A New Approach to Compiling Adaptive Programs[*]

Jens Palsberg[1]   Boaz Patt-Shamir[2]   Karl Lieberherr[2]

[1] MIT Laboratory for Computer Science, NE43-340, 545 Technology Square, Cambridge, MA 02139, USA, `palsberg@theory.lcs.mit.edu`
[2] Northeastern University, College of Computer Science, 161 Cullinane Hall, Boston, MA 02115-9959, USA, {`boaz,lieber`}`@ccs.neu.edu`

Abstract. An adaptive program can be understood as an object-oriented program where the class graph is a parameter, and hence the class graph may be changed without changing the program. The problem of compiling an adaptive program and a class graph into an object-oriented program was studied by Palsberg, Xiao, and Lieberherr in 1995. Their compiler is efficient but works only in special cases. In this paper we present and prove the correctness of a compiler that handles the general case. The compiler first computes a finite-state automaton and then uses it to generate efficient code.

1   Introduction

Object orientation has demonstrated that properties such as encapsulation, inheritance, late binding, etc. are useful in the discipline of software engineering. However, object-oriented languages suffer from a certain inherent "rigidity" which makes software re-use sometimes awkward and laborious. This property can be intuitively explained as follows. The key feature of most object-oriented languages is that the description of actions (usually called "methods" in this context) is attached to the description of types ("classes"). While this characteristic property is useful in many cases, it has been observed (see, e.g., [2]) that changes in the structure of data (i.e., class definitions) may necessitate re-writing large portions of the action code (i.e, method definitions), even if essentially, the underlying algorithm remains the same.

Let us illustrate this point with a simple example. Consider the following two scenarios. In one scenario, we are given a data structure named *company* which describes fully a commercial firm, and our task is to write a function *sumSalary* which computes the total sum of salaries on *company*'s payroll. In the second scenario, we are given a data structure called *airplane* which describes the current state of an airplane, and our task is to write a function *sumWeight* which finds the current total cargo weight. Naturally, the *company* and *airplane* structures are different, and it seems that there is no escape from writing each of the two functions *sumSalary* and *sumWeight* from scratch. However, after a

---

[*] To appear in *Science of Computer Programming*.

second thought (or perhaps after writing dozens of functions...) one sees that *sumSalary* and *sumWeight* are doing essentially the same thing. Loosely speaking, the algorithm for both *sumSalary* and *sumWeight* is as follows: "given an object, scan all its sub-objects of a certain kind, and apply a (commutative and associative) combining operation to these subobjects to obtain the final result." The difference between the code for *sumSalary* and *sumWeight* is due solely to the difference in the specific structure of the input, and not to differences in the underlying algorithms. Moreover, the detailed description of the data structures (which contributes most of the complexity in the code in the examples above) is, in fact, readily provided to the programmer! It would be desirable to enable programmers to specify a generic algorithm which could be automatically tailored to fit the application at hand according to a given description of the structure of the application.

Scenarios such as the one sketched above (which are quite common in the practice of software development) constitute the main motivation for *adaptive programs* [7, 11, 6]. Informally (a formal description is given in Section 2), an adaptive program is a program where the complete description of its data structures is a parameter. Employing the idea of object orientation, actions are associated with types, and in adaptive programs this means that *action code is associated with partially-specified data structures*. Of course, an adaptive program cannot be executed. To get an executable program, an adaptive program has to be *specialized*, in the sense of partial evaluation [5], with a complete description of the actual data structures to be used.

Let us outline the way adaptive programs can be used in our example from above. The basic concepts of adaptive programs will be informally introduced as we proceed. Adaptive programs consist of *traversal specifications* and *code wrappers*. Traversal specifications select objects according to their classes, and code wrappers associate actions with the selected objects. For example, a traversal specification of the form $[A, B]$ is interpreted as "all objects of class B which are subobjects of an object of class A." With the proper code wrapper attached to class B, the interpretation of the adaptive program could be "for each object of class B contained in the class A object, add its value to a *sum* variable."

In Figure 1 we give a complete description of the adaptive program informally sketched above, which illustrates the concise nature of this language. The advantage of adaptive programs is that they adapt automatically to changes in the class structure. For each particular application, we just need to provide a class graph that describes it and a renaming which maps adaptive-program identifiers to class names from the graph. A class graph is a labeled directed graph which expresses the "has-a" and "is-a" relations among the classes. The combination of an adaptive program and a class graph contains all the details required for execution: the operations that are to be applied to objects are fully defined, and the desired traversals (in the *company* example, finding all subobjects of type *salary*) can be automatically generated from the class graph.

Adaptive programming is related to functional programming with iterators and folders. Instead of writing a traversal specification, one might first use a

```
OPERATION void add(counter& total)
  TRAVERSE                                        traversal specification
    [Container, Item]          meaning: find all Item subobjects of Container
  WRAPPER Item                                        behavior (C++ code)
    (@ total = total + value; @)
```

```
RENAME                   company scenario    RENAME                  airplane scenario
  add => sumSalary,                            add => sumWeight,
  Container => Company,                        Container => Airplane,
  Item => Salary                               Item => Weight
```

Fig. 1. Top: an adaptive program. Bottom: renamings for two scenarios.

traversal routine to extract a list of the relevant objects, and then do a fold on
that list. The advantage of adaptive programming is that the traversal routine is
succinctly specified and automatically generated from the traversal specification.
This is particularly convenient when we want to change the class graph but not
the traversal specification. An advantage of typed functional programming is that
iterators and folders can be defined at a meta-level as type-dependent functions.
This, however, requires the set of types to be smaller than the untyped set of class
graphs that we use in this paper. If the advantages of adaptive and functional
programming were to be combined, a useful first step would be to define a typed
universe of class graphs, where the types provide more information than, say,
meta-classes. We leave such developments to future work.

Syntax for traversal specifications, etc. can easily be added to an existing
object-oriented language. See [6] for numerous examples of adaptive program-
ming in an extension of C++.

Systems which support adaptive programming have been available since 1991,
and are being successfully used at Northeastern University, Xerox PARC, and
other places [1]. The core of the compiler provided by these tools was presented
and proved correct in [11]. The current compiler, despite being quite useful in
many practical cases, is not general in the sense that there are certain com-
binations of adaptive programs and class graphs which the compiler rejects. If
a program and a class graph cannot be compiled, then the program has to be
rewritten (as discussed in [11]). This defeats the original motivation of adaptive
programs, namely the automation of adaptiveness.

In this paper, we present a new compiler which has the desired features of
generality in the sense that it is applicable to any combination of adaptive pro-
gram and class graph, and *optimality* in the sense that it generates traversal
code based on a minimized automaton. Informally, the main idea is as follows.
While the old compilation algorithm used the class graph directly to generate
traversal code, the new compiler uses the class graph to construct a finite au-
tomaton which is used, in turn, to generate the traversal code. The concept of

intermediate automaton enables us to apply standard minimization techniques to ensure that the size of the traversal code is optimal.

We prove the correctness of the compiler with respect to the original semantics for adaptive programs, as described in [11, 6]. Our proof consists of two stages. First, we define a variant of the original semantics, and prove that it is equivalent to the original one. Then we show how to construct automata which implement the new semantics. Informally, the purpose of defining the new semantics is to help us to deal with the subclass relation in the construction.

The remainder of this paper is organized as follows. In Section 2 we define basic notions and recall the original semantics of adaptive programs. In Section 3 we give a new semantics of adaptive programs and prove that it is equivalent to the old one, and in Section 4 we give a compilation algorithm for adaptive programs and prove it correct with respect to the new semantics. Due to lack of space, most proofs are omitted; they can be found in [9].

2   The Semantics of Adaptive Programs

In this section we recall from [11] the definitions of graphs, paths, class graphs, object graphs, traversal specifications, wrappers, and the semantics of adaptive programs. We also define the semantics of an object-oriented target language. The target language is slightly different from the one used in [11].
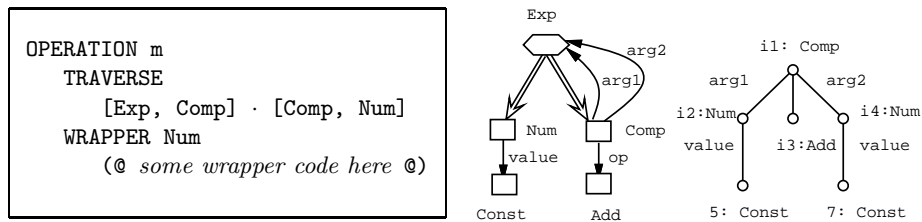

Fig. 2. Left: an adaptive program. Middle: a class graph. Right: an object graph.

As a running example throughout the paper we consider the adaptive program, the class graph, and the object graph in Figure 2.

2.1   Graphs

A labeled directed graph is a triple (V, E, N) where V is a set of nodes, N is a set of labels, and E is a set of edges where $E \subseteq V \times N \times V$. If $(u, l, v) \in E$, then u is the source, l is the label, and v is the target of (u, l, v). We will write (u, l, v) as $u \xrightarrow{l} v$.

2.2 Paths

Given a graph $G = (V, E, N \cup \{ \diamond \})$ where $\diamond \notin N$, a *path* is a sequence $v_0 l_1 v_1 l_2 \ldots l_n v_n$ where $v_0, \ldots, v_n \in V$, and for all $0 \le i < n$ we have that $v_i \xrightarrow{l_{i+1}} v_{i+1} \in E$, and $l_1, \ldots, l_n \in N \cup \{ \diamond \}$. We call v_0 and v_n the *source* and *target* of the path, respectively. If $p_1 = v_0 \ldots v_i$ and $p_2 = v_i \ldots v_n$, then we define the concatenation $p_1 \cdot p_2 = v_0 \ldots v_i \ldots v_n$. Notice that $p_1 \cdot p_2$ contains only one copy of the meeting point v_i. Let P_1 and P_2 be sets of paths such that all paths in P_1 have target v, and all paths of P_2 have source v. Then we define

$$P_1 \cdot P_2 = \{p \mid p = p_1 \cdot p_2 \text{ where } p_1 \in P_1 \text{ and } p_2 \in P_2\} \ .$$

For the remainder of this subsection, let R denote an arbitrary set of paths. We are mainly interested in the paths obtained by removing a prefix containing only $\diamond$-labeled edges. First, we define an auxiliary function Reduce which will be used to define the concepts we are interested in. Intuitively, $\mathsf{Reduce}(R)$ is the set of paths obtained by removing all $\diamond$ prefixes from each path in R. Formally, for a path set R we define

$$\mathsf{Reduce}(R) = \{v_n \ldots v_{n+m} \mid \exists v_0, v_1 \ldots v_{n-1} \text{ such that } v_0 \diamond v_1 \diamond \ldots \diamond v_n \ldots v_{n+m} \in R\}$$

We now arrive at our main definitions. For a node u, we define $\mathsf{Select}(R, u)$ to be the set of suffixes of paths in R that start with u after skipping a leading $\diamond$-labeled prefix. Formally:

$$\mathsf{Select}(R, u) = \{v_0 \ldots v_n \mid v_0 \ldots v_n \in \mathsf{Reduce}(R), v_0 = u\} \ .$$

Finally, we define $\mathsf{Car}(R, u)$ to be the set of the first edges in $\mathsf{Select}(R, u)$, and for a given label l, we define $\mathsf{Cdr}(l, R, u)$ to be the set of tails of $\mathsf{Select}(R, u)$ where the head has label l. Formally:

$$\mathsf{Car}(R, u) = \{v_0 \xrightarrow{l_1} v_1 \mid v_0 l_1 v_1 \ldots v_n \in \mathsf{Select}(R, u)\}$$
$$\mathsf{Cdr}(l, R, u) = \{v_1 \ldots v_n \mid v_0 l_1 v_1 \ldots v_n \in \mathsf{Select}(R, u), l_1 = l\} \ .$$

2.3 Class and Object Graphs

The following notion of class graph is akin to those presented in [8] and [10]. The set $\mathsf{ClassName}$ is a set of class names which can be used in class graphs. The predicate $\mathsf{Abstract}$ is true for names of abstract classes, and it is false otherwise. If a class is not abstract, it is said to be *concrete*.

A *class graph* is a finite labeled directed graph, intended to represent the (static) class structure of a program. Formally, it is defined as follows. Each node is an element of $\mathsf{ClassName}$. Each edge is labeled by an element of $N \cup \{ \diamond \}$, where $(N, \le)$ is a totally ordered set of labels, such that $\diamond \notin N$. If $l \in N$, then the edge $u \xrightarrow{l} v$ indicates that the class represented by u has an instance variable with name l and with a type represented by v. Such an edge is called

a *construction* edge. Intuitively, a construction edge $u \xrightarrow{l} v$ in the class graph represents the fact that the l-component of objects of class u is an object of class v. We require that the construction edges outgoing from a node are labeled with distinct labels.

The edges $u \xrightarrow{\diamond} v$ are called *subclass* edges; they represent the fact that v is a subclass of u. In a class graph, only abstract classes have outgoing subclass edges. A class graph is *flat* if for every node u where Abstract(u), all outgoing edges are subclass edges. Following [11] we henceforth assume that all class graphs are flat.

For example, in the class graph of Figure 2, Exp is an abstract class (indicated by a hexagon), and the other four classes are concrete (indicated by rectangles). The edges from Exp to Num and Comp are subclass edges (indicated by double arrows), and the other four edges are construction edges (indicated by regular arrows). Clearly, the class graph is flat.

An *object graph* is a finite labeled directed graph, intended to represent a run-time object structure. Formally, it is defined as follows. Each node represents an object, and the function Class maps each node to "its class," that is, the name of a concrete class. Each edge is labeled by an element of N. The edge $u \xrightarrow{l} v$ indicates that the object represented by u has a component object represented by v. For each node u and each label $l \in N$, there is at most one outgoing edge from u with label l. For example, the object graph in Figure 2 contains six nodes, each representing an object.

2.4 Traversal Specifications

A *traversal specification* denotes a set of paths. Formally, it is an expression generated by the grammar

$$D ::= [A, B] \mid D \cdot D \mid D + D$$

where A and B are class names.

The semantics of traversal specifications is intuitively summarized in the following table.

Specification D	PathSet$_G(D)$	Source(D)	Target(D)
$[A, B]$	All paths from A to B in G	A	B
$D_1 \cdot D_2$	PathSet$_G(D_1) \cdot$ PathSet$_G(D_2)$	Source(D_1)	Target(D_2)
$D_1 + D_2$	PathSet$_G(D_1) \cup$ PathSet$_G(D_2)$	Source(D_1)	Target(D_1)

For a traversal specification to be meaningful, it has to be *well formed.* A traversal specification is well formed if (1) it determines a *source* node and a *target* node, (2) each concatenation has a "meeting point," and (3) each union of a set of paths preserves the source and the target. Formally, the predicate WF is defined in terms of the two functions Source and Target given in the table

above, and the following recursive definition.

$$
\begin{aligned}
\mathsf{WF}([A, B]) &= \text{true} \\
\mathsf{WF}(D_1 \cdot D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \mathsf{Target}(D_1) =_{\text{nodes}} \mathsf{Source}(D_2) \\
\mathsf{WF}(D_1 + D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\quad (\mathsf{Source}(D_1) =_{\text{nodes}} \mathsf{Source}(D_2)) \wedge \\
&\quad (\mathsf{Target}(D_1) =_{\text{nodes}} \mathsf{Target}(D_2))
\end{aligned}
$$

If G is a class graph and D is a well-formed traversal specification, then $\mathsf{PathSet}_G(D)$ is a set of paths in G from $\mathsf{Source}(D)$ to $\mathsf{Target}(D)$, as defined in the table above.

Lemma 1. *If* $\mathsf{WF}(D)$, *then* (i) $\mathsf{PathSet}_G(D)$ *is well defined and* (ii) *each path in* $\mathsf{PathSet}_G(D)$ *starts in* $\mathsf{Source}(D)$ *and ends in* $\mathsf{Target}(D)$.

We henceforth assume that all traversal specifications are well formed. We shall use traversal specifications to denote path sets in class graphs and object graphs. For example, let D be the traversal specification of the adaptive program in Figure 2, and let G be the class graph from Figure 2. Let us denote paths as strings. Then using the standard notation of regular expressions [3], and denoting by $L(E)$ the language defined by a regular expression E, we have that

$$
\mathsf{PathSet}_G(D) = L(\mathrm{Exp}\ (\diamond\ \mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2})\ \mathrm{Exp})^+ \diamond\ \mathrm{Num})\ .
$$

2.5 Adaptive Programs

Following [11], we define adaptive programs as follows. First, define a *wrapper map* to be a mapping of class names to code segments called wrappers (the idea is that when an object is visited during the traversal of an adaptive program, the appropriate wrapper code will be executed). Now, an *adaptive program* is a pair (D, W), where D is a traversal specification, and W is a wrapper map. Intuitively, given an object graph Ω and a node o in Ω, the interpretation of an adaptive program (D, W) is roughly "for the subgraph of Ω reachable from o: traverse the objects on paths induced by D in depth-first order, and execute the wrapper code specified by W for each object visited." Formally, the semantics is given by the function Run defined as follows.

$$
\mathsf{Run}(D, W)(G, \Omega, o) = \mathsf{Execute}_W(\mathsf{Traverse}(\mathsf{PathSet}_G(D), \Omega, o))
$$
$$
\text{where}\quad \mathsf{Traverse}(R, \Omega, o) = \begin{cases} H \text{ if } \Omega \vdash_s o : R \rhd H, \text{ for some } H \\ \bot \text{ otherwise} \end{cases}
$$

If Ω is an object graph, o a node in Ω, R a path set over G, and H a sequence of objects, then the judgment

$$
\Omega \vdash_s o : R \rhd H
$$

means that when traversing the object graph Ω starting in o, and guided by the path set R, then H is the *traversal history*, that is, the sequence of objects

that are traversed. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_s o_i : \mathsf{Cdr}(l_i, R, \mathsf{Class}(o)) \rhd H_i \qquad \forall i \in 1..n}{\Omega \vdash_s o : R \rhd o \cdot H_1 \cdot ... \cdot H_n} \qquad \begin{array}{l} \text{if } \mathsf{Car}(R, \mathsf{Class}(o)) = \\ \quad \{\mathsf{Class}(o) \xrightarrow{l_i} w_i \mid i \in 1..n\}, \\ o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and} \\ l_j < l_k \text{ for } 1 \le j < k \le n. \end{array}$$

The label s of the turnstile indicates "semantics." Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_s o : R \rhd o} \qquad \text{if } \mathsf{Car}(R, \mathsf{Class}(o)) = \emptyset.$$

Notice that $\mathsf{Traverse}$ is well defined: if both $\Omega \vdash_s o : R \rhd H_1$ and $\Omega \vdash_s o : R \rhd H_2$, then $H_1 = H_2$. This can be proved by induction on the structure of the derivation of $\Omega \vdash_s o : R \rhd H_1$.

The call $\mathsf{Execute}_W(H)$ executes in sequence the wrappers for the class of each of the objects in H. We leave $\mathsf{Execute}_W$ unspecified, since its definition depends on the language in which the code wrappers are written.

Example. Let $R = \mathrm{Exp}\ (\diamond\ \mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2})\ \mathrm{Exp})^+ \diamond \mathrm{Num})$. We get:

$$\begin{aligned} \mathsf{Car}(R, \mathsf{Class}(i1)) &= \mathsf{Car}(R, \mathrm{Comp}) \\ &= \left\{ \mathrm{Comp} \overset{\mathrm{arg1}}{\to} \mathrm{Exp}, \mathrm{Comp} \overset{\mathrm{arg2}}{\to} \mathrm{Exp} \right\} \\ \mathsf{Cdr}(\mathrm{arg1}, R, \mathsf{Class}(i1)) &= L(\mathrm{Exp}\ (\diamond\ \mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2})\ \mathrm{Exp})^* \diamond \mathrm{Num}) \\ \mathsf{Cdr}(\mathrm{arg2}, R, \mathsf{Class}(i1)) &= L(\mathrm{Exp}\ (\diamond\ \mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2})\ \mathrm{Exp})^* \diamond \mathrm{Num}) \end{aligned}$$

Let R' denote $\mathsf{Cdr}(\mathrm{arg1}, R, \mathsf{Class}(i1)) = \mathsf{Cdr}(\mathrm{arg2}, R, \mathsf{Class}(i1))$. Clearly we have $\mathsf{Car}(R', \mathsf{Class}(i2)) = \emptyset$ and $\mathsf{Car}(R', \mathsf{Class}(i4)) = \emptyset$. Let Ω be the object graph in Figure 2. Assuming arg1 < arg2 in the total order of the labels, we get the following derivation:

$$\frac{\Omega \vdash_s i2 : R' \rhd i2 \qquad \Omega \vdash_s i4 : R' \rhd i4}{\Omega \vdash_s i1 : R \rhd i1\ i2\ i4}$$

Thus, the traversal history is $i1\ i2\ i4$.

2.6 The Target Language

We will compile adaptive programs into an object-oriented target language without inheritance. A program in the target language is a partial function which maps a pair of a class name and a method name to a method. A method is a tuple of the form $\langle l_1.m_1, \ldots, l_n.m_n \rangle$, where $l_1 \ldots l_n \in N$ and $m_1 \ldots m_n$ are method names. When invoked, such a method executes by invoking $l_i.m_i$ in order.

If Ω is an object graph, o a node in Ω, m a method name, P a program in the target language, and H a sequence of objects, then the judgment

$$\Omega \vdash_c o : m : \mathsf{P} \rhd H$$

means that when sending the message m to o, we get a traversal of the object graph Ω starting in o so that H is the traversal history. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_c o_i : m_i : \mathsf{P} \rhd H_i \qquad \forall i \in 1..n}{\Omega \vdash_c o : m : \mathsf{P} \rhd o \cdot H_1 \cdot ... \cdot H_n} \quad \begin{array}{l} \text{if } \mathsf{P}(\mathsf{Class}(o), m) = \langle l_1.m_1 \ldots l_n.m_n \rangle \\ \text{and } o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n. \end{array}$$

The label c of the turnstile indicates "code". Intuitively, the rule says that when sending the message m to o, we check if o understands the message, and if so, we invoke the method. Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_c o : m : \mathsf{P} \rhd o} \quad \text{if } \mathsf{P}(\mathsf{Class}(o), m) = \langle \rangle.$$

Given a program in the target language, it is straightforward to generate, for example, a C++ program.

3   A Simplified Semantics of Adaptive Programs

In this section we specify a new semantics of adaptive programs, and prove that it is equivalent to the one given in Section 2. The main difference between the new and the old semantics is the way they treat the subclass relation. To emphasize the difference, we use the term "words" for paths without subclass edges. Informally, the idea is as follows.

The semantics of adaptive programs which was given in Section 2 has the following property. When a path set is used to guide a traversal, $\diamond$-labels are skipped along the way by the operations Car and Cdr. In this section, we define a simpler semantics which has the property that all $\diamond$-labels are removed before the traversal begins. The new semantics greatly simplifies the compiling algorithm presented in Section 4. Our notion of word is related to that of "calling path" in [4].

Our first step is to define functions transforming path sets into strings (words), while deleting abstract classes. Define a *word* to be a sequence $v_0 l_1 v_1 l_2 \ldots v_n$ where $v_0, \ldots, v_{n-1}$ are names of concrete classes, $l_1, \ldots, l_{n-1} \in N$, and v_n is the name of either an abstract or a concrete class. Next, we define the function $\mathsf{SimplifyPath}$ which maps paths to words as follows. Given a path p, the function $\mathsf{SimplifyPath}$ is the string obtained from p by removing all $\diamond$ labels and abstract class names, except for the last class name in p. Observe that if p is a path in a flat class graph, then $\mathsf{SimplifyPath}(p)$ is a word. To see that, recall that in flat class graph, every outgoing edge of an abstract class is a subclass edge, and every outgoing edge of a concrete class is a construction edge. Thus, in a path, except for the last class, a class is abstract if and only if the following label is $\diamond$. Finally, for a path set R, we define $\mathsf{Simplify}(R) = \{\mathsf{SimplifyPath}(p) \mid p \in R\}$.

Example. Let D be the traversal specification of the adaptive program in Figure 2, and let G be the class graph from Figure 2. We have:

$$\mathsf{Simplify}(\mathsf{PathSet}_G(D)) = L((\mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2}))^+\ \mathrm{Num})\ .$$

Next, we define traversal of objects in terms of strings. Let R denote a set of strings. We use the functions First and Chop, defined as follows:

$$\mathsf{First}(R) = \{x \mid \exists\alpha.(x\alpha \in R)\}$$
$$\mathsf{Chop}(R, x) = \{\alpha \mid x\alpha \in R\}\ .$$

If Ω is an object graph, o a node in Ω, R a word set, and H a sequence of objects, then the judgment

$$\Omega \vdash_n o : R \rhd H$$

means that when traversing the object graph Ω starting in o, and guided by the word set R, then H is the traversal history. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_n o_i : \mathsf{Chop}(\mathsf{Chop}(R, \mathsf{Class}(o)), l_i) \rhd H_i \qquad \forall i \in 1..n}{\Omega \vdash_n o : R \rhd o \cdot H_1 \cdot ... \cdot H_n}$$

$$\text{if } \mathsf{First}(\mathsf{Chop}(R, \mathsf{Class}(o))) = \{l_i \mid i \in 1..n\},$$
$$o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and } l_j < l_k \text{ for } 1 \le j < k \le n.$$

The label n of the turnstile indicates "new semantics."

For example, let

$$S = L((\mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2}))^+\ \mathrm{Num})\ .$$

Notice that S is a set of words now. We get:

$$\mathsf{First}(\mathsf{Chop}(S, \mathsf{Class}(i1))) = \mathsf{First}(\mathsf{Chop}(S, \mathrm{Comp}))$$
$$= \{\mathrm{arg1}, \mathrm{arg2}\}$$
$$\mathsf{Chop}(\mathsf{Chop}(S, \mathsf{Class}(i1)), \mathrm{arg1}) = L((\mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2}))^*\ \mathrm{Num})$$
$$\mathsf{Chop}(\mathsf{Chop}(S, \mathsf{Class}(i1)), \mathrm{arg2}) = L((\mathrm{Comp}\ (\mathrm{arg1} + \mathrm{arg2}))^*\ \mathrm{Num})\ .$$

Let S' denote $\mathsf{Chop}(\mathsf{Chop}(S, \mathsf{Class}(i1)), \mathrm{arg1}) = \mathsf{Chop}(\mathsf{Chop}(S, \mathsf{Class}(i1)), \mathrm{arg2})$. Clearly, we have that $\mathsf{First}(\mathsf{Chop}(S, \mathsf{Class}(i2))) = \emptyset$ and $\mathsf{First}(\mathsf{Chop}(S, \mathsf{Class}(i4))) = \emptyset$. Let Ω be the object graph in Figure 2. Carrying on our assumption that $\mathrm{arg1} < \mathrm{arg2}$, we get the following derivation:

$$\frac{\Omega \vdash_n i2 : S' \rhd i2 \qquad \Omega \vdash_n i4 : S' \rhd i4}{\Omega \vdash_n i1 : S \rhd i1\ i2\ i4}$$

Thus, in this case, the new semantics gives the same traversal history as the old semantics. We now prove that the new semantics is equivalent to the one given in Section 2. We start with two lemmas.

Lemma 2. *For a path set R and a concrete class u,*

$$\{l \mid (u \xrightarrow{l} v) \in \mathsf{Car}(R, u) \text{ for some } v\} = \mathsf{First}(\mathsf{Chop}(\mathsf{Simplify}(R), u)) \ .$$

Lemma 3. *For a path set R, a concrete class u, and a label $l \in N$,*

$$\mathsf{Simplify}(\mathsf{Cdr}(l, R, u)) = \mathsf{Chop}(\mathsf{Chop}(\mathsf{Simplify}(R), u), l) \ .$$

Theorem 1. $\Omega \vdash_s o : R \triangleright H$ *if and only if* $\Omega \vdash_n o : \mathsf{Simplify}(R) \triangleright H$.

4   Compiling Adaptive Programs

The compiler of [11] will reject the adaptive program and class graph of Figure 2, as discussed in [11]. The reason is that the code that would be generated looks as follows.

```
CLASS Comp                          CLASS Num
    VAR arg1, arg2: Exp
    METHOD m                            METHOD m
        arg1.m; arg2.m                      – Wrapper code here
    END                                 END
END                                 END
```

This code does not correctly handle objects that are simply Nums, such as $i2$ in Figure 2. When the message m is sent directly to $i2$, it executes the wrapper code even though the execution has not processed any Comp object first.

We now present a compiling algorithm which can compile all combinations of adaptive programs and class graphs. The presentation proceeds in two steps. First we show how to compute a representation of a set of paths, and then we use that representation to generate code.

4.1   Automata

Given a class graph G and a traversal specification D, we will represent the word set $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$ by a finite state non-deterministic automaton. In this section we show how to compute this representation.

We start by constructing automata for traversal specifications of the form $[A, B]$. We will use the notation that if V is a set, then V_{in} and V_{out} are two distinguished copies of V where elements are subscripted by in and out, respectively.

If $G = (V, E, N \cup \{\diamond\})$ is a class graph and $A, B \in V$, then $\mathsf{Auto}_G(A, B)$ is a finite state automaton defined as follows:

- the set of states is $V_{in} \cup V_{out}$,
- the alphabet is $V \cup N \cup \{\diamond\}$,
- the start state is A_{in},
- there is just one accept state B_{out}, and

– the transitions are

$$v_{in} \xrightarrow{v} v_{out} \text{ if } v \in V$$
$$u_{out} \xrightarrow{l} v_{in} \text{ if } u \xrightarrow{l} v \in E \text{ where } l \in N \cup \{\diamond\} \ .$$

An example of the construction is given later in the section.

Lemma 4. *Let G be a class graph, and let A, B be two classes in G. Then* $L(\mathsf{Auto}_G(A, B)) = \mathsf{Paths}_G(A, B)$.

Next, we show how to construct automata for an arbitrary directive. For this, we define three operations on automata as follows. Let $L(M)$ denote the language accepted by an automaton M.

– If M_1, M_2 are automata, then $M_1 + M_2$ is the automaton such that $L(M_1 + M_2) = L(M_1) \cup L(M_2)$. $M_1 + M_2$ can be computed by standard methods [3].
– If M_1, M_2 are automata, then $M_1 \cdot M_2$ is the automaton defined as follows. The states of $M_1 \cdot M_2$ are the disjoint union of the states of M_1 and the states of M_2, together with a fresh state m. The start state of $M_1 \cdot M_2$ is the start state of M_1. The accept states of $M_1 \cdot M_2$ are the accept states of M_2. The transitions of $M_1 \cdot M_2$ are the union of the transitions of M_1 and the transitions of M_2, together with ϵ-transitions from each final state of M_1 to m, and ϵ-transitions from m to each state in M_2 which can be reached from the start state of M_2 by a sequence of ϵ-transitions followed by one non-ϵ-transition. For an example of this construction, see below.
– If M is an automaton which only accepts paths in some class graph, then $\mathsf{Simplify}(M)$ is the automaton defined as follows. The states of $\mathsf{Simplify}(M)$ are those of M together with a fresh state s. The start state of $\mathsf{Simplify}(M)$ is that of M. The only accept state of $\mathsf{Simplify}(M)$ is s. The transitions of $\mathsf{Simplify}(M)$ are defined as follows.

$u \xrightarrow{\epsilon} v$ if $u \xrightarrow{\epsilon} v$ is a transition of M

$u \xrightarrow{a} v$ if $u \xrightarrow{a} v$ is a transition of M where $a \in N \cup \{u \mid \neg\mathsf{Abstract}(u)\}$

$u \xrightarrow{\epsilon} v$ if $u \xrightarrow{a} v$ is a transition of M where $a \in \{\diamond\} \cup \{u \mid \mathsf{Abstract}(u)\}$

$u \xrightarrow{l} s$ if there is a path in M from u to an accept state of M which consist

of one l-transition followed by a sequence of ϵ-transitions.

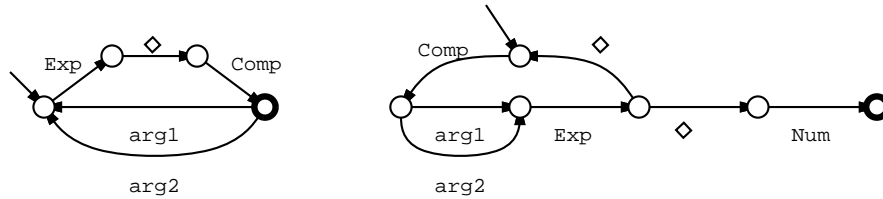For an example of this construction, see below.

Lemma 5. *Given automata M_1 and M_2, $L(M_1 \cdot M_2) = L(M_1) \cdot L(M_2)$, and* $L(\mathsf{Simplify}(M_1)) = \mathsf{Simplify}(L(M_1))$.

Finally, for a traversal specification D and a class graph G, define $A_G(D)$ recursively as follows.
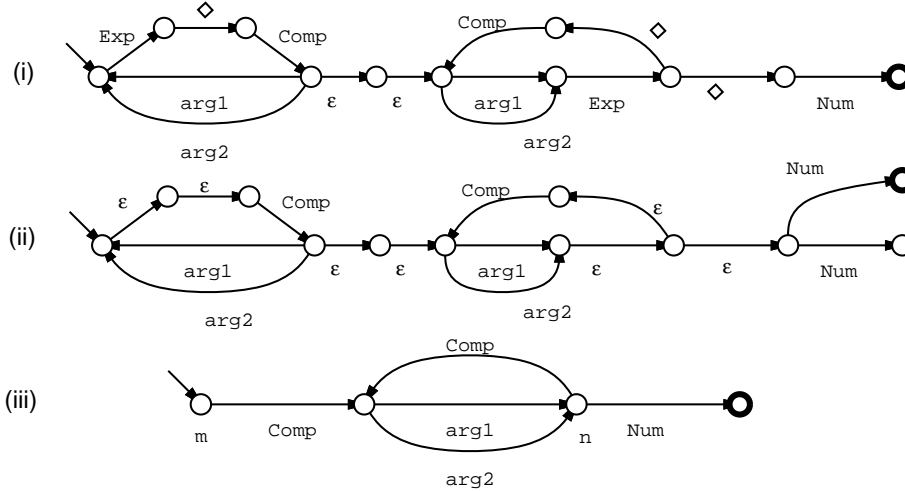
$$A_G([A, B]) = \mathsf{Auto}_G(A, B)$$
$$A_G(D_1 \cdot D_2) = A_G(D_1) \cdot A_G(D_2)$$
$$A_G(D_1 + D_2) = A_G(D_1) + A_G(D_2)$$

Clearly, $A_G(D)$ accepts precisely $\mathsf{PathSet}_G(D)$. Hence, we can compute an automaton which accepts the word set $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$. However, the resulting automaton is non-deterministic, and thus cannot be used directly to guide traversals. The next step in our construction is therefore to determinize the automaton accepting $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$ using the standard subset construction. Finally, we minimize the automaton using the standard algorithm (see [3]). In Section 4.2, we show how to use the resulting automaton to produce deterministic code.

Example. In Figure 2 we have $D = D_1 \cdot D_2$ where $D_1 = [\mathrm{Exp}, \mathrm{Comp}]$ and $D_2 = [\mathrm{Comp}, \mathrm{Num}]$. First, we display the two automata $A_G(D_1)$ and $A_G(D_2)$.

We display final states as fat circles. Next, we display the automaton $A_G(D)$, see (i), the automaton $\mathsf{Simplify}(A_G(D))$, see (ii), and the minimal deterministic automaton which accepts $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$, see (iii). For later use, in (iii) we have labeled two of the states (m and n).

4.2   The Compiling Algorithm

We are now in a position to explain how to generate code. We use the following notation. Given an automaton M and a state s, let $\mathsf{Outgoing}_M(s)$ denote the set of all transitions $s \xrightarrow{a} s'$.

For an automaton M, we define P_M to be a program in the target language by the following rule. The method names in P_M are the states of M, defined as follows.

1. If $s \xrightarrow{c} s'$, where c is a class name, and $\mathsf{Outgoing}_M(s') = \{s' \xrightarrow{l_i} m_i \mid i \in 1..n\}$, then $P_M(c, s) = \langle l_1.m_1 \ldots l_n.m_n \rangle$, where $l_j < l_k$ for $1 \leq j < k \leq n$.
2. Otherwise, if c is a class name used in M and s is a state in M, then $P_M(c, s) = \langle \rangle$.

If we in case (2) have access to the class graph from which the automaton M was generated, then we can avoid the generation of many unreachable methods. In a target language with inheritance, the empty methods can be placed in superclasses, thus reducing code size further. The wrapper code would by an implementation be inserted into the methods generated from case (1).

Example. Given the deterministic automaton shown above, the compiling algorithm emits the following code (written in a programming language-like notation). For simplicity, we have omitted four empty methods, two for each class, generated from case (2).

```
CLASS Comp                              CLASS Num
    VAR arg1, arg2: Exp
    METHOD m    // case (1)                 METHOD m    // case (2)
        arg1.n; arg2.n                          – No code here
    END                                     END
    METHOD n    // case (1)                 METHOD n    // case (1)
        arg1.n; arg2.n                          – Wrapper code here
    END                                     END
END                                     END
```

Notice that two method names are needed to distinguish if a Num object is reached via a Comp object or not. In the former case (method n), the wrapper code should be executed, in the latter case (method m), it should not.

The example indicates the consequence of the potentially large size of the deterministic automaton which accepts $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$: massive wrapper code duplication, in the worst case. Notice that if we change the class graph of the example such that class Num can be reached from several classes, say Comp_1, Comp_2, etc, then each class Comp_i gets two methods m and n.

We conclude this paper with a proof that the compiling algorithm is correct.

Theorem 2. *If M_s is a deterministic automaton which accepts a word set, then*

$$\Omega \vdash_n o : L(M_s) \rhd H \quad \textit{if and only if} \quad \Omega \vdash_c o : s : P_M \rhd H \ .$$

Proof. Suppose first that $\Omega \vdash_n o : L(M_s) \rhd H$ is derivable. We proceed by induction on the structure of the derivation of $\Omega \vdash_n o : L(M_s) \rhd H$. Since $\Omega \vdash_n o : L(M_s) \rhd H$ is derivable, we have that

$H = o \cdot H_1 \cdot \ldots \cdot H_n$

$\mathsf{First}(\mathsf{Chop}(L(M_s), \mathsf{Class}(o))) = \{l_i \mid i \in 1..n\}$

$o \xrightarrow{l_i} o_i$ is in $\Omega, i \in 1..n,$

$l_j < l_k$ for $1 \leq j < k \leq n,$ and that

$\Omega \vdash_n o_i : \mathsf{Chop}(\mathsf{Chop}(L(M_s), \mathsf{Class}(o)), l_i) \rhd H_i$ is derivable for all $i \in 1..n$.

There are two cases. First, if we have $\mathsf{Chop}(L(M_s), \mathsf{Class}(o)) = \emptyset$, then also $\mathsf{First}(\mathsf{Chop}(L(M_s), \mathsf{Class}(o))) = \emptyset$, so $n = 0$, and $H = o$. Moreover, there is no u such that $s \overset{\mathsf{Class}(o)}{\to} u$ is in M, so $P_M(\mathsf{Class}(o), s) = \langle\rangle$, and hence $\Omega \vdash_c o : s : P_M \rhd o$ is derivable, which is the desired conclusion.

Second, if $\mathsf{Chop}(L(M_s), \mathsf{Class}(o)) \neq \emptyset$, then $s \overset{\mathsf{Class}(o)}{\to} u$ is in M_s for some u, and

$$\mathsf{First}(\mathsf{Chop}(L(M_s), \mathsf{Class}(o))) = \{l_i \mid u \overset{l_i}{\to} s_i \text{ is in } \mathsf{Outgoing}_{M_s}(u)\} \ .$$

Thus, the side condition of the rule for $\vdash_c$ is satisfied. By the induction hypothesis, $\Omega \vdash_c o_i : s_i : P_M \rhd H_i$ is derivable for all $i \in 1..n$. We conclude that $\Omega \vdash_c o : s : P_M \rhd H$ is derivable.

The converse is proved similarly.

By combining Theorem 1 and Theorem 2, we obtain our compiler correctness result.

Corollary 1. *For a class graph G, a traversal specification D, a deterministic automaton M_s which accepts $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$, an object graph Ω, a node o in Ω, and a traversal history H, we have*

$$\Omega \vdash_s o : \mathsf{PathSet}_G(D) \rhd H \quad \text{if and only if} \quad \Omega \vdash_c o : s : P_M \rhd H \ .$$

In summary, compilation of an adaptive program proceeds by first computing an automaton M which accepts $\mathsf{Simplify}(\mathsf{PathSet}_G(D))$ and then generating the program P_M.

5 Conclusion

We have presented a general compiling algorithm for a core language of adaptive programs. The algorithm generates efficient code, but the algorithm itself may in some cases be slower than the previous algorithm of [11]. In future work, we will attempt to combine the two approaches, by using [11] as a source of ideas for optimizations that apply in useful cases.

References

1. Version 5.5 of the Demeter Tools/C++, which generates C++ code, is available through the Demeter home page: `http://www.ccs.neu.edu/research/demeter/`.

2. Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.

3. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

4. Walter L. Hürsch and Linda M. Seiter. Automating the evolution of object-oriented systems. In *International Symposium on Object Technologies for Advanced Software*, 1996. To appear.

5. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

6. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0–534–94602–X.

7. Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

8. Karl J. Lieberherr and Cun Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.

9. Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. Technical Report NU-CCS-95-22, College of Computer Science, Northeastern University, 1996.

10. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

11. 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.