Compiling Adaptive Programs by Partial Evaluation

Peter Thiemann

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

Abstract. An adaptive program is an object-oriented program which is abstracted over the particular class structure. This abstraction fosters software reuse, because programmers can concentrate on specifying how to process the objects which are essential to their application. The compiler of an adaptive program takes care of actually locating the objects. The adaptive programmer merely writes a traversal specification decorated with actions. The compiler instantiates the specification with the actual class structure and generates code that traverses a collection of objects, performing visits and actions according to the specification.
Earlier work on adaptive programming merely stated but never verified that compilation of adaptive programs is nothing but partial evaluation. We employ an algebraic framework based on derivatives of traversal specifications to develop an interpretive semantics of adaptive programming. This semantics is naturally staged in up to three stages. Compilation can be achieved using a standard partial evaluator. Slight changes in the binding-time properties yield several variants of the compiler, by trading compile-time computations for run-time computations.

Key words: object-oriented programming, semantics, compilation

1 Introduction

An adaptive program [16, 18, 15, 19] is an object-oriented program which is abstracted over the particular class structure. Adaptive programming moves the burden of navigating through a linked structure of objects of many different classes from the programmer to the compiler. The key idea is to only specify the landmarks for navigation and the actions to be taken at the landmarks, and leave to the compiler the task of generating traversal code to locate the "landmark" classes and to perform the actions.

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

Second, adaptive programming is attractive for programming in an evolving environment. Here, "evolving" means that classes, instance variables, and methods are added, deleted, and renamed, as customary in refactoring [17, 5, 8]. In this situation, many adaptive programs need merely be recompiled without change, thus alleviating the tedious work of refactoring considerably.

An adaptive program consists of two parts: a traversal specification and wrapper (action) specifications. The traversal specification mentions classes whose objects must (or must not) be visited in a certain order and the instance variables that must (or must not) be traversed. A wrapper specification links a class to an action that has to be performed when the traversal first encounters an object of that class or when it finally leaves the object.

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

1.1 Related work

Due to the high-level programming style of adaptive programs, their compilation is an interesting problem. Palsberg et al [19] define a formal semantics for adaptive programs, formalizing Lieberherr's original approach to compilation [15], and identify a number of restrictions. A subsequent paper [18] removes the restrictions and simplifies the semantics, but leads to a compilation algorithm which runs in exponential time in the worst case. Both papers rely on the theory of finite automata and employ standard constructions, like minimization and the powerset construction (which leads to the exponential worst case behavior).

Finally, Lieberherr and Patt-Shamir [16] introduce further generalizations and simplifications which lead to a polynomial-time compilation algorithm. However, whereas the earlier algorithms perform "static compilation", which processes all compile-time information at compile time, their polynomial-time algorithm performs "dynamic compilation", which means that a certain amount of compile-time information is kept until run time and hence compile-time work is spread over the code implementing the traversal. They employ a different notion of traversal specification than in their earlier work.

The algebraic approach is based on a notion of derivatives which is closely related to quotients of formal languages [9] and to derivatives of regular expressions [6, 7, 4]. The formal underpinnings of this approach are elaborated in a companion paper [23].

Evaluation algorithms for attribute grammars [13, 14] also rely on traversals of tree structures. These traversals are usually specified indirectly as dependences between attribute values. So the paths are inferred from the operations, not vice versa. Many have a notion of "threaded" attributes, which are implicitly passed along by the evaluator to all nodes that mention the attribute [20].

1.2 Contribution of this work

The starting point of this work is a different notion of traversal specification (slightly extended with respect to [23]), which generalizes the earlier work on adaptive programming of Lieberherr, Palsberg and others [15, 19, 18]. In our framework, compiling and running an adaptive program has three steps,

1. normalization of the traversal specification and generation of the set of iterated derivatives;
2. generation of the traversal code from the set of derivatives and the class graph;
3. running the traversal code on an object graph.

Initially, we develop an interpretive semantics of adaptive programs (aka denotational semantics) which implements these three steps. Considered from the point of view of partial evaluation, it is clear that the specification has three stages. In stage one, only the traversal specification is known and the compiler can normalize it and precompute a "state skeleton", that is, the set of states of a finite automaton which implements the traversal of an object graph. In stage two, the class structure becomes available and the compiler can construct the actual traversal code. In stage three, the object graph becomes available and the actual traversal takes place. These stages correspond naturally to binding times in the sense of partial evaluation. We have employed a multi-stage partial evaluation system for Scheme [24] to obtain a staged compiler exactly as outlined above. Thus, we have substantiated the claim of Palsberg et al [18] that "To get an executable program, an adaptive program has to be *specialized*, in the sense of partial evaluation [10], with a complete description of the actual data structures to be used."

Of course, we gain the usual benefits of partial evaluation. Once the interpretive semantics has been verified, the compiler is correct due to the correctness of the partial evaluator.

By varying the staging inside the interpreter, we obtain a compiler either for static compilation or for dynamic compilation. Once the framework is in place, it is easy to experiment with different and more powerful notions of traversal specifications or variations of the semantics. Again, partial evaluation supplies compilers at the push of a button.

Overview Section 2 establishes some formal preliminaries and defines a semantics of adaptive programs. Section 3 defines semantic functions on traversal specifications and wrapper specifications. Finally, it defines the semantics via an interpreter. Section 4 explains the required steps to specialize this interpreter and how to achieve static as well as dynamic compilation. This is followed by a discussion of two variants of the semantics which are readily implemented by modifying the interpreter. Section 6 considers extensions and further work, and Section 7 concludes.

The papers assumes some knowledge of the Scheme programming language.

There is a companion paper which formalizes the algebra of traversal specifications and proves that normalized specifications yield implementations which are minimal in some sense. It is available from the author's Website http://www.informatik.uni-freiburg.de/ thiemann/papers.

2 Semantics of Adaptive Programs

This section first recalls the basic concepts of class graphs and object graphs used to define the semantics of adaptive programs. Then, we define traversal specifications and wrapper specifications and use them to define a semantics of adaptive programs.

2.1 Graphs

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

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

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

2.2 Class graphs and object graphs

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

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

An edge $u \xrightarrow{\diamond} v$ is a *subclass edge*, indicating that v is a subclass of u. Without lack of generality [3, 19, 16] we assume that class graphs are *simple*, i.e., every class is either *abstract* (all outgoing edges are subclass edges) or *concrete* (all outgoing edges are construction edges). In addition, if $u \xrightarrow{\diamond} v \in \mathcal{E}_C$ then v is concrete.

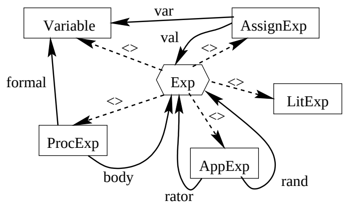

Fig. 1. Example class graph [18]

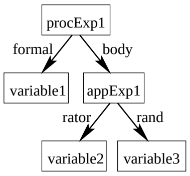

Fig. 2. Example object graph

Figure 1 shows an example class graph with an abstract class *Exp* and five concrete classes *ProcExp*, *AssignExp*, *AppExp*, *LitExp*, and *Variable*. Dashed arrows indicate subclass edges, solid arrow indicate construction edges. Class *Exp* has the remaining classes as subclasses. Class *ProcExp* has two instance variables, `formal` of class *Variable* and `body` of class *Exp*. Class *AssignExp* also has two instance variables, `var` and `val` of class *Variable* and *Exp*, respectively. *AppExp* has two instance variables `rator` and `rand` of class *Exp*.

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

Figure 2 shows an example object graph corresponding to the class structure in Fig. 1. The object *procExp1* has class *ProcExp*, *appExp1* has class *AppExp*, and *variable1*, *variable2*, and *variable3* all have class *Variable*.

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

2.3 Traversal specifications

A traversal specification answers the question "where do we go from here?" at some point of a traversal of an object or class graph. Hence, a traversal specification is either a path to an object of class B, a path leading through an instance variable l, a concatenation of specifications, or an alternative of specifications. Figure 3 introduces the syntax.

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

Fig. 3. Traversal specifications

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

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

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

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

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

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

2.4 Abstract semantics

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

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

where

$$
\begin{array}{ll}
\mathsf{Seq}(o_0, \Pi) & = o_0 \mathsf{Seq}(o_1, \Pi_1) \ldots \mathsf{Seq}(o_n, \Pi_n) \\
\text{where} & \\
\{l_1, \ldots, l_n\} & = \{l \in \mathcal{N} \mid Avlw \in \Pi, v \in (\Diamond\mathcal{C})^*\} \quad l_i < l_{i+1} \\
\Pi_i & = \{w \in \mathcal{C}(\mathcal{N}\mathcal{C})^* \mid Avl_i w \in \Pi, v \in (\Diamond\mathcal{C})^*\} \\
\{o_i\} & = \{o' \mid o_0 \xrightarrow{l_i} o' \in \mathcal{E}_O\}
\end{array}
$$

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

1. the definition of $l_1, \ldots, l_n$ works because of our restriction to simple class graphs, specifically, it is not possible that the traversal misses a construction edge in a superclass because every superclass is abstract;
2. the o_i are uniquely determined in every recursive expansion of $\mathsf{Seq}()$ because the construction edges of the object graph are deterministic.

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

The semantics is quite subtle because the properties of the class graph determine the traversal of the object graph. That means, a traversal continues in the object graph as long as there is a successful path in the class graph. Therefore, parts of the object graph are traversed and wrappers are executed unnecessarily.

2.5 Comparison

In the work of Palsberg and others, a primitive traversal specification has the form $[A, B]$ and denotes the set of all paths from class A to class B in the class graph. In order to construct a sensible semantics, their specifications must be well-formed, which means that there must be a middle point in a concatenation and that sources and targets of the branches of an alternative must coincide. Their formal framework lacks via-paths (as does our companion paper [23]), but their implementation supports them as well as ours does. It is straightforward to provide a translation from their well-formed specifications to ours and to prove that the translation preserves the semantics [22, Sec. 3.1, Lemma 2].

3 An interpretive semantics

The interpretive semantics of adaptive programs takes five inputs, a traversal specification, a wrapper specification, a class graph, the root of an object graph, and a list of parameters for the adaptive program. We develop representations and algorithms for each of these in the Scheme language. Finally, we discuss the implementation of the traversal.

3.1 Traversal specification

We will use a traversal specification to denote the current state of a traversal of a class or object graph. Whenever we track an instance variable or examine the class of a node in the graph, the state of the traversal changes. This change of state is captured by two derivative functions which compute the new state.

The function, $\Phi_X(\rho)$ (pronounced "ρ is final for X"), determines if ρ is a final state for class name X. For a simple path A, X must be a subclass of A to qualify as a final state. A concatenation is final for X if both components are final for X. An alternative is final for X if either component is final for X. A via-path is never final.

The class derivative, Δ, defined in Fig. 4 maps a traversal specification ρ and a class name X to $\rho' = \Delta_X(\rho)$. The traversal specification ρ' is the new

$$
\begin{aligned}
\Phi_X(A) &= X \in \mathsf{Subclasses}(A)\\
\Phi_X(\rho_1 \cdot \rho_2) &= \Phi_X(\rho_1) \wedge \Phi_X(\rho_2)\\
\Phi_X(\rho_1 + \rho_2) &= \Phi_X(\rho_1) \vee \Phi_X(\rho_2)\\
\Phi_X(\{l\}\rho) &= \text{false}\\[1ex]
\Delta_X(A) &= A\\
\Delta_X(\rho_1 \cdot \rho_2) &= \begin{cases} \Delta_X(\rho_2) & \text{if } \Phi_X(\Delta_X(\rho_1))\\ \Delta_X(\rho_1)\cdot\rho_2 & \text{otherwise}\end{cases}\\
\Delta_X(\rho_1 + \rho_2) &= \Delta_X(\rho_1) + \Delta_X(\rho_2)\\
\Delta_X(\{l\}\rho) &= \{l\}\Delta_X(\rho)\\[1ex]
\Delta^{l'}(A) &= A\\
\Delta^{l'}(\rho_1 \cdot \rho_2) &= \Delta^{l'}(\rho_1)\cdot\rho_2\\
\Delta^{l'}(\rho_1 + \rho_2) &= \Delta^{l'}(\rho_1) + \Delta^{l'}(\rho_2)\\
\Delta^{l'}(\{l\}\rho) &= \begin{cases} \Delta^{l'}(\rho) & \text{if } l = l'\\ \{l\}\Delta^{l'}(\rho) & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 4. Derivative of a traversal specification

state. In the case of a simple path A, the derivative is the class itself. For a concatenation $\rho_1 \cdot \rho_2$, if ρ_1 has become final then the new state is the derivative of ρ_2. Otherwise, the new state is the derivative of ρ_1 concatenated with ρ_2. The derivative of an alternative is the alternative of the derivatives. The class derivative of a via-l-path is simply pushed through the via-l-path.

The label derivative, $\Delta^{l'}(\rho)$, maps a traversal specification and a label to a new traversal specification. A simple path A is not affected by traversal of label l'. In a concatenation, the label can only affect the first component. In a union, the label can affect both components. In a via-l-path, the label l removes itself from the traversal specification, recursively. Any other via-path remains in the specification.

The relation of these three functions to the sets of paths upon which the semantics is based is easily stated and proved.

Proposition 1. *If* $(A_0, l_1, A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_0, \rho)$ *then* $(A_1, \ldots, A_n) \in \mathsf{RPathSet}(A_1, \rho')$ *where* $\rho' = \Delta_{A_1}(\Delta^{l_1}(\rho))$*. Furthermore, if* $n = 1$ *then* $\Phi_{A_1}(\rho')$*.*

3.2 Wrapper specification

Figure 5 shows an example of a concrete specification building on Figures 1 and 2 [19]. It defines an adaptive operation `print-free-vars` which traverses an expression object, prints all the bound variables, and returns a list of them.

The line `(traverse variable)` is the traversal specification: visit all objects of class `variable`. There are two wrapper specifications, one for class `procexp` and another for class `variable`. The `procexp` wrapper declares a `prefix` method, which must be executed whenever the traversal *enters* an object of class `procexp`,

```
(define-operation
  (print-free-vars bound-vars)
  (traverse variable)
  (wrapper
   procexp
   (prefix
    (set! bound-vars
          (cons (value-of-field (value-of-field this 'formal) 'id)
                bound-vars)))
   (suffix
    (set! bound-vars
          (cdr bound-vars))))
  (wrapper
   variable
   (prefix
    (let ((name (value-of-field this 'id)))
      (if (not (member name bound-vars))
        (display name))))))
```

Fig. 5. An adaptive program

and a `suffix` method, to be executed whenever the traversal of an object of class `procexp` is finished. The `variable` wrapper defines only a `prefix` method.

The bodies of the methods show a few particulars from our simple object system. Each method can access the current object through `this`. The parameter of the operation is visible and is updated destructively. The function `value-of-field` takes an object and the name of an instance variable and returns its current value in the object.

It is worthwhile mentioning that the wrappers of a `procexp` object are executed regardless whether any `variable` object is reachable through its instances. This is exactly what the abstract semantics (see [19] and also Section 2.4) prescribes. In general, this might be a problem if it is a requirement that wrappers are only executed along successful paths.

3.3 Traversal code

The heart of the traversal code is the procedure `visit` shown in Figure. For simplicity, we only concern ourselves with prefix wrappers, the implementation of suffix wrappers is analogous.

The `visit` procedure accepts three parameters, the current object `this`, the current state of the traversal `state`, and a list of the current values of the operation's variables `parms`. It performs the traversal of the object graph below `this` according to `state` and returns the list the variables' values after completing the traversal. All of these parameters change during the traversal. In addition, `visit` refers to the list of names of the operation's variables in the

```
(define (visit this state parms)
  (let* ((this-class (class-of this))
         (this-state (delta state this-class))
         (children (to-visit this-class this-state))
         (prefix-body (find-prefix-wrapper op-spec this-class))
         (parms0 (if prefix-body
                     (apply
                      ((wrapper-semantics prefix-body) this)
                      parms)
                     parms)))
    (let loop ((children children)
               (parms parms0))
      (if (null? children)
          parms
          (let* ((field-name (car children))
                 (edge-state (lab-delta this-state field-name)))
            (loop (cdr children)
                  (visit (value-of-field this field-name)
                         edge-state
                         parms)))))))

(define (wrapper-semantics formals body)
  (eval `(lambda (this)
           (lambda ,formals
             ,@body
             (list ,@formals)))
        (interaction-environment)))
```

Fig. 6. Naive traversal code

specification, `formals`, and to the wrapper specification to locate the code for the wrappers, `(find-wrapper op-spec this-class)`.

To traverse an object `visit` first determines the class `this-class` of the current object. Next, it computes the new state `this-state` of the traversal using `delta`, which implements the first derivative function. From the class and the state, it computes a list, `children`, of names of instance variables which are to be tracked.

Now `visit` checks if there is a prefix wrapper for `this-class`. If so, it constructs the text of a function that takes the current object and the current values of the operation's variables as a parameter, executes the wrapper, and returns the final values of the variables. The function `wrapper-semantics` maps the text of the wrapper, `prefix-body`, and the names of the variables, `formals` to a corresponding function, the semantics of the wrapper. It accomplishes this mapping using a combination of Scheme's `eval` and `apply` primitives[1]. The resulting semantics is applied to the current object and the current values of the variables. It returns their new values.

Finally, the `children` are visited in the specified order, threading the values of the operation's variables through the computation. Whenever `visit` calls itself recursively, it adjusts the state using `lab-delta` which implements $\Delta^l(\rho)$.

It is remarkable that the procedure `visit` indeed implements the semantics of compiled programs as defined in [19] and Section 2.4. In particular, the whole burden of controlling the traversal is in the function `to-visit`, it does not matter whether or not a state is final.

Clearly, the trivial implementation of `to-visit` which just returns the names of all instance variables of the class is not the one intended by Lieberherr et al. To do a proper job, the function `to-visit` must simulate the traversal starting from the current class and the current state using the class graph and thus check whether *any* final state might be reachable. Here is the specification:

$$\texttt{to-visit}(A,\rho) = \{l \in \mathcal{N} \mid (A, l, \ldots) \in \mathsf{RPathSet}_{\mathcal{G}_C}(A,\rho)\}$$

This specification is simplified with respect to the one in Section 2.4 because A is the class of the current object and hence concrete. Clearly, `to-visit` can be computed by searching for paths in a graph $\mathcal{G}$ where each node is a pair of a class name and a derivative of ρ and there is an l-edge from (A,ρ) to (A',ρ') iff $A \xrightarrow{l} A' \in \mathcal{E}_C$ and $\rho' = \Delta_{A'}(\Delta^l(\rho))$. With this graph, the specification becomes

$$\begin{aligned}\texttt{to-visit}(A,\rho) = \{l \in \mathcal{N} \mid\ & A \xrightarrow{l} A' \in \mathcal{E}_C,\\ & \exists \text{path in } \mathcal{G} \text{ from } (A', \Delta_{A'}(\Delta^l(\rho))) \text{ to } (A'', \rho'')\\ & \text{where } \Phi_{A''}(\rho'')\}.\end{aligned}$$

This function is readily implemented, yet it is inefficient to search for the paths over and over again.

[1] `interaction-environment` is a predefined identifier in R5RS Scheme [11]. It contains the current top-level bindings.

4 Specialization

We employ an off-the-shelf partial evaluator for Scheme [24] for transforming our interpretive semantics into a compiler for adaptive programs. This partial evaluator belongs to the offline guild, ie, it relies on a preliminary binding-time analysis which propagates the binding times of the program's arguments to every expression. Thus it discovers for each expression the earliest time at which it can be evaluated.

First, the adaptive program (traversal specification and wrapper specification) is available. Next, the class graph becomes available, and finally the object graph and an initial object to start the traversal. Therefore, it is natural to consider this specialization task as a three level process.

In the first stage, we can normalize the traversal specification. This normalization is elaborated in a companion paper [23]. In that paper, it is shown that a normalized traversal specification leads to a compiled program with a uniformly minimal number of specialized variants of the `visit` function. In addition, the first stage can precompute a table of all possible derivatives of the traversal specification. Arguably, this stage might be regarded as a preprocessing step. No interesting specialized code is generated, the specializer merely performs the normalization and reconstructs the remaining parts of the program, inserting the precomputed table.

In the second stage, the class graph becomes available. This stage constructs the graph $\mathcal{G}$ mentioned at the end of Section 3.3 and performs all the necessary path computations in advance. At the end, every node of $\mathcal{G}$ is marked whether or not it has a path to a final node. The construction of this graph and the marking phase can be supported by a partial evaluator which memoizes static function calls. Our partial evaluator does not, so the code shown above must be reorganized. Specifically, the `to-visit` function must be tabulated so that the function used in the `visit` procedure is merely a table lookup. Any programmer concerned with efficiency would have programmed in this way from the beginning.

The second stage is more interesting because it creates specialized variants of the `visit` procedure according to the state and the class of the visited object. The specialized code does not mention states and derivatives anymore. The control structure is completely reified in the code. We regard this as the actual compilation step.

In the third stage, the object graph becomes available. All that is left to do is call a suitable entry point – dictated by the object's class – of the specialized traversal code.

4.1 Binding-time improvements

It is a common problem in partial evaluation that code subjected to specialization must be slightly changed so that the partial evaluator correctly infers the intended binding-time properties. The interpretive semantics of adaptive programming is no exception.

```
(define (visit this class state parms)
  (let ((actual-class (class-of this)))
    (let loop ((classes (subclasses class)))
      (if (null classes)
          (error "illegal object graph")
          (let ((this-class (car classes)))
            (if (equal? actual-class this-class)
                (let ((this-state (delta state this-class)))
                  ...)
                (loop (cdr classes)))))))))
```

Fig. 7. `visit` after application of The Trick

However, the required changes are minimal and rely on a standard trick, "The Trick" [10]. The problem is with the following code fragment from Fig. 6:

```
(define (visit this state parms)
  (let* ((this-class (class-of this))
         (this-state (delta state this-class))
    ...)))
```

Considering the binding times, `this` and `parms` are the current object and the operations's variables, which are available in stage 3. The current `state`, however, is already available in stage 2, and so should be all state information. Unfortunately, `this-class` depends on `this`, so it belongs to stage 3, and `this-state` depends on `this-class`, so it also belongs to stage 3. This results in the binding-time analysis annotating almost everything in visit as stage 3 so that no specialization can happen.

The solution is to apply "The Trick" to `this-class` as follows. The procedure `visit` gets an extra argument `class` which is a superclass of the class of `this`. This `class` value belongs to stage 2 because it can be easily extracted from the class graph. With this information we can replace the stage 3 value `(class-of this)` with a stage 2 value, as demonstrated in Figure 7. Thus, we are replacing a single stage-3-test which is known to be a member of `(subclasses class)` by a stage-2-loop over this set.

No further modifications were necessary, except the insertion of a memoization point to improve sharing in the compiled programs. Also the variations considered in Section 5 below do not lead to further problems. In particular, the partial evaluator processes the function `wrapper-semantics` as it stands (including `apply` and `eval` [21]).

4.2 Dynamic compilation

A later work on compiling adaptive programs [16] has introduced dynamic compilation. In this framework (but translated in the terms of this paper), the compiler need not generate different versions of `visit`, but rather the propagation of state information is deferred to the actual traversal. Furthermore, it is not necessary to perform the initial normalization step (it only decreases the number of variants specialized from `visit`), so that only the classic two stages remain.

But what is left to specialize in this situation? Considering the code fragment in Figure 7, we see that only `class` can be propagated before the actual traversal takes place. Therefore, in the specialized program, there will be one variant of `visit` for each class. This variant contains the prefix and suffix methods for the class (if any). It is also possible – using The Trick on `children` – to hard wire the references to the instance variables into the code. We did not pursue this last idea because its outcome did not seem worthwhile.

5 Variations

It is easy to experiment with different variants of the semantics, by modifying the interpreter accordingly. Partial evaluation yields the corresponding compilers for free. This section considers two variants, one which restricts the execution of wrappers only to the nodes on successful paths in the object graph, and one which implements a different semantics of path expressions. The variants are independent, so any combination of them is possible.

5.1 A variation on wrappers

It would be desirable to change the semantics of adaptive programs so that wrappers are only executed on paths that reach a final state. As it stands, a traversal continues in the object graph, executing wrappers, until there is no path in the class graph to a final state. While the traversal cannot be avoided, the execution of the wrappers can be avoided in our approach. The corresponding change in the semantics (Sec. 2.4) is to replace the class graph by the object graph, ie, to use $\mathsf{RPathSet}_{\mathcal{G}_O}(o, \rho)$ to control the traversal instead of $\mathsf{RPathSet}_{\mathcal{G}_C}(\mathsf{Class}(o), \rho)$.

The idea of the implementation is to defer the execution of wrappers as long as it is not clear that the current path reaches a final state. This is easily accomplished in a functional language. The procedure `visit` gets one more parameter `wrapped` and it returns an indication whether or not the traversal has reached a final state, besides the new values of the operation's variables. The parameter, `wrapped`, is the composition of all prefix wrappers up to the last final state encountered in the traversal.

Whenever the traversal enters an object, `visit` composes the prefix wrapper (if any) with the accumulated wrappers in `wrapped`.

```
(let ((new-wrapped
       (lambda parms
         (apply ((wrapper-semantics prefix-body) this)
                (apply wrapped parms)))))
  ...)
```

The newly constructed wrapper contains a "partially applied" use of the wrapper's semantics to the current object `this`. The resulting function maps a list of the variable's old values to their new values.

If the current state is final, `visit` applies `new-wrapped` to the values of the variables. The `wrapped` parameter for traversing the children is the identity function `(lambda parms parms)`. Before returning from a final state, the suffix wrapper (if any) is executed and the result is paired with "true" because the traversal has encountered a final state.

If the current state is not final, `visit` passes the composed wrapper `new-wrapped` down to the traversal of the instances. If any of these traversals returns "true", indicating that it has found a successful path, the suffix wrapper is executed. Moreover, as soon as a traversal has returned "true", the remaining instances must be traversed with the identity function as the `wrapped` parameter. Otherwise, the prefix wrapper of the current state (and the accumulated wrappers in `wrapped`) would be executed more than once.

The actual code is slightly more complicated due to special cases, which arise since prefix and suffix wrappers are optional.

5.2 A variation on traversal specifications

The current semantics of traversal specifiations has a few glitches. For example, the traversal specification $A \cdot A$ is satisfied by the one-element path (A). Therefore, it is not possible to specify that, eg, three instances of A are encountered before a traversal is successful. Likewise, it is not possible to specify a traversal that reaches A so that it first goes through label l and then continues through label l'. The semantics prescribes that $\{l\}\{l'\}A$ is equivalent to $\{l'\}\{l\}A$.

By changing the derivative functions accordingly, we can implement the changes suggested above.

6 Extensions and further work

Our framework is not restricted to single source and target classes. In fact, the "single source" restriction can be lifted by introducing a new abstract superclass from which all source classes inherit. Since we do not insist in well-formed traversal specifications [19], multiple targets arise naturally. These restrictions have also been lifted in the later work on compiling adaptive programs [16].

Further operators like negation and intersection could be allowed for traversal specifications. It is easy to extend the derivative functions accordingly. In the database community [1, 12, 2], more expressive path expressions have been considered, including l^{-1} (find an object o so that the current one is the value of instance variable $o.l$) and $\mu(l)$ (find the closest reachable object with instance variable l) [25]. These would also be interesting to investigate.

Although adaptive programming has been conceived in the context of object-oriented programming, it is perfectly reasonable to also use this paradigm in the context of functional programming. The common sum-of-product types are well-suited for this. For example, in the declaration of an algebraic datatype in Haskell,

data AClass = C_1 l_{11}=t_{11} ... l_{11n}=t_{1n} | ... | C_m l_{m1}=t_{m1} ... l_{mn}=t_{mn}

the adaptive programmer would consider `AClass` an abstract class and `C1`, ..., `Cm` concrete classes with `lij` the names of the instance variables.

7 Conclusion

We have demonstrated that the compilation of adaptive programs is easily implemented using an of-the-shelf partial evaluator. We have experimented with different semantics and also with different stagings. We have found that static and dynamic compilation can be achieved essentially from the same specification by varying the binding times. Thus we have clarified the relation between the two compilation strategies.

References

1. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, 1997.
2. Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 122–133, Tucson, Arizona, May 1997. ACM Press.
3. Paul L. Bergstein. Object-preserving class transformations. In *OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 299–313. ACM, November 1991. SIGPLAN Notices (26)11.
4. Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
5. W. Brown, R. Malveau, H. McCormick, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1998.
6. J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
7. John H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.
8. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
9. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
11. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available electronically as http://www.neci.nj.nec.com/homepages/kelsey/r5rs.ps.gz.

12. Michael Kifer, Wong Kim, and Yehoshua Sagiv. Querying object oriented databases. In Michael Stonebraker, editor, *Proceedings of the SIGMOD International Conference on Management of Data*, volume 21 of *SIGMOD Record*, pages 393–402, New York, NY, USA, June 1992. ACM Press.
13. Donald Ervin Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.
14. Donald Ervin Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 5:95–96, 1971. Correction to [13].
15. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
16. Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of object structures: Specification and efficient implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.
17. W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champain, 1992.
18. Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, September 1997.
19. 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.
20. S. Doaitse Swierstra, P. R. Azero Alocer, and João Saraiava. Designing and implementing combinator languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS-Tutorial*, pages 150–206. Springer-Verlag, 1999.
21. Peter Thiemann. Towards partial evaluation of full Scheme. In Gregor Kiczales, editor, *Reflection'96*, pages 95–106, San Francisco, CA, USA, April 1996.
22. Peter Thiemann. An algebraic approach to compiling adaptive programs. http://www.informatik.uni-freiburg.de/ thiemann/papers/adaptive-report.ps.gz, October 1999.
23. Peter Thiemann. An algebraic foundation for adaptive programming. http://www.informatik.uni-freiburg.de/ thiemann/papers/adaptive-lncs.ps.gz, October 1999.
24. Peter Thiemann. *The PGG System—User Manual*. Universität Freiburg, Freiburg, Germany, February 1999. Available from ftp://ftp.informatik.uni-freiburg.de/iif/thiemann/pgg/.
25. Jan Van den Bussche and Gottfried Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, volume 760 of *Lecture Notes in Computer Science*, pages 267–282, 1993.