Theory and Practice of Object Systems 3(2):75-85, 1997.

Class-graph Inference for Adaptive Programs

Jens Palsberg[*]

Purdue University

Abstract

Software generators can adapt components to changes in the architectures in which the components operate. The idea is to keep the architecture description separate and let the software generator mix it with specifications of each component. Adaptation is done by regeneration: when the architecture changes, the components are regenerated.

A software component will usually be written with a particular architecture in mind. This raises the question: how much has it committed to the particular structure of that architecture? To put it in a nutshell: How flexible is a given software component?

In this paper we study this question in the setting of Lieberherr's adaptive programming. Lieberherr uses class graphs as the architecture and so-called adaptive programs as the software components. We present a polynomial-time class-graph inference algorithm for adaptive programs. The algorithm builds a representation of the set of class graphs with which a given adaptive program can work. It also decides if the set is non-empty, and if so it computes a particularly simple graph in the solution set. Several toy programs have been processed by a prototype implementation of the algorithm.

[*]Jens Palsberg, Dept of Computer Science, Purdue University, W Lafayette, IN 47907, USA; email: `palsberg@cs.purdue.edu`.

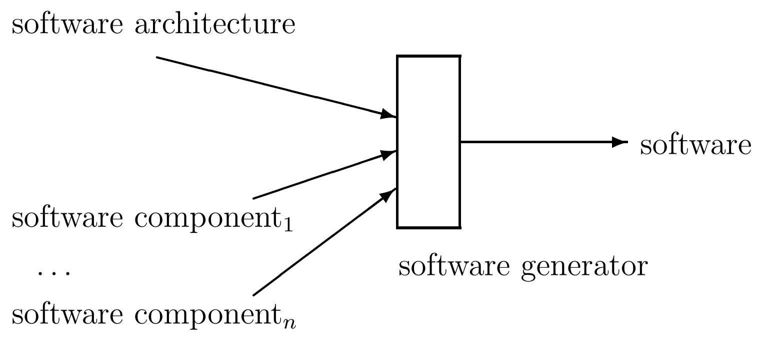

Figure 1: Software generator.

1 Introduction

1.1 Background

Software generators can adapt components to changes in the architectures in which the components operate. The idea is to keep the architecture description separate and let the software generator mix it with specifications of each component. Adaptation is done by regeneration: when the architecture changes, the components are regenerated. The functionality of such a software generator is illustrated in Figure 1.

There are several approaches to software generation in this style, including:

- **Databases.** In a database, the architecture is the schema, and a software component is a query [3]. When the schema changes, a query need not change but may have to be recompiled.
- **Software architecture.** Baltzer studies multi-targeted program generators [2] and states about regeneration:

  > "This will allow these generated components to remain compliant as their architecture evolves and to become reusable assets that can be adapted for use in multiple architectures. By eliminating the need to manually revise code generators for each architectural change, this component adaptation technology will remove major cost, expertise, and predictability barriers to the use of synthesis technology in complex systems and place adaptation to an evolving architecture on an equal footing with evolution of the component itself" [2].

- **Aspect-oriented programming.** Kiczales and his coworkers study aspect-oriented programming where the code of several components is woven together by a software generator [10]. Typically, one of the components provides the basic functionality and the others enhance the behavior in various ways.

```
OPERATION print-nested
   TRAVERSE
      [Exp,Comp,Num]
   WRAPPER Num
      (@ print value @)
```

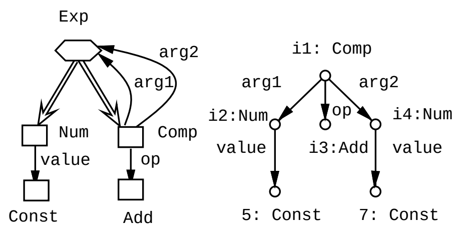

Figure 2: Left: an adaptive program. Middle: a class graph. Right: an object graph.

- **Adaptive programming.** Lieberherr's adaptive programming uses class graphs as the architecture [12]. The class graph represents subclass relationships and the types of instance variables. Software components are written in a style that relies as little as possible on the class graph.

A software component will usually be written with a particular architecture in mind. This raises the question: how much has it committed to the particular structure of that architecture? To put it in a nutshell:

> **Fundamental question.** How flexible is a given software component?

Related questions: With which architectures will the component work? Do two given components have compatible sets of architectures that they can work with?

In this paper we study this question in the setting of Lieberherr's adaptive programming [12]. We present an algorithm which, when given a component, produces a summary of the architectures that the component can work with, or decides that none exists. The algorithm is in linear time and is simple to implement. This makes us optimistic that similar algorithms may be found for other settings than adaptive programming. One application might be for databases where multiple views can be defined. Our algorithm may help generate an integrated view for different queries.

Let us first take a closer look at adaptive programming and restate the above question in that setting.

1.2 Adaptive Programming

The goal of adaptive programming is to make programs adaptable to changes in the forms of their data structures. In essence, the idea is to program in a style where code may remain untouched when the forms of data structures change.

In object-oriented programs, each data structure is an object and each object is instantiated from a class. In such a program, the form of an object is given by the types of its instance variables and the subclass relationships in the program. (We assume a C++ style type system where each type is a class name.) This structural information can be summarized in a *class graph* where each node is a class, and each edge either indicates the type of an instance variable or a subclass relationship. For example, see Figure 2. The class graph contains five nodes that each represent a class, it contains subclass edges (double arrows) that indicate that `Num` and `Comp` are subclasses of `Exp`, and it contains construction edges (single arrows) that indicate that, for example, the class `Num` has an instance variable named `value` of type `Const`.

The approach to adaptive object-oriented programming taken by Lieberherr and his colleagues [12, 13] is to let the class graph be separate rather than having it as an integral part of the program text. Once an actual class graph is given, the adaptive program and the class graph can be compiled to an efficient object-oriented program by a software generator [17, 22]. Thus, if the adaptive program can remain untouched when the class graph changes, then such a change only entails a recompilation.

An adaptive program in the style of Lieberherr uses the constructs of *traversal specifications* and *wrappers*. An example program is shown in Figure 2. The program uses a traversal specification (written after `TRAVERSE`) to denote a set of paths in any given class graph. Different class graphs yield different path sets. The traversal specification `[Exp,Comp,Num]` denotes the set of paths from `Exp` to `Num` that go via `Comp`. This set of paths is then used to traverse objects. Intuitively, a given object and the objects in its instance variables are traversed in a depth-first fashion according the denoted path set. Along the way, the traversal executes wrapper code (written after `WRAPPER`). For example, the wrapper for `Num` is executed each time an object of class `Num` is encountered. In summary, the adaptive program in Figure 2 prints the value field of all `Num` objects nested inside a `Comp` object. If we write the program directly in a traditional object-oriented language, then it looks something like this:

```
CLASS Comp INHERITS Exp              CLASS Num INHERITS Exp
   VAR arg1, arg2:  Exp
   METHOD print-nested                  METHOD print-nested
      arg1.aux; arg2.aux                   — No code here
   END                                  END
   METHOD aux                           METHOD aux
      arg1.aux; arg2.aux                   print value
   END                                  END
END                                  END
```

Suppose now that we extend the class graph in Figure 2 with a new class `If` which becomes a subclass of `Exp` and has three outgoing constructions edges `cond`, `then`, and `else`, all to `Exp`. The adaptive program stays the same, so all we need to do is to regenerate.

A class graph contains most of the type information which usually is written explicitly in a, say, C++ program. Given an adaptive program and a class graph, it is straightforward to define type correctness by a predicate

$$\text{check} \quad : \quad \text{Program} \times \text{Class Graph} \rightarrow \text{Bool} \ ,$$

as shown in Section 3. For example, consider an assignment

```
x := y
```

where `x` and `y` are instance variables. We can type check this assignment by first looking up the types of `x` and `y` in the class graph, and then checking that in the class graph the type of `y` is a subclass of the type of `x`.

Let us now restate the question "how flexible is a software component?" in the setting of class graphs and propagations patterns. The class graph defines the types of data that are manipulated by the propagation patterns, so flexibility and typability are closely related. The question can be stated as follows.

> **Fundamental question.** Given an adaptive program, with respect to which class graphs is it type correct?

In particular, does there at all exist such a class graph? We call this the *class-graph inference problem.* This problem is akin to type inference for object-oriented programs, but it seems harder because both the type of instance variables and the subclass relationships are unknown and must be inferred.

The difference between the problems of type checking, typability, and class-graph inference is made explicit by stating the problems in the following way:

- **Type checking.** Given a class graph, is the program type correct with respect to that graph?
- **Typability.** Does there exist a class graph such that the program is type correct with respect to that graph?
- **Class-graph inference.** Find all class graphs with respect to which the program is type correct.

In Section 3 we will formalize these questions. Traditional type checking and type inference algorithms handle the first question. In this paper we address the last two questions. The key difference between the first and the last two problems is that in the former case the subclass relationships are known, but in the latter case they are not. All published algorithms on type checking and type inference for object-oriented programs, for example [5, 6, 15, 20, 18, 19, 1, 14, 7], rely on knowing the subclass relationships, and if there is a separate notion of typing and subtyping, then, of course, the algorithms rely on knowing the definition of subtyping. Hence, they do not apply to the typability and class-graph inference problems that we study in this paper.

We present a polynomial-time algorithm for typability and class-graph inference. It first derives a set of type constraints from the program text, and from them it builds a representation of the wanted set of class graphs. It also decides if the set is non-empty, and if so it computes a particularly simple graph in the solution set. Several toy programs have been type checked by a prototype implementation of the algorithm.

If several parts of an adaptive program are developed independently, then fitting the pieces together will in practice mean finding a common class graph. In general, there need not be a "most general" class graph with which a program is type correct, so even when candidate graphs for each piece are available, they may be of little use. Our algorithm can take the combined program and generate the wanted set of class graphs.

The above questions were first addressed by Pedersen in his Master's thesis [23] (I was the advisor of Pedersen). The approach of Pedersen was to use the notion of control-flow analysis of Palsberg and Schwartzbach [18, 19] (in [18, 19] this analysis is called "type inference") to generate approximate type information. From this information, he built an approximate solution to the class-graph inference problem. There are cases where the second part of his algorithm must give up and not produce a result. We will use the same example language as Pedersen, but our algorithm will give an exact solution, and it will be in $O(n)$ time where n is the size of the program (Pedersen's algorithm is in $O(n^3)$ time). Both Pedersen's and our approach use constraints, but they are different forms of constraints. Pedersen used constraints for control-flow analysis and we use constraints that define typability in a traditional C++-style way.

In the following section we recall the basic notions of adaptive programming, in Section 3 we define type correctness, and in Section 4 we present our class-graph inference algorithm.

2 Adaptive Programs

Our example language is called Adaptive-BOPL and it can intuitively be understood as follows:

Adaptive-BOPL = BOPL + traversal specifications + wrappers.

BOPL is the object-oriented language studied by Palsberg and Schwartzbach in [19]. In this section we briefly summarize the syntax and semantics of Adaptive-BOPL, following Pedersen [23].

BOPL is a small language with classes, objects, methods, instance variables, assignments, message sends, integers, booleans, self, super, and a few other standard constructs. Here is part of the grammar for BOPL expressions:

Exp	`::=`	`self`	(the self metavariable)
	|	Identifier	(variable or argument)
	|	Identifier `new`	(instantiation of a class)
	|	Identifier `:=` Exp	(assignment)
	|	Exp `.` Identifier `(` $\text{Exp}_1, \ldots, \text{Exp}_k$ `)`	(message send)
	|	...	

In the style of C++, each type is generated from the grammar:

Type	`::=`	`Void`	(the type of nil)
	|	`Int`	(the type of integers)
	|	`Bool`	(the type of booleans)
	|	C	(C is a class name)

For example, if a parameter has type `C`, where `C` is a class name, then we can pass arguments that are instances of `C` or subclasses of `C`.

Notice that "type = class" and "subtyping = subclassing." In its published form, Lieberherr's adaptive programming [12] is fundamentally based on these design choices. As such, type checking is inherently more restrictive than in some languages where types and classes are separate, for example, Bruce's language TOOPLE [5, 4]. It is not clear how to add, say, a function type constructor or an object-type constructor to the adaptive programming framework, and it is not clear if the algorithm presented in this paper could be extended to such a setting.

We use the grammars for BOPL expressions and BOPL types as the basis for creating Adaptive-BOPL. On top of that we put the notions of class graph, object graph, traversal specification, and wrapper from [17, 22], as follows.

2.1 Class graphs and object graphs

A class graph is a finite directed graph, where each node is a type, and each edge either represents a subclass relationship or the type of an instance variable. An edge of the form $v \Leftarrow u$ is a subclass edge which represents that the class v is a subclass of u. If G is a class graph and there is a path of subclass edges from u to v, then we write $v \leq_G u$. (Notice that we represent "v is a subclass of u" by both the notation of Lieberherr et al, $v \Leftarrow u$, and by the more often used notation $v \leq_G u$.) An edge of the form $u \xrightarrow{l} v$ is a construction edge which represents that the class u has an instance variable with name l and with type v. We require that the outgoing construction edges from a node are labeled with distinct labels. This corresponds to the requirement found in most languages that the names of instance variables in a class must be distinct. The nodes are divided into three disjoint sets: the abstract classes, the concrete classes, and $\{\texttt{Int}, \texttt{Bool}, \texttt{Void}\}$. We require that in the expression "C `new`", C must be the name of a concrete class. This corresponds to the principle found in, for example, Java that one cannot create instances of an abstract class.

In accordance with many object-oriented languages, we require that the subclass edges form an acyclic relation, that there are no subclass edges to or from `Bool` or `Int`, and that there are no subclass edges from `Void` (these requirements were not present in [17, 22].) Following [17, 22] we also require that only abstract classes can have outgoing subclass edges, and that only concrete classes can have outgoing construction edges. We have imposed these restrictions to make the set up for the class-graph inference problem realistic. As we shall see, without these restrictions the problem is trivially solvable. An explanation of this point will be given after we have completed the precise definition of the class-graph inference problem, see Section 4.

Examples of the above concepts are provided by the class graph in Figure 2, where `Exp` is an abstract class (indicated by a hexagon), and the other four classes are concrete (indicated by rectangles). Clearly, the class graph satisfies all the above requirements.

An *object graph* is a finite directed graph, intended to represent a run-time object structure. Each node represents an object, and the function Class maps each node to "its class", that is, the name of a concrete class. An edge of the form $u \xrightarrow{l} v$ indicates that the object represented by u has an instance variable with name l, and that in that variable is stored the object represented by v. We require that the edges outgoing from a node are labeled with distinct labels.

For example, the object graph in Figure 2 contains six nodes, each representing an object.

2.2 Traversal specifications

A traversal specification denotes a set of paths. For simplicity, this paper considers only traversal specifications of the form

$$[\mathtt{A}_1,\mathtt{A}_2,\ldots,\mathtt{A}_n]$$

where $\mathtt{A}_1,\mathtt{A}_2,\ldots,\mathtt{A}_n$ are class names. This choice is only made for notational convenience; it has no impact at all on the class-graph inference problem that we consider in this paper. Notice that $\mathtt{A}_1,\mathtt{A}_2,\ldots,\mathtt{A}_n$ can be both abstract and concrete. If G is a class graph, then the above traversal specification denotes the set of paths in G that begin in $\mathtt{A}_1$, and then go to $\mathtt{A}_2,\ldots,\mathtt{A}_n$ in that order. Once this set of paths has been computed, it will be used to guide traversals of objects, as explained in the following.

2.3 Propagation patterns

An Adaptive-BOPL program is a collection of *propagation patterns*. Each propagation pattern has the form

```
OPERATION Name(arg_1: Type_1,...,arg_k: Type_k) returns Type
  TRAVERSE [A_1,A_2,...,A_n]
  WRAPPER Classname_i
  PREFIX (@ <BOPL expression> @)
  ...
  WRAPPER Classname_j
  SUFFIX (@ <BOPL expression> @)
  ...
```

where $\mathtt{Classname}_i$ and $\mathtt{Classname}_j$ are names of concrete classes. It can be thought of as a powerful method of the class $\mathtt{A}_1$. Just like a method, it has a header with a name, some arguments and their types, and a return type. It can be invoked by sending the message Name to an instance of the class $\mathtt{A}_1$, or to an instance of a subclass of $\mathtt{A}_1$. This will start a depth-first traversal of that instance, guided by the paths denoted by $[\mathtt{A}_1,\mathtt{A}_2,\ldots,\mathtt{A}_n]$. Specifically, the traversal will go to subobjects that are instances of $\mathtt{A}_2,\ldots,\mathtt{A}_n$ in a depth-first manner. Along the way, some wrapper code may be executed. For example, consider the *prefix* wrapper

```
    WRAPPER Classname_i
    PREFIX (@ <BOPL expression> @)
```

and suppose that an instance of `Classname`$_i$ is encountered during the traversal. In that case, the BOPL expression will be executed right away, before the traversal continues. In the case of a *suffix* wrapper, say

```
    WRAPPER Classname_j
    SUFFIX (@ <BOPL expression> @)
```

the traversal continues until all subobjects of `Classname`$_j$ have been traversed, and only then the BOPL expression is executed.

The invocation of a propagation pattern returns a result. In the header

```
  OPERATION Name(arg_1: Type_1,...,arg_k: Type_k) returns Type
```

the type of the result is specified after `returns`. The expression that evaluates to the result must be the body of a suffix wrapper of the propagation pattern. Thus, the body of each suffix wrapper of the propagation pattern `Name` must be an expression of type `Type`. During a traversal, one suffix wrapper can refer to the result of another suffix wrapper. For example, consider the propagation pattern

```
  OPERATION M() returns T
    TRAVERSE [A,B,C]
    WRAPPER B
    SUFFIX ...
    WRAPPER C
    SUFFIX ...
```

and the class graph

$$A \xrightarrow{b} B \xrightarrow{c} C \ .$$

Suppose the suffix wrapper for C has just been executed and that it produced the value v of type T. The next suffix wrapper to be executed is the one for B, and this wrapper "knows" that it is executed after a traversal starting with the edge labeled c. It can refer to the result of that traversal, the value v, with the identifier `ResMc`. This identifier is composed of three parts. First the fixed "`Res`", then the name of the propagation pattern "`M`", and finally the name of the variable "`c`". In general, one can reference the result of a subtraversal starting in the instance variable, say, `Var`, by writing `ResNameVar`, where `Name` is the name of the propagation pattern.

If we want to write a propagation pattern with name `Name`, traversal specification just `[A`$_1$`]`, and just one wrapper for `A`$_1$ itself, then we can use the syntactic sugar

```
  METHOD A_1.Name(arg_1: Type_1,...,arg_k:Type_k) returns Type
    (@ <BOPL expression> @)
```

Following [17, 22], we assume that the propagation patterns have different names, and that wrappers only can be attached to concrete classes. It is not clear what would be the meaning of a program where a wrapper was attached to an abstract class, so we have not considered such programs.

2.4 Example

For an example of an adaptive program, see the appendix. The program is a hand-translated version of a C++ program from Lieberherr's book [12, p.214]. We have extended it with code that creates a little graph and runs the depth-first traversal, see the body of

```
METHOD Main.Execute .
```

Lieberherr states that the program was written with the class graph in Figure 3 in mind. That class graph contains the classes for vertices, adjacency lists, etc, that are used in the program. The program contains:

- the main method (`METHOD Main.Execute`),
- 12 other methods (`METHOD ...`), and
- 4 propagation patterns (`OPERATION ...`).

A depth-first traversal begins by invoking the method `DFT` of an object of class `Adjacency`. As argument, the method takes the entire graph. The depth-first traversal proceeds by invoking the propagation pattern `DFTmark`, which invokes the propagation pattern `UncondDFT`, which in turn invokes the propagation pattern `Find`. (The fourth propagation pattern `AddAdjacency` is used during creation of the graph.) In outline, the three propagation patterns divide the work between them as follows:

- `DFTmark`: locates unmarked edges.
- `Find`: finds outgoing edges from the current vertex.
- `UncondDFT`: recursively invokes `DFT`.

In each case, the use of traversal specifications relieves the programmer from writing some tedious code, and it makes the program adaptable to changes in the class graph. For example, Lieberherr [12, pp.217] shows how to extend the program to do depth-first traversal of graphs with two kinds of edges. To do this, he changes only the class graph, not the adaptive program.

3 Type Checking

We will now define a notion of type correctness for adaptive programs. It will be done using constraints, and this leads to closely related formulations of type checking, typability, and class-graph inference.

Suppose we are given an adaptive program and a class graph. Type checking proceeds much like in C++, The only difference is that here we have to go to the class graph to find some of the type information, as described in the following.

Let $\mathcal{T}$ denote the set of types. We will use the term *substitution* to mean a mapping from type variables to types. We will also allow a substitution φ to be applied to a type $T \in \mathcal{T}$; in that case $\varphi(T) = T$.

Following [19, Ch.4] we define type correctness using constraints. Related definitions of typability for functional programs can be found in [24, 25, 8, 21, 16, 11]. The approach can be summarized as follows:

1. Introduce a type variable for each expression in the program,
2. Derive constraints from the program text; the constraints are expressed in terms of the type variables, and
3. Solve the constraints, or decide that no solution exists.

The basic idea is that, by definition: "the program is typable if and only if the constraints are solvable." In the setting of adaptive programs there are at least three variations of step (3):

- **The type checking problem.** The class graph is known.
- **The typability problem.** The class graph is not known.
- **The class-graph inference problem.** We want all class graphs for which the constraints are solvable.

Here follow precise definitions of these notions.

Definition 3.1 Assume a set $\mathcal{V}$ of type variables. Let G be a class graph, and let φ be a substitution. A *construction constraint* is of the form $C \xrightarrow{l} V$, where C is a class name, and $V \in \mathcal{V}$. A set Q of construction constraints is satisfied by G and φ, written

$$G, \varphi \models Q$$

provided that for every constraint $C \xrightarrow{l} V$ in Q we have that

$$C \xrightarrow{l} \varphi(V) \text{ is a construction edge in } G\ .$$

A *subclass constraint* is of the form $W \leq W'$, where W, W' are of the forms T or V, where $T \in \mathcal{T}$ and $V \in \mathcal{V}$. A set R of subclass constraints is satisfied by G and φ, written

$$G, \varphi \models R$$

provided that for every constraint $W \leq W'$ in R we have that

$$\varphi(W) \leq_G \varphi(W') \ .$$

An AP-system S over $\mathcal{V}$ is a pair (Q, R) of a finite set of construction constraints Q, and a finite set of subclass constraints R, where all type variables are drawn from $\mathcal{V}$, and where if $C \xrightarrow{l} V$ and $C \xrightarrow{l} V'$ belong to Q, then $V = V'$. An AP-system $S = (Q, R)$ is satisfied by G and φ, written

$$G, \varphi \models S$$

if both Q and R are satisfied by G and φ. □

We will now show how to generate an AP-system from a program P. For simplicity, let us assume that a preprocessor has transformed the program such that all instance variables and parameters have distinct names. Given such a program, we assign a type variable to every expression, and to every declaration of a formal argument, as follows.

The construct:	Written:	Is assigned the type variable:
expression in a wrapper for class C	`e`	$[\![\texttt{e}]\!]_C$
declaration of formal argument	`f`	$\langle \texttt{f} \rangle$

Let $\mathcal{V}(P)$ denote the set of these type variables. Constraints are generated from the program text in the following way (we show the cases from the BOPL grammar above; other expressions can be handled similarly.)

- For every expression of the form `self` occurring in a wrapper for the class C, the subclass constraint

  $$[\![\texttt{self}]\!]_C = C \ .$$

  This constraint reflects that wrappers can be attached to only concrete classes and that concrete classes cannot have subclasses; hence `self` must be an instance of C. If we drop these two requirements, then the constraint would change to $[\![\texttt{self}]\!]_C \leq C$.

- For every instance variable `x` used in wrappers for the class C, the construction constraint

  $$C \xrightarrow{\texttt{x}} [\![\texttt{x}]\!]_C \ .$$

  This constraint reflects that only concrete classes can have outgoing construction edges. Thus, the variable `x` cannot be inherited.

- For every declaration of a parameter `f` of a propagation pattern, where `f` has declared type `T`, the subclass constraint

  $$\langle \texttt{f} \rangle = \texttt{T} \ ;$$

  The constraints for message send, see below, allow subtyping between the types of actual and formal parameters.

- For every reference to a parameter `f` (of a propagation pattern) occurring in a wrapper for the class C, the subclass constraint

  $$[\![\texttt{f}]\!]_C = \langle \texttt{f} \rangle \ ;$$

- For every instantiation of the form `D new` occurring in a wrapper for the class C, the subclass constraint

  $$[\![\texttt{D new}]\!]_C = \texttt{D} \ ;$$

- For every assignment `x:=e`, occurring in a wrapper for the class C, the subclass constraints

  $$\begin{array}{rcl} [\![\texttt{e}]\!]_C & \leq & [\![\texttt{x}]\!]_C \ ; \\ [\![\texttt{x:=e}]\!]_C & = & \texttt{Void} \ ; \end{array}$$

- For every message send $\texttt{e.m(e}_1\texttt{,...,e}_k\texttt{)}$ occurring in a wrapper for the class C, and for every propagation pattern of the form

  ```
  OPERATION m(arg₁: T₁,...,argₖ: Tₖ) returns T
    TRAVERSE [A,...]
    ...
  ```

  the subclass constraints

  $$\begin{array}{rcl} [\![\texttt{e}]\!]_C & \leq & \texttt{A} \\ [\![\texttt{e}_1]\!]_C & \leq & \texttt{T}_1 \\ \ldots & & \\ [\![\texttt{e}_k]\!]_C & \leq & \texttt{T}_k \\ \texttt{T} & \leq & [\![\texttt{e.m(e}_1\texttt{,...,}e_k\texttt{)}]\!]_C \ ; \end{array}$$

  Notice that the first of these constraints is the *only* one that uses information from a traversal specification, and that this information is just the source class. Thus, we can handle any other language of traversal specifications, as long as the source class is always unique.

- For every wrapper in a propagation pattern of the form

```
OPERATION m(arg_1: T_1,...,arg_k: T_k) returns T
  TRAVERSE [...]
  WRAPPER C
  SUFFIX (@ e @)
  ...
```

the subclass constraint

$$[\![\mathtt{e}]\!]_{\mathtt{C}} \leq \mathtt{T} \; ,$$

and for each occurrence in `e` of an identifier of the form `Resmx` the subclass constraint

$$\mathrm{T} \leq [\![\mathtt{Resmx}]\!]$$

and the construction constraint

$$C \xrightarrow{\mathtt{x}} [\![\mathtt{x}]\!]_C \; .$$

Each constraint of the form $A = B$ denotes the two subclass constraints $A \leq B$ and $B \leq A$.

Type checking proceeds by first generating an AP-system and then solving the following type-checking problem. We also define the typability problem and the graph-inference problem. Recall that G ranges over class graphs, S over AP-systems, and φ over substitutions.

- **The type-checking problem.** Given S, given G, $\exists\varphi : \; G, \varphi \models S$?
- **The typability problem.** Given S, $\exists G$, $\exists\varphi : \; G, \varphi \models S$?
- **The class-graph inference problem.** Given S, find all G, $\exists\varphi : \; G, \varphi \models S$.

Solving the type-checking problem is straightforward [19]; solving the two other problems is considered next.

4 Class-graph Inference

In this section we present our class-graph inference algorithm. As a non-trivial example, we consider the program in the appendix. As we shall see, our algorithm produces exactly the class graph in Figure 3 from the program.

Intuitively, our algorithm proceeds by first generating an AP-system from the given program, and then collapsing strongly connected components in that AP-system.

First we show that a solution to the subclass constraints of an AP-system can easily be transformed into a solution to the whole AP-system.

Lemma 4.1 *If (Q, R) is an AP-system, then (Q, R) is satisfiable if and only if R is satisfiable.*

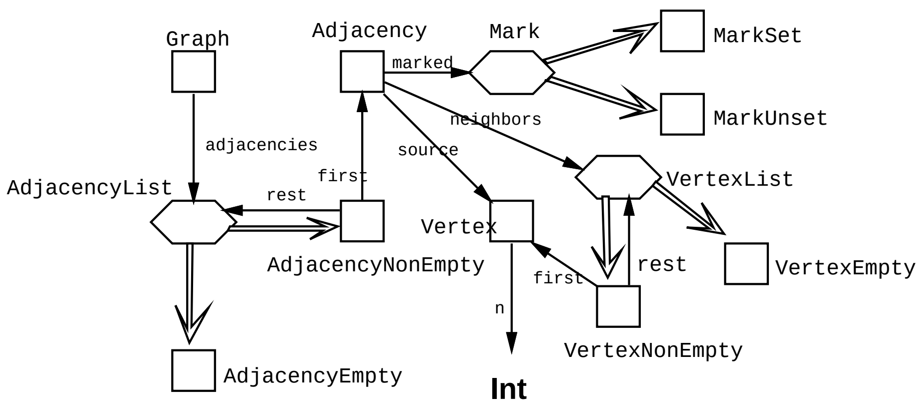

Figure 3: The class graph from Lieberherr's book, p.213.

Proof. By definition, if $G, \varphi \models (Q, R)$, then $G, \varphi \models R$. Conversely, suppose $G, \varphi \models R$. Obtain G' from G by removing all construction edges. Obtain G'' from G' by, for every construction constraint $C \xrightarrow{l} V$ in Q, inserting the construction edge $C \xrightarrow{l} \varphi(V)$, possibly extending the set of nodes. Clearly, $G'', \varphi \models (Q, R)$. □

Next we define a convenient representation of the subclass constraints. The idea is to make a quotient with an equivalence relation that reflects the restrictions on class graphs imposed in Section 2.1.

Definition 4.2 Let R be a finite set of subclass constraints. Let M be the set of types and type variables used in R. The constraints then define a relation on M. Let $\leq^*$ be the reflexive, transitive closure of that relation. For $x, y \in M$, define $x \equiv y$ if and only if

1. $x \leq^* y$ and $y \leq^* x$, or
2. $x \leq^* y$ and $x \in \{\texttt{Int}, \texttt{Bool}\}$, or
3. $x \leq^* y$ and $y \in \{\texttt{Int}, \texttt{Bool}, \texttt{Void}\} \cup \{C \mid C \text{ is the name of a concrete class }\}$.

It is straightforward to prove that $\equiv$ is an equivalence relation. Notice that if $G, \varphi \models S$, and $x \equiv y$, where $x, y \in M$, then $\varphi(x) = \varphi(y)$.

Let $\pi : M \rightarrow M/\!\equiv$ be the standard projection function of elements of M to the $\equiv$-equivalence classes of M. For $m, m' \in M/\!\equiv$, define $m \sqsubseteq m'$ if and only if $\exists x \in m \; \exists y \in m' : x \leq^* y$. Clearly, $\sqsubseteq$ is a partial order on $M/\!\equiv$. We say that $M/\!\equiv$ is *consistent*, provided

$$\text{if } C, C' \in M \text{ are types, and } C \equiv C', \text{ then } C = C'.$$

In other words, if $M/\!\equiv$ is consistent, then each $m \in M/\!\equiv$ contains either one or no types. Thus, no two types are on the same subclass cycle.

If φ is a substitution, and G is a class graph, then $M/\!\equiv$ is satisfied by φ and G, written

$$G, \varphi \models M/\!\equiv$$

if for all $x, y \in M$ with $\pi(x) \sqsubseteq \pi(y)$, we have $\varphi(x) \leq_G \varphi(y)$. $\square$

Theorem 4.3 *$G, \varphi \models R$ if and only if $G, \varphi \models M/\!\equiv$.*

Proof. Straightforward. $\square$

We can now prove our main result which relates satisfiability and consistency. Consistency is easy to check, so via this theorem we get an algorithm for checking satisfiability.

Theorem 4.4 *$M/\!\equiv$ is satisfiable if and only if $M/\!\equiv$ is consistent.*

Proof. Suppose first that $G, \varphi \models M/\!\equiv$. If $C, C' \in M$ are types, and $C \equiv C'$, then we can without loss of generality assume that $C \leq^* C'$. It follows that $\pi(C) \sqsubseteq \pi(C')$, so $C = \varphi(C) \leq_G \varphi(C') = C'$. There are three cases. If $C' \leq^* C$, we get $C' \leq_G C$, so we must have $C = C'$, since G does not contain subclass cycles. If $C = \texttt{Int}$, then we must have $C' = \texttt{Int}$, and thus $C = C'$, since G does not contain subclass edges to $\texttt{Int}$ (similarly if $C = \texttt{Bool}$). The third case is similar. We conclude that $C = C'$, so $M/\!\equiv$ is consistent.

Conversely, suppose $M/\!\equiv$ is consistent. For $m \in M/\!\equiv$, define

$$\mathsf{vertex}(m) = \begin{cases} C & \text{if there is a type } C \in m \\ D_m & \text{otherwise, where } D_m \text{ is a fresh class name.} \end{cases}$$

Let G be the graph that has vertices

$$\{ \, \mathsf{vertex}(m) \mid m \in M/\!\equiv \, \} \ ,$$

and edges defined such that if $\mathsf{vertex}(m), \mathsf{vertex}(m')$ are distinct vertices, then

$$(\mathsf{vertex}(m), \mathsf{vertex}(m')) \text{ is an edge if and only if } m' \sqsubseteq m.$$

Since $M/\!\equiv$ is consistent, we get that G is a class graph. Let φ be the substitution defined such that if $V \in \mathcal{V}$, then

$$\varphi(V) = \mathsf{vertex}(\pi(V)) \ .$$

Clearly, $G, \varphi \models M/\!\equiv$. $\square$

Notice that without the restrictions on class graphs imposed in Section 2.1, we can drop the concept of consistency because the graph G from the proof of Theorem 4.4 will automatically be a class graph (there would be no requirements to satisfy). Hence, in that case $M/\equiv$ is always satisfiable. If other restrictions on class graphs are imposed, then it suffices to refine Definition 4.2, Theorem 4.3, and Theorem 4.4.

The second half of the proof of Theorem 4.4 gives a way of generating a class graph from a satisfiable constraint set. Unfortunately, this graph can be large. We will now define an equivalence relation $\equiv'$ which is larger than $M/\equiv$ and which will lead to a smaller graph. Intuitively, $\equiv'$ merges equivalence classes in $M/\equiv$ if at least one of them does not contain a type, and the collapse of them will not destroy consistency. The idea is that if we draw the Hasse-diagram for the $\sqsubseteq$-ordering, and we find

$$y \longrightarrow x \ ,$$

where x is a $\sqsubseteq$-maximal equivalence class, where there are no other equivalence classes strictly $\sqsubseteq$-larger than y, where y is the $\sqsubseteq$-greatest among all equivalence classes $\sqsubseteq$-less than x, and where x contains only type variables, then we can merge x and y. Similarly for the dual situation where x is a minimal equivalence class. The merging of equivalence classes can be repeated, leading to the following definition.

Definition 4.5 Define $\equiv'$ on M to be the least equivalence relation such that

- if $x \equiv y$, then $x \equiv' y$,
- if (1) $y \leq^* x$, (2) $y \leq^* C$ implies $y \equiv C$ for any type C, (3) $z \leq^* x$ implies either $y \leq^* z$ or $z \leq^* y$, and (4) the set $\{z \mid y \leq^* z \leq^* x\}$ is totally ordered by $\leq^*$, then $x \equiv' y$,
- if (1) $x \leq^* y$, (2) $C \leq^* y$ implies $y \equiv C$ for any type C, (3) $x \leq^* z$ implies either $y \leq^* z$ or $z \leq^* y$, and (4) the set $\{z \mid x \leq^* z \leq^* y\}$ is totally ordered by $\leq^*$, then $x \equiv' y$,

□

Theorem 4.6 *If $\equiv$ is consistent, then $\equiv'$ is consistent.*

Proof. Straightforward. □

We can now summarize the algorithms for typability and class-graph inference. Given a program, first generate the corresponding AP-system. (If n is the size of the program, then there will be generated $O(n)$ constraints using $O(\log n)$ space.) Consider now just the subclass constraints and compute $M/\equiv$. (This mainly involves sorting the elements of M, and then collapsing strongly connected components of M, and also collapsing certain components with the components of `Int`, `Bool`, and `Void`. This can be done in $O(n \log n)$ time.) To decide typability, we check if $M/\equiv$ is consistent.

(This check can be interleaved with the computation of $M/{\equiv}$ without changing the complexity.) Finally, to do class graph inference, note that $M/{\equiv}$ is a convenient representation of all wanted class graphs (Lemma 4.1 and Theorem 4.3.) To obtain a particular graph which we can present to the programmer, compute $M/{\equiv'}$ (in almost-linear time), and use the construction in the second half of the proof of Theorem 4.4 to construct a class graph.

Three toy programs have been type checked by a prototype implementation of the typability algorithm. For the program in the appendix, there is first generated 563 subclass constraints. If we consider a Hasse-diagram presentation of $M/{\equiv}$, then there are 35 edges in that diagram. Finally, the Hasse-diagram presentation of $M/{\equiv'}$ has 6 subclass edges, and moreover, when the algorithm has inserted construction edges as outlined in the proof of Lemma 4.1, then the result is exactly the graph in Figure 3. For two other adaptive programs that are not shown in this paper, the corresponding numbers are 326/12/3 and 173/11/4, as summarized in the following table.

	# lines	# constraints	size of $M/{\equiv}$	size of $M/{\equiv'}$
Program 1	142	563	35	6
Program 2	87	326	12	3
Program 3	73	173	11	4

5 Conclusion

Using a few well-known graph algorithms we demonstrated how to do class-graph inference for adaptive programs. We also presented a technique for generating a small graph that can be shown as an example to the user.

The presented algorithm does not consider the details of the traversal specifications in the adaptive program: it only uses the source node of each traversal specification. If a traversal specification is of the form

$$\texttt{[A}_1\texttt{,A}_2\texttt{,...,A}_n\texttt{]}$$

then it makes sense to require that each of the generated class graphs contain a path from A_1 via A_2, etc, to A_n. This requirement is called compatibility in [12, 22]. Class-graph inference under the requirement of compatibility remains an open problem.

We can view an adaptive program as having the functionality

$$\text{adaptive program} \quad : \quad \text{Class Graph} \times \text{Input} \rightarrow \text{Output}\ .$$

The software generator takes an adaptive program and a class graph and does partial evaluation [9] to produce an object-oriented program with the functionality

$$\text{object-oriented program} \quad : \quad \text{Input} \rightarrow \text{Output}\ .$$

Can we assign a useful type to an adaptive program? This requires a type for both of the arguments (the class graph and the input), and a type for the result (the output). The type of the first argument will denote a set of class graphs. This paper takes a first step in the direction of understanding and generating such sets of class graphs.

Acknowledgments The author thank the referees for many helpful comments on a draft of the paper. This work has been supported by the National Science Foundation under grant number CCR-9402486 (Software Engineering), and by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation). This work was done while the author was visiting MIT in 1995/1996.

Appendix: Example Program

```
// Depth-first traversal program from Karl Lieberherr's book, p.214.

// Operations to maintain the adjacency list
OPERATION AddAdjacency(newAdjacency:Adjacency) returns AdjacencyList
  TRAVERSE [AdjacencyList,AdjacencyEmpty]
  WRAPPER AdjacencyEmpty
  PREFIX (@ (AdjacencyNonEmpty new).InitAdjacencyNonEmpty
                 (newAdjacency,self)
         @)
  WRAPPER AdjacencyNonEmpty
  SUFFIX (@ restA := ResAddAdjacencyrestA; self @)

// Initializer for AdjacencyNonEmpty
METHOD AdjacencyNonEmpty.InitAdjacencyNonEmpty
           (newAdjacency:Adjacency; tailA:AdjacencyEmpty)
           returns AdjacencyNonEmpty
  (@ firstA := newAdjacency; restA := tailA; self @)

// Operations to maintain the vertex list

OPERATION AddVertex(newVertex:Vertex) returns VertexList
  TRAVERSE [VertexList,VertexEmpty]
  WRAPPER VertexEmpty
  PREFIX (@ (VertexNonEmpty new).InitVertexNonEmpty(newVertex,self) @)
  WRAPPER VertexNonEmpty
  SUFFIX (@ restV := ResAddVertexrestV; self @)

// Initializer for VertexNonEmpty
METHOD VertexNonEmpty.InitVertexNonEmpty
           (newVertex:Vertex; tailV:VertexEmpty) returns VertexNonEmpty
  (@ firstV := newVertex; restV := tailV; self @)

// The Depth-first traversal operation

METHOD Adjacency.DFT(g:Graph) returns Void
  (@ marked.DFTmark(g,self) @)

OPERATION DFTmark(g:Graph; adj:Adjacency) returns Void
  TRAVERSE [Mark,MarkUnset]
  WRAPPER MarkUnset
  SUFFIX (@ adj.aPrint(); adj.SetMark(MarkSet new); adj.UncondDFT(g) @)
```

```
OPERATION UncondDFT(g:Graph) returns Void
  TRAVERSE [Adjacency,VertexList,Vertex]
  WRAPPER Vertex
  PREFIX (@ (g.Find(self)).DFT(g) @)

OPERATION Find(v:Vertex) returns Adjacency
  TRAVERSE [Graph,Adjacency]
  WRAPPER AdjacencyNonEmpty
  SUFFIX (@ if ResFindFirstA isnil
            then ResFindRestA
            else ResFindFirstA
            end
         @)
  WRAPPER Adjacency
  SUFFIX (@ if v.equal(source) then self else nil end @)

// Vertex operations
METHOD Vertex.InitVertex(Value:Int) returns Vertex
  (@ n:=Value; self @)

METHOD Vertex.equal(t:Vertex) returns Bool
  (@ n = t.GetValue() @)

METHOD Vertex.GetValue() returns Int
  (@ n @)

// Graph operations
METHOD Graph.InitGraph() returns Graph
  (@ adjacencies := AdjacencyEmpty new; self @)

METHOD Graph.InsertAdjacency(a:Adjacency) returns Adjacency
  (@ adjacencies := adjacencies.AddAdjacency(a); a @)

// Adjacency operations
METHOD Adjacency.InitAdjacency(s:Vertex) returns Adjacency
  (@ neighbors := VertexEmpty new;
     marked := MarkUnset new;
     source := s;
     self
  @)

METHOD Adjacency.SetMark(m:Mark) returns Mark
  (@ marked := m @)
```

```
METHOD Adjacency.InsertVertex(target:Vertex) returns Vertex
  (@ neighbors := neighbors.AddVertex(target); target @)

METHOD Adjacency.aPrint() returns Void
  (@ source.GetValue() print @)

// Main Program: creates a little graph and runs DFT.
METHOD Main.Execute() returns Void
  (@ cg := (Graph new).InitGraph();

     v1 := (Vertex new).InitVertex(1);
     ...
     v10:= (Vertex new).InitVertex(10);

     a1 := (Adjacency new).InitAdjacency(v1);
     ...
     a10:= (Adjacency new).InitAdjacency(v10);

     a1.InsertVertex(v8);
     a3.InsertVertex(v4);
     a3.InsertVertex(v2);
     a5.InsertVertex(v3);
     a5.InsertVertex(v1);
     a6.InsertVertex(v8);
     a6.InsertVertex(v5);
     a8.InsertVertex(v9);
     a8.InsertVertex(v7);
     a9.InsertVertex(v10);

     cg.InsertAdjacency(a1);
     ...
     cg.InsertAdjacency(a10);

     a6.DFT(cg)
  @)
```

References

[1] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. *Software – Practice & Experience*, 25(9):975–995, September 1995. Preliminary version in Proc. ECOOP'93, Seventh European Conference on Object-Oriented Programming, Springer-Verlag (*LNCS* 707), pages 247–267, Kaiserslautern, Germany, July 1993.

[2] Robert Baltzer. Multi-targeted program generators. http://www.isi.edu/software-sciences/multi-gen/multi-gen.html, 1996.

[3] Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, pages 441–451, 1987.

[4] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 285–298, 1993.

[5] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93, ACM SIGPLAN Eighth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 29–46, 1993.

[6] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA'95, ACM SIGPLAN Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 169–184, 1995.

[7] Carl Gunter and John Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[8] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

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

[10] Gregor Kiczales. Aspect-oriented programming. http://www.parc.xerox.com/spl/projects/aop/, 1996.

[11] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

Preliminary version in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[12] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[13] 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.

[14] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[15] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[16] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.

[17] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In *Proc. ESOP'96, European Symposium on Programming*, pages 280–295. Springer-Verlag (*LNCS* 1058), Linkoeping, Sweden, April 1996.

[18] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

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

[20] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. *Science of Computer Programming*, 23(1):19–53, 1994.

[21] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, 1996.

[22] 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.

[23] Carsten Pedersen. Generation of class graphs from adaptive programs. Master's thesis, University of Aarhus, January 1996.

[24] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.

[25] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.