size proportional to 2^n.

Lemma 3. *For all A and ρ:* $\mathsf{RPathSet}(A, \rho) = \mathsf{RPathSet}(A, \mathrm{norm}(\rho))$.

3.2 Second normal form

Further simplifications are possible for specifications in normal form. First, define another ordering on words. A word v is less than or equal to a word $w \cdot A_{n+1}$ if v is a subsequence of w followed by a non-empty ascending sequence of superclasses of A_{n+1}.

Definition 1. *Let $m \geq 0$. Define $v \preceq_m w$ iff there is some $n \geq 0$ such that $v = A_1 A_2 \cdots A_n B_1 B_2 \cdots B_{m+1}$ and there exist $\alpha_1, \ldots, \alpha_{n+1}$ such that $w = \alpha_1 A_1 \alpha_2 A_2 \cdots A_n \alpha_{n+1} A_{n+1}$ where $A_{n+1} \in \mathsf{Subclasses}(B_1)$ and, for all $1 \leq i \leq m$, $B_i \in \mathsf{Subclasses}(B_{i+1})$.*

Write $v \preceq w$ if there exists some m such that $v \preceq_m w$.

We will be most interested in the special case where $m = 0$. It turns out that $\preceq_0$ is related to the reverse inclusion of the corresponding path sets.

Proposition 1. *The relation $\preceq$ is a partial ordering on the set of normalized traversal words.*

Corollary 1. *The relation $\preceq_0$ is a partial ordering on traversal words.*

Lemma 4. *For all words v and w, if $w \preceq_0 v$ then, for all A,* $\mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, w)$.

The proof is by induction on v.

The lemma does not extend to $\preceq_m$ for $m > 0$, hence it does not hold for $\preceq$. To see this, consider three distinct class names A, B, and D so that $A \cdot B \preceq D$ if $D \in \mathsf{Subclasses}(A)$ and $A \in \mathsf{Subclasses}(B)$. Now, $\mathsf{RPathSet}(D, D)$ contains (D), but $(D) \notin \mathsf{RPathSet}(A \cdot B,)$, since every element of the latter contains A.

As a corollary, we have the following additional law.

Lemma 5. *Suppose $w \preceq_0 v$ then $v + w = w$.*

Proof. We need to show that, for each A, $\mathsf{RPathSet}(A, v + w) = \mathsf{RPathSet}(A, w)$. The inclusion $\mathsf{RPathSet}(A, w) \subseteq \mathsf{RPathSet}(A, v + w)$ is obvious by definition. The reverse inclusion is Lemma 4.

Definition 2. *A traversal specification ρ is in* second normal form *(2NF) if it is in normal form $\rho \equiv w_1 + \ldots + w_n$ and, for all $i, j \in \{1, \ldots, n\}$, if $i \neq j$ then $w_i \not\preceq_0 w_j$.*

To get norm to produce traversal specifications in 2NF, it suffices to have sort not just remove duplicates but also remove each word w if there is another word v such that $v \preceq_0 w$. Call this modified function sort$'$.

$$\begin{aligned}
\partial_X(A) &\equiv A \\
\partial_X(A \cdot w) &\equiv \begin{cases} w & \text{if } X = A \\ A \cdot w & \text{otherwise} \end{cases} \\
\partial_X(w_1 + \ldots + w_n) &\equiv \partial_X(w_1) + \ldots + \partial_X(w_n)
\end{aligned}$$

Fig. 5. Derivative of a traversal specification

3.3 Formal derivatives

Another view of a traversal specification is to consider it as a state in a traversal. For example, the specification A means "continue until you find some object of class A", and the specification $B \cdot A$ means "search for some B-object and then continue looking for some A-object". Clearly, whenever a traversal visits a node of class X in the object graph, the traversal specification might change to reflect the new state of the traversal. For example, if the traversal hits upon a B-object in state $B \cdot A$, the traversal continues through B's instance variables in state A. In principle, the new state should be $B \cdot A + A$ because, by definition, the traversal should still look for As following some B. However, Lemma 5 shows that A is equivalent to $B \cdot A + A$.

Formally, when a traversal in state ρ encounters an X-object, the new traversal specification for the instance variables is the *derivative* of the old specification ρ with respect to the class X, that is $\partial_X(\rho)$. The definition in Fig. 5 assumes that ρ is already in normal form.

Lemma 6. *If ρ is in (second) normal form then so is* $\text{sort}'(\partial_X(\rho))$*, for all X.*

Computation of $\partial_X(\rho)$ takes time linear in the length of ρ. Subsequent normalization boils down to a single run of sort$'$ on the derivative, which takes $O(n^2 \log n)$ where n is the length of ρ (since each comparison may take time linear in n).

3.4 The state skeleton

Let $\text{Der}(\rho)$ be the set of all iterated derivatives of a traversal specification ρ. Since it only depends on ρ, it is possible to precompute $\text{Der}(\rho)$ before a concrete class graph is given.

The complexity for computing $\text{Der}(\rho)$ is clearly its size $|\text{Der}(\rho)|$ for some ρ in normal form. First, for words the table

w	$\text{Der}(w)$	$\lvert\text{Der}(w)\rvert$
A	$\{A\}$	1
$A \cdot w$	$\{A \cdot w\} \cup \text{Der}(w)$	$1 + \lvert\text{Der}(w)\rvert$

determines the number of derivatives. That is, $|\text{Der}(w)|$ is linear in the size of w.

For a general specification $\rho = w_1 + \ldots + w_n$ which also includes alternatives $\mathrm{Der}(w_1 + \ldots + w_n) \subseteq \{v_1 + \ldots + v_n \mid v_i \in \mathrm{Der}(w_i)\}$ (some alternatives may be deleted due to Lemma 5), it holds that $|\mathrm{Der}(w_1 + \ldots + w_n)| \leq |\mathrm{Der}(w_1)| \cdot \ldots \cdot |\mathrm{Der}(w_n)| \sim |w_1| \cdots |w_n|$. Depending on the structure of ρ, $|w_1| \cdots |w_n|$ can range from linear in $|\rho|$ (for $n = 1$) to exponential in $|\rho|$. To demonstrate the latter, consider the following example. Let $w_i = A_i \cdot B_i \cdot C$ where all A_i and B_i and C are distinct. In this case, $|\mathrm{Der}(w_1 + \ldots + w_n)| = 2^n + 1$ (by straightforward induction using Lemma 5) whereas the size of $w_1 + \ldots + w_n$ is linear in n. Therefore, the exponential bound is tight.

3.5 Compiling traversal specifications

Compiling a traversal specification means to compute the set of its iterated derivatives. In the case of a multiplicative specification [16] (which does not use $+$), compilation takes linear time. Firstly, normalization of a specification boils down to removing repeated class names, which can be done in linear time. Secondly, computing the normalized derivative takes unit time. Finally, the number of normalized derivatives is linear in the size of the multiplicative specification.

For general specifications, compilation takes exponential time in the worst case, for two reasons. First, the normalization of a traversal specification may take exponential time and, second, the traversal specification may have an exponential number of derivatives.

From now on we can safely assume that the function $\partial_X(\rho)$ only deals with specifications in 2NF. Once the elements of $\mathrm{Der}(\rho)$ have been computed, the compiler assigns numbers to them and reduces the computation of $\partial_X(\rho)$ to a constant-time table lookup.

Starting from a specification ρ and a designated source class A, a compiled traversal specification is a quadruple (A, P, ρ_0, ∂) where $\rho_0 \in P$ is the initial traversal specification in 2NF, $P = \mathrm{Der}(\rho_0)$ is the set of normalized iterated derivatives of ρ_0, and $\partial : P \times \mathcal{C} \to P$ is the table of derivatives.

4 Adaption to a class structure

Given a specific class graph $\mathcal{G}_C = (\mathcal{C}, \mathcal{E}_C, \mathcal{N} \cup \{\diamond\})$ and a compiled traversal specification (A, P, ρ_0, ∂), the next task is to produce a target program which implements the traversal. The abstract model of the target program is a finite automaton, the traversal automaton. We describe its construction and prove its correctness with respect to the traversal specification. Then, we show that it is uniformly minimal, ie., it has the least number of states among those automata that implement the semantics of ρ_0 and work for all possible class graphs.

4.1 Traversal automaton

The first step towards the target program is the construction of the traversal automaton. The traversal automaton is a non-deterministic finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ [9] where

– $Q = \mathcal{C} \times \{\text{in}, \text{out}\} \times P$ is the set of states;
– $\Sigma = \mathcal{C} \cup \mathcal{N} \cup \{\diamond\}$ is the alphabet;
– $q_0 = (A, \text{in}, \rho_0)$ is the initial state;
– $F = \{(A, \text{out}, \rho) \mid A \in \mathcal{C}, \rho \in P$ and there exists some B and ρ' such that $\rho = B + \rho' \vee \rho = B, A \in \mathsf{Subclasses}(B)\}$ is the set of final states;
– $\delta((A, \text{in}, \rho), A) = \{(A, \text{out}, \rho)\}$ and $\delta((A, \text{out}, \rho), l) = \{(B, \text{in}, \partial_A(\rho)) \mid A \xrightarrow{l} B \in \mathcal{E}_C\}$ defines the transition function.

As usual, $L(\mathcal{A}, q)$ is the language recognized from initial state $q \in Q$.

Inheritance edges in the class graph are the only sources of non-determinism in $\mathcal{A}$. All transitions on construction edges are deterministic because there is at most one construction edge with a given source u and label l.

The next Lemma shows that the automaton $\mathcal{A}$ indeed implements the semantics of the traversal specification ρ_0.

Lemma 7. *The automaton $\mathcal{A}$ recognizes $\mathsf{RPathSet}_{\mathcal{G}_O}(A, \rho_0)$.*

We actually prove a more general claim: for all $A \in \mathcal{C}$, for all $\rho \in P$, $L(\mathcal{A}, (A, \text{in}, \rho)) = \mathsf{RPathSet}(A, \rho)$. This can be shown using induction on the length n of a path $(A_0, l_1, A_1, l_2, \ldots, l_n, A_n)$.

4.2 Minimal traversal automaton

The step from the traversal automaton to generated code is simple. The states of the automaton correspond to a set of mutually recursive procedures/methods, each of which implements one step of the traversal and – possibly – an action. If there were equivalent states they would have to employ equivalent code to continue the traversal. In other words, the compiled code would suffer from code duplication. Hence, the traversal automaton should have as few states as possible.

Standard results from automata theory [9, Sec. 3.4] show that a minimal (deterministic) finite automaton always exists. This minimal automaton has the property that for all its states q and q', if $q \neq q'$ then $L(\mathcal{A}, q) \neq L(\mathcal{A}, q')$, that is, q and q' are *distinguishable*. Theorem 1 below demonstrates that the set $\text{Der}(\rho)$ generates distinguishable states under certain assumptions on the class graph.

Since we want to compute the uniformly minimal automaton, ie., an automaton which works regardless of the actual class graph, we make the following two assumptions.

Assumption 1 (REACHABILITY) *Let B be an arbitrary class. For all concrete classes A there is a label $l \neq \diamond$ such that $A \xrightarrow{l} B \in \mathcal{E}_C$.*

Assumption 2 (RICHNESS) *Given a specification ρ, there is always a sufficient number of classes not mentioned in ρ.*

These assumptions are technical devices to enable a proof of the following development. They are not the weakest possible assumptions, but rather they are chosen to make the proofs palatable. They are not meant to be imposed on class graphs submitted to the compiler. If a class graph violates the assumptions, then the generated automaton may not be minimal for this particular class graph.

The intuition behind the assumptions is to guarantee the existence of a suitably connected set of classes which are not mentioned in the specification which is compiled. This is usually true because any given specification only mentions a very small subset of the classes in a system.

From now on, REACHABILITY and RICHNESS are implicitly assumed for all statements.

Theorem 1. *Let $\rho_1, \rho_2 \in \mathrm{Der}(\rho)$, all in 2NF, such that $\rho_1 \not\equiv \rho_2$.*
For all $A \in \mathcal{C}$, $\mathsf{RPathSet}(A, \rho_1) \neq \mathsf{RPathSet}(A, \rho_2)$.

We need a number of auxiliary lemmas to prove this theorem. First, we establish that the inclusion of path sets for words implies that the words are related by $\preceq_0$.

Lemma 8. *Let v and w be words in normal form.*
Suppose $\forall A \,.\, \mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, w)$. Then $w \preceq_0 v$.

The proof is by induction on the length of w.

Since Lemma 4 provides the other implication, we have proved the following theorem.

Theorem 2. *Let v and w be words in normal form.*
$\forall A \,.\, \mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, w)$ if and only if $w \preceq_0 v$.

Exploiting that $\preceq_0$ is a partial order immediately yields the following.

Corollary 2. *Let v and w be words in normal form. $\forall A \,.\, \mathsf{RPathSet}(A, v) = \mathsf{RPathSet}(A, w)$ if and only if $w \equiv v$.*

This result for words extends to traversal specifications in strong 2NF. To obtain the strong form of 2NF, we replace $\preceq_0$ by $\preceq$ in the definition of 2NF.

Theorem 3. *Suppose ρ and ρ' are in strong 2NF. Then $\forall A \,.\, \mathsf{RPathSet}(A, \rho) = \mathsf{RPathSet}(A, \rho')$ if and only if $\rho \equiv \rho'$.*

Theorem 1 follows immediately from Theorem 3. Theorem 3 characterizes all specifiable traversals since it establishes a one-to-one correspondence between traversal specifications in strong 2NF and specifiable path sets.

From another point of view, we have proved a normal form result for a certain kind of regular expressions, namely traversal specifications. Theorem 3 completely characterizes them, by putting the languages in one-to-one correspondence with expressions in normal form.

1. $(A, \mathrm{in}, \rho_0) \in Q$.
2. If $q = (A, \mathrm{in}, \rho) \in Q$ then let $q' = (A, \mathrm{out}, \rho)$ in
   - $q' \in Q$; and
   - if $q' \in F$ then active(q'); and
   - if active(q') then active(q); and
   - if active(q') then $(q, A, q') \in \delta$.
3. If $q = (A, \mathrm{out}, \rho) \in Q$ then
   for each $A \xrightarrow{l} B \in \mathcal{E}_C$ let $q' = (B, \mathrm{in}, \partial_A(\rho))$ in
   - $q' \in Q$; and
   - if active(q') then active(q); and
   - if active(q') then $(q, l, q') \in \delta$.

Fig. 6. Constraint system specifying those states of $\mathcal{A}$ which are reachable and active

4.3 Generation of the automaton

The naive construction of the automaton $\mathcal{A}$ is too costly because not all states of $\mathcal{A}$ are accessible from the initial state. Fortunately, it is easy to restrict the construction of the state space of $\mathcal{A}$ so that only accessible states are constructed.

Likewise, naive use of $\mathcal{A}$ to control a traversal leads to unnecessary visits because some states of $\mathcal{A}$ are *sink states*. A state q is a sink state if there is no path from q to a final state or, equivalently, $L(\mathcal{A}, q) = \emptyset$. The non-sink states are easily identified by marking all those states that have a path to a final state (analogous to the construction of an automaton for INIT(L) from an automaton for L [9]). The remaining unmarked states are sink states.

Both properties can be computed in one traversal of the reachable part of the automaton. This traversal takes $O(|\mathcal{E}_C| \cdot |\mathrm{Der}(\rho_0)|)$ time. The constraint system in Fig. 6 specifies the traversal. In the specification, the predicate active(q) is true iff q is not a sink state. It can be implemented using standard techniques in the complexity given above.

5 Extensions and further work

The algebraic approach using derivatives of traversal specifications is also suitable for dynamic compilation [14]. In this case, the compilation time is constant for each class (i.e., linear in the size of the class structure) and the amount of work left to run time is comparable to that in Lieberherr and Patt-Shamir's approach [14]. However, their approach employs a different notion of traversal specification.

It is easy to generalize the framework to multiple source classes since traversal specifications do not mention source classes to begin with. Also the adaption to multiple target classes requires no change in the method, due to our removal of the notion of well-formedness. Well-formedness was only a real requirement in the early work which relied on identifying the set of states of the traversal automaton with the set of classes [17]. The later works have been able to dispense with well-formedness, too [14].

Further operators like negation and intersection could be allowed for traversal specifications. While the algebraic approach seems to work with these operators in principle, its impact on normal forms and the remaining development of Sec. 4 has been left to further investigation. In the database community [1, 12, 2], more expressive path expressions have been considered, including l^{-1} (find an object o so that the current one is the value of instance variable $o.l$) and $\mu(l)$ (find the closest reachable object with instance variable l) [21]. These would also be interesting to investigate.

6 Conclusion

We have presented a new algebraic foundation for compiling adaptive programs. Our approach provides a simple and intuitive algorithm based on formal derivatives of traversal specifications, while maintaining and verifying the previously established complexity bounds. We have implemented the compilation of adaptive programs using partial evaluation, thus substantiating earlier claims in this regard. We hope that this new perspective provides further insight into the structure of adaptive programs.

References

1. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, 1997.
2. Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 122–133, Tucson, Arizona, May 1997. ACM Press.
3. Paul L. Bergstein. Object-preserving class transformations. In *OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 299–313. ACM, November 1991. SIGPLAN Notices (26)11.
4. Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
5. W. Brown, R. Malveau, H. McCormick, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1998.
6. J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
7. John H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.
8. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
9. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

11. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available electronically as http://www.neci.nj.nec.com/homepages/kelsey/r5rs.ps.gz.
12. Michael Kifer, Wong Kim, and Yehoshua Sagiv. Querying object oriented databases. In Michael Stonebraker, editor, *Proceedings of the SIGMOD International Conference on Management of Data*, volume 21 of *SIGMOD Record*, pages 393–402, New York, NY, USA, June 1992. ACM Press.
13. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.
14. Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of object structures: Specification and efficient implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.
15. W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champain, 1992.
16. Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, September 1997.
17. Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.
18. Peter Thiemann. An algebraic approach to compiling adaptive programs. http://www.informatik.uni-freiburg.de/ thiemann/papers/adaptive-report.ps.gz, October 1999.
19. Peter Thiemann. Aspects of the PGG system: Specialization for standard Scheme. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, 1999. Springer-Verlag.
20. Peter Thiemann. *The PGG System—User Manual.* Universität Freiburg, Freiburg, Germany, February 1999. Available from ftp://ftp.informatik.uni-freiburg.de/iif/thiemann/pgg/.
21. Jan Van den Bussche and Gottfried Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, volume 760 of *Lecture Notes in Computer Science*, pages 267–282, 1993.

Proof of Proposition 1

Proof. Reflexivity of $\preceq$ is obvious.

For transitivity, let $v \preceq w$ as in the definition and suppose $u \preceq v$. Then $u = u' \cdot u'' \cdot C_1 \cdots C_{r+1}$, for some $r \geq 0$, where u' is a subsequence of $A_1 A_2 \cdots A_n$, $u'' = B_{i_1} \cdots B_{i_l}$ is a subsequence of $B_1 B_2 \cdots B_m$ and $B_{m+1} \in \mathsf{Subclasses}(C_1)$ and, for $1 \leq i \leq r$, $C_i \in \mathsf{Subclasses}(C_{i+1})$.

Now, $u \preceq w$ because u' is a subsequence of $\alpha_1 A_1 \alpha_2 A_2 \cdots A_n \alpha_{n+1}$, and u'' starts the ascending sequence of superclasses: $A_{n+1} \in \mathsf{Subclasses}(B_{i_1})$ and $B_{i_j} \in \mathsf{Subclasses}(B_{i_{j+1}})$ (since $v \preceq w$), $B_{i_l} \in \mathsf{Subclasses}(C_1)$ and, for $1 \leq i \leq r$, $C_i \in \mathsf{Subclasses}(C_{i+1})$.

For antisymmetry, let $v \preceq w$ and $w \preceq v$ and perform an induction on w.

1. Case B: assume $v \preceq B$ and $B \preceq v$. Since $v \preceq B$, $v = C_1 \cdots C_{r+1}$ where $B \in \mathsf{Subclasses}(C_1)$ and $C_i \in \mathsf{Subclasses}(C_{i+1})$. Now $B \preceq v$ means $B \preceq C_1 \cdots C_{r+1}$, hence $C_{r+1} \in \mathsf{Subclasses}(B)$. By antisymmetry of $\subseteq$, $B = C_1 = \ldots = C_{r+1}$ and therefore, by normalizedness of v, $r = 0$ and $v = B$.
2. Case $B \cdot w$: Suppose $v \preceq B \cdot w$ and $B \cdot w \preceq v$. If $v \preceq B \cdot w$ then either $v = B \cdot v'$ and $v' \preceq w$ or $v \preceq w$.
   If $v = B \cdot v'$ then we have $B \cdot w \preceq B \cdot v'$ and $B \cdot v' \preceq B \cdot w$, that is, $w \preceq v'$ and $v' \preceq w$. By induction, we conclude that $w \equiv v'$ and hence $B \cdot w \equiv v$.
   If $v \preceq w$ would hold, then transitivity yields that $B \cdot w \preceq v \preceq w$. But this would be possible only if $w = D$ and $B \in \mathsf{Subclasses}(D)$ and $D \in \mathsf{Subclasses}(B)$ so that $D = B$, which contradict the normlizedness of w. Hence, this case is impossible.

Proof of Lemma 4

Proof. Induction on v.

1. Case B. If $w \preceq_0 B$ then $w = D$ and $B \in \mathsf{Subclasses}(D)$. By definition of path sets, for all A, $\mathsf{RPathSet}(A, D) \subseteq \mathsf{RPathSet}(A, B)$.
2. Case $B \cdot v'$. If $\pi \in \mathsf{RPathSet}(A, B \cdot v')$ then, by definition, $\exists \pi_1 \in \mathsf{RPathSet}(A, B)$ so that $\exists \pi_2 \in \mathsf{RPathSet}(B, v')$ so that $\pi = \pi_1 \cdot \pi_2$.
   There are two cases for $w \preceq_0 B \cdot v'$.
   (a) If $w = B \cdot w'$ then $w' \preceq_0 v'$. Hence, by induction $\pi_2 \in \mathsf{RPathSet}(B, w')$ and since $\pi_1 \in \mathsf{RPathSet}(A, B)$ we find that $\pi = \pi_1 \cdot \pi_2 \in \mathsf{RPathSet}(A, B \cdot w') = \mathsf{RPathSet}(A, w)$.
   (b) If $w \preceq_0 v'$ then, by induction, $\mathsf{RPathSet}(B, v') \subseteq \mathsf{RPathSet}(B, w)$, so $\pi_2 \in \mathsf{RPathSet}(B, w)$. Since π_1 is a path from A to B, we have $\pi_1 \cdot \pi_2 \in \mathsf{RPathSet}(A, w)$.

Proof of Lemma 7 We need an auxiliary lemma.

Lemma 9. $(A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A, \rho)$ *iff there exists some* D *such that* $A \xrightarrow{l} D \in \mathcal{E}_C$ *and* $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(D, \partial_A(\rho))$.

Proof. Induction on the structure of ρ (in 2NF). First for $\rho = w$, a word.

1. Case B. $(A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A, B)$ iff $(A, l, A_1, \ldots, A_n) \in \mathsf{Paths}(\mathcal{G}_C)$ and $A_n \in \mathsf{Subclasses}(B)$ iff $A \xrightarrow{l} A_1 \in \mathcal{E}_C$ and $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, B)$. This proves the claim since $\partial_A(B) = B$.
2. Case $B \cdot w$. $(A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A, B \cdot w)$ iff $(A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A, B) \cdot \mathsf{RPathSet}(\mathsf{Target}(B), w)$ iff $(A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A, B) \cdot \mathsf{RPathSet}(B, w)$ iff there exist π_1, π_2 with $\pi_1 \in \mathsf{RPathSet}(A, B)$ and $\pi_2 \in \mathsf{RPathSet}(B, w)$ so that $(A, l, A_1, \ldots, A_n) = \pi_1 \cdot \pi_2$.
   There are two cases. If $\pi_1 = (A)$ then the above is equivalent to $A = B$ and $\pi_2 = (A, l, A_1, \ldots, A_n) \in \mathsf{RPathSet}(B, w) = \mathsf{RPathSet}(A, w)$ iff (by induction) there exists $A \xrightarrow{l} A_1 \in \mathcal{E}_C$ and $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, \partial_A(w))$. Since $A = B$, $\partial_A(B \cdot w) = \partial_A(A \cdot w) = w$ and w does not start with A, by normalizedness. Hence $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, \partial_A(B \cdot w))$ iff $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, \partial_A(A \cdot w))$ iff $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, w)$ iff $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, \partial_A(w))$.
   In the other case, $\pi_1 = (A, l, A_1 \ldots, A_i) \in \mathsf{RPathSet}(A, B)$, for $i \geq 1$, and $\pi_2 = (A_i, \ldots, A_n) \in \mathsf{RPathSet}(B, w)$. Applying the induction to π_1 yields $A \xrightarrow{l} A_1 \in \mathcal{E}_C$ and $(A_1, \ldots, A_i) \in \mathsf{RPathSet}(A_1, \partial_A(B)) = \mathsf{RPathSet}(A_1, B)$. Therefore $(A_1, \ldots, A_i) \cdot (A_i, \ldots, A_n) \in \mathsf{RPathSet}(A_1, B) \cdot \mathsf{RPathSet}(B, w) = \mathsf{RPathSet}(A_1, B \cdot w) \subseteq \mathsf{RPathSet}(A_1, \partial_A(B \cdot w))$ with $\partial_A(B \cdot w) \in \{w, B \cdot w\}$ and Lemma 4 providing the last inclusion.
3. Case $w_1 + w_2$. Immediate by appealing to induction.

Proof. (of Lemma 7) The claim needs to be generalized to: for all $A \in \mathcal{C}$, for all $\rho \in P$, $L(\mathcal{A}, (A, \mathrm{in}, \rho)) = \mathsf{RPathSet}(A, \rho)$. Use induction on the length n of a path $(A_0, l_1, A_1, l_2, \ldots, l_n, A_n)$.

1. Case $n = 0$: $(A_0) \in L(\mathcal{A}, (A, \mathrm{in}, \rho))$ iff $A_0 = A$ and $(A, \mathrm{out}, \rho) \in F$ iff $A_0 = A$ and ($\rho = B$ or $\rho = B + \rho'$) and $A \in \mathsf{Subclasses}(B)$ thus $(A_0) \in \mathsf{RPathSet}(A, \rho)$.
   For the other inclusion let $\rho = w_1 + \ldots + w_n$. If $(A_0) \in \mathsf{RPathSet}(A, \rho)$, then $A_0 = A$ and, for some $1 \leq i \leq n$, $(A) \in \mathsf{RPathSet}(A, w_i)$. Now $(A) \in \mathsf{RPathSet}(A, w_i)$ implies $w_i = B$ such that $A \in \mathsf{Subclasses}(B)$. Hence $\rho = B$ or $\rho = B + \rho'$ so that $(A) \in L(\mathcal{A}, (A, \mathrm{in}, \rho))$.
2. Case $n > 0$: Consider a path $(A_0, l_1, A_1, l_2, \ldots, l_n, A_n)$.
   $(A_0, l_1, A_1, l_2, \ldots, l_n, A_n) \in L(\mathcal{A}, (A, \mathrm{in}, \rho))$ iff $A_0 = A$ and $(l_1, A_1, l_2, \ldots, l_n, A_n) \in L(\mathcal{A}, (A, \mathrm{out}, \rho))$ iff there exists $A \xrightarrow{l_1} B \in \mathcal{E}_C$ and $(A_1, l_2, \ldots, l_n, A_n) \in L(\mathcal{A}, (B, \mathrm{in}, \partial_A(\rho)))$. The induction hypothesis is applicable since $\partial_A(\rho) \in P$ and the length of the path is strictly less than n. Therefore, the last expression is equivalent to $A \xrightarrow{l_1} B \in \mathcal{E}_C$ and $(A_1, l_2, \ldots, l_n, A_n) \in \mathsf{RPathSet}(B, \partial_A(\rho))$. By Lemma 9 this is equivalent to $(A_0, l_1, A_1, l_2, \ldots, l_n, A_n) \in \mathsf{RPathSet}(A, \rho)$, which proves the claim.

Proof of Lemma 10 This is an auxiliary result to prove Lemma 8.

Lemma 10. *If, for all A and $\pi \in \mathsf{RPathSet}(A, w)$, B occurs in π, then B occurs in w.*

Proof. Suppose for a contradiction that B does not occur in w but $\forall A\,.\,\forall\pi \in \mathsf{RPathSet}(A, w)\,.\,B$ occurs in π. By RICHNESS there is some $A \neq B$ and $\pi \in \mathsf{RPathSet}(A, w)$ with a minimal number of Bs (at least one must be present). Letting $A_0 = A$ and $w = A_1 \ldots A_n$, $n \geq 1$, it must be that $\pi = A_0\alpha_1 A_1 \ldots \alpha_n A'_n$, for some $A'_n \in \mathsf{Subclasses}(A_n)$.

Since $A \neq B$ and, for all i, $A_i \neq B$, by assumption, it must be that either $A'_n = B$ or B occurs in one of the α_i. In the first case, by RICHNESS and REACHABILITY, there is some α'_n, which does not mention B, and $A''_n \neq B$ so that $\pi' = A_0\alpha_1 A_1 \ldots \alpha'_n A''_n \in \mathsf{RPathSet}(A, w)$ with fewer Bs than π. This contradicts to the minimality of π.

In the other case, by RICHNESS and REACHABILITY, there exists some path α'_i, which does not mention B, from A_i to A_{i+1} or (if $i = n-1$) from A_{n-1} to A'_n. Replacing α_i by α'_i in π yields a path with fewer Bs than π. This is again a contradiction to the minimality of π.

Proof of Lemma 8

Proof. Induction on w.

1. Case B. Show that $\forall A\,.\,\mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, B)$ implies $v = v' \cdot D$ where $B \in \mathsf{Subclasses}(D)$.
   $\forall A\,.\,\mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, B)$ iff $\forall A\,.\,\mathsf{RPathSet}(A, v) \subseteq \{A\alpha E \in \mathsf{Paths}(\mathcal{G}_C) \mid E \in \mathsf{Subclasses}(B)\}$ implies $v = v' \cdot D$ where $B \in \mathsf{Subclasses}(D)$.
2. Case $B \cdot w$. Show that $\forall A\,.\,\mathsf{RPathSet}(A, v) \subseteq \mathsf{RPathSet}(A, B \cdot w)$ implies $B \cdot w \preceq_0 v$.
   Since $\mathsf{RPathSet}(A, B \cdot w) = \mathsf{RPathSet}(A, B) \cdot \mathsf{RPathSet}(B, w)$ it must be that, for all $\pi \in \mathsf{RPathSet}(A, v)$, B occurs in v by Lemma 10. Hence, we can write $v = v' \cdot B \cdot v''$ or $v = B \cdot v''$. Clearly, $\mathsf{RPathSet}(B, v'') \subseteq \mathsf{RPathSet}(B, w)$ (otherwise the originally assumed inclusion would be invalidated), so that we obtain $w \preceq_0 v''$ by induction. By the definition of $\preceq_0$, we conclude that $B \cdot w \preceq_0 v$.

Proof of Theorem 3

Proof. The "if" part is trivial. It remains to prove the "only if" part.

Since ρ and ρ' are both in strong 2NF, they can be written as $\rho = w_1 + \ldots + w_k$ and $\rho' = w'_1 + \ldots + w'_m$ where each of the w_i and w'_j is a normalized word.

Let M be the set of class names occuring in ρ and ρ'. Consider $w_1 = A_1 \cdots A_n$ and the path $\pi = A\alpha_1 A_1 \ldots \alpha_n A_n \in \mathsf{RPathSet}(A, A_1 \cdots A_n)$. By RICHNESS, we can assume that none of the α_i mentions a class name in M and, by REACHABILITY, we can fix the last latter of π to A_n.

Since $\mathsf{RPathSet}(A, \rho) = \mathsf{RPathSet}(A, \rho')$, there must be some j so that $\pi \in \mathsf{RPathSet}(A, w'_j)$. Hence, w'_j has one of the forms $A_{i_1} \cdots A_{i_l} \cdot A'$ or $A_{i_1} \cdots A_{i_l} \cdot A_n \cdot A'$, where $A_n \in \mathsf{Subclasses}(A')$ and $i_1, \ldots, i_l$ is a subsequence of $1, \ldots, n-1$.

Applying the same reasoning to w'_j, we find that there exists some i so that w_i has one of the forms $A_{j_1} \cdots A_{j_p} \cdot A''$ or $A_{j_1} \cdots A_{j_p} \cdot A' \cdot A''$ or $A_{j_1} \cdots A_{j_p} \cdot A_n \cdot A''$ or $A_{j_1} \cdots A_{j_p} \cdot A_n \cdot A' \cdot A''$, where $A_n \in \mathsf{Subclasses}(A')$, $A' \in \mathsf{Subclasses}(A'')$, and $j_1, \ldots, j_p$ is a subsequence of $i_1, \ldots, i_l$. But this means that $w_i \preceq w_1$, which is ruled out by the strong 2NF. Therefore, $i = 1$ and $w_1 = w'_j$.

Iterating this argument, we find that $k = m$ and, after reordering, $w_i = w'_i$, so that $\rho = \rho'$.