Modular Adaptive Programming

A dissertation presented

by

Therapon Skotiniotis

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

August, 2010

Abstract

Adaptive Programming (AP) provides advanced code modularization for traversal related concerns in object-oriented programs. Computation in AP programs consists of *(i)* a graph-based model of a program's class hierarchy, *(ii)* a navigation specification, called a *strategy*, and *(iii)* a visitor class with specialized methods executed before and after traversing objects. Strategy specifications abstract over graph nodes and edges allowing for certain modifications to the program's class hierarchy without affecting visitor behavior. Despite the benefits of AP there are also limitations; hardcoded name dependencies between strategies and the class hierarchy as well as non-modular adaptive code (strategies and visitors). These limitations hamper adaptive code reuse and make composition and extension of adaptive code difficult.

To address these limitations we define What You See Is What You Get (WYSIWYG) strategies, constraints and *Demeter Interfaces*. WYSIWYG strategies guarantee the order of strategy nodes in selected paths simplifying the semantics of strategies and leading to more predictable behavior. We further extend AP systems to enforce the interface between WYSIWYG strategies and visitors by limiting visitor operations to strategy nodes. This limitation makes dependencies between WYSIWYG strategies and visitors explicit. Constraints provide a new mechanism that allows programmers to define invariants on the graph-based model of a program's hierarchy thereby making programmer's assumptions explicit and verifiable at compile time. Finally, Demeter Interfaces provide *(i)* an interface between the program's class hierarchy and both strategies and visitors, *(ii) constraints* on the structure of a class hierarchy that implements a Demeter interface and *(iii)* the ability to parametrize adaptive code.

While our combination of extensions to AP provide modular, reusable and resilient adaptive programs, our definition of WYSIWYG strategies also leads to a new simpler algorithm for calculating all valid paths for a strategy and a straightforward code generation process. Code generation results in a program whose size is polynomial in the size of the strategy and the input program's class hierarchy. For a strategy with k nodes and a program with n classes, our generation algorithm produces at most $(k+2) \times n$ methods.

Acknowledgments

During my time as a graduate student there have been many people that have, in their way, helped and supported me and my work. First and foremost, the members of my dissertation committee, Professors Mitchell Wand, Pete Manolios and Ralf Laemmel. Their support, suggestions and tireless efforts gave me direction and helped me complete this dissertation. My advisor, Karl Lieberherr, whose infinite patience, devotion, and support has been invaluable throughout my tenure as a graduate student. I would also like to thank Professor Matthias Felleisen for his support, and whose work ethic and dedication to his work acted as a catalyst and as an example to follow.

I would also like to thank the members of the Demeter Team, Johan Ovlinger, Doug Orleans, Pengcheng Wu, Jeffrey Palm, Bryan Chadwick and Ahmed Abdelmeged for providing a fertile environment for me to develop as a researcher and person. The members of the Programming Research Laboratory that provided a vibrant and stimulating research environment, I cannot begin to list all the things that I have learned from all of them.

I am also grateful to my friends and colleagues in Boston, Christos Dimoulas, Vassileios Koutavas, Dimitris Vardoulakis, Dimitris Kanoulas and my roommate Evangelos Kanoulas. They are the reason I was able to call Boston my home away from home. They were always there for me.

I would like to thank my family. My parents for their patience, support and understanding. My siblings, Michael and Frosoula for believing in me and for always being there for me. My family in Greece for their support and for not letting me forget who I am and where I come from. Especially, I thank my cousin Michael who has always been by my side, in good and bad times.

Last but not least, I want to thank Stavroula, for her love, support and understanding even at times when I did not deserve it.

I am eternally grateful.

Contents

List of Figures

CHAPTER 1

Introduction

This dissertation provides a new definition and interpretation of strategies (called WYSIWYG strategies) in Adaptive programs and presents the design and specification of two extensions to Adaptive Programming, *Constraints* and *Demeter Interfaces*. These three additions to Adaptive Programming (AP) facilitate modular, reusable and resilient adaptive programs assisting with evolutionary software development. We further show how WYSIWYG strategies provide additional benefits during compilation of Adaptive programs.

1.1 Evolutionary Software Development

Software evolution is omnipresent in software development. The need for bigger more complex software has lead development methods to adopt an evolutionary software development process. Software is thus developed using an iterative and incremental lifecycle [8, 17, 26]. Each iteration adds a new feature on top of the initial core implementation. Evolution is the way by which software is developed today.

Despite the central role of evolution in the software development process writing software that is easy to evolve is difficult. Iterations alter a system's code base in order to:

- introduce a new feature;

- correct errors;
- refactor code, *i.e.*, modify the implementation but not the program's behavior in order to improve technical and/or managerial objectives and
- extend existing functionality to meet changing requirements.

Programmers are required to perform these alterations at the right location(s) in the code base and check the behavior of their alterations both in isolation and in relation with the rest of the system.

To facilitate evolutionary programming paradigms, languages supporting these paradigms provide concepts and mechanisms to assist programmers. To this end Adaptive Programming (AP) [29] has been developed as an extension to Object-Oriented (OO) programming.

1.1.1 The Law of Demeter and Adaptive Programming

Dependencies between software units such as one unit using another or accessing information inside another unit directly affect software evolution [49, 37]. The Law of Demeter (LoD) [27] was developed as a design rule to help limit dependencies that negatively affect software evolution.

The general formulation of the Law of Demeter states [18]:

> Each unit should have only limited knowledge about other units: only units "closely related" to the current unit.

Unit here refers to a programming module, *e.g.*, a function, a method, a class, an object etc. The definition of *closely related* is intentionally left vague so that it can be adapted accordingly. For example, in the case of methods closely related units are the method's arguments and its enclosing class' fields, in the case of a Java class closely related units are the classes in the enclosing package.

We can view the LoD as a more specific case of Low Coupling [15] that aims to reduce coupling between units. Coupling is a measure of how strongly one class is connected to, has knowledge of, or relies upon other classes. In the light of software evolution modifications to classes that exhibit high coupling causes modifications to their related classes leading to a ripple effect throughout the code base. More specific coupling metrics, *e.g.*, coupling between object classes (CBO) [9] and the coupling factor (CF) [11], are used to predict evolution-sensitive code [9]; code that can cause problems upon evolution, or, the estimated effort of managing the impact of changes.

Although OO designs that follow the LoD are easier to evolve, they also lead to class implementations with a large number of methods. These method implementations contain method calls on their closely related objects, which in turn call methods on their closely related objects, etc. Also, there might be an increase in the number of arguments passed to some of these methods. Adaptive programming allows programmers to organize all the methods associated with a particular functional task (or algorithm) into a module which hides the original low-level methods.

As an example consider the class structure for an integration test framework for web pages. Through the use of web testing APIs (*i.e.*, WebDriver [41]) we can represent a web page, along with it contents, as a Java class. WebDriver allows programmers to represent web pages as Java objects, called *PageObject*s and is responsible for communicating between a browser and a PageObject. The communication is bi-directional, Java programs can obtain information from the browser concerning the contents of the currently loaded URL as well perform actions on the web page through the PageObject. Typically, a PageObject contains other PageObjects as fields. Each field represents a segment of the web page under test.

Consider a PageObject class (`Main`) for a web page that decomposes the page's content into five segments; a header, a footer, and three columns, left, center and right column. Each segment is represented as a field inside `Main` that is also a PageObject. The PageObject for the left column (`LColumn`) is further decomposed into a login PageObject (`Login`) and an FAQ PageObject (`FAQ`). The `Login` PageObject contains two fields, one for providing the user's login (`lfield` of type `LField`) and one for providing the user's password (`pfield` of type `PField`). Both `LField` and `PField` are PageObjects attached to the the two text fields on the web page where users input their login and password respectively.

We will consider two test cases for the login operation. The first test case will successfully login and the second test case will provide a malformed email address. In the case where a user provides a malformed email address (*i.e.*, `a@@a.com`) the web page refreshes and sets the user name text field to red. The code for the two test cases needs to gain access to the PageObjects that represent the user name text field and the password text field. In the first test case we provide a valid email and password and click the login button. In the second use case we provide a malformed email and a password, click the login button and then fetch the PageObject that represents the user's login name to verify that it is indeed highlighted red.

Writing a test for the login operation starting from `Main` requires that we access the two fields, `lfield` and `pfield`, inside `Login`. Directly accessing the user login and password field, *e.g.*,

```
main.lcolumn.login.lfield.sendKeys("joe@joe.com")[1]
```

breaks the LoD. If we follow the LoD then the task needs to be encapsulated as a new method, *e.g.*, `setLoginName()` in each of the classes involved. At each class this new method implementation delegates to its immediate neighbor. Once we reach an object of `LField` we then call its `sendKeys()` method. We repeat this process in order to implement a getter method from `Main` to retrieve the PageObject stored in `lfield` and `pfield` respectively.

[1]The method `sendKeys` is provided by WebDriver as a mechanism to send keyboard input to a web page's text field.

```
aspect ToLoginName {
  declare strategy: toLoginName: from Main to LField;
  declare traversal: public void setLoginName(String name): toLoginName(LFieldV);
  declare traversal: public LField getLoginTextBox(): toLoginName(LFieldGetV);
}

import org.openqa.selenium.∗;

class LFieldV {
  String name;
  public LField(String name) { this.name = name; }
  public void before (lField host){ host.sendKeys(this.name); }
}


class LFieldGetV {
  LField res;
  public void before (LField host){ this.res = host; }
  public LField return() { this.res; }
}
```

Figure 1.1: Setter and getter for the login field using DAJ.

We can view any solution that follows the LoD as a composition of two tasks *(i)* find the path to the type(s) we are interested in and *(ii)* perform computation at the points of interest along the path. Knowing the type(s) of interest and the path(s) to reach them the code required to navigate through objects is repetitive, boilerplate and uninteresting. Adaptive programming encapsulates these two tasks and alleviates the programmer from writing the repetitive, boilerplate, code for navigating through objects. To facilitate this separation between navigation and computation AP tools allow programmers to provide *strategy specifications*, that define an abstract path specification, and a visitor [15] class that defines computation to be performed along this path.

Figure 1.1 shows the AP implementation, in DAJ [44], for setting and retrieving the login field starting from an object of type `Main`. The methods `setLoginName` and `getLoginTextBox` are *adaptive methods* that are also introduced into the class `Main`.[2] Both methods use the strategy `"from Main`

[2]An adaptive method is introduced into the strategy's source class, in this case the

to LField", the setLoginName adaptive method uses the visitor LFieldV and the getLoginTextBox adaptive method uses the visitor LFieldGetV. Adaptive methods that consume arguments must be used with visitor's whose constructor method matches these arguments. At runtime the values passed as arguments to the adaptive method are forwarded to the visitor's constructor.

The LFieldV visitor implementation consists of a before method that executes before traversing an object of type LField. DAJ analyzes the class hierarchy to identify paths that satisfy the strategy specification, *i.e.*, a path starting from an object of type Main and terminating at an object of type LField [28]. A call to the method setLoginName first creates an instance of PFieldV, passing any arguments given to the adaptive method to the visitor's constructor method, and then starts the traversal along the calculated path(s) executing any applicable methods in the LFieldV visitor. When traversal reaches an LField object the visitor call the sendKeys method sending the string name as input. The The LFieldGetV visitor implementation is similar but also contains a return method. The return method is called by DAJ at the end of the traversal and the value returned by the return method is the value returned by the adaptive method. With these two adaptive methods in place our test code can now obtain and set the values for lfield inside Login.[3]

AP programs separate navigation and computation along traversals making it possible to reuse strategy specifications and visitor implementations. Furthermore, adaptive methods continue to function[4] as long as modifications to the class hierarchy provide at least one path that satisfies the adaptive method's strategy specification.

There are however limitations to the current AP approach. Strategy

source of the strategy toLoginName is Main.

[3]Similar adaptive methods can be used to obtain and set values for pfield inside Login.

[4]The AP program compiles and runs but may have different behavior.

specifications introduce dependencies on class names. These hardcoded name dependencies make strategy specifications less reusable. Also, visitor implementations rely on certain invariants that strategy specifications sometimes violate.

Consider an addition to the web page's content where we add the capability for first time users to register after they have been send an invitation from an existing user. To accommodate for user registration we add HTML code on the left column of the web page that requires new users to input their email address and a secret invitation key they have received with their invitation in order to start their setup process.

The new HTML code is represented as a new PageObject `Registration` that is part of `LColumn` with a new button for registration. `Registration` contains two fields, `lfield` of type `LField` and `rfield` of type `RField`. Our `setLoginName` method used in our original login test will now input the email address `joe@joe.com` in both instances of `LField` (one under `Login` and one under `Registration`) since the visitor encounters both objects in its traversal. This does not break our login test since the login functionality does not take into account other input fields on the web page. Our test for illegal email address however breaks. Setting an invalid email address that will go to both instance of `LField` encountered while traversing from `Main` to `LField` will cause the first occurrence (from `Login`) to be flagged and colored red. The `getLoginTextBox` adaptive method however will fetch the second instance of `LField` (from `Registration`) that will not be colored red, thus breaking the tests assertion. The implicit assumption that there is one unique `LField`

These limitations admitted by AP decrease code reuse, hamper evolution and make iterative development difficult.

1.2 My Thesis

> WYSIWYG strategies, Constraints and Demeter Interfaces facilitate modular, reusable and resilient Adaptive programs that better support iterative software development. Furthermore, WYSIWYG strategies express a subset of valid paths that is more intuitive and allows for simpler code generation than previous approaches.

The adoption of WYSIWYG strategies provides a simple strategy semantics and guarantees the order of strategy nodes in valid paths. WYSIWYG strategies define an interface between the program's structure and the visitor's behavior. The strategy exposes only strategy nodes to the visitors and in a specific order, shielding visitor code from modifications to the intermediate nodes increasing reusability and resilience. The restriction on the order of strategy nodes that is set by WYSIWYG strategies simplifies the code generated by AP tools.

Constraints are used to define properties (uniqueness of a subpath and absence of subpaths) on the paths matched by a strategy. A constraint specification makes explicit the invariants that a visitor implementation expects from a path. Constraints are statically checked and provide early detection of inappropriate uses of adaptive code.

Demeter interfaces encapsulate an abstraction of a program's data structure (an interface class graph) along with strategy specifications. The abstraction provided by the interface class graph facilitates reuse. The clear separation between adaptive code and the program's structure leads to more modular adaptive programs.

1.2.1 Outline and research contributions

This dissertation is structured as follows:

- The following chapter (2) introduces adaptive programming and its limitations in detail along with an introduction to WYSIWYG strategies, constraints and Demeter interfaces and their usage.

- In Chapter 3 we define strategy automata and their properties and give the definition of our algorithm for calculating all valid paths given a strategy and a class graph along with our analysis for its runtime complexity. We also provide an abstract definition of our generation algorithm along with our definitions for object graphs and object graph slice. The chapter concludes with our proof for object path correctness that shows how our generation algorithm selects the correct object paths for a given strategy $\mathcal{SG}$ and class graph $\mathcal{CG}$.

- Chapter 4 defines our AP language, APCORE , and gives its semantics as a translation to CLASSICJAVA . We further provide a type preservation theorem and its proof showing that given a well typed APCORE program of type t our generation algorithm generates a well typed CLASSICJAVA program of type t.

- Demeter Interfaces are introduced in chapter 5. The chapter includes examples and a study based on the implementation of a design by contract system for Java implemented using DIs.

- The dissertation concludes with a section on related work and future directions.

The main contributions of the research presented in this dissertation can be summarized as follows:

1. definition of WYSIWYG strategies, constraints and Demeter Interfaces

2. a formal model for AP with WYSIWYG strategies and constraints defined as a translation to CLASSICJAVA

3. a new algorithm for calculating the valid paths p in a class graph $\mathcal{CG}$ given a strategy $\mathcal{SG}$.

4. a polynomial code generation algorithm that does not rely on runtime information during object traversal. While previous attempts to encode traversals without runtime information resulted into exponentially large programs, the new algorithm generates programs whose size is polynomial to the size of their WYSIWYG strategy and the input program's class hierarchy.

CHAPTER 2

Adaptive Programming

In this chapter of the dissertation we informally describe DAJ [44], one of the tools that supports AP, and discuss benefits and limitations of AP through examples. The chapter concludes with an introduction to each of our extensions to AP, WYSIWYG, constraints and Demeter interfaces, and how these extensions tackle the current limitations of AP.

2.1 Programming in DAJ

DAJ programs consist of:

- a set of Java class definitions,
- a set of *traversal* files,
- a set of specialized *visitor* classes and
- a *class dictionary*.

DAJ uses and extends AspectJ [42]'s inter-type declarations. A traversal file is an AspectJ aspect that defines strategy and traversal definitions as inter-type declarations. Visitors in DAJ are an extension to the standard visitor design pattern [15] and allow for three special methods, `before` and `after` methods are called before and after traversing an object and a `return` method called after traversal completion.

A class dictionary is a textual representation of the program's class structure and has a dual role in an AP program; a class dictionary defines a graph based representation of the class hierarchy and can be used to define an LL(k) grammar for a language. DAJ generates a lexer and parser from the class dictionary that can be used to parse sentences in the language defined and create object instances.

We first give a short introduction to AspectJ's inter-type declarations and then return to class dictionaries, traversal files and DAJ visitors. A more detailed exposition of AspectJ and its inter-type declaration feature is available in the AspectJ Programmer's Manual [43].

2.1.1 AspectJ's inter-type declarations

AspectJ is a general purpose aspect oriented programming (AOP) language implemented as an extension to Java. The AspectJ language provides *aspects* as a new programming construct. Aspects, like classes, can contain methods and fields but also *pointcut definitions*, *advice* and *inter-type declarations*. This subsection will cover inter-type declarations.

Inter-type declarations can be used to

- introduce new methods or fields to a class
- add Java annotations
- alter a class's superclass
- add new interface names to the list of interfaces that a class implements

Figure 2.1 shows a small example aspect that uses inter-type declarations to add the method `sayHello` to the class `Main`. The result of this inter-type declaration is the addition of the method

```
public void sayHello(){System.out.println("Hello");}
```

```
aspect Test {
  public void Main.sayHello(){System.out.println{"Hello"}; }
  declare parents: Main implements Serializable;
}
```

Figure 2.1: Sample AspectJ aspect using inter-type declarations.

inside the definition of class `Main`. The AspectJ compiler signals a compile time error if the addition of `sayHello` conflicts with another method definition with the same name.[1]

Inter-type declarations can introduce `private` methods as well. The modifier `private` limits the usage of this method to the aspect definition that introduced the method. For `private` inter-type declarations the AspectJ compiler internally renames method and/or field names with fresh names avoiding conflicts.

The second kind of inter-type declarations which are of the form `declare` *spec* : *args*, were *spec* is a predefined category, *e.g.*, `parents`, `error` etc. In Figure 2.1 the second inter-type declaration adds the interface `Serializable` to the list of interfaces implemented by `Main`. The class `Main` is responsible for implementing all method signatures in `Serializable`.

The facilities provided by AspectJ's inter-type declarations allow for the addition of methods to existing classes without creating subclasses or directly editing existing code.

2.1.2 Class Dictionaries

A class dictionary is a textual representation of a program's class hierarchy. The class dictionary defines a graph, called a *class graph*, where each class is a node and each inheritance edge expands into two edges; one edge with the superclass as its source and the subclass as its target labeled with $\diamond\downarrow$, and a second edge with the subclass as its source and the superclass as its target

[1]We direct the interested reader to the AspectJ programmers guide [43] for details about errors and conflicts due to inter-type declarations.

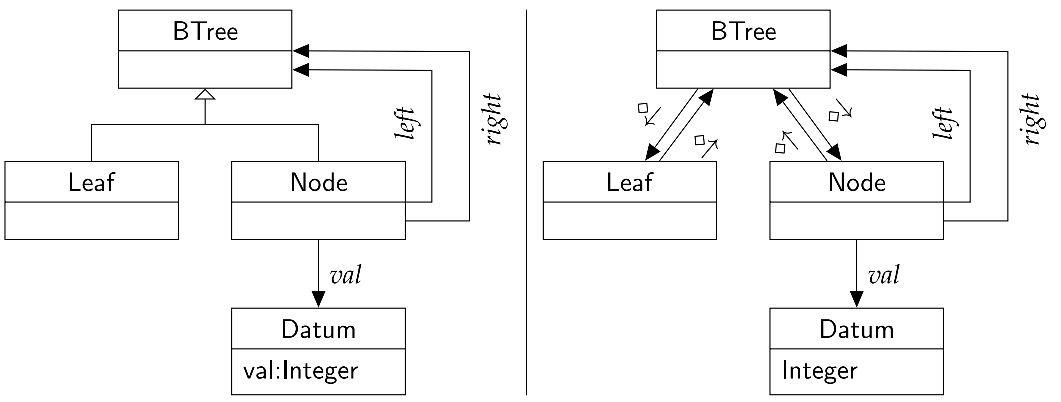


Figure 2.2: UML diagram for `BTree` and its corresponding class graph

```
BTree : Node | Leaf.
Leaf = "leaf".
Node = "data" Datum
          "left" <l> BTree
          "right" <r> BTree.
Datum = <val> Integer.
```

```
data 1
    left leaf
    right leaf
```

Figure 2.3: The class dictionary for binary trees (left) and sample input in the language specified by the class dictionary grammar (right).

labeled with ◇↑ (Figure 2.2). Each containment edge is a unidirectional edge between a class and its member with the field's name as the edge label.

It is not necessary to provide a class dictionary with every DAJ program. DAJ can extract the class dictionary automatically through reflection. A class dictionary defines a subset S of the programs classes and limits the applicability of AP code to the subset S.

However, the class dictionary can also serve as a grammar specification that DAJ uses to generate a lexer and parser. DAJ can then parse valid sentences in the language defined by the grammar and create the corresponding object instances.

Figure 2.3 (left) contains the class dictionary for the binary tree example. Each line in the class dictionary defines an abstract or a concrete class and its terminated by a period. Abstract classes are defined using a colon, `:`, with the name of the abstract class on the left of the colon and the names of the subclasses on the right. Alternative subclasses are defined with a parallel line `|`. Class field names are enclosed in $< >$ and the keyword `common` defines fields inside abstract classes, *e.g.*,

```
BTree :  Node | Leaf common <val> Integer
```

defines the abstract class `BTree` with a field named `val` of type `Integer` and two subclasses `Node` and `Leaf`.

Concrete classes are defined using the equal sign, `=`, with the name of the class on the left of the equal sign and the name of the class's fields on the right. The last line in Figure 2.3 defines the concrete class `Datum` with one field of type `Integer`. Class dictionaries provide special syntax for parametrized classes. `List(A)` $\sim$ `{A}` defines a parametrized list with type parameter `A` that can contain zero or more elements, for example we can create a node with a list of subtrees by changing the definition of `Node` in Figure 2.3 to be

```
Node = "data" Datum "subtrees" List(BTree).
```

Class dictionaries are used to define an LL(k) grammar. We can then create object instances by parsing sentences in the language defined by this grammar. Each line of the class dictionary is a production rule. DAJ provides some predefined terminal classes such as `Integer`, `String` etc.; syntactic tokens are enclosed inside double quotes. For example, in the definition of `Node` in Figure 2.3 the first field `Datum` is preceded by the token `data`, the second field by the token `left` and the third field by the token `right`. Figure 2.3 (right) shows a sample input sentence that will generate a binary tree with one node containing the number 1 as its datum and two leafs as the left and right subtrees.

```
aspect TreeAP{
  declare strategy: toLeafs: from BTree to Leaf;
  declare traversal: public void print(): toLeafs(TPrintV);
}
```

Figure 2.4: Traversal file for Binary Trees

2.1.3 Traversal Files

Traversal files and visitors are defined against the class graph. DAJ extends AspectJ's `declare` statement to include the categories `strategy` and `traversal`. A declare strategy inter-type declaration takes two arguments an identifier (`id`) and a strategy specification (`ss`) separated by a colon. The inter-type declaration binds the strategy specification, `ss` to the identifier `id`. The strategy specification is written in a domain specific language, *e.g.*, in Figure 2.4 `toLeafs` is the name of the strategy and `from BTree to Leaf` is the strategy specification. DAJ supports a from-to notation and a graph-based notation for strategy specifications. The strategy in Figure 2.4 can also be written as `BTree -> Leaf` using graph-based notation. The purpose of strategy specifications is to define abstract paths based on the types of objects. Abstract paths are used to select paths at runtime made up of a sequence of objects connected through containment or inheritance relationships that are *expansions* of an abstract path. A path p' is an expansion of a path p if we can obtain a non-empty prefix of p from p' by removing nodes from p'. Put differently, we can obtain p' from p by adding nodes from the class graph. For example the strategy in `TreeAP` selects all object paths starting from an instance of type `BTree` that lead to an instance of type `Leaf`.

Strategy specifications use directives to include or exclude nodes from paths. Strategy specification directives come in two flavors, class directives that refer to class names and edge directives that refer to edges in the class graph. From-to notation uses `from` to specify the path's source node; `to` to specify the path's target node; `via` to specify nodes or edges that a path should contain; `bypassing` to specify nodes or edges a path should exclude. For example

```
from BTree via Node to Leaf
```

uses class directives to select all paths starting at an object of type `Btree` to an object of type `Leaf` that goes through one (or more) object(s) of type `Node`.

Similarly the strategy

```
from BTree bypassing Node to Leaf
```

selects all paths starting at an object of type `BTree` to an object of type `Leaf` that do not contain objects of type `Node`.

Using the graph based syntax we can define class directives for each edge. Source and target nodes are annotated with `source:` and `target:`, *e.g.*,

```
source:BTree -> Node bypassing { Leaf }
Node -> target:Leaf
```

defines a strategy with `BTree` as the source and `Leaf` as the target. The strategy selects paths starting from a `BTree` going through a `Node` with no `Leaf` nodes in between and continues from `Node` to reach a `Leaf`.

Edge directives are of the form `->` *Type*,*fieldName*,*Type*. For example

```
from BTree via (->Node,l,BTree) to Leaf
```

allows paths from an instance of type `BTree` to an instance of type `Leaf` that go through instance(s) of type `Node` that contain a field with name `l` of type `BTree`. We can also define *any* type using the pattern `*` or a set of types as a comma separated list inside curly braces. For example the strategy specification

```
from BTree via {Leaf, Node} to *
```

will select paths starting from `BTree`, going through either object(s) of type `Leaf`, or of type `Node`, or both, and terminate at a node of any type. The language used for strategy specifications allows programmers to abstract over classes, edges and whole subgraphs of the class hierarchy.

2.1.4 DAJ Visitors

DAJ visitors are an extension to the visitor pattern [15] that provide specific meaning to a set of predefined method names, `before`, `after` and `return`. All `before` and `after` methods are one argument methods and *must* return `void`. A `before` method executes before traversing an object that is a subtype[2] of the `before` method's argument type. Similarly, an `after` method executes after traversing an object that is a subtype of the `after` method's argument type. In Figure 2.6, the first `before` method executes before traversing an object that is a subtype of `BTree`.

A `return` method takes no arguments and its return type has to match the return type given in the method signature used in the traversal inter-type declaration. The `return` method executes after all valid paths have been traversed. For example, in Figure 2.5 we define the adaptive method `toString` and uses the `ToStringV` visitor. The adaptive method's return type matches the visitor's `return` method's return type. By replacing the visitor `ToStringV` with `TPrintV` (Figure 2.6) without changing the adaptive method's return type to `void` yields a compile time error.

Strategies are connected to visitors through traversal inter-type declarations. A traversal inter-type declaration takes as input a method signature, a strategy name and the name of a visitor class. The method signature is used to introduce a new method inside the source class of the strategy. We call these methods *adaptive* methods. Arguments to the adaptive methods

[2]The subtype relation is reflexive.

```
aspect TreeAP{
  declare strategy: toLeafs: from BTree to Leaf;
  declare traversal: public String toString(): toLeafs(ToStringV);
}
```

```
class ToStringV {
  int tabs;
  String tostring;

  // properly initialize fields
  public void ToStringV(){
    tabs = 0;
    tostring = new String("");
  }
  protected void writeTabsLn(int t, String s) { tostring.concat(writeTabs(t) + s + "\n"); }
  public void before(Leaf host) { writeTabsLn(tabs,"LEAF"); }
  public void before(Node host) {
    tabs++;
    writeTabsLn(tabs, "NODE : ");
  }
  public void before(Datum host) { tostring.concat(host.integer.toString()); }
  public void after(Node host) { tabs −= 1; }
  protected String writeTabs(int t){
    String res = new String("\n");
    while(t > 0) {
      res = res.concat("\t");
      t = t − 1;
    }
    return res;
  }
  // return final result
  public String return() { return this.tostring; }
}
```

Figure 2.5: An AP implementation of `toString` using `return` inside a visitors.

```
class TPrintV {
  int tabs;
  public void TPrintV(){ tabs = 0; }
  protected void write(String s) { System.out.print(s); }
  public void writeTabsLn(int t, String s) { System.out.println(writeTabs(this.t) + s); }
  public void before(Leaf host) { writeTabsLn(tabs,"LEAF"); }
  public void before(Node host) {
    tabs++;
    writeTabsLn(tabs,"NODE : ");
  }
  public void before(Datum host) { write(host.integer.toString()); }
  public void after(Node host) { tabs -= 1; }
  protected String writeTabs(int t){
    String res = new String("\n");
    while(t > 0) {
      res = res.concat("\t");
      t -= 1;
    }
    return res;
  }
}
```

Figure 2.6: A Binary Tree pretty printing Visitor implementation.

must match the arguments to the Visitor's constructor. Values passed as arguments to the adaptive method are used as arguments to the Visitor's constructor. Also, the return type of the adaptive method must be a subtype of the return type of the visitor's `return` method. The value returned by the visitor's `return` method is the value returned by the adaptive method.

The adaptive method's implementation is generated from the strategy specification and the visitor definition. From the strategy specification and the program's class hierarchy DAJ calculates all valid paths and generates the necessary visit methods, pointcuts, advice and calls to the visitor methods inside all the relevant classes.[3] The adaptive method's implementation first creates a new instance of the visitor class, passing the adaptive method's arguments to the visitor's constructor, and then traverses the class hierarchy by calling the generated visit method(s). The binary tree ex-

[3] The details of the algorithm and its implementation can be found in [28].

ample, Figure 2.4, declares a traversal that introduces the method `public void print()` into the abstract class `BTree`. The method's implementation traverses from `BTree` to all `Leaf` objects executing the visitor `TPrintV` along the way (Figure 2.6).

2.2 Benefits of AP

In this subsection we show some of the benefits and pitfalls of AP using as an example a system for manipulating a collection of music media. We take the examples through iterations, extensions and refactorings exemplifying the adaptive nature of AP.

2.2.1 A Music Media collection

Our running example involves a media collection. Even though the media collection example is at first sight a simple program, we have identified that similar programs are common in multi-tier, specifically three-tier software architectures for building web services. Our media example is a simplification of a recurring use case for systems that rely on web services. We first give a brief overview of a simple multi-tier architecture for web services and discuss a simple use case scenario. We then proceed with our simplified media collection example.

A popular multi-tier architecture (Figure 2.7) for web services is the three-tier architecture which comprised of

Presentation tier The presentation tier is the topmost level of the application that displays information obtained from other tiers. Typically a web server serving static or dynamic content is in the presentation tier.

Application tier The application tier (also referred to as the logic layer) controls the application's functionality and the processing of data.

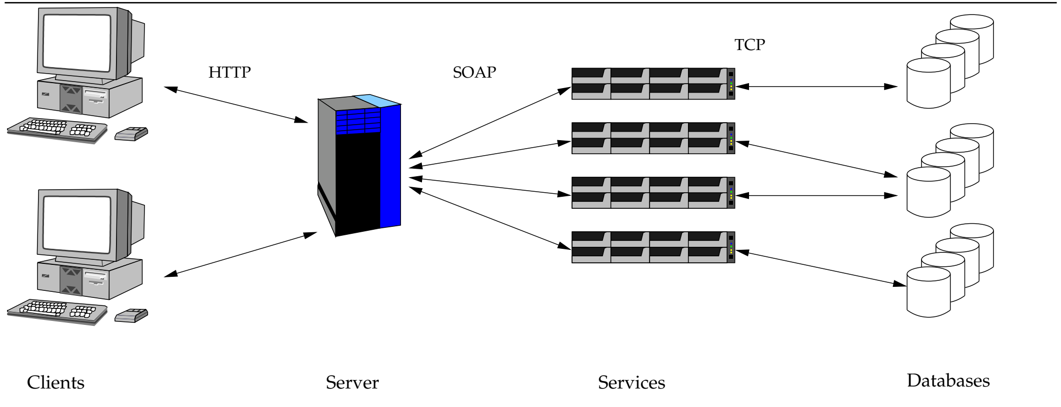


Figure 2.7: A three-tier architecture.

The application tier contains all the services that the presentation tier depends on.

Data tier The data tier is responsible for storing and retrieving data. The data tier typically consists of database servers.

As an example consider an internet site that allows visitors to search, listen to, and, buy music. The site relies on multiple services, provided by different providers that allow for searching and retrieving music content.

The communication between a web server and service providers is governed by an interface that defines the available operations, input data, output data and any exceptions for the service. The interface is defined using WSDL (Web Services Description Language) [46]. The WSDL uses an XML schema to define its interface with supporting tools that generate code, for example in Java, to capture the data structured defined in the interface. On the side of the database services use object-relational mapping libraries (*e.g.*, Hibernate [10]) to map database tables and query results to classes.

Web services typically try to maintain two different data representations, one for clients (requests) and one for providers (databases). For example, consider that all services and database systems are using the same WSDL and database schema. A Java service maintains a set of class files to capture database tables and database query results and a second set of classes for the WSDL interface. The service implementation is responsible for mapping values between the two different data representations. It is important to note that the separation between data representations allows for service implementations to shield clients from changes to the tables and query results.

Consider the case of our music internet site and the operation of searching for songs. The service will provide the WSDL interface that contains a method for searching for music songs and returning a list of matches. Consider the interactions between each tier when a client visits the web site and searches for songs by an artist. The web server receives an HTTP request and fires a request to the service. The web server builds the request by consulting the WSDL interface in order to identify the appropriate method call and build the appropriate data structure for the method's arguments and populate these data structures with HTTP requests arguments.

The service receives the request from the web server and sends the appropriate query to the database. The database returns a set of results to the service. The service receives the query results, performs any necessary processing and then maps the results to the data structure defined by the WSDL method and sends it back to the web server. The web server receives the reply and presents the result to the clients in HTML.

Web services contain data translators to perform the translation from one data representation to another. The implementation of these data translators can be rather large. The number of translators gets even larger when the service has to communicate with multiple different databases that store data under different database schemas. In these situations web services need to have different translators, one for each different database schema. A similar problem occurs on the web server side when the hosted site communicates with multiple different services in order to obtain information (*e.g.*, a meta search site for airline tickets that searches multiple airline companies).

With AP we can implement operations on data representations, as well as implementing translators from one representation to another, with ease. The use of strategies allows programmers to specify the parts of the data representation they are interested in while visitors capture operations on data representations that can be easily reused. Also, AP implementations for operations and data translators adapt to modifications of the data representation (either from the WSDL interface or from the database) making it easier to maintain. We show some of these benefits, and explore some of the shortcomings, of AP using a simple music collection example.

A music collection is made up of zero or more music media. Our first iteration deals with CDs and DVDs each with a title and artist as attributes. CDs contain a list of tracks and a price where tracks consist of a song title and the song's duration in minutes and seconds. Price is an integer representing the cost of a CD. DVDs consist of two song lists and a price.

Figure 2.8 defines the class dictionary for our media collection that includes CDs and DVDs.[4] We define the following operations as adaptive methods for our media collection:

- count the total number of songs in the collection
- search for a song in the collection given a search string
- pretty print all media; print the title, artist and an enumerated list of songs with the duration of each song
- update the price of a all media by a given amount
- calculate the total cost of the collection.

Figure 2.9 shows the content of `MediaCollectionAP` which defines one adaptive method for each operation. `MediaCollectionAP` defines two strate-

[4]The class `Ident` is provided by DAJ as a wrapper for Java's `String` with extra functionality and operations for parsing.

```
import java.util.*;

MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = "disk" Media MediaCollection.
Media : CD | DVD common "title" Title "artist" Artist.
CD = "tracks" SongList "price" Price.
DVD = "side-1" <a> SongList "side-2" <b> SongList "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.
SongList : MtSongList | ConsSongList.
MtSongList = ";".
ConsSongList = Song SongList.
Song = "song title" Title "duration" Duration.
Duration = <min> Integer ":" <sec> Integer.
```

Figure 2.8: Class dictionary for a collection of music media that includes CDs and DVDs.

gies, the first (`toSongs`) begins at a `MediaCollection` instance and navigates to all reachable `Song` instances; the second strategy (`toPrice`) starts from a `MediaCollection` object and navigates to all reachable `Price` instances.

Each traversal declaration in Figure 2.9 introduces a new adaptive method inside the `MediaCollection` class. The adaptive method `countSongs` takes no arguments and returns the total number of songs reached through the `toSongs` strategy in a collection using the `SongCounterV` visitor (Figure 5.28). The `SongCounterV` extends `CounterV` incrementing `counter` when the traversal reaches a `Song` instance. The adaptive method `searchSongTitle` takes a string `t` as input and returns a `SongList` instance containing all `Song` instances reached through the `toSongs` strategy whose title matches the pattern `.* t .*` using the `SongSearchV` visitor (Figure 2.11).

`SongSearchV` extends `SearchV`; `SearchV` is a general search visitor that provides methods for creating the appropriate regular expression given a string `s` and a method for checking if the string `s` matches the regular expression. `SongSearchV` specializes `SearchV` by checking for a match against the title of any `Song` instance encountered during traversal. `SongSearchV` maintains a local song list and stores `Song` instances whose title matches

```
aspect MediaCollectionAP {
  declare strategy : toSongs : from MediaCollection to Song;
  // Return the total number of songs in a collection.
  declare traversal: int countSongs() : toSongs(SongCounterV);
  // Return a list of songs in the collection that match .* t .*
  declare traversal: SongList searchSongTitle(String t): toSongs(SongSearchV);
  // String representation of all the songs in a collection.
  declare traversal: String asStringCount() : toSongs(AsStringCounterV);

  declare strategy: toPrice : from MediaCollection to Price;
  // Change all prices by delta.
  declare traversal: void updatePriceBy(int delta) : toPrice(UpdatePriceV);
  // Return the total cost of the collection.
  declare traversal: int getTotalPrice() : toPrice(PriceAdderV);
}
```

Figure 2.9: The traversal file that defines operations on a `MediaCollection` as adaptive methods.

```
class CounterV {
  int count;

  public CounterV() { this.count = 0; }
  public CounterV(int i) { this.count = i; }
  public int return() { return this.count; }
}
```

```
class SongCounterV extends CounterV {
  public SongCounterV() { super(); }

  public SongCounterV(int i) { super(i); }
  public void before(Song host) {
    this.count += 1;
  }
}
```

Figure 2.10: `CountV` is a generic counting visitor extended by `SongCounterV` for counting `Song` instances.

the regular expression. The before method defined in `SongSearchV` depends on another adaptive method `getSongTitle` defined in `SongAP` (Figure 2.14). `SongAP` introduces two getter methods, one for title and one for duration, as adaptive methods using two strategies `songToTitle` and `songToDuration`. Each method uses a specialized version of the `StringV` visitor (Figure 2.15) that returns a string representation of a song's title (`IdentV`) and a song's duration (`DurationV`).

The adaptive method `asStringCount` returns a string representation of a collection using the `AsStringCounterV` visitor (Figure 2.12). The visitor `AsStringCounterV` extends the `AsStringV` visitor. The `AsStringV` vis-

```
import java.util.regex.Pattern;

class SearchV {
  protected String s;
  SearchV(String s){
    this.s = s;
  }
  protected boolean match(String regexp,
      String s) {
    return Pattern.matches(regexp, s);
  }
  protected String makeRE(String s){
    return ".*" + s + ".*";
  }
}
```

```
class SongSearchV extends SearchV {
  private SongList res;
  SongSearchV(String s){
    super(s);
    this.res = new MtSongList();
  }
  public void before(Song host){
    if (match(makeRE(s),
             host.getSongTitle())){
      res = new ConsSongList(host, res);
    }
  }
  public SongList return() { return res; }

}
```

Figure 2.11: `SearchV` is a generic search visitor that constructs the appropriate regular expression and provides a method for checking a string against its regular expression. `SongSearchV` extends `SearchV` checking for a match between the pattern and a `Song` instance's title.

itor provides a default implementation for representing a media collection as a string. Visitor `AsStringCounterV` extends `AsStringV`'s behavior and prints an enumerated song list for music media using the adaptive method `stringAndCount` (Figure 2.14). The aspect `SongListAP` introduces the method `stringAndCount` along with the `CounterAndStringV` visitor (Figure 2.12) and navigates from a `Media` instance to all reachable `Song` instances. The `CounterAndStringV` visitor maintains a string representation of each song prepended with the song's index.

The `updatePriceBy` adaptive method takes as input the increment (or decrement) in price for music media and the `UpdatePriceV` (Figure 2.13) visitor updates for each CD and DVD instance the price field accordingly. Finally, the `getTotalPrice` adaptive method adds all price instances in a collection and returns the grand total.

Adaptive programs can accommodate changes to the topology of the class graph for which strategy declarations select non-empty paths. As a first extension we would like to extend our media collection to include

```
class AsStringV extends StringV{
  public void AsStringV() { super();}
  protected String writeTabs(){
    StringBuffer res = new StringBuffer();
    for (int i = 0; i < tabs ; i++) { res.
          append(TAB); }
    return res.toString();
  }
  public void before(Media host){
    tabs++;
    sb.append(host.getTitle() + NL);
    sb.append(host.getArtist() + NL);
  }
  public void after(Media host) { tabs; }
  public void before(Song s){
    sb.append(writeTabs()).
      append(s.getSongTitle()).
      append(TAB).
      append(" [").
      append(s.getSongDuration()).
      append("] ").
      append(NL);
  }
  public String return(){ return sb.
        toString(); }
}
```

```
class AsStringCounterV extends AsStringV
      {
  AsStringCounterV() { super(); }

  public void before(Media host){
    super.before(host);
    sb.append(host.stringAndCount());
  }
  public void before(Song host) { }
}
```

```
class CounterAndStringV extends
      AsStringV {
  private int counter;
  CounterAndStringV(){
    super();
    this.counter = 0;
  }
  public void before(Song host) {
    counter++;
    sb.append(TAB + counter).append(")
          ");
    super.before(host);
  }
}
```

Figure 2.12: Visitors that return string representations of objects.

```
class UpdatePriceV {
  protected int change;
  UpdatePriceV(int i){ this.change = i; }
  public void before(Price p) {
    p.integer =
      new Integer(p.integer.intValue() +
            change);
  }
}
```

```
class PriceAdderV {
  int total;
  PriceAdderV() { this.total = 0; }
  public void before(Price p) {
    total += p.integer.intValue();
  }
  public int return(){
    return this.total;
  }
}
```

Figure 2.13: Visitors to update a CD's or a DVD's price (left) and to sum all media prices (right).

```
aspect SongAP {
  declare strategy : songToTitle : from Song to Title;
  // getter for a song title
  declare traversal: String getSongTitle() : songToTitle(IdentV);

  declare strategy : songToDuration : from Song to Duration;
  //getter for a song's duration
  declare traversal: String getSongDuration() : songToDuration(DurationV);
}
```

```
aspect SongListAP {
  declare strategy : songListToSong : from SongList to Song;
  // string represetation of a list of songs
  declare traversal: String show() : songListToSong(CounterAndStringV);

 declare strategy : mediaToSong : from Media to Song;
  declare traversal: String stringAndCount() :
                    mediaToSong(CounterAndStringV);
}
```

Figure 2.14: The traversal file that defines operations on a `Song` as adaptive methods (top) and the traversal file that defines operations on a `SongList` as adaptive methods (bottom).

```
class StringV {
  protected static String NL = "\n";
  protected static String TAB = "\t";
  protected int tabs;
  protected StringBuffer sb;

  public void StringV() {
    this.sb = new StringBuffer();
    this.tabs = 0;
  }
  public String return(){ return sb.toString(); }
}
```

```
class IdentV extends StringV{
  public void IdentV() { super(); }
  public void before(Title host) {
    this.sb.append(host.ident.toString());
  }
}
```

```
class DurationV extends StringV{
  public void DurationV() { super();}
  public void before(Duration host) {
    this.sb.append(host.min.toString())
      .append(" : ")
      .append(host.sec.toString());
  }
}
```

Figure 2.15: `IdentV` and `DurationV` (bottom) retrieve the title (bottom left) and a song's duration (bottom right).

```
import java.util.*;

MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = "disk" Media MediaCollection.
Media : CD | DVD  | MP3 common "title" Title "artist" Artist.
MP3 = Song.
CD = "tracks" SongList "price" Price.
DVD = "side-1" <a> SongList "side-2" <b> SongList "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.
SongList : MtSongList | ConsSongList.
MtSongList = ";".
ConsSongList = Song SongList.
Song = "song title" Title "duration" Duration.
Duration = <min> Integer ":" <sec> Integer.
```

Figure 2.16: Class dictionary with support for MP3s.

songs that we have in mp3 format. An mp3 file has a title and artist name which represent the album's title and artist's name. The mp3 song however captures a single song with a song title and duration. We extend our existing class dictionary for our media collection to incorporate mp3 files as an extra alternative to the `Media` class (Figure 2.16, modifications are in a grey background). Updating the class dictionary with the one given in Figure 2.16 we obtain a valid AP program. The already defined adaptive methods maintain their original behavior; `countSongs`, `searchSongTitle` and `asStringCount` perform their original task and their results include MP3 instances. The adaptive methods `updatePriceBy` and `getTotalPrice` correctly update CD and DVD instances and are not affected by the addition of mp3s in our media collection.

As a second extension we want to include genres in our representation of our media collection. Our collection becomes a list of genres, *e.g.*, classical, rock, blues, pop, r&b etc. We alter `MediaCollection` to be a list of `Genre`'s where `Genre` is an abstract class with a field holding a list of music media. Each kind of music genre becomes a subclass of `Genre` (Figure 2.17).

```
MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = "genre" Genre MediaCollection.
Genre : Classical | Rock | Blues | POP | RandB common MediaList.
MediaList : Mt | ConsMedia.
Mt = .
ConsMedia = "disk" Media MediaList.
Media : CD | DVD | MP3 common "title" Title "artist" Artist.
MP3 = Song.
CD = "tracks" SongList "price" Price.
DVD = "side-1" <a> SongList "side-2" <b> SongList "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.
SongList : MtSongList | ConsSongList.
MtSongList = ";".
ConsSongList = Song SongList.
Song = "song title" Title "duration" Duration.
Duration = <min> Integer ":" <sec> Integer.
```

Figure 2.17: The class dictionary for a media library with genres.

Replacing our original class dictionary for our collection with the class dictionary from Figure 2.17 gives a valid AP program. Despite the modification to the class dictionary the originally defined adaptive methods still behave the same; they still apply to all the songs and media in our collection. As a small refactoring to the class dictionary in Figure 2.17 we can introduce a new abstract class to capture the common attribute `Price` in `CD` and `DVD`. We add an abstract class `PricedMedia` that extends `Media` and contains a member of type `Price` and we have `CD` and `DVD` extend `PricedMedia`. The originally defined adaptive methods adapt to our refactoring and their implementations will still take into account all media and songs in our collection.

Even though each preceding extension modified the class graph the modification was such that allowed all adaptive methods to accommodate for the change and maintain their original behavior. Each extension altered the set of paths selected by a strategy in our original program. The changes to the paths either

- introduce (or remove) a node to a path and no visitor depends on this node (*e.g.*, `PricedMedia`),
- does not remove nodes from the paths for which a visitor has a before (or after) method,
- does not disrupt the sequence of nodes in a path for which a visitor has a before (or after) method, and,
- introduces new paths for the same strategy that work correctly with the current visitors.

2.3 Pitfalls of AP

It is easy to perform modifications to the class dictionary that yield valid AP programs but generate adaptive methods with undesired behavior.

Consider changing our original media collection example so that instead of representing our collection as a collection of music media containing songs, we represent our collection as a list of songs where each song has an attribute specifying the different media that contain the song (Figure 2.18). Replacing our original class dictionary with the one in Figure 2.18 yields a valid AP program with `asStringCount` adaptive method exhibiting undesired behavior. A call to `asStringCount` results in printing a long list of songs without artist names or titles from CDs and DVDs.[5]

Our modification did not cause strategies to fail, *i.e.*, a strategy with an empty set of valid paths, but has caused a change in the sequence of nodes found in paths from `MediaCollection` to `Song`. `Media` is no longer part of the paths selected by `toSongs` and the visitor used with `asStringCount` (`AsStringCounterV`) has before and after methods defined for `Media`. The problem stems from the fact that DAJ visitors can have before or after methods on types that are not explicitly mentioned in the attached strategy.

[5]The remaining adaptive methods behave as expected.

```
import java.util.*;

MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = Song MediaCollection.
Song = "song title" Title "duration" Duration "media" MediaList.
Duration = <min> Integer ":" <sec> Integer.
MediaList : MtMediaList | ConsMediaList.
MtMediaList = ";".
ConsMediaList = Media MediaList.
Media : CD.
CD = "title" Title "artist" Artist "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.

// Util Classes
SongList : MtSongList | ConsSongList.
MtSongList = .
ConsSongList = Song SongList.
```

Figure 2.18: A media collection is a list of songs with each song having a list of media that contain the song.

Visitors that depend on types of nodes not mentioned in the strategy are brittle. Modifications to the class dictionary that remove nodes that a visitor depends on still result to valid paths selected by the strategy. Visitor implementations that depend on such nodes run the risk of adapting to modifications with undesired behavior.

As another extension to our original media collection example consider the addition of a recommendation list to each `Media` class. A recommendation list is made up of zero or more `Media` instances (Figure 2.19)

Replacing our original class dictionary with the one in Figure 2.19 yields a valid AP program. The behavior of the originally defined adaptive methods changes drastically; given a circular `MediaCollection` object any call to an adaptive method results in a non-terminating execution. Running our adaptive program on an acyclic `MediaCollection` yields duplicating behavior for each media instance that is used as a recommendation, *e.g.*, for an object that contains a CD d and it is also used as a recommendation for

```
import java.util.*;

MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = "disk" Media MediaCollection.
Media : CD | DVD common "title" Title "artist" Artist
                        "recommendations" RecList .
RecList : MtRecList  |  ConsRecList.
MtRecList = .
ConsRecList = Media RecList.
CD = "tracks" SongList "price" Price.
DVD = "side-1" <a> SongList "side-2" <b> SongList "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.
SongList : MtSongList | ConsSongList.
MtSongList = ";".
ConsSongList = Song SongList.
Song = "song title" Title "duration" Duration.
Duration = <min> Integer ":" <sec> Integer.
```

Figure 2.19: A media collection with support for a recommendations list.

another CD d', d is printed twice.

The problem is the generality with which DAJ interprets strategy specifications and allows *any* node to appear as part of a path between two strategy nodes. This general interpretation allows for valid paths that include cycles.[6] Programmers have to rewrite their strategies to include bypassing directives for each node in the strategy in order to avoid infinite paths that contain strategy nodes. Rewriting strategies in this way to avoid cycles becomes cumbersome, brittle and error prone.

2.3.1 Lack of Abstractions in AP

It is important to note that AP lacks abstractions that allow for the reuse of adaptive behavior. Consider our original media collection example and the adaptive method used to search for songs inside our collection. To extend our media collection to allow for searches on media titles we have to create

[6]Infinite paths.

a new visitor, extending `SearchV` and repeat the code in `SongSearchV` but instead of adding a before method for `Song` we have to add it for `Media`. Furthermore, we need to define a new adaptive method similar to `getSongTitle` to obtain the title for a media instance. The new adaptive method is similar to the `getSongTitle` but uses a strategy whose source node is `Media` rather than `Song`. DAJ does not provide any mechanism for abstracting and reusing adaptive code that deals with similar graphs. The lack of abstracting adaptive code decreases reuse and increases maintenance costs.

2.4 WYSIWYG Strategies, Constraints and Demeter Interfaces

Even though AP allows for programs to adapt to data structure modifications we have identified specific shortcomings of AP that interfere with modularity, code reuse, and resilience of AP programs:

- Strategy interpretation is too general. For any strategy s with strategy nodes a and b a valid path can contain any type of node between a and b (including a and/or b). Developers cannot rely on a strict ordering of strategy nodes in selected paths and have to guard their code against unwanted loops in selected paths. Development becomes difficult and error prone.

- Brittle interfaces between strategy and visitor. Even though behavior is defined at the same abstraction level as the strategy, AP tools allow for behavior specifications to directly refer to nodes that are not explicitly mentioned in the strategy. Referring to nodes not explicitly mentioned in the strategy adds a dependency on the part of the path that we anticipate to change. These extra dependencies make the development of behavior specifications difficult.

- Lack of abstractions for AP code (strategy and behavior). Strategies and behavior attached to strategies cannot be easily reused. The names used in the strategy and behavior specification are the same as the names given to the data structures in the program. Reuse across programs that have the same data structure but have chosen different names cannot be easily abstracted and reused. Similarly, data structures within the same program that share similar topology cannot share the same AP code. The hard-coded name dependencies between the program's data structure and AP code limits reuse and increases software maintainability.

To address these shortcomings we extend AP with

- WYSIWYG strategies and constraints; WYSIWYG strategies provide a new way for interpreting strategy specifications while constraints can be used to define path invariants for a class graph,
- enforcing an interface between strategies and visitors; a well typed AP program contains visitors that can only have before or after methods on types explicitly mentioned in their attached strategy, and,
- Demeter Interfaces provide a new mechanism that abstract over common graph structures and adaptive code providing a better mechanism for reuse both within and across programs.

WYSIWYG strategies are a new way of interpreting strategy specifications. Given a strategy s with a set of strategy nodes V, a path p is a valid path if and only if there exists a path p' in s such that p is an expansion of p' and for every node n found between a pair of strategy nodes a and b in p, $n \notin V$. We refer to this notion of expansion as a *pure expansion*.

Constraints are a new extension to AP that allows programmers to explicitly specify extra constraints on the class graph. These constraints are invariants that the programmer expects the class graph to satisfy. Constraints are expressed using strategy specifications and properties about the set of valid paths selected by these strategies, *e.g.*, absence of certain paths and cardinality of the set of valid paths for a strategy.

Demeter Interfaces allow for abstracting adaptive code into a new module. A Demeter Interface is made up of an interface class graph (ICG) –that serves as an abstraction of a class graph– along with strategies and visitors defined against the ICG. A Demeter interface is then mapped onto a class dictionary by providing mappings between the edges and nodes in the ICG to the edges/paths and nodes in the class dictionary. The adaptive code defined in a Demeter interface gets generated according to the mapping provided for the specific class dictionary.

In the remainder of this section we reexamine our media example using our new extensions and explain their usage. We start with the original media collection program.

With WYSIWYG strategies and the enforcement of a strict interface between strategy and visitor where each visitor can only contain before and/or after methods on types explicitly mentioned in the visitor's attached strategy, our original media collection program is invalid. Figure 2.20 shows the new `MediaCollectionAP` traversal file with altered strategies and a new strategy (`toSongsViaMedia`) used in the definition of `asStringCount`. Strategy definitions use our graph-based notation and each strategy explicitly mentions node repetitions. Originally our `toSongs` strategy was defined as `from MediaCollection to Song`; under WYSIWYG strategies however the selected paths for `toSongs` include all songs reachable through the first element of `MediaCollection`. Any path through `ConsCollection` followed by `MediaCollection` is *not* a valid path under WYSIWYG strategies for `toSongs`. To allow for such paths the strategy specification has to explicitly contain an edge from `ConsCollection` to `MediaCollection`.

Furthermore, `asStringCount` originally used the `toSongs` strategy along with the `AsStringCounterV` visitor. The `AsStringVisitor` however con-

```
aspect MediaCollectionAP {
  declare strategy : toSongs : source:MediaCollection → ConsCollection
                               ConsCollection → MediaCollection
                               ConsCollection → target:Song;
  // Return the total number of songs in a collection.
  declare traversal: int countSongs() : toSongs(SongCounterV);
  // Return a list of songs in the collection that match .* t .*
  declare traversal: SongList searchSongTitle(String t): toSongs(SongSearchV);

  declare strategy : toSongsViaMedia : source:MediaCollecton → ConsCollection
                                       ConsCollection → MediaCollection
                                       ConsCollection → Media
                                       Media → target:Song;
  // String representation of all the songs in a collection.
  declare traversal: String asStringCount() :
    toSongsViaMedia(AsStringCounterV);

  declare strategy: toPrice : source:MediaCollection → ConsCollection
                              ConsCollection → MediaCollection
                              ConsCollection → target:Price;
  // Change all prices by delta.
  declare traversal: void updatePriceBy(int delta) : toPrice(UpdatePriceV);
  // Return the total cost of the collection.
  declare traversal: int getTotalPrice() : toPrice(PriceAdderV);
}
```

Figure 2.20: Modified traversal file for media collections with WYSIWYG strategies.

tains methods for `Media` and `Song` breaking the condition imposed on visitors. We replace `toSongs` in the definition of `asStringCount` with the new strategy `toSongsViaMedia`. We also modify `SongListAP.trv` and use a new visitor (`SongListAsStringV`) for the adaptive method `show` (Figure 2.21). With the preceding changes to the original media collection example we have a valid AP program under WYSIWYG strategies. Performing the same extensions with the same program modifications for adding support for MP3 media, genres and refactoring to introduce `PricedMedia` yield a valid AP program with adaptive methods correctly adapting their behavior.

Consider changing the representation in our media collection to be a list of songs where each song has an attribute specifying the different media that contain the song. Replacing our class dictionary with the one given in Figure 2.18 gives us an *invalid* AP program. The strategies `toSongsViaMedia`

```
aspect SongListAP {
  declare strategy : songListToSong : source:SongList → ConsSongList
                                      ConsSongList → SongList
                                      ConsSongList → target:Song;
  // string represetation of a list of songs
  declare traversal: String show() : songListToSong(SongListAsStringV);

  declare strategy : mediaToSong : source:Media → target:Song;
  declare traversal: String stringAndCount() :
                     mediaToSong(CounterAndStringV);

}
```

```
class SongListAsStringV extends AsStringV {
  private int counter;
  SongListAsStringV(){
    super();
    this.counter = 0;
  }
  public void before(Song host) {
    counter++;
    sb.append(TAB + counter).append(")  ");
    super.before(host);
  }
}
```

Figure 2.21: The altered `SongListAP` traversal file and the new visitor `SongListAsStringV`.

and `mediaToSong` (Figure 2.21) select no paths in the new class dictionary causing a compile time error rejecting a modification that was previously allowed by DAJ but had undesired behavior.

As another extension consider our original music media collection and the addition of a recommended list of music media. Applying the same modifications as before we can replace our class dictionary with the one given in Figure 2.19 giving a valid AP program. Unlike the behavior of this program under DAJ, calling the `asStringCount` adaptive method does not duplicate songs and execution terminates. The WYSIWYG strategy selects paths via `Media` to `Song` but disallows paths of the form `MediaCollection`, `ConsCollection`, `Media`, `ConsRecList`, `Media` etc. A second occurrence of `Media` is not allowed by WYSIWYG strategies stopping the traversal from

```
aspect MediaCollectionAP {
  declare strategy : toSongs : source:MediaCollection → ConsCollection
                               ConsCollection → MediaCollection
                               ConsCollection → target:Song;
  // Return the total number of songs in a collection.
  declare traversal: int countSongs() : toSongs(SongCounterV);
  // Return a list of songs in the collection that match .* t .*
  declare traversal: SongList searchSongTitle(String t): toSongs(SongSearchV);

  declare strategy : toSongsViaMedia : source:MediaCollecton → ConsCollection
                                       ConsCollection → MediaCollection
                                       ConsCollection → Media
                                       Media → target:Song;
  // String representation of all the songs in a collection.
  declare traversal: String asStringCount() :
    toSongsViaMedia(AsStringCounterV);

  declare strategy: toPrice : source:MediaCollection → ConsCollection
                              ConsCollection → MediaCollection
                              ConsCollection → target:Price;
  // Change all prices by delta.
  declare traversal: void updatePriceBy(int delta) : toPrice(UpdatePriceV);
  // Return the total cost of the collection.
  @constraint{unique(from Media to Price)}
  declare traversal: int getTotalPrice() : toPrice(PriceAdderV);
}
```

Figure 2.22: Media collection's traversal file with constraint annotations.

navigating through `RecList` instances.

Constraints allow programmers to define invariants that they expect the class graph to satisfy. In our music media example observe that the adaptive method `getTotalPrice` expects to find one price for each music media. Consider the scenario where a CD or a DVD also includes prices for each individual song. Calculating the total cost of our media collection using the `getTotalPrice` adaptive methods yields the wrong result. With constraints programmers can explicitly express invariants that they expect the underlying class graph to honor. We define constraints as annotations to adaptive method declarations. Figure 2.22 shows the traversal file for `MediaCollection` with a constraint annotation on the adaptive method declaration for `getTotalPrice`. The constraint specifies that there should be one and only one path that starts from a `Media` object and leads to a `Price` object. With this constraint in place, the extensions to allow individual song prices for CDs and DVDs as well as a CD and DVD price yields an invalid AP program. Constraints enforce programmer specified invariants about the underlying class graph at compile time. Constraints provide an extra mechanism for developers to define and have finer control over what they consider to be "correct" class graphs controlling in this way the set of correct program evolutions.

2.4.1 Abstracting over AP code with DIs

Consider extending our original media collection application to support searching based on media titles. The extension in DAJ requires a new visitor similar to `SongSearchV` but with a before method on `Media` rather than `Song`. Also, we need to add a new helper adaptive method to `Media` that returns the media's title, similar to the adaptive method `getSongTitle` (Figure 2.23). The AP code for searching for a song title and searching for a media title have a lot in common. The two search visitors `SongSearchV` and `TitleSearchV` differ in the type they use to store search results and the methods that they use to acquire the title of a song and a music media.

A Demeter Interface (DI) is similar to a traversal file and includes an abstraction of the class graph called the *interface class graph* (ICG) and a sequence of strategy and traversal declarations. An ICG is similar to a class dictionary definition with abstract names for nodes instead of actual class names. Figure 2.24 shows the DI that abstracts the search functionality in our media collection example. The ICG captures the structural similarities between the relevant subgraphs involved in searching for a song or media title. Adaptive code (strategies, traversals and visitors) are defined against the ICG. Visitor definitions can refer to adaptive methods defined in the DI or methods available to all types (*e.g.*, `toString`). Figure 2.25 contains the definitions of the visitors used in the `Search` DI. A Demeter Interface

```
aspect MediaAP {
  declare strategy : mediaToSong : source:Media → target:Song;
  declare traversal: String stringAndCount() :
                         mediaToSong(CounterAndStringV);

  declare strategy : mediaToTitle : source:Media → target:Title bypassing SongList;
  // return a Media's title as a string
  declare traversal: String getMediaTitle(): mediaToTitle(IdentV);
}
```

```
class TitleSearchV extends SearchV {
  private Collection res;

  TitleSearchV(String s){
    super(s);
    this.res = new MtCollection();
  }
  public void before(Media host){
    if (match(makeRE(s), host.getMediaTitle())){
      res = new ConsCollection(host, res);
    }
  }
  public Collection return() { return res; }
}
```

Figure 2.23: The new adaptive method `getMediaTitle` defined on `Media` instances and the new visitor `TitleSearchV`.

```
di Search {
  //ICG
  Collection : Mt | Cons.
  Mt = .
  Cons = Element Collection.
  Element = Value.
  Value = Id.
  Id = .

  // Strategies
  declare strategy: toEle: source:Collection → Cons
                           Cons → Collection
                           Cons → target:Element;

  declare strategy: toVal: source:Element → target:Value;

  // Traversals
  declare traversal: Collection search(String s): toEle(ValSearchV);
  @constraint{unique(toVal)}
  declare traversal: Value getValue(): toVal(IdV);
}
```

Figure 2.24: DI for searching for a `Val` through a `Collection`.

```
class ValSearchV extends SearchV {
  private Collection res;
  ValSearchV(String s){
    super(s);
    this.res = new Mt();
  }
  public void before(Ele host){
    if (match(makeRE(s),
              host.getValue())){
      res = new Cons(host, res);
    }
  }
  public SongList return() { return res; }

}
```

```
class IdV extends StringV{
  public void IdV() { super(); }
  public void before(Id host) {
    this.sb.append(host.toString());
  }
  public String return(){ return sb.
       toString(); }
}
```

Figure 2.25: Visitor definitions used with the `Search` DI.

needs to be applied to a concrete class dictionary in order for the DI's behavior to be available. The definition of class dictionaries is extended to include a list of DIs that a class dictionary implements and a new section in the class dictionary defines a mapping between a DI's ICG and the class dictionary's nodes and edges. The mapping of a DI to a class dictionary takes an optional list of name pairs that indicate new names for adaptive methods defined in the DI and a list of mappings that maps all nodes and edges of the DI to nodes and edges/paths in the class dictionary. Figure 2.26 shows a new class dictionary for our media collection. The new class dictionary implements the `Search` DI and maps the DI twice. The first mapping generates a new adaptive method inside `MediaCollection` with the name `mediaSearch` instead of `search` and a method `getValue` inside `Media`. The mapping is comprised of a list of `use` statements, each use statement takes two arguments an ICG edge and a graph-based strategy over the nodes in the class graph. We specify ICG edges as a triple where the first element is either an inheritance edge (`=>`) or a has-a edge (`->`) followed by the edge source and the edge target. For example in the first mapping in Figure 2.26 `Collection` objects are mapped to `MediaCollection` objects and `Mt` objects

```
import java.util.*;
cd MediaCollection {
//ICG
MediaCollection : MtCollection | ConsCollection.
MtCollection = "end".
ConsCollection = "disk" Media MediaCollection.
Media : CD | DVD common "title" Title "artist" Artist.
CD = "tracks" SongList "price" Price.
DVD = "side-1" <a> SongList "side-2" <b> SongList "price" Price.
Title = Ident.
Artist = Ident.
Price = Integer.
SongList : MtSongList | ConsSongList.
MtSongList = ";".
ConsSongList = Song SongList.
Song = "song title" Title "duration" Duration.
Duration = <min> Integer ":" <sec> Integer.

//Mappings
  for Search ((search, mediaSearch)) {
    use (=>, Collection, Mt)
      as MediaCollection -> MtCollection,
    use (=>, Collection, Cons)
      as MediaCollection -> ConsCollection,
    use (->,Cons, Element)
      as ConsCollection -> Media,
    use (->, Element, Value)
      as Media -> Title bypassing SongList,
    use (->, Value, Id)
      as Title -> Ident
  }

  for Search ((search, songSearch)) {
    use (=>, Collection, Mt)
      as SongList -> MtSongList,
    use (=>, Collection, Cons)
      as SongList -> ConsSongList,
    use (->, Cons, Element)
      as ConsSongList -> Song,
    use (->, Element, Value)
      as Song -> Title,
    use (->, Value, Id)
      as Title -> Ident
   }
}
```

Figure 2.26: The new media collection class dictionary that implements the search DI. The DI is mapped twice; once for searching for media titles `titleSearch` and a second time for searching for song titles `songSearch`.

are mapped to `MtCollection` objects. `Element` objects are mapped to `Media` objects and `Value` objects are mapped to `Title` objects reachable from `Media` objects that do not traverse through `SongList` objects.

The second mapping in Figure 2.26 introduces a new adaptive method into `SongList` with the name `songSearch` instead of `search` and an adaptive method with the name `getValue` inside `Song`.

Our extension to DAJ takes a class dictionary and all DIs implemented by the class dictionary and uses the mappings to rewrite the strategies in the DI and generate appropriate versions of the visitors rewriting abstract names using their mapped class names. Any constraints defined inside the DIs are rewritten based on the mappings provided and reevaluated to verify that the mapping does not violate any constraints.

The class dictionary in Figure 2.26 does not introduce a method for searching songs inside `MediaCollection` but instead introduces a method for searching for songs inside `SongList`. To gain access to `songSearch` method from the `MediaCollection` class we can introduce an adaptive method that navigates to `SongList` and calls `songSearch` (Figure 2.27).

A DI only depends on names in the ICG, adaptive methods introduced by the DI or methods available from `Object`. DIs are self contained allowing for separate development and testing. Instantiating each ICG name as a new class with all the necessary inheritance and has-a relationships we obtain a valid AP program that we can execute and test separately. DIs increase modularity and reuse of AP code.

```
aspect MediaCollectionAP {
  //elided ...
  declare strategy : toSongList : source:MediaCollection → ConsCollection
                                  ConsCollection → MediaCollection
                                  ConsCollection → target:SongList;

  declare traversal: SongList searchSongs(String s): toSongList(SSongV);

  declare strategy : toSong : source:SongList → ConsSongList
                              ConsSongList → SongList
                              ConsSongList → target:Song;


  declare traversal: SongList songListAppend(SongList s): toSong(AppendSongListV);

}
```

```
class SSongV {
  private String s;
  private SongList res = new MtSongList();

  SSongV(String s) { this.s = s;}
  public void before(SongList host){ res = host.searchSong(s).songListAppend(res); }
  public SongList return() { return res; }
}
```

```
class AppendSongListV {
  private SongList res;

  AppendSongListV(SongList sl) { this.res = sl;}
  public void before(Song host){ res = new ConsSongList(host,res); }
  public SongList return() { return res; }
}
```

Figure 2.27: Introducing an adaptive method to `MediaCollection` for searching song titles.

CHAPTER 3

WYSIWYG Strategies and Traversal Automata

In this chapter we present the main abstractions used to capture the semantics of WYSIWYG strategies. We simplify our model and consider strategies, class graphs and object graphs as simple graphs, where strategy graphs further define a unique source and target node. We define the notion of embedded strategies and object graph conformance. We define the notion of embedded strategies where a strategy graph's node set is a subset of the class graph's node set. An object graph conformance conforms to a given class graph if for each node (edge) in the object graph there exists a corresponding node (edge) in the class graph.

Based on our abstraction of strategy graphs and class graphs we introduce *traversal automata*, the main construct used to calculate all valid paths for a WYSIWYG strategy in a class graph. We formally define the necessary conditions that a traversal automaton must have and give an algorithm for constructing a traversal automaton. We prove our algorithm correct and show how to use a traversal automaton to guide a traversal (also called a *walk*) over a conforming object graph. We further show that a complete walk starting an object o, guided by a traversal automaton selects all prefixes of valid paths in the object graph for a strategy $\mathcal{SG}$, class graph $\mathcal{CG}$ and a conforming object graph O.

The following section provides definitions and notation used throughout this chapter. In section 3.2 we provide a simple model for class graphs, strategies and object graphs. In section 3.3 we give a description of the compilation problem for AP in terms of automata. In section 3.4 we give the definition for traversal automata and in section 3.5 we provide our algorithm for constructing a traversal automaton given a strategy and a class graph and prove our algorithm correct. In the final section 3.6 we provide an algorithm for traversing an object graph guided by a traversal automaton. We also show that our guided traversal algorithm upon termination visits all object paths p in a conforming object graph O such that $meta(p)$ is a prefix of some valid path q in the class graph $\mathcal{CG}$.

3.1 Graphs and Automata

In this section we describe our notation for graphs, paths, automata and their operations.

A *graph* is defined as a pair consisting of two finite sets, a set of nodes and a set of edges, $G = (V, E)$. The set of edges E contains pairs of nodes, *e.g.*, (v_1, v_2), that represent the edge $v_1 \rightarrow v_2$ in G. For a graph G we define the following functions

- $G.nodes$ returns the set of nodes V in G,
- $G.edges$ returns the set of edges E in G,
- $G.outgoing(v)$ returns a set of edges $O = \{e \mid e \in G.edges \wedge e.source = v\}$.

For an edge $e = (v_1, v_2)$ we define $e.source = v_1$ and $e.target = v_2$. A node $v \in G$ is said to be a *sink* node if and only if $G.outgoing(v) = \emptyset$. We define a path p as a sequence of nodes $v_1, v_2, \ldots, v_n$ with the following operations:

- *p.first* returns the first node v_1 in p,
- *p.last* returns the last node v_n in p and
- *p.tail* returns $v_2, \ldots, v_n$.

A path p is said to be *in* graph G, denoted as $path(p, G)$, if for every two consecutive nodes v_i and v_{i+1}, $(v_i, v_{i+1}) \in G.edges$. A path p is said to be a *prefix* of another path q, denoted as $p \sqsubseteq q$ if p can be written as $v_1, \ldots, v_n$ and q can be written as $v_1, \ldots, v_n, v_{n+1}, \ldots, v_m$. Our prefix operation is reflexive, *i.e.*, $p \sqsubseteq p$. The function *allPrefixes* consumes a path p and returns all paths q such that $q \sqsubseteq p$. We also define concatenation of two paths $p = p_1, \ldots, p_n$ and $p' = p'_1, \ldots, p'_m$ as $p \bullet p' = p_1, \ldots, p_n, p'_2, \ldots, p'_m$ if and only if $p_n = p'_1$.

We define the predicate $expansion(p, q, R)$ (read p is an expansion of q, modulo R) to hold between two paths p and q and a set of nodes R if and only if

1. *p.first* = *q.first* and *p.last* = *q.last*.
2. p is obtainable from q by inserting zero or more nodes and each inserted node is *not* in the set R.

For a given graph G we also define the predicate $reaches(G, v_1, v_n, allowed)$ to hold iff $allowed \subseteq G.nodes$ and $\exists p : path(p, G)$ and $p = v_1, \ldots, v_n$ and $\forall i : 1 < i < n, v_i \in allowed$.

A *deterministic finite state automaton* (DFA) is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite set of states
- Σ is a finite set of symbols called the *alphabet*
- δ is a total transition function $\delta : Q \times \Sigma \longmapsto Q$
- $q_0 \in Q$ is the initial state

- $F \subseteq Q$ is the set of final states

We use the dot notation to refer to elements of the quintuple of a DFA, *e.g.*, $N.\Sigma$ refers to the alphabet Σ of automaton N. Also, a state q is a *stuck* state in automaton N, written as $stuck(N, q)$, iff

1. $q \notin N.F$, and
2. $\forall \alpha : \alpha \in N.\Sigma \Rightarrow (q, \alpha, q) \in N.\delta$.

We generalize δ to $\delta^* : Q \times \Sigma^* \longmapsto Q$ where

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, ab) = \delta^*(\delta(q, a), b)$

where ϵ is the empty sequence of symbols. Given a sequence of symbols σ a *simulation* of σ on an automaton N starting at state q is defined as $N.\delta^*(q, \sigma)$. If a starting state is not given the simulation starts at the automaton's initial state, *e.g.*, q_0.

For automata we also define the $\perp$-closure of an automaton N by adding a distinguished node $q_\perp$ to $N.Q$ and update $N.\delta$ to include

- for each $\alpha \in N.\Sigma$ the transition $(q_\perp, \alpha, q_\perp)$ and
- for each state $q \in N.Q$ and for each $\alpha \in N.\Sigma$ such that $N.\delta(q, \alpha)$ is undefined, we update the transition function to include $(q, \alpha, q_\perp)$.

Given a DFA N we define the prefix-closure of the language defined by N as $Prefix(L(N)) = \bigcup\limits_{w \in L(N)} \{t \mid t \sqsubseteq w\}$. Using the automaton N we can obtain N' such that $L(N') = Prefix(L(N))$ by making each state in N from which we can reach a state $q \in N.F$ final in N'.

3.2 The Model

In this section we formally present a simple model for class graphs, strategies, and, object graphs. In the interest of simplifying our algorithm and its correctness proof we restrict class graphs to simple unlabeled graphs and strategies to be embedded WYSIWYG strategies with one target node that must be a sink node. Restricting class graphs to simple graphs appears to disallow classes that contain one or more fields of the same type. We can use a refactoring that will allow us to model such classes using graphs. For each field f_i of type t_i in class C that shares the same type with another field f_j also in C we create a fresh class t_i' that contains a single field f of type t_i. The new class contains the same method names and signatures as t_i and forwards all calls to f. We then update the field definition of f_i in C to be t_i'.

A *class graph* ($\mathcal{CG}$) is a graph and a *strategy graph* embedded in a specific class graph ($\mathcal{SG}\langle\mathcal{CG}\rangle$) is also a graph with a distinguished source node *source* and a distinguished target node *target* such that

1. *target* is a sink node.
2. $\mathcal{SG}.nodes \subseteq \mathcal{CG}.nodes$ [*Embedded*].
3. $\forall n \in \mathcal{SG}.nodes, reaches(\mathcal{SG}, \mathcal{SG}.source, n, \mathcal{SG}.nodes)$ and $reaches(\mathcal{SG}, n, \mathcal{SG}.target, \mathcal{SG}.nodes)$. [*No dead nodes*]
4. $\forall e \in \mathcal{SG}.edges : \exists p : path(p, \mathcal{CG})$ where $expansion(p, (e.source, e.target), \mathcal{SG}.nodes)$ [*Compatibility*]

The preceding conditions help simplify and minimize the size of the strategy graph. We restrict the strategy graph to have a single target node (condition 1) and require that the nodes in the strategy graph are a subset of the nodes in the class graph (condition 2) removing the need to map between the two node sets. The third condition ensures that each node in the strategy graph contributes to some path that will reach the target node disallowing irrelevant nodes. Finally, the compatibility condition (condition 4) ensures that each edge in the strategy graph contributes to some path in the class graph that will lead us to the target node, disallowing redundant (or irrelevant) strategy edges.

A path $q = q_1, \ldots, q_n$ is said to satisfy a WYSIWYG strategy $\mathcal{SG}\langle\mathcal{CG}\rangle$ iff there exists a path $s = s_1, \ldots, s_m i$ and $path(s, \mathcal{SG})$, and each node s_i in s is also in q and we can decompose q into subpaths q' such that

$$q = s_1, q'_1, s_2, \ldots, s_{m-1}, q'_{m-1}, s_m$$

and for all j, $0 \leq j < m$, expansion$((s_j, q'_j, s_{j+1}), (s_j, s_{j+1}), \mathcal{SG}.nodes)$ holds.

An *object graph* O that conforms to a class graph $\mathcal{CG}$ ($O\langle\mathcal{CG}\rangle$) is a graph with a total function $meta : O.nodes \longmapsto \mathcal{CG}.nodes$ such that: $\forall e \in O.edges$ $(meta(e.source), meta(e.target)) \in \mathcal{CG}.edges$. We also say that the type of object o is $meta(o)$. The function $meta$ is lifted to paths in O, *e.g.*, for a path $p = p_1, \ldots, p_n$, $meta(p) = meta(p_1), \ldots, meta(p_n)$. We also say that a path p in O satisfies a strategy graph $\mathcal{SG}$ embedded in $\mathcal{CG}$ if and only if $satisfies(meta(p), \mathcal{SG})$.

Given a class graph $\mathcal{CG}$, a strategy graph $\mathcal{SG}$ and a conforming object graph O our goal is to enumerate the set of paths p in O such that $meta(p)$ is a prefix of a path q such that $path(q, \mathcal{CG})$ and $satisfies(q, \mathcal{SG})$. We achieve this by constructing a class graph automaton $N_{\mathcal{CG}}$, a strategy graph automaton $N_{\mathcal{SG}}$ and an object graph automaton N_O such that,

- $N_{\mathcal{CG}}$ selects all paths in $\mathcal{CG}$ starting at $\mathcal{SG}.source$ and terminating at $\mathcal{SG}.target$,
- $N_{\mathcal{SG}}$ selects all paths comprised of nodes from $\mathcal{CG}.nodes$ that are expansions of paths in $\mathcal{SG}$ starting with $\mathcal{SG}.source$ and terminating with $\mathcal{SG}.target$, and,
- N_O selects all paths in the object graph O that start at an object o and terminate at an object o' such that $meta(o) = \mathcal{SG}.source$ and $meta(o') = \mathcal{SG}.target$.

We can obtain the set of paths p that we are looking for using automata intersection. First we take the prefix closure of the automata intersection of the strategy graph automaton and the class graph graph automaton, *i.e.*, $N_{sc} = \mathit{Prefix}(N_{\mathcal{SG}} \cap N_{\mathcal{CG}})$. We then take the intersection between N_{sc} and the prefix closure of the object graph, *e.g.*, $\mathit{Prefix}(N_{\mathcal{SG}} \cap N_{\mathcal{CG}}) \cap \mathit{Prefix}(N_O)$.

The intersection between the strategy graph automaton and the class graph automaton gives as a new automaton (N_{sc}) that accepts all prefixes of valid paths in the class graph. The intersection between N_{sc} and the prefix closure of the object graph automaton results in an automaton that accepts all prefixes of valid paths in the object graph.

3.3 Path Selection

Abstractly speaking, compilation of an AP program involves the selection of paths in a graph G given a selector S. In AP, the strategy graph corresponds to the selector S and the class graph corresponds to the graph G. Ultimately, in AP the selected paths in the class graph are used to traverse over an object graph. We only consider object graphs that *conform* to the class graph in question. We define conformance between an object graph O and a class graph $\mathcal{CG}$ if for each edge (v_1, v_2) in O $(\mathit{meta}(v_1), \mathit{meta}(v_2))$ is in $\mathcal{CG}$, where *meta* is a function that returns the runtime type of an object o in O. Therefore, by conformance, selected paths from the class graph provide the set of all possible valid paths in a conforming object graph.

Even though in this chapter we use simple graphs to represent WYSIWYG strategies, class graphs and object graphs to address the path selection problem, automata theory [16] has also been used to explain the path selection problem in AP [35, 36, 33]. We describe how each of the graphs involved, strategy graph, class graph and object graph can be turned into a deterministic finite automaton (DFA). First we sketch an algorithm on how we can turn the strategy graph into a DFA and then describe the modification to the algorithm to accommodate for class graphs and object graphs. We then proceed to map operations on automata, specifically automata intersection, to the compilation approaches taken by AP tools. AP tools fall into one of the following two compilation approaches,

1. generating code given a strategy and a class graph, *e.g.*, DemeterJ, and,

2. navigating over an object graph O given a strategy $\mathcal{SG}$ and a class graph $\mathcal{CG}$ where O conforms to $\mathcal{CG}$, *e.g.*, DJ and DAJ.

3.3.1 Strategy Graph Automaton

Given a strategy graph $\mathcal{SG}$ with a source node s and a target node t, and a class graph $\mathcal{CG}$ such that $\mathcal{SG}$ is embedded in $\mathcal{CG}$ (*i.e.*, $\mathcal{SG}.nodes \subseteq \mathcal{CG}.nodes$) we build a DFA $N_{\mathcal{SG}} = \langle Q, \Sigma, \delta, q_0, F \rangle$ using the following steps:

1. $\Sigma = \mathcal{CG}.nodes$. The automaton's alphabet is the same as the node names in $\mathcal{CG}$.

2. $F = \{q_{\mathcal{SG}.target}\}$. The state $q_{\mathcal{SG}.target}$ corresponds to the target node v in $\mathcal{SG}$.

3. $Q = \{q_v \mid v \in \mathcal{SG}.nodes\} \cup \{q_0\}$. Each node in $\mathcal{SG}$ becomes a state in the automaton and we add a distinguished initial state q_0.

4. $\forall q_v \in Q \setminus F : \delta(q_v, v') = \begin{cases} q_{v'} & \textit{if } (v, v') \in \mathcal{SG}.edges \\ q_v & \textit{if } v' \in \Sigma \setminus \mathcal{SG}.nodes \end{cases}$
   For each edge $e = (v, v')$ in $\mathcal{SG}.edges$, such that $v \notin F$ we add a transition from the automaton's state that corresponds to v to the automaton's state that corresponds to v'. The transition is labeled with v'. We also add self loops to each state, except the final state, in $N_{\mathcal{SG}}$ labeled with all the symbols in Σ other than the node names in $\mathcal{SG}$, (*i.e.*, $\mathcal{CG}.nodes \setminus \mathcal{SG}.nodes$).

5. We add a transition labeled with s from our initial state q_0 to q_s and the $\perp$-closure of $N_{\mathcal{SG}}$ to δ.

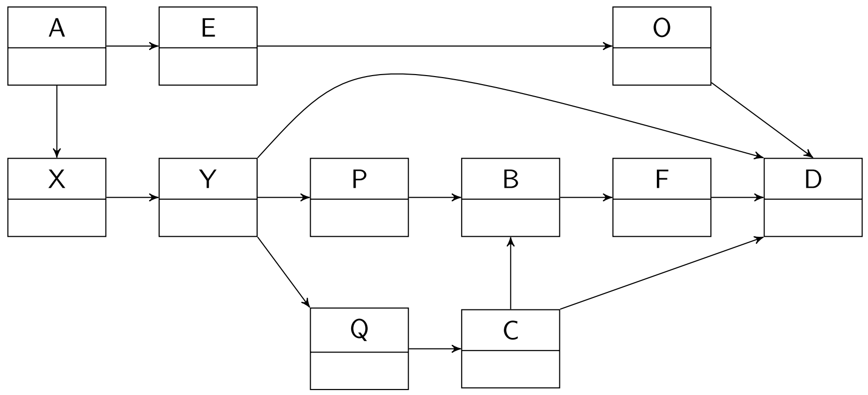


Figure 3.1: Class Graph

Using the class graph in Figure 3.1 and the strategy graph in Figure 3.2(a) we obtain the DFA $N_{\mathcal{SG}}$ given in Figure 3.2(b). The language defined by $N_{\mathcal{SG}}$ can be given as a regular expression,

$$L(N_{\mathcal{SG}}) = A\mathcal{X}^*(B \mid C)\mathcal{X}^*D$$

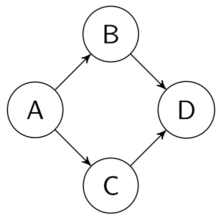
where $\mathcal{X} = (v_1 \mid \ldots \mid v_n)$ such that $v_i \in \mathcal{CG}.nodes \setminus \mathcal{SG}.nodes$.

3.3.2 Class Graph and Object Graph Automaton

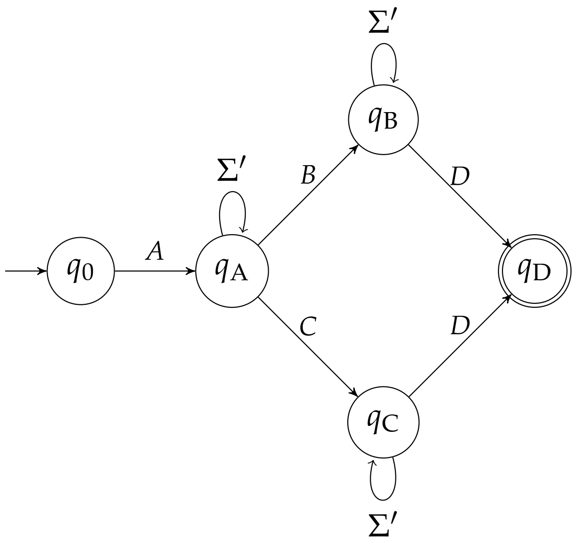
We use a similar algorithm to create an automaton $N_{\mathcal{CG}}$ for a class graph $\mathcal{CG}$. The automaton $N_{\mathcal{CG}}$ selects all the paths in $\mathcal{CG}$ that start from the class $\mathcal{SG}.source$ and terminate at the class $\mathcal{SG}.target$. Given a strategy graph $\mathcal{SG}$ with a source node s and a target node t, and a class graph $\mathcal{CG}$ such that $\mathcal{SG}$ is embedded in $\mathcal{CG}$ (*i.e.*, $\mathcal{SG}.nodes \subseteq \mathcal{CG}.nodes$) we build a DFA $N_{\mathcal{CG}} = \langle Q, \Sigma, \delta, q_0, F\rangle$ using the following steps:

1. $\Sigma = \mathcal{CG}.nodes$. The automaton's alphabet is the same as the node names in $\mathcal{CG}$.

2. $Q = \{q_v \mid v \in \mathcal{CG}.nodes\} \cup \{q_0\}$. Each node in $\mathcal{CG}$ becomes a state in the automaton and we add a distinguished initial state q_0.



(a) Strategy graph with source A and target D



(b) Resulting DFA from $\mathcal{SG}$ and $\mathcal{CG}$ where $\Sigma' = \mathcal{CG}.nodes \setminus \mathcal{SG}.nodes = (X, Y, P, Q, F, E, O)$. For readability all transitions to/from $q_\perp$ are excluded.

Figure 3.2: The strategy $\mathcal{SG}$ and the traversal automaton using the class graph from Figure 3.1.

3. $F = \{q_{\mathcal{SG}.target}\}$. The state $q_{\mathcal{SG}.target}$ corresponds to the target node v in $\mathcal{CG}$.

4. $\delta = \{(q_v, v', q_{v'}) \mid (v, v') \in \mathcal{CG}.edges\}$. For each edge e in $\mathcal{CG}$ we add a transition from the automaton's state that corresponds to e's source node to the automaton's state that corresponds to e's target node. The transition is labeled with e's target node name. For example, given an edge $e = (v, v')$ in $\mathcal{CG}$ we generate a transition $(q_v, v', q_{v'})$ in $N_{\mathcal{CG}}.\delta$.

5. We add a transition labeled with s from our initial state q_0 to q_s and the $\perp$-closure of $N_{\mathcal{CG}}$ to δ.

Figure 3.3 shows the class graph automaton using the class graph from Figure 3.1 and the strategy from Figure 3.2(a). The language defined by the

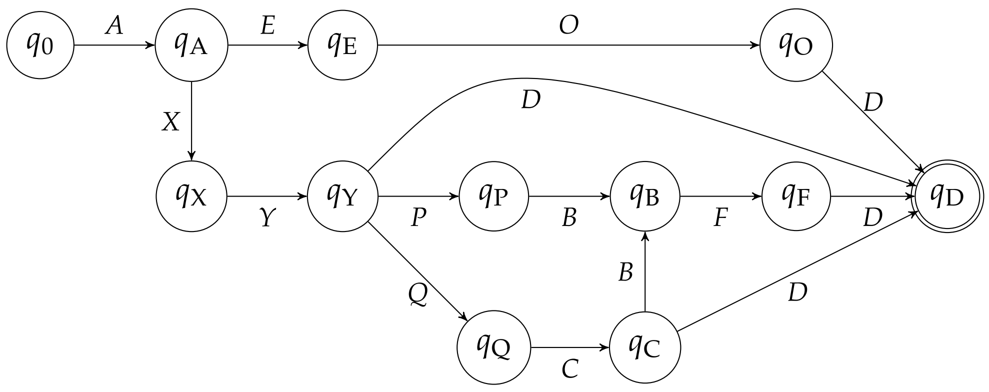


Figure 3.3: Class Graph Automaton

class graph automaton $N_{\mathcal{CG}}$ in Figure 3.3 can be given as

$$L(N_{\mathcal{CG}}) = A(EO \mid XY(\epsilon \mid (P \mid QC)BF \mid QC))D$$

In the case of an object graph, the algorithm is similar to the case for the class graph but the algorithm takes as input an object graph O a strategy graph $\mathcal{SG}$, and a start object o. The object graph automaton N_O selects all the paths in O that start from the node o, terminate at a node o', such that $meta(o) = \mathcal{SG}.source$ and $meta(o') = \mathcal{SG}.target$. We build the DFA N_O using the following steps:

1. $\Sigma = \mathcal{CG}.nodes$. The automaton's alphabet is equal to the set of node names in the class graph.

2. $Q = \{q_o \mid o \in O.nodes\} \cup \{q_0\}$. For each node o in the object graph we create a corresponding state q_o.

3. $F = \{q_o \mid meta(o) = \mathcal{SG}.target\}$. All states that correspond to nodes in the object graph whose runtime type is equal to the strategy's target are final.

4. $\delta = \{(q_o, meta(o'), q_{o'}) \mid (o, o') \in O.edges\}$. For each edge in the object graph between two nodes o and o' we create a transition between q_o and $q_{o'}$ labeled with the runtime type of o'.

5. We add $(q_0, meta(o), q_o)$, where o is the starting object provided, and the $\perp$-closure of N to δ.

Thus, given a strategy graph $\mathcal{SG}$, a class graph $\mathcal{CG}$ and a conforming object graph O with a start object o, we can obtain a strategy automaton $N_{\mathcal{SG}}$, a class graph automaton $N_{\mathcal{CG}}$ and an object graph automaton N_O.

3.3.3 AP Compilation Approaches Using Automata

We can explain the two compilation approaches taken by AP tools using automata operations on the strategy graph automaton, class graph automaton and the object graph automaton.

3.3.3.1 Generative Approach: DemeterJ

The DemeterJ [50] AP tool consumes a strategy and a Java program along with a class graph for the Java program and generates Java code to perform traversals based on the input strategy. To generate Java code for traversing runtime object graphs that conform to the input class graph, DemeterJ calculates a *traversal graph* [28]; a graph that contains all valid paths in the class graph.

The process for calculating all valid paths for a given strategy $\mathcal{SG}$ over a given class graph $\mathcal{CG}$ can be formulated as the automata intersection of the strategy graph automaton $N_{\mathcal{SG}}$ and the class graph automaton $N_{\mathcal{CG}}$. Taking the intersection of the strategy graph automaton and the class graph automaton gives us an automaton, N_{sc}, that recognizes the language of all complete WYSIWYG paths, *i.e.*, paths that terminate with $\mathcal{SG}.target$. DemeterJ generates code (methods) within classes that are responsible for traversing any conforming object graph.

DemeterJ's code generation consults the traversal graph in order to identify the correct classes in the program and performs runtime checks on objects in order to avoid `null` values. The traversal graph is viewed as a non-deterministic finite automaton (NFA). The generated methods simulate the NFA with each method having as an argument a list of tokens. The list of tokens tracks the NFA's states that the automaton is currently on given the objects traversed [7].

The intersection of the strategy graph automaton and the class graph automaton N_{sc} can replace the traversal graph and can be used to generate code in DemeterJ. Given that N_{sc} is a DFA, code generation is simpler than in the case of the traversal graph; methods do not need to consume a list of tokens, there is at most one current state during a run of the DFA on any input.

Given that at runtime object graphs are *not* restricted to have all the edges and nodes found in their conforming class graph we can have object graphs that do not contain a valid path for the strategy. An object graph can, however, contain prefixes of valid paths. Performing a traversal over such an object graph starting at an object o will result in visiting all prefixes of valid paths that start at $meta(o)$, *e.g.*, $meta(o, \ldots, o_n) \sqsubseteq p$ where $p \in L(N_{sc})$. We accommodate for these prefixes in our automaton by taking the prefix closure of N_{sc}, $Prefix(N_{sc})$. For example, the prefix closure of the intersection of the strategy automaton in Figure 3.2(b) and the class graph automaton in Figure 3.3 $L(N) = Prefix(L(N'))$ is given in Figure 3.4. Given an object graph O we can obtain the set of valid prefixes for O by simulating each path in O on N.

Given N (Figure 3.4) consider as an example an object graph O consisting of the object a such that $meta(a) = A$. The set of paths p such that p is a prefix of a valid path in $\mathcal{CG}$ is the set $\{a\}$. If we consider another object graph $a \rightarrow o \rightarrow e \rightarrow d$ where the runtime type for an object x is X, then again the set of valid prefixes is $\{a\}$. As a last example, lets consider the object graph $a \rightarrow x \rightarrow y \rightarrow q \rightarrow c \rightarrow b \rightarrow f \rightarrow d$. Observe that the object graph contains one path and the path is not a WYSIWYG path. The set of valid prefixes is then $\{a, ax, axy, axyq, axyqc\}$.

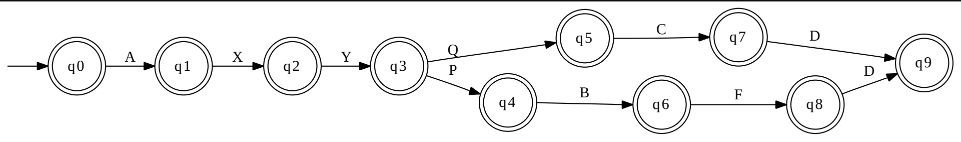


Figure 3.4: Automaton representing *Prefix*$(N_{\mathcal{SG}} \cap N_{\mathcal{CG}})$. For readability all transitions to/from $q_{\perp}$ are excluded.

3.3.3.2 Reflective Approach: DJ/DAJ

Tools like DJ [34] and DAJ [44] provide the same capabilities as DemeterJ but can do so without generating code.[1] Instead both DJ and DAJ interrogate runtime objects in order to calculate the valid paths during traversals. Both DJ and DAJ rely on the underlying runtime system's ability to obtain runtime types for objects as well as construct the complete class graph from an object graph.

After calculating the class graph, DJ and DAJ proceed by constructing the traversal graph from the strategy and the calculated class graph and use the traversal graph as a guide in order to traverse the object graph. At each object DJ/DAJ reflectively acquires the objects runtime type t and then uses t and the traversal graph (an NFA) to decide where to go next.

As in the case with DemeterJ, the traversal graph can be replaced by the prefix closure of the intersection of the strategy graph automaton and the class graph automaton, *e.g.*, $N_{sc} = \mathit{Prefix}(N_{\mathcal{SG}} \cap N_{\mathcal{CG}})$. Traversing the object graph will follow the same pattern but instead of interrogating the traversal graph, we now interrogate N_{sc}.

Previous work on compilation methods for AP [35, 36, 28] defined strategies differently. In [35, 36] strategies are given as series-parallel graphs that cannot express loops in strategy graphs. Unlike WYSIWYG strategies, series-parallel strategies select paths that allow for extra occurrences of strategy node names in selected paths. In [28], strategies are more gen-

[1]DAJ can do both, use reflection or generate code.

eral; strategies are graphs selecting paths that allow for extra occurrences of strategy node names in selected paths. Both of these approaches to strategies allow extra occurrences of strategy nodes to appear in selected paths. Trying to capture this property in our strategy graph automaton construction implies that at each state of the automaton we should have self loop iterations labeled with strategy node names.[2] Adding these types of self loops yields a non-deterministic finite automaton (NFA). Therefore, the intersection of the strategy graph automaton, the class graph automaton and the object graph automaton yield an NFA. Using an NFA to generate code that traverses all paths selected by $\mathcal{SG}$ in an object graph O we can either determinize the NFA to obtain a DFA and use the generated DFA to generate code [36, 35], or, simulate an NFA by passing to each generated method tokens that represent the state(s) the NFA is currently on during its simulation [28]. Determinization of the NFA can generate an exponentially large DFA and thus generate an exponentially large program,[3] while simulating an NFA incurs a runtime cost.

With WYSIWYG strategies we are able to give a simple construction for our traversal automaton that is deterministic.

3.4 Correctness Criteria for Traversal Automata

A traversal automaton N for class graph $\mathcal{CG}$ and a strategy graph $\mathcal{SG}$ (Definition 1), is an automaton with $\Sigma = \mathcal{CG}.nodes$ that is used to guide a traversal of an object graph O conforming to $\mathcal{CG}$ only through paths that satisfy the given strategy. Intuitively, N must have the following properties:

soundness – all words in the language $L(N)$ of the traversal automaton N satisfy the strategy.

[2]Our construction in section 3.3.1 explicitly disallows these types of self loops.

[3]An example strategy and class graph that leads to exponentially large program is given in [28] §6.2.

completeness – all paths in the class graph that satisfy the strategy are in $L(N)$.

optimality – the automaton moves to state $q_\perp$ for all paths that cannot lead to a target. For any path p in class graph $\mathcal{CG}$ such that there does not exist a path q in $\mathcal{CG}$ and $q \in L(N)$ and $p \sqsubseteq q$ then $N.\delta^*(q_0, p) = q_\perp$.

We would like the traversal automaton to capture all words that satisfy the strategy (*soundness*) but also capture all the paths in the given class graph that satisfy the strategy (*completeness*). Furthermore, we would like the traversal automaton to transition to its stuck state ($q_\perp$) as soon as a symbol α is read from the input that we know cannot lead us to the final state (*optimality*).

Definition 1. ***(Traversal Automaton)***
An automaton $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a traversal automaton *for a class graph $\mathcal{CG}$ and an embedded strategy $\mathcal{SG}$ if and only if*

1. *$\Sigma = \mathcal{CG}.nodes$. The automaton's alphabet is the same as the node names in $\mathcal{CG}$.*

2. *$\forall q : q \in L(N) \Rightarrow satisfies(q, \mathcal{SG})$. All words in the language Ł(N) satisfy $\mathcal{SG}$.*

3. *$\forall q : path(q, \mathcal{CG}) \wedge satisfies(q, \mathcal{SG}) \Rightarrow q \in L(N)$. All the paths in the class graph $\mathcal{CG}$ that satisfy $\mathcal{SG}$ are in the language $L(N)$.*

4. *$\forall p : path(p, \mathcal{CG}) \wedge \neg stuck(N, N.\delta^*(q_0, p)) \Rightarrow \exists q : path(q, \mathcal{CG}) \wedge p \sqsubseteq q \wedge satisfies(q, \mathcal{SG})$. For simulations of the automaton on a class graph path p that leave the automaton in a non-stuck state (i.e., all states other than $q_\perp$) there exists a path q in the class graph that satisfies $\mathcal{SG}$ and $p \sqsubseteq q$.*

$$\begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, "A"] & q_{\mathrm{A}} \arrow[r, "E"] & q_{\mathrm{E}} \arrow[r, "O"] & q_{\mathrm{O}} \arrow[r, "D"] & q_{\mathrm{D}} \end{tikzcd}$$

Figure 3.5: Object graph automaton N_O for object graph $a \rightarrow e \rightarrow o \rightarrow d$.

3.5 Constructing the Traversal Automaton

Given a strategy graph $\mathcal{SG}$ and a class graph $\mathcal{CG}$ we construct a traversal automaton N using the following steps:

1. $\Sigma = \mathcal{CG}.nodes$. The automaton's alphabet is the same as the node names in $\mathcal{CG}$.

2. $Q = \{q_v \mid v \in \mathcal{SG}.nodes\} \cup \{q_0\}$. Each node in $\mathcal{SG}$ becomes a state in the automaton N and we also add a distinguished start node q_0.

3. $F = \{q_{\mathcal{SG}.target}\}$. The state $q_{\mathcal{SG}.target}$ corresponds to the target node v in $\mathcal{SG}$.

4. $$\forall q_v \in Q \setminus F : \delta(q_v, v') = \begin{cases} q_{v'} & \text{if } (v,v') \in \mathcal{SG}.edges \\ q_v & \text{if } \exists v_2 : (v,v_2) \in \mathcal{SG}.edges \wedge v' \in A \\ & reaches(\mathcal{CG}, v, v', A) \wedge reaches(\mathcal{CG}, v', v_2, A) \\ & \text{where } A = \Sigma \setminus \mathcal{SG}.nodes \end{cases}$$

5. We add $(q_0, \mathcal{SG}.source, q_{\mathcal{SG}.source})$ and the $\perp$-closure of N to δ.

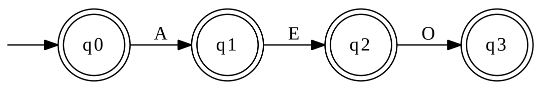
Figure 3.8 shows the TA for the strategy graph in Figure 3.8(b) and the class graph given in Figure 3.1 (page 55). The language defined by the automaton in Figure 3.8(b) can be given as

$$L(N) = A(X \mid Y \mid P \mid Q)^*(BF^* \mid C)D$$

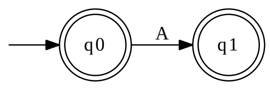
The language defined by the traversal automaton N in Figure 3.8(b) defines paths that satisfy the strategy $\mathcal{SG}$ from Figure 3.8(a) (Definition 1, condition 2). Observe that for all paths p in the class graph that satisfy the strategy, p is in the language defined by the automaton N in Figure 3.8(b), (Definition 1, condition 3). However the reverse implication does not hold; there are words $w \in L(N)$ that are valid paths according to the strategy but invalid paths for the class graph, *e.g.*, $(A, Q, B, D) \in L(N)$ but the path (A, Q, B, D) is an invalid path for the class graph in Figure 3.1.

The strategy graph automaton $N_{\mathcal{SG}}$ (§ 3.3.1, page 56) corresponds to our Traversal Automaton in that it defines the set of valid paths as words of the language defined by the automaton. Figure 3.8(b) shows the Traversal Automaton (TA) for the same class graph and strategy graph. The traversal automaton's construction is similar to the construction of the strategy graph automaton, the difference lies in the way we calculate self loops on each automaton state. In the case of the TA, we add a self loop on a state q_{i} labeled v if the node v appears on a path p in the class graph from node i to node j and (i, j) is an edge in the strategy graph. Therefore, the TA excludes paths in the class graph that may never lead to q_D that the $N_{\mathcal{SG}}$ automaton allows (*e.g.*, A, E, O). The language defined by the TA contains all WYSIWYG paths in the class graph and extra paths that cannot be present in a conforming object graph. The strategy graph automaton, on the other hand, contains all WYSIWYG paths in the class graph as well as extra paths that *can* be present in a conforming object graph.

For example, given the strategy from Figure 3.2(a) with the strategy graph automaton $N_{\mathcal{SG}}$ (Figure 3.2(b)), traversal automaton N_{TA} (Figure 3.8) and class graph automaton $N_{\mathcal{CG}}$ (Figure 3.3), consider the object graph $a \rightarrow e \rightarrow o \rightarrow d$ and its object graph automaton N_O (Figure 3.5), then the paths selected by $\mathit{Prefix}(N_{TA}) \cap \mathit{Prefix}(N_O)$ (Figure 3.7) and the paths selected by $\mathit{Prefix}(N_{\mathcal{SG}}) \cap \mathit{Prefix}(N_O)$ (Figure 3.6(a)) differ. The extra loop transitions in $N_{\mathcal{SG}}$ allow for paths that are not prefixes of any valid path for the strategy. The construction of our TA only includes transitions between non-stuck states if the node in the class graph contributes to a path between strategy nodes. Thus the TA only selects paths that are prefixes of some valid path



(a) Automaton $N = \mathit{Prefix}(N_{\mathcal{SG}}) \cap \mathit{Prefix}(N_O)$.



(b) Automaton $M = \mathit{Prefix}(N_{TA}) \cap \mathit{Prefix}(N_O)$.

Figure 3.6: The difference between using the TA and the strategy graph automaton to calculate the set of paths the object graph $a \rightarrow e \rightarrow o \rightarrow d$.

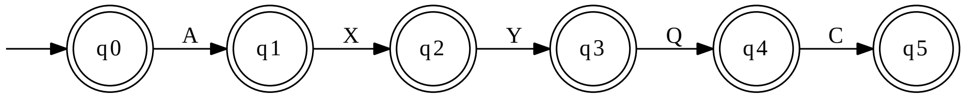


Figure 3.7: Intersection of $\mathit{Prefix}(TA)$ and the prefix closure of the object graph automaton for object graph $a \rightarrow x \rightarrow y \rightarrow q \rightarrow c \rightarrow b \rightarrow f \rightarrow d$.

for the strategy in the class graph.

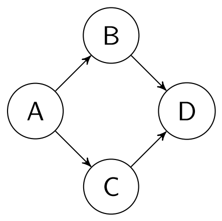
By taking the intersection of the strategy graph automaton and the class graph automaton we obtain an automaton that contains only the valid WYSIWYG paths in the class graph.

With a TA there is no need to take the TA's intersection with the class graph automaton. Instead we can obtain the set of valid paths in an object graph by taking the intersection of the prefix closure of the TA and the prefix closure of the object graph, *e.g.*, $\mathit{Prefix}(TA) \cap \mathit{Prefix}(N_O)$.

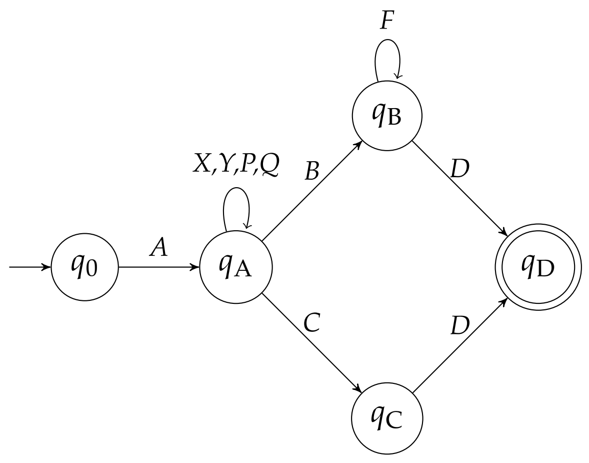
Consider for example the TA in Figure 3.8(b) and the object graph a where $\mathit{meta}(a) = A$. The intersection of the prefix closed TA and the prefix closed object graph automaton accepts the string A. As another example consider the object graph $a \rightarrow x \rightarrow y \rightarrow q \rightarrow c \rightarrow b \rightarrow f \rightarrow d$, then the intersection of the prefix closure of TA and the prefix closure of the object graph automaton accepts the language $\{A, AX, AXY, AXYQ, AXYQC\}$ (Figure 3.7).

Theorem 1. ***(Correctness)***

The traversal automaton constructed from a class graph $\mathcal{CG}$ and a strategy graph



(a) Strategy graph with source A and target D



(b) Resulting traversal automaton from $\mathcal{SG}$ and $\mathcal{CG}$. For readability all transitions to/from $q_\perp$ are excluded.

Figure 3.8: The strategy $\mathcal{SG}$ and the traversal automaton using the class graph from Figure 3.1 (page 55).

$\mathcal{SG}$ satisfies Definition 1, page 62.

Proof. let N be the traversal automaton obtained from a class graph $\mathcal{CG}$ and a strategy $\mathcal{SG}$ using the 5-step procedure given in § 3.5, we show that N satisfies all the properties of a traversal automaton.

1. Immediate.

2. We start by showing that every word r in $L(N)$ must start with the node $\mathcal{SG}.source$ and end with $\mathcal{SG}.target$. By definition we know that r is in $L(N)$ iff we can trace a path through N starting by $N.q_0$ and ending with a state in $N.F$ whose sequence of labels are the same as r. By construction, there is one edge going out of q_0 whose label is $\mathcal{SG}.source$. Therefore, every word in $L(N)$ starts with $\mathcal{SG}.source$. Also, by construction, we have a single final state labeled $q_{\mathcal{SG}.target}$. By the definition of $\mathcal{SG}$, $\mathcal{SG}.target$ is a sink node. Therefore, all the edges

leading into $q_{\mathcal{SG}.target}$ are created by step 4 (q_m, n, q_n), *i.e.*, all the edges leading into the final state of N are labeled $\mathcal{SG}.target$.

Now we show that any word $r = r_1, r_2, \ldots r_n$ in $L(N)$ satisfies the strategy. Let $s = s_1, s_2, \ldots, s_k$ be the longest subsequence of r such that $s_i \in \mathcal{SG}.nodes$. Therefore $expansion(r, s, \mathcal{SG}.nodes)$ holds. As we have shown above $r_1 = \mathcal{SG}.source$, therefore, $s_1 = r_1 = \mathcal{SG}.source$ because otherwise, we can prepend $\mathcal{SG}.source$ to s and get a longer subsequence of r whose nodes are in $\mathcal{SG}.nodes$. Likewise, $s_k = r_n = \mathcal{SG}.target$. By the definition of *satisfies*, if we can show that s is a path in $\mathcal{SG}$, then we can conclude that $satisfies(r, \mathcal{SG})$.

First, we show that s is in $L(N)$ and then show that s is a path in $\mathcal{SG}$. By construction, the nodes in r but not in s are added as self loop labels by step 4 (q_m, s, q_m). Since we can trace a path through N whose labels are the same as r, we can also trace a path through N whose labels are the same as s by avoiding self loops.

Finally, we show that s is a path in $\mathcal{SG}$. Since s is a word in $L(N)$, by construction we know that each symbol s_i (in $\mathcal{SG}.nodes$) labels a transition that leads to state q_{s_i} (added by step 4 (q_m, n, q_n)). Furthermore, every two consecutive symbol s_i and s_{i+1} show up as labels on two transitions, one leading into state q_{s_i} and the other leading out of q_{s_i} and into $q_{s_{i+1}}$. The second transition can only exist if there is an edge (s_i, s_{i+1}) in $\mathcal{SG}.edges$. Therefore, s is a path in $\mathcal{SG}$.

3. Let $p = p_0, p_1, \ldots, p_n$ be a path in $\mathcal{CG}$ and $satisfies(p, \mathcal{SG})$. We show that a simulation of p on N starting at the initial state terminates with N in the final state.

We note that, by construction, only symbols in $\mathcal{SG}$*.nodes* can transition N from one state to another (through one of the transitions added by step 4 (q_m, n, q_n)) such that both states are not the $q_\perp$ state. Symbols in $\mathcal{CG}$*.nodes* $\setminus$ $\mathcal{SG}$*.nodes* can either transition N through one of the self-loops added by step 4 (q_m, s, q_m) to the same state or to the stuck state $q_\perp$.

We first show that nodes in p that are not in $\mathcal{SG}$*.nodes* can not transition N to the stuck state $q_\perp$ and therefore can not change the state of N. Suppose that $p_i \notin \mathcal{SG}$*.nodes* is the first node in p to transition N from a non stuck state into the stuck state $q_\perp$. Let p_h be its closest predecessor node that is in $\mathcal{SG}$*.nodes* and p_j be its closest successor node that is in $\mathcal{SG}$*.nodes*. The symbols p_h and p_j always exist and are uniquely identifiable by the definition of *satisfies*. Then p_i must not be mentioned on any self-loop at q_{p_h}, otherwise, p_i will not transition N to $q_\perp$. For that to happen, p_i must be unreachable in $\mathcal{CG}$ from p_h through nodes not in $\mathcal{SG}$*.nodes* or cannot reach p_j in $\mathcal{CG}$ through nodes not in $\mathcal{SG}$*.nodes* or both. The first condition, contradicts our assumption that p_h can reach p_i through a path in $\mathcal{CG}$ (a subpath of p) that does not contain any node in $\mathcal{SG}$*.nodes*. The second condition, contradicts our assumption that p_j is reachable from p_i through a path in $\mathcal{CG}$ that does not contain any node in $\mathcal{SG}$*.nodes*.

We now show that nodes in p that are in $\mathcal{SG}$*.nodes* can only transition N to its only final state $q_{\mathcal{SG}.target}$. By definition of *satisfies*, there exists a path $s = s_0, \ldots, s_k$ in $\mathcal{SG}$ that contains all nodes in p that are in $\mathcal{SG}$*.nodes* and $p_0 = s_0 = \mathcal{SG}$*.source*, which by construction transitions N from the initial state q_0 to $q_{\mathcal{SG}.source}$. For any following symbol s_i

where $i > 0$, there must be an edge (s_{i-1}, s_i) in $\mathcal{SG}.edges$ because, by the definition of *satisfies*, s is a path in $\mathcal{SG}$. Therefore, by construction, there is a transition $(q_{s_{i-1}}, s_i, q_{s_i})$ (added by step 4 (q_m, n, q_n)) in N. Furthermore, by the definition of *satisfies*, $p_n = s_k = \mathcal{SG}.target$, which leaves N in state $q_{\mathcal{SG}.target}$ which is the only final state of N by construction.

4. Given N at state q_0 consider a simulation of N given the input $p = p_1, p_2, \ldots, p_n$, where $path(p, \mathcal{CG})$, that leaves N in a state q_m we need to show that $\exists q : path(q, \mathcal{CG})$ and $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$. We proceed by cases on $p.last$.

   - $p.last \in N.F$ then select $p = q$, and we know that $q \in L(N)$ thus $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$.

   - $p.last = q_0$. By Lemma 1 we know that there exists a path p in $\mathcal{CG}$ and $p.first = \mathcal{SG}.source$ such that given N at state $q_{\mathcal{SG}.source}$ a simulation of N on $p.tail$ leaves N in state $q_{\mathcal{SG}.target}$. By the construction of N we know that q_0 has one and only one transition $(q_0, \mathcal{SG}.source, q_{\mathcal{SG}.source})$ that leaves N in a non-stuck state. Thus given N at state q_0 a simulation of p will leave N in state $q_{\mathcal{SG}.target}$. Thus, $p = q$, and we know that $q \in L(N)$, therefore $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$.

   - $p.last \in \mathcal{SG}.nodes \backslash N.F$. By construction of N we know that N just completed a transition between two distinct states q_1, q_2. We have already shown (Lemma 1) that from any state $q' \in N.Q \backslash \{q_0, q_\perp, \}$ we can always find a path r in $\mathcal{CG}$ such that a simulation of N at state q' on input $r.tail$ leaves N in state $q_{\mathcal{SG}.target}$. We also know that p takes N from q_0 to q_m where q_m is not a stuck state or an accepting state. We also know that $r.first = p.last$ thus we can construct $q = p \bullet r$ and $path(q, \mathcal{CG})$ and $p \sqsubseteq q$.

We need to show that $satisfies(q, \mathcal{SG})$. But we have already shown that for all $w \in L(N)$ then $satisfies(w, \mathcal{SG})$ and we know that $q \in L(N)$.

- $p.last \notin \mathcal{SG}.nodes$. We know that a simulation of N at state q_0 on input p leaves N in state q_m. By construction of N, $(q, v, q) \in N.\delta$ iff $v \in \mathcal{CG}.nodes$ and $v \notin \mathcal{SG}.nodes$. Thus the last transition taken by N while simulating p was a self loop on state q_m. We also know by construction of N that for all $v \in \mathcal{CG}.nodes$, $(q_m, v, q_m) \in N.\delta$ iff there exists $q_k \in N.Q$ and path $r = r_1, r_2, \ldots, r_n$ in $\mathcal{CG}$ such that

$$\begin{array}{ll} r_1 = m & \wedge \\ r_n = k & \wedge \\ \forall j : 1 < j < n : r_j \notin \mathcal{SG}.nodes & \wedge \\ \exists i : 1 < i < n : r_i = v & \end{array}$$

We can thus select transitions

$$(q_m, r_i + 1, q_m), (q_m, r_i + 2, q_m), \ldots, (q_m, r_{n-1}, q_k), (q_m, r_n, q_k)$$

By construction we know that the transitions labeled with $r_{i+1}, r_{i+2}, \ldots, r_{n-1}$ are defined in $N.\delta$. We also know by construction that $(q_m, r_n, q_k) \in N.\delta$. We have shown (Lemma 1) that for any state $q_i \in N.Q$ such that $q_i \notin \{q_0, q_\perp, q_{\mathcal{SG}.target}\}$ we can construct a path p' in $\mathcal{CG}$ such that given an N at state q_i simulating N on input $p'.tail$ leaves N in state $q_{\mathcal{SG}.target}$. We can thus construct the path $q = p \bullet (r_i, \ldots r_n \bullet p')$ such that $path(q, \mathcal{CG})$ and $p \sqsubseteq q$. Since $q \in L(N)$ we can also conclude that $satisfies(q, \mathcal{SG})$.

$\square$

Lemma 1. *Given a class graph $\mathcal{CG}$, an embedded strategy $\mathcal{SG}$ and the traversal automaton N for $\mathcal{SG}$ and $\mathcal{CG}$ for every state q_v in $N.Q\backslash\{q_\perp, q_0, q_{\mathcal{SG}.target}\}$, $\exists p' : path(p', \mathcal{CG}) \wedge p'.first = v$ such that $N.\delta^*(q_v, p'.tail) = q_{\mathcal{SG}.target}$.*

Proof. By the definition of $\mathcal{SG}$ (no dead states condition),

$$\exists r : path(r, \mathcal{SG}) \wedge r.first = v \wedge r.last = \mathcal{SG}.target$$

By the definition of $\mathcal{SG}$ (compatibility condition) we know that

$$\forall e \in \mathcal{SG}.edges : \exists r' : path(r', \mathcal{CG}) \wedge \\ expansion(r', (e.source, e.target), \mathcal{SG}.nodes)$$

Let $r' = r'_1, \ldots, r'_m$. By the definition of *expansion* and by construction of N, we know that $r'_1 = e.source$ and $r'_m = e.target$ and

$$\forall i : \;\; 1 < i < m : \;\; r'_i \notin \mathcal{SG}.nodes \qquad \wedge \\ (q_{r'_1}, r'_i, q_{r'_1}) \in N.\delta$$

Thus r'_i cannot move N to a stuck state. Observe that $q_{r'_1} = q_{e.source}$ and $q_{r'_n} = q_{e.target}$ for $e \in \mathcal{SG}.edges$. Thus

$$(q_{r'_1}, r'_n, q_{r'_n}) = (q_{e.source}, r'_n, q_{e.target})$$

and by construction of N we know that $(q_{e.source}, r'_n, q_{e.target}) \in N.\delta$. We can thus select

$$(q_{r'_1}, r'_2, q_{r'_1})(q_{r'_1}, r'_3, q_{r'_1}) \ldots (q_{r'_1}, r'_n, q_{r'_n})$$

and move N from $q_{r'_1}$ to $q_{r'_n}$.

We can repeat the same process for each strategy edge e_j connecting two consecutive nodes in r and obtain a set of paths π_j such that each $\pi_j.tail$ simulated on N at state $q_{e_j.source}$ leaves N in state $q_{e_j.target}$. Concatenating all π_j returns a path π in $\mathcal{CG}$ such that given N at state q_v, a simulating of N on $\pi.tail$ takes N from q_v to $q_{\mathcal{SG}.target}$. □

3.6 Walking an Object Graph

In this section, we define the operation $\texttt{walk}(O\langle\mathcal{CG}\rangle, N, o)$ which takes an object graph $O\langle\mathcal{CG}\rangle$ that conforms to a class graph $\mathcal{CG}$, a traversal automaton N created from $\mathcal{CG}$ and a strategy $\mathcal{SG}$, and an starting object (o) in $O.nodes$ and traverses O guided by traversal automaton N. We also prove that if `walk` terminates it returns a set P that is the largest set containing all paths p in O starting with object o such that $meta(p)$ is a prefix of a path q in O and q satisfies $\mathcal{SG}$. We say that a path p in O is valid under $\mathcal{SG}$, denoted $valid(p, \mathcal{SG})$, iff

$$\exists q : path(q, \mathcal{CG}) \wedge meta(p) \sqsubseteq q \wedge satisfies(q, \mathcal{SG})$$

```
Input:  og : an object graph that conforms to a class graph cg
        ta : a traversal automaton
        os : an object in the object graph.
Output: a walk of og starting a os and guided by ta
1 return walkHelper (os, og,ta,ta.$q_0$,{os},$\emptyset$)
```

Figure 3.9: `walk`(og,ta,os)

Theorem 2. *For all $\mathcal{CG}$ and for all traversal automata N, and all object graphs $O\langle\mathcal{CG}\rangle$ and o_s such that $o_s \in O.nodes$ and $meta(o_s) = \mathcal{SG}.source$, let $P =$ $\texttt{walk}(O, N, o_s)$ then for all paths p in P, $valid(p, \mathcal{SG})$ holds.*

Proof.

By induction on the depth of recursive calls to `walkHelper`. Consider the first call to `walkHelper`. $P = \emptyset$ and we know that $meta(o_s) = \mathcal{SG}.source$. We also have that N is in state q_0 and by construction of N we know that there is a transition $(q_0, meta(o_s), q_{\mathcal{SG}.source})$. By Lemma 1 we know that there exists a path q such that $path(q, \mathcal{CG})$ and a simulation of N on $\mathcal{SG}.source, q$ moves N to $\mathcal{SG}.target$. The function `walkHelper` adds o_s to P (line 9). Thus,

```
Input:
    os    : the starting object, os ∈ og and $meta(\text{os}) = \mathcal{SG}.source$.
    og    : an object graph that conforms to a class graph cg
    ta    : a traversal automaton whose alphabet is cg.$nodes$
    state : the current state of the automaton ta
    reach : set of nodes from og to process
    P     : $\{allPrefixes(p) \mid \exists o' : o' \in \text{reach} \wedge p.first = \text{os}$
                                    $path((p, o'), \text{og}) \wedge$
                                    $\exists w : w \in L(\text{ta}) \wedge meta(p) \sqsubseteq w\}$
Output: set of paths $P'$ such that
    $\forall p : path(p, \text{og}) \wedge p.first = \text{os} \wedge$
        $\exists w : w \in L(\text{ta}) \wedge meta(p) \sqsubseteq w \iff$
        $o_1, \ldots, o_n \in P'$
if reach = ∅ then
    return P;
end
foreach $o$ ∈ reach do
    state' = ta.$\delta(meta(o), \text{state})$;
    if state' $== q_\perp$ then
        return P;
    end
    if $P = \emptyset$ then $P = \{o\}$;
    else foreach $p \in P$ do
        if $(p.last, o) \in$ og.$edges$ then
            P = P ∪ $\{p, o\}$;
        end
    end
    reach' = $\{o' \mid e \in \text{og}.outgoing(o) \wedge e.target = o'\}$ ;
    P = walkHelper (os, og, ta, state', reach', P);
end
return P;
```

Figure 3.10: `walkHelper`(os, og,ta,state,reach, P)

$\mathcal{SG}.source, p \in L(N)$ and by Definition 1 we can deduce that $satisfies((\mathcal{SG}.source, p), \mathcal{SG})$

Assume that after k recursive calls to `walkHelper` the theorem holds and consider the $k+1$ recursive call. By the definition of `walkHelper` line 16 is the only recursive call.

At call $k+1$ if reach is empty then we return P and by the induction hypothesis all paths in P satisfy $\mathcal{SG}$.

If reach is not empty then for each element o of reach we have two cases (lines 4 - 17):

1. $\mathsf{ta}.\delta(meta(o), \mathsf{state}) = q_{\perp}$ In this case, we return P and by the induction hypothesis we know that each element of P satisfies $\mathcal{SG}$.

2. $\mathsf{ta}.\delta(meta(o), \mathsf{state}) \neq q_{\perp}$ For each $p \in P$ we update P to include p, oc (lines 9 - 14) if there exists an edge $(p.last, \mathsf{oc}) \in O$, or, if P is the empty set we add the simple path o to P. By the induction hypothesis we know that $valid(p, \mathcal{SG})$. Let q' be the automaton's new state after reading in $meta(\mathsf{oc})$. Since $q' \neq q_{\perp}$ we know that by Lemma 1 there exists an input r such that simulating ta at state q' on r moves ta to its target node. Let $h = meta(p), meta(\mathsf{oc}), r$, then $h \in L(N)$ and by Definition 1 we can deduce $satisfies(h, \mathcal{SG})$. Finally, we create a new set reach' (line 15) that contains the all immediate neighbors of oc in O and return the result of the recursive call to walkHelper passing the same start object, object graph, traversal automaton, the new automaton state state', the new set of reachable nodes reach', and the set P.

$\square$

Theorem 3. *Given a class graph $\mathcal{CG}$, an embedded WYSIWYG strategy $\mathcal{SG}$, a traversal automaton N for $\mathcal{CG}$ and $\mathcal{SG}$, a conforming object graph O for $\mathcal{CG}$ and a starting object o_s such that $o_s \in O.nodes$, $meta(o_s) = \mathcal{SG}.source$, and, $P =$* `walk`(O, N, o_s) *then P is maximal.*

Proof. By contradiction. Assume that there exists a path p in O ($path(p, O)$) such that $valid(p, \mathcal{SG})$ and $p \notin P$. Let $p = p_1, p_2, \ldots, p_n$, an consider the first call to `walkHelper`$(o_s, O, N, N.q_0, \{o_s\}, \emptyset)$, where reach $= \{o_s\}$. Since reach is not the empty set (Figure 3.10, line 1), `walkHelper` iterates over the elements of reach (Figure 3.10, lines 4 - 17). We know that $N.\delta(meta(o_s), q_0) = q_{\mathcal{SG}.source}$ by construction of N and $P = \{o_s\}$ (Figure 3.10, lines 9 - 14). The recursive call to `walkHelper`$(o_s, O, N, q_{\mathcal{SG}.source}, \text{reach}', P)$ (Figure 3.10, line 16) where reach' contains all immediate neighbors of o_s in O (Figure 3.10, line 15). Thus, $p_2 \in$ reach'. Inspecting the execution of the first recursive call we know that reach is not the empty set and that state $= q_{\mathcal{SG}.source}$. Thus walkHelper iterates over all elements o in reach including p_2. By our assumption that $valid(p, \mathcal{SG})$ and by the properties of the traversal automaton N we know that $N.\delta(meta(p_2), q_{\mathcal{SG}.source}) = q$ and that $q \neq q_\perp$. Therefore $o_s p_2$ will become an element of P.

A similar reasoning can be used to show that for all pairs (p_i, p_{i+1}), for $1 \leq i < n$ in p the prefix $o_s p_2, \ldots, p_{i+1}$ is a member of P.

□

CHAPTER 4

AP translation to CLASSICJAVA

In this chapter we provide the semantics for a core subset of Demeter, called APCORE, as a translation from APCORE to CLASSICJAVA [14]. We also prove that given a well typed APCORE program at type t our translation generates a well typed CLASSICJAVA program at type t.

4.1 Notation

Before we discuss APCORE's abstract syntax and our translation we first introduce notation that we use throughout this chapter.

We use use an overbar over a symbol, *e.g.*, $\overline{\mathcal{D}}$, to denote a sequence of zero or more elements $\mathcal{D}$. We index individual elements of the list using a subscript, $i \in [0, n] : \mathcal{D}_i$. We also use the notation $\mathcal{D}_1 \ldots \mathcal{D}_n$ as another way to denote a sequence and we deploy this second notation in situations where confusion between indexes may arise.

We use C to range over class names and V to range over visitor names and B to range over both class names and visitor names. We use $\mathbb{SG}$ to denote the set of strategy graphs $\mathcal{SG}$, $\mathbb{CG}$ to denote the set of class graphs $\mathcal{CG}$ and $\mathbb{TG}$ to denote the set of traversal graphs $\mathcal{TG}$. We overload the symbol $\cdot$ and use it for both list concatenation and list append. We use $\bullet$ for the empty list and overload the symbol $\setminus$ to denote set difference but also an update or addition of an element in a list. For example, $\overline{f \mapsto v} \setminus f_i \mapsto v'$

will update the value pointed to by f_i if $f_i \in \overline{f}$, otherwise if $f_i \notin \overline{f}$ we add $f_i \mapsto v'$ to the list $\overline{f \mapsto v}$.

We use $\emptyset$ to denote the empty set and we use $S[x/y]$ to denote replacement of the element y with the element x in the set S. If $y \notin S$ then the replacement does not alter S. Replacements can be sequenced, *e.g.*, $S[x/y][p/q]$ performs the replacements one at at time in sequence, first we replace y with x and on the resulting set we then replace q with p. We also use the notation $C^{\mathtt{k}}$ to annotate the node C with `k`. We use $\mathbb{C}$ for a set of class names and $\mathbb{C}^{\mathtt{k}}$ for a set of annotated class nodes. We use the symbol $\mathbb{P}$ denote a set of paths and overload the functions *source* and *target* to return the source (or target) of an edge or path.

4.2 APCORE

4.2.1 APCORE Syntax

The surface syntax for APCORE is given in Figure 4.1. A complete program $\mathcal{P}$ in APCORE consists of a list of class and/or visitor definitions $\overline{\mathcal{D}}$ followed by an expression e representing the program's main method. A class definition must provide a name for the class followed by the keyword `extends` and the name of its superclass. The name of the superclass cannot be omitted except for the predefined class `Object`. A class definition comprises of a list of field definitions $\overline{C\ f};$ followed by a unique constructor method, a list of instance methods, and a list of *adaptive methods*.

A visitor definition is similar to a class definition but can only extend other visitors. A visitor definition must provide a name for the visitor followed by the keyword `extends` and the name of its *supervisitor*. The name of the supervisitor cannot be omitted except for the build in `Visitor`; `Visitor` plays the same role for visitors as the role played by `Object` for classes. A visitor definition comprises of a list of field definitions, a unique constructor

$\mathcal{P}$	$::= \overline{\mathcal{D}}\ e$	Programs
$\mathcal{D}$	$::= \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{C\ f;}\ \mathcal{K}\ \overline{\mathcal{M}}\ \overline{\mathcal{A}}\}$	Class
	$\mid\ \texttt{visitor}\ V\ \texttt{extends}\ V\ \{\ \overline{C\ f;}\ \mathcal{K}\ \overline{\mathcal{M}}\ \overline{\mathcal{V}}\}$	Visitor
$\mathcal{K}$	$::= B(\overline{C\ x}, \overline{C\ y})\{\texttt{super}(\overline{x}); \overline{\texttt{this}.f := y;}\}$	Constructors
$\mathcal{M}$	$::= t\ m(\overline{C\ x})\{e\}$	Method
$\mathcal{A}$	$::= @\{\mathcal{CC}\}\ t\ m(\overline{C\ x})\ \texttt{with}\ \mathcal{SG}[V]$	Adaptive Method
$\mathcal{V}$	$::= \texttt{before}\ \ \{\overline{C\ \rightarrow \{e\}}\}$	Before Method
	$\mid\ \texttt{after}\ \ \{\overline{C\ \rightarrow \{e\}}\}$	After Method
	$\mid\ t\ \texttt{return}\ \ \{e\}$	Return Method
$\mathcal{CC}$	$::= tt \mid ff \mid \texttt{empty}(\mathcal{SG}) \mid \texttt{unique}(\mathcal{SG})$	Constraints
	$\mid\ \texttt{not}\ \mathcal{CC} \mid \texttt{or}(\mathcal{CC},\mathcal{CC}) \mid \texttt{and}(\mathcal{CC},\mathcal{CC})$	
e	$::= x \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) \mid e.f \mid e.f := e$	Expressions
	$\mid\ \texttt{super}.m(\overline{e}) \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid (C)e \mid e;e$	
t	$::= C$	Class Types
B	$::= C \mid V$	Class or Visitor Types
C	$::=$ *class name or* `Object`	
V	$::=$ *visitor name or* `Visitor`	
m	$::=$ *method name*	
f	$::=$ *field name*	
x	$::=$ *variable name or* `this`	
$\mathcal{SG}$	$::=$ *strategy graph*	

Figure 4.1: AP surface syntax.

method, a list of instance methods and a list of *visitor methods*.

Fields are defined as a pair with the first element being the field's type and the second element the field's name. A constructor method $\mathcal{K}$ has the same name as the class or visitor and takes the same number of arguments as the *total* number of class or visitor fields, including inherited fields. The body of the constructor method is fixed, first the superclass' constructor is called passing the appropriate arguments and then the current class' fields are initialized.

A method definition $\mathcal{M}$ is made up of a return type t, the method name m, a list of arguments $\overline{C\ x}$ and an expression as the method body. An adaptive method $\mathcal{A}$ is similar to a method definition with a constraint annotation @$\{\mathcal{CC}\}$ and with the method's body replaced with the keyword `with`, a *strategy specification* $\mathcal{SG}$ and a visitor name `V`.

Constraints are either *primitive* (*e.g.*, `empty`, `unique`, *tt*, *ff*) or composites created using constraint operators (*e.g.*, `not`, `or`, `and`). The arguments to primitive constraints are strategy specifications. The arguments to constraint operators are constraints. We leave the syntactic definition of a strategy specification unspecified but provide its definition here.[1]

Definition 2. ***(Strategy Specification)***
A strategy specification $\mathcal{SG}$ *is a graph* $G = (V, E, n_s, n_t)$ *with the following conditions:*

- *a unique node* $n_s \in V$ *as the source node,*
- *a unique node* $n_t \in V$ *as the target node,*
- *for every node* $n \in V$, *n is reachable from* n_s *and* n_t *is reachable from n*

We also assume the following operations on strategy definitions:

- $\mathcal{SG}$.*source* returns the source node (n_s) of strategy $\mathcal{SG}$,
- $\mathcal{SG}$.*target* returns the target node (n_t) of strategy $\mathcal{SG}$,
- $\mathcal{SG}$.*nodes* returns the node set V of strategy $\mathcal{SG}$, and,
- $\mathcal{SG}$.*edges* returns the set of edges E of strategy $\mathcal{SG}$.

Visitor definitions are similar to class definitions, instead of adaptive methods visitor definitions contain *visitor methods*. Visitor methods have the special names `before`, `after`, and `return`; `before` and `after` method definitions consist of a list of type and expression pairs, *i.e.*, $C \to \{e\}$ and do not specify a return type. Each type-expression pair represents a visit

[1]The exact meaning and usage of strategy specifications is explained in later subsections. The definition of strategy specifications is similar to the definition of strategy graph from § 3.

method with argument type C and method body e. A `return` method definition has a return type and an expression as the method's body.

Expressions in our grammar define typical expressions found in OO languages, *e.g.*, object creation with `new`, method invocation, field access and field update, local scoping with `let`, an if expression, sequencing and casting. Calls a super class' methods start with the special variable `super` followed by the method name and a list of arguments. Mathematical expressions using basic mathematical operators are also expressions in our language.

4.2.2 Static type system for APCORE

Type rules for APCORE begin with Figure 4.5 and start with the $\vdash_{\mathrm{P}}$ rule. A well typed APCORE program satisfies some simple predicates and relations, these are given in Figures 4.2, 4.3, and 4.4. The first three predicates in Figure 4.2 assert that class names are unique, and within a class definition field names and method names are unique. The relation $<_p^c$ captures direct class subtypes and $\leq_p^c$ captures class subtypes, direct or transitive. The predicate WFClasses($\overline{\mathcal{D}}$) asserts that the class hierarchy is an order. We use $\in_p^f$ to identify direct fields of a class and $\in_p^c$ to check that a type C or a supertype of C is a member of a list of types $\overline{D}$. The final two predicates in Figure 4.2 assert that classes which are extended are defined (CompleteClasses($\overline{\mathcal{D}}$)) and that method overriding preserves the method's type (ClassOverridesOK($\overline{\mathcal{D}}$)).

Figure 4.3 contains predicates that deal with Visitors and follow a similar pattern as the predicates for classes. The first predicate in Figure 4.3 asserts that visitor names and class names are distinct. The following three predicates assert that visitor names are unique and within a visitor definition field names and method names are unique. The relations $<_p^v$ and $\leq_p^v$ define the direct and transitive subtype relationship for visitor types;

ClassesOnce($\overline{\mathcal{D}}$) *Unique class names*

$$\texttt{class}\ C\ \cdots\ \{\cdots\}\cdots$$
$$\texttt{class}\ C'\ \cdots\ \{\cdots\} \in \overline{\mathcal{D}} \Longrightarrow C \neq C'$$

1MethodPerClass($\overline{\mathcal{D}}$) *Inside a class method names are unique*

$$\texttt{class}\ C\ \cdots\ \{\ \cdots\ t\ m(\cdots)\cdots t'\ m'(\cdots)\cdots\} \in \overline{\mathcal{D}} \Longrightarrow m \neq m'$$

1FieldPerClass($\overline{\mathcal{D}}$) *Inside a class field names are unique*

$$\texttt{class}\ C\ \cdots\ \{\ \cdots C\ f;\ \ \cdots C'\ f';\ \cdots\ \} \in \overline{\mathcal{D}} \Longrightarrow f \neq f'$$

$<^c_p$ *Direct Subclass*

$$C <^c_p C' \iff \texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\} \in \overline{\mathcal{D}}$$

$\leq^c_p$ *Subclass*

$$\leq^c_p\ \equiv\ \textit{transitive, reflexive closure of}\ <^c_p$$

WFClasses($\overline{\mathcal{D}}$) *Class hierarchy is an order*

$$\leq^c_p\ \textit{is antisymmetric}$$

$\in^f_p$ *Direct field*

$$C \in^f_p C' \iff \texttt{class}\ C \cdots \{\cdots C'\ f;\cdots\} \in \overline{\mathcal{D}}$$

$\in^c_p$ *List membership for list of Classes*

$$C \in^c_p \overline{D} \iff \exists D_i : C \leq^c_p D_i$$

CompleteClasses($\overline{\mathcal{D}}$) *Classes that are extended are defined*

$$\textit{range}(<^c_p) \subseteq \textit{domain}(<^c_p) \cup \{\texttt{Object}\}$$

ClassOverridesOK($\overline{\mathcal{D}}$) *Class method overriding preserves the type*

$$\texttt{class}\ C\ \cdots\{\cdots m(\cdots)\cdots\}\ \cdots\ \texttt{class}\ C'\ \cdots\{\cdots m(\cdots)\cdots\} \in \overline{\mathcal{D}}$$
$$\Longrightarrow \textit{mtype}(m, C) = \textit{mtype}(m, C')\ \vee\ C \not\leq^c_p C'$$

Figure 4.2: Predicates on class definitions.

DistinctVCNames($\overline{\mathcal{D}}$) *Visitor and class names are distinct*

$$\texttt{visitor}\ V\ \cdots\ \{\cdots\}\cdots\ \texttt{class}\ C\ \cdots\ \{\cdots\} \in \overline{\mathcal{D}} \implies V \neq C$$

VisitorsOnce($\overline{\mathcal{D}}$) *Unique visitor names*

$$\texttt{visitor}\ V\ \cdots\ \{\cdots\}\cdots\ \texttt{visitor}\ V'\ \cdots\ \{\cdots\} \in \overline{\mathcal{D}} \implies V \neq V'$$

1MethodPerVisitor($\overline{\mathcal{D}}$) *Inside a visitor method names are unique*

$$\texttt{visitor}\ V\ \cdots\ \{\ \cdots\ t\ m(\cdots)\ \cdots t'\ m'(\cdots)\ \cdots\} \in \overline{\mathcal{D}} \implies m \neq m'$$

1FieldPerVisitor($\overline{\mathcal{D}}$) *Inside a visitor field names are unique*

$$\texttt{visitor}\ V\ \cdots\ \{\ \cdots\ C\ f; \cdots C'\ f';\ \cdots\ \} \in \overline{\mathcal{D}} \implies f \neq f'$$

$<^v_p$ *Direct Subvisitor*

$$V <^v_p V' \iff \texttt{visitor}\ V\ \texttt{extends}\ V'\ \{\cdots\} \in \overline{\mathcal{D}}$$

$\leq^v_p$ *Subvisitor*

$$\leq^v_p\ \equiv\ \textit{transitive, reflexive closure of}\ <^v_p$$

$\prec:$ *Direct Subtype*

$$B \prec: B' \iff B <^c_p B' \lor B <^v_p B'$$

$\preceq:$ *Subtype*

$$B \preceq: B' \iff B \leq^c_p B' \lor B \leq^v_p B'$$

WFVisitors($\overline{\mathcal{D}}$) *Visitor hierarchy is an order*

$\leq^v_p$ *is antisymmetric*

CompleteVisitors($\overline{\mathcal{D}}$) *Visitors that are extended are defined*

$$\textit{range}(<^v_p) \subseteq \textit{domain}(<^v_p) \cup \{\texttt{Visitor}\}$$

VisitorOverridesOK($\overline{\mathcal{D}}$) *Visitor method overriding preserves the type*

$$\texttt{visitor}\ V\ \cdots\{\cdots m(\cdots)\cdots\}\ \cdots\ \texttt{visitor}\ V'\ \cdots\{\cdots m(\cdots)\cdots\} \in \overline{\mathcal{D}}$$
$$\implies \textit{mtype}(m, V) = \textit{mtype}(m, V') \lor V \not\leq^v_p V'$$

Figure 4.3: Predicates on visitor definitions (Part I).

OneReturnMethod($\overline{\mathcal{D}}$) *Visitors have one* `return` *method*

$$\begin{array}{l} \texttt{visitor}\ V\ \cdots\ \{\cdots\ t\ \texttt{return}\ \{\cdots\}\ \cdots \\ \qquad \cdots\ t\ m(\cdots)\ \cdots\} \in \overline{\mathcal{D}} \\ \Longrightarrow m \neq \texttt{return} \end{array}$$

WFBeforeMethod($\overline{\mathcal{D}}$) *Unique types inside before methods*

$$\begin{array}{l} \texttt{visitor}\ V\ \cdots\ \{\cdots \\ \qquad \texttt{before}\ \ \{\cdots\ C\ \rightarrow \{\cdots\}\ \cdots \\ \qquad\qquad \cdots\ C'\ \rightarrow \{\cdots\}\ \cdots\}\} \in \overline{\mathcal{D}} \\ \Longrightarrow C \neq C' \end{array}$$

WFAfterMethod($\overline{\mathcal{D}}$) *Unique types inside after methods*

$$\begin{array}{l} \texttt{visitor}\ V\ \cdots\ \{\cdots \\ \qquad \texttt{after}\ \ \{\cdots\ C\ \rightarrow \{\cdots\}\ \cdots \\ \qquad\qquad \cdots\ C'\ \rightarrow \{\cdots\}\ \cdots\}\} \in \overline{\mathcal{D}} \\ \Longrightarrow C \neq C' \end{array}$$

Figure 4.4: Predicates on visitor definitions (Part II).

these are similar to the subtype relations given for classes in Figure 4.2. The relations $\prec:$ and $\preceq:$ generalize over the class and visitor subtype relations. The last three predicates in Figure 4.3 assert that the visitor type hierarchy is an order (WFVisitors($\overline{\mathcal{D}}$)), visitors that are extended are defined (CompleteVisitors($\overline{\mathcal{D}}$)), and, that method overriding inside visitor definitions preserves the method's signature (VisitorOverridesOK($\overline{\mathcal{D}}$)).

Figure 4.4 contains predicates that deal with visit methods. The predicate OneReturnMethod($\overline{\mathcal{D}}$) asserts that there is a unique return method inside a visitor definition. The last two predicates in Figure 4.4 assert that the types given in the list of type-expression pairs in a `before` or `after` method are disjoint.

The type rules for APCORE classes and visitors are given in Figure 4.5 ($\vdash_{\mathrm{D}}$), the type rules for methods including adaptive methods and visit methods are given in Figure 4.6. Typing of adaptive methods deals with con-

$\boxed{\vdash_P \mathcal{P} : t}$

$$\frac{\begin{array}{c}\text{ClassesOnce}(\overline{\mathcal{D}}) \quad \text{VisitorsOnce}(\overline{\mathcal{D}}) \quad \text{DistinctVCNames}(\overline{\mathcal{D}}) \\ \text{1MethodPerClass}(\overline{\mathcal{D}}) \quad \text{1FieldPerClass}(\overline{\mathcal{D}}) \quad \text{CompleteClasses}(\overline{\mathcal{D}}) \\ \text{1MethodPerVisitor}(\overline{\mathcal{D}}) \quad \text{1FieldPerVisitor}(\overline{\mathcal{D}}) \quad \text{CompleteVisitors}(\overline{\mathcal{D}}) \\ \text{WFBeforeMethod}(\overline{\mathcal{D}}) \quad \text{WFAfterMethod}(\overline{\mathcal{D}}) \quad \text{OneReturnMethod}(\overline{\mathcal{D}}) \\ \text{ClassOverridesOK}(\overline{\mathcal{D}}) \qquad \text{VisitorOverridesOK}(\overline{\mathcal{D}}) \\ \text{WFClasses}(\overline{\mathcal{D}}) \qquad \text{WFVisitors}(\overline{\mathcal{D}}) \\ \overline{\mathcal{D}} \vdash_D \mathcal{D}_i : \mathsf{OK} \qquad \overline{\mathcal{D}}, [\,] \vdash_e e : t\end{array}}{\vdash_P \overline{\mathcal{D}}\, e : t}$$

$\boxed{\overline{\mathcal{D}} \vdash_D \mathcal{D} : \mathsf{OK}}$

$$\frac{\begin{array}{c}\overline{\mathcal{D}}, C \vdash_K \mathcal{K} : \mathsf{OK} \\ \overline{\mathcal{D}} \vdash_t C_i \qquad i \in [1, n] \\ \overline{\mathcal{D}}, C \vdash_M \mathcal{M}_j : \mathsf{OK} \qquad j \in [1, m] \\ \overline{\mathcal{D}}, C \vdash_A \mathcal{A}_k : \mathsf{OK} \qquad k \in [1, p]\end{array}}{\overline{\mathcal{D}} \vdash_D \texttt{class}\ C\ \texttt{extends}\ D\ \{C_1\ f_1; \ldots C_n\ f_n;\ \mathcal{K}\ \mathcal{M}_1 \ldots \mathcal{M}_m\ \mathcal{A}_1 \ldots \mathcal{A}_p\} : \mathsf{OK}}$$

$$\frac{\begin{array}{c}\overline{\mathcal{D}}, V \vdash_K \mathcal{K} : \mathsf{OK} \\ \overline{\mathcal{D}} \vdash_t C_i \qquad i \in [1, n] \\ \overline{\mathcal{D}}, V \vdash_M \mathcal{M}_j : \mathsf{OK} \qquad j \in [1, m] \\ \overline{\mathcal{D}}, V \vdash_{VM} \mathcal{V}_k : \mathsf{OK} \qquad k \in [1, p]\end{array}}{\overline{\mathcal{D}} \vdash_D \texttt{visitor}\ V\ \texttt{extends}\ Z\ \{C_1\ f_1; \ldots C_n\ f_n;\ \mathcal{K}\ \mathcal{M}_1 \ldots \mathcal{M}_m\ \mathcal{V}_1 \ldots \mathcal{V}_p\} : \mathsf{OK}}$$

Figure 4.5: Type Rules for Classes and Visitors

straint evaluation. The rules for constraint evaluation are given in Figure 4.9, auxiliary predicates and definitions are given in Figures 4.7 and 4.8. Type rules for expressions are given in Figures 4.10 and 4.11. Auxiliary definitions used as part of the type rules for APCORE can be found in Figures 4.12 (field lookup), Figure 4.13 (constructor method lookup), and Figure 4.14 (method lookup).

Typing of an APCORE program begins with the rule $\vdash_P \overline{\mathcal{D}}\ e : t$ where first we verify that the predicates concerning class and visitor definitions hold for $\overline{\mathcal{D}}$ (Figure 4.5). We then check that each class or visitor definition is well typed $\overline{\mathcal{D}} \vdash_D \mathcal{D}_i : \mathsf{OK}$. The type of the whole program is given by the type of the main expression e at the empty type environment $[\,]$.

The rules for checking classes and visitors ($\overline{\mathcal{D}} \vdash_{\mathrm{D}} \mathcal{D} : \mathsf{OK}$) follow a similar pattern, for classes we check that the constructor method is well typed, each of the field types are well defined and that methods and adaptive methods are well typed. The case for visitor definitions is similar but instead of adaptive methods we check that all visit methods are well typed.

A constructor method is well typed (Figure 4.6) if the constructor's arguments match the class or visitor fields, both defined and inherited (rule $\overline{\mathcal{D}}, B \vdash_{\mathrm{K}} \mathcal{K} : \mathsf{OK}$). For a well typed method we check that the return type and the method argument types are well defined and that the body of the method under a type environment that binds `this` and each argument appropriately is of type t (rule $\overline{\mathcal{D}}, B \vdash_{\mathrm{M}} \mathcal{M} : \mathsf{OK}$). Note that the typing of a method's body allows for subsumption (Figure 4.11 rule TSUB) and that methods return class types only.

For visit methods we have two rules, one for dealing with `before` and `after` visit methods and one for `return` methods. For before and after methods we check that for each type-expression pair $C_i \rightarrow \{ e_i \}$, the type provided is well defined and that the expression under a type environment that appropriately binds `this` to the current class' type and `host` to C_i is of type `Object`. The typing rule allows for subsumption here as well. For the `return` method the rule is similar but only types the expression e under a type environment that only binds `this` and verifies that the type of e is a subtype of the `return`'s method declared returned type.

Finally, for a well typed adaptive method we first check that the method's return type, argument types and the visitor name are well defined. We then verify that the strategy's source node has the same name as the adaptive method's defining class and that the node names in the strategy are a subset of the program's set of class names. The next premise verifies that the constraint attached to the adaptive method evaluates to true and that the visitor's constructor method argument types match the adaptive method's argument types. Also, the visitor's return method is a subtype of the adap-

$$\boxed{\overline{\mathcal{D}}, B \vdash_{\mathrm{K}} \mathcal{K} : \mathsf{OK}}$$

$$\frac{\mathit{ftypes}(B) = \overline{C'} \cdot \overline{C''}}{\overline{\mathcal{D}}, B \vdash_{\mathrm{K}} B(\overline{C'\ x}, \overline{C''\ y})\{\mathtt{super}(\overline{x}); \overline{\mathtt{this}.f := y;}\} : \mathsf{OK}}$$

$$\boxed{\overline{\mathcal{D}}, B \vdash_{\mathrm{M}} \mathcal{M} : \mathsf{OK}}$$

$$\frac{\overline{\mathcal{D}} \vdash_{\mathrm{t}} t \qquad \overline{\mathcal{D}} \vdash_{\mathrm{t}} C_i \qquad \overline{\mathcal{D}}, [\mathtt{this} : B, \overline{x : C}] \vdash_{\mathrm{s}} e : t}{\overline{\mathcal{D}}, B \vdash_{\mathrm{M}} t\ m(\overline{C\ x})\{\ e\ \} : \mathsf{OK}}$$

$$\boxed{\overline{\mathcal{D}}, V \vdash_{\mathrm{VM}} \mathcal{V} : \mathsf{OK}}$$

$$\frac{\begin{array}{c}\overline{\mathcal{D}} \vdash_{\mathrm{t}} C_i \qquad \alpha \in \{\mathtt{before}, \mathtt{after}\} \\ \overline{\mathcal{D}}, [\mathtt{this} : V, \mathtt{host} : C_i] \vdash_{\mathrm{s}} e_i : \mathtt{Object}\end{array}}{\overline{\mathcal{D}}, V \vdash_{\mathrm{VM}} \alpha\ \{\overline{C \to \{\ e\ \}}\} : \mathsf{OK}} \qquad \frac{\overline{\mathcal{D}} \vdash_{\mathrm{t}} t \qquad \overline{\mathcal{D}}, [\mathtt{this} : V] \vdash_{\mathrm{s}} e : t}{\overline{\mathcal{D}}, V \vdash_{\mathrm{VM}} t\ \mathtt{return}\ \{\ e\ \} : \mathsf{OK}}$$

$$\boxed{\overline{\mathcal{D}}, C \vdash_{\mathrm{A}} \mathcal{A} : \mathsf{OK}}$$

$$\frac{\begin{array}{c}\overline{\mathcal{D}} \vdash_{\mathrm{t}} t \qquad \overline{\mathcal{D}} \vdash_{\mathrm{t}} C_i \qquad \overline{\mathcal{D}} \vdash_{\mathrm{V}} V \\ \mathcal{SG}.\mathit{source} = C \qquad \mathcal{SG}.\mathit{nodes} \subseteq \overline{\mathcal{D}}.\mathit{cnames} \\ \overline{\mathcal{D}} \vdash_{\mathrm{sg}} \mathcal{CC} : \mathit{tt} \qquad \mathit{ktype}(V) = \overline{C} \to V \\ \mathit{mtype}(\mathtt{return}, V) = \overline{t'} \to t'_{,} \qquad t'' \leq_p^c t \\ \mathit{visitTypes}(V) \subseteq \mathcal{SG}.\mathit{nodes}\end{array}}{\overline{\mathcal{D}}, C \vdash_{\mathrm{A}} @\{\mathcal{CC}\}\ t\ m(\overline{C\ x})\ \mathtt{with}\ \mathcal{SG}[V] : \mathsf{OK}}$$

Figure 4.6: Type Rules for Methods

tive method's return type. The last premise checks that the set of types used inside the visitor's before and after methods (*visitTypes* is defined in Figure 4.22) is a subset of the strategy's node names. The last premise imposes our strict interface between strategy and visitor definitions.

To explain constraint evaluation we first introduce the definitions for class graph and traversal graph.

4.2.2.1 Strategy and Traversal Graphs

The definitions of a class graph and a traversal graph rely on labeled graphs.[2] We define a graph $G = (V, E)$ as a tuple containing the set of node names V and a set of edges E. The edge set for a normal graph is a set of ordered pairs of nodes. We define a labeled graph as $G = (V, E, L)$ where L is a set of labels. The set of edges for a labeled graph is a set of ordered 3-tuples, (C, x, C'), consisting of a node $C \in V$, a label $x \in L$ and a node $C' \in V$. We use the functions *nodes* and *edges* on graphs that return the set of nodes and the set of edges of a graph respectively.

For normal graphs we define paths to be a sequence of nodes such that each pair of nodes in a path is in the graph's set of edges. For a labeled graph a path is a sequence of node pairs interposed by labels, *e.g.*, $C_1x_1C_2 \ldots C_{n-1}x_{n-1}C_n$ such that each node-label-node triple is in the graph's edge set. We also talk about paths in labeled graphs where we only consider node names and exclude labels, *i.e.*, $C_1x_1C_2 \ldots C_{n-1}x_{n-1}C_n$ can also be given as $C_1C_2 \ldots C_{n-1}C_n$ when we are only concerned with path nodes.

A *class graph* (Figure 4.7) is a representation of a program's class definitions as a labeled graph where classes represent nodes and inheritance and containment relationships (*i.e.*, fields) represent edges. Containment edges are labeled with field names, *i.e.*, for a class C with a field f of type C' the class graph will contain an edge (C, f, C'). Inheritance edges are special and are represented with two edges, one with an upward direction going from the subclass to the superclass and one with a downward direction going from the superclass to the subclass. Therefore a class graph contains two edges for each inheritance relation between two classes; one edge is labeled with $\diamond\uparrow$ starting at the subclass and ending at the superclass and one edge labeled with $\diamond\downarrow$ starting at the superclass and ending at the subclass.

Given a class graph $\mathcal{CG}$ and two nodes C and C' we can calculate the

[2]A more general treatment of traversal graphs can be found in [6, 28]

$\boxed{cg(\overline{\mathcal{D}}) = \mathcal{CG}}$ *Class graph*

$$
\begin{array}{llll}
cg(\overline{\mathcal{D}}) = (V, E, L) & & & \\
\textit{where}\ V & = & \{C \mid & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\} \in \overline{\mathcal{D}} \quad \vee \\
& & & \texttt{class}\ C'\ \texttt{extends}\ C\ \{\cdots\} \in \overline{\mathcal{D}} \quad \vee \\
& & & \texttt{class}\ C'\ \texttt{extends}\ C''\ \{\cdots C\ f; \cdots\} \in \overline{\mathcal{D}} \quad \}
\end{array}
$$

$$
\begin{array}{lll}
E & = & \{(C', \diamond\!\downarrow, C) \mid \texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\} \in \overline{\mathcal{D}}\} \quad \cup \\
& & \{(C, \diamond\!\uparrow, C') \mid \texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\} \in \overline{\mathcal{D}}\} \quad \cup \\
& & \{(C, f, C') \mid \texttt{class}\ C\ \texttt{extends}\ C''\ \{\cdots C'\ f; \cdots\} \in \overline{\mathcal{D}}\}
\end{array}
$$

$$
\begin{array}{lll}
L & = & \{f \mid \texttt{class}\ C \cdots \{\cdots C'\ f; \cdots\} \in \overline{\mathcal{D}}\} \quad \cup \\
& & \{\diamond\!\uparrow, \diamond\!\downarrow\}
\end{array}
$$

$\boxed{\mathcal{SG}.pathset = \mathbb{P}}$ *All strategy satisfying paths in $\mathcal{SG}$*

$$
\begin{array}{llll}
\mathcal{SG}.pathset = \{\overline{C} \mid & C_0 = \mathcal{SG}.source & \wedge & C_n = \mathcal{SG}.target \quad \wedge \\
& C_i \in \mathcal{SG}.nodes & \wedge & (C_i, C_{i+1}) \in \mathcal{SG}.edges\}
\end{array}
$$

$\boxed{\mathcal{CG}.paths(C, C) = \mathbb{P}}$ *Paths in $\mathcal{CG}$ given source and target types*

$$
\begin{array}{llll}
\mathcal{CG}.paths(A, B) = \{(C_0, l_1, C_1), \ldots, (C_{n-1}, l_n, C_n) \mid & & & \\
& C_0 & = & A \quad \wedge \\
& C_n & = & B \quad \wedge \\
& (l_1 \ldots l_n) & = & (\diamond\!\downarrow^*\ f)?(\diamond\!\uparrow^*\ f\ \diamond\!\downarrow^*)^*(f\ \ \diamond\!\uparrow^*)? \\
& & & \textit{where}\ f \notin \{\diamond\!\uparrow, \diamond\!\downarrow\}\}
\end{array}
$$

$\boxed{\text{CPExpansion}(\overline{C}, \overline{C}, \mathbb{C})}$ *Path Expansion*

$$
\begin{array}{l}
\text{CPExpansion}(p, p', R) \iff p = C_0 \ldots C_n \quad \wedge \quad p' = C'_0 \ldots C'_m \quad \wedge \quad n \leq m \quad \wedge \\
\qquad \forall i : 0 \leq i < n : \\
\qquad\quad \exists j, l : 0 \leq j, l < m : \\
\qquad\qquad j < l \quad \wedge \quad C_i = C'_j \quad \wedge \quad C_{i+1} = C'_l \quad \wedge \\
\qquad\qquad \forall k : j < k < l : C'_k \notin R
\end{array}
$$

Figure 4.7: Auxiliary predicates for path sets (Part I).

set of all paths in $\mathcal{CG}$ that start from C and finish at C'. Figure 4.7 defines $\mathcal{CG}.\mathit{paths}(C, C)$ that returns a set $\mathbb{P}$ of all paths in $\mathcal{CG}$ that start from C and finish at C' and for each such path, the path's labels match the regular expression $(\diamond\!\downarrow^*\ f)?\ (\diamond\!\uparrow^*\ f\ \diamond\!\downarrow^*)^*\ (f\ \diamond\!\uparrow^*)?$ where $f \not\in \{\diamond\!\uparrow, \diamond\!\downarrow\}$. The pattern disallows paths that contain segments where the labels are a mixture of consecutive downward and upward inheritance edges.[3] A strategy specification is itself a graph (Definition 2) and it is used to define a set of abstract paths that an adaptive method will traverse. We thus have to analyze the program's class definitions to discover the concrete paths that satisfy the abstract paths. This begs the definition of how do we relate a concrete path to a given abstract path; the definition is given by *path expansion* (CPExpansion in Figure 4.7).

Path expansion consumes two paths (sequences of class names), p_1 and p_2, and a set of class names R and verifies that for each consecutive pair of nodes, C_i and C_j, in p_1 there exists a subsequence in p_2 that starts with C_i, ends with C_j and all class names between C_i and C_j in p_2 are not in R. We will use this last condition– to disallow nodes in R – to define WYSIWYG strategies by having $R = \mathcal{SG}.\mathit{nodes}$ (Figure 4.8, the use of *allpaths* in *createtg*).

A traversal graph is a labeled annotated graph that captures all the concrete paths in a $\mathcal{CG}$ that satisfy the abstract paths defined by a $\mathcal{SG}$. We define a labeled annotated graph as $G = (V^{\mathrm{k}}, E^{\mathrm{k}}, L^{\mathrm{k}})$ where V^{k} is a finite set of annotated nodes, E^{k} is a finite set of edges, L^{k} is a finite set of labels and all annotations for all three sets are drawn from a set $\alpha = \{\mathbb{Z} \cup \mathtt{s}\}$ where $\mathtt{s}$ is a special symbol. A labeled annotated edge is a labeled edge where both nodes and the label are annotated.

Our algorithm for calculating the traversal graph given a strategy graph $\mathcal{SG}$ and a set of class definition $\overline{\mathcal{D}}$ is given in Figure 4.8 by *gettg*.

Informally, the algorithms starts by creating the class graph for the set

[3]We could also capture the pattern with the *complement* of the language corresponding to $\sigma(.^*(\diamond\!\uparrow^+\ \diamond\!\downarrow^+)\ \mid\ (\diamond\!\downarrow^+\ \diamond\!\uparrow^+).^*)$

$\boxed{gettg(\mathbb{SG}, \overline{\mathcal{D}}) = \mathbb{TG}}$ *Traversal graph given a strategy and class definitions*

$$\frac{\mathcal{SG} = (V, E, n_s, n_t) \quad \mathcal{TG}_\emptyset = (\emptyset,\emptyset,\emptyset) \quad cg(\overline{\mathcal{D}}) = \mathcal{CG} \\ createtg(E, V, \mathcal{CG}, 1, \mathcal{TG}_\emptyset) = \mathcal{TG}}{gettg(\mathcal{SG}, \overline{\mathcal{D}}) = \mathcal{TG}}$$

$\boxed{createtg(E, V, \mathbb{CG}, \mathbb{Z}, \mathbb{TG}) = \mathbb{TG}}$ *Calculate the traversal graph*

$$\frac{}{createtg(\emptyset, V, \mathcal{CG}, k, \mathcal{TG}) = \mathcal{TG}}$$

$$\frac{\begin{array}{c} \mathcal{TG}_1 = (V_1, E_1, L_1) \quad copy(\mathcal{CG}, k) = \mathcal{CG}^{\mathrm{k}} \\ E = (C_1, C_2) \cup E' \quad copy((V, \emptyset, \emptyset), k) = (V', \emptyset, \emptyset) \\ allpaths(\mathcal{CG}^{\mathrm{k}}, C_1^{\mathrm{k}}, C_2^{\mathrm{k}}, V') = (V^{\mathrm{k}}, E^{\mathrm{k}}, L^{\mathrm{k}}) \\ V_2 = V_1 \cup V^{\mathrm{k}}[C_1^{\mathrm{s}}/C_1^{\mathrm{k}}][C_2^{\mathrm{s}}/C_2^{\mathrm{k}}] \\ E_2 = E_1 \cup E^{\mathrm{k}}[C_1^{\mathrm{s}}/C_1^{\mathrm{k}}][C_2^{\mathrm{s}}/C_2^{\mathrm{k}}] \\ L_2 = L_1 \cup L^{\mathrm{k}} \quad \mathcal{TG}_2 = (V_2, E_2, L_2) \\ createtg(E', V, \mathcal{CG}, k+1, \mathcal{TG}_2) = \mathcal{TG} \end{array}}{createtg(E, V, \mathcal{CG}, k, \mathcal{TG}_1) = \mathcal{TG}}$$

$\boxed{allpaths(\mathbb{CG}, \mathbb{C}, \mathbb{C}, \mathbb{C}) = \mathbb{CG}}$ *Reduce the class graph to represent path expansions.*

$$\begin{array}{lcccl} \multicolumn{5}{l}{allpaths(\mathcal{CG}, C_s, C_t, R) = (V, E, L)} \\ s.t.\ \ \forall p\colon p \in \mathcal{CG}.paths(C_s, C_t) & \wedge & \text{CPExpansion}((C_s, C_t), p, R) & \wedge \\ C_i \in p & \iff & C_i \in V & \wedge \\ l_i \in p & \iff & l_i \in L & \wedge \\ (C_i, l_i, C_{i+1}) \in p & \iff & (C_i, l_i, C_{i+1}) \in E & \end{array}$$

$\boxed{copy(\mathbb{CG}, \mathbb{Z}) = \mathbb{CG}}$ *Copy and annotate the class graph*

$$\begin{array}{llcl} \multicolumn{4}{l}{copy((V, E, L), k) = (V^{\mathrm{k}}, E^{\mathrm{k}}, L^{\mathrm{k}})} \\ s.t. & n \in V & \iff & n^{\mathrm{k}} \in V^{\mathrm{k}} \\ & l \in L & \iff & l^{\mathrm{k}} \in L^{\mathrm{k}} \\ & (n_1, l, n_2) \in E & \iff & (n_l^{\mathrm{k}}, l^{\mathrm{k}}, n_2^{\mathrm{k}}) \in E^{\mathrm{k}} \end{array}$$

$\boxed{\mathcal{TG}.paths(C, C) = \mathbb{P}}$ *Paths in $\mathcal{TG}$ given source and target types*

$$\begin{array}{lllll} \mathcal{TG}.paths(A, B) = \{(C_0, l_1, C_1), \ldots, (C_{n-1}, l_n, C_n) \mid & C_0 & = & A & \wedge \\ & C_n & = & B & \wedge \\ & (C_i, l_{i+1}, C_{i+1}) & \in & \mathcal{TG}.edges\} & \end{array}$$

Figure 4.8: Auxiliary predicates for path sets (Part II).

of classes given as input. Then for each strategy edge $e_i = (C_1, C_2)$ the algorithm creates a copy of the class graph (G) and annotates all class graph nodes and labels with the number i. Using the annotated class graph the algorithm then finds all path expansions for edge e_s in G and represents them as a graph G'. In the graph G' we then replace $C_1^{\mathtt{i}}$ with $C_1^{\mathtt{s}}$ and $C_2^{\mathtt{i}}$ with $C_2^{\mathtt{s}}$ to denote that these two nodes are strategy nodes. Once all the strategy edges have been processed the algorithm takes the union of all the resulting annotated graphs giving the final traversal graph for strategy $\mathcal{SG}$ and the set of class definitions $\overline{\mathcal{D}}$.

In Figure 4.8 *gettg* returns the traversal graph for a strategy $\mathcal{SG}$ and a set of class definitions $\overline{\mathcal{D}}$. The main function that creates the traversal graph is *createtg*. The function *gettg* creates an empty traversal graph and passes the set of strategy edges E, the set of strategy nodes V, the class graph $\mathcal{CG}$, the number 1 (the edge index to start) and the empty traversal graph to *createtg*. The function *createtg* calculates the traversal graph using an accumulator; *createtg* loops over the strategy edge set E, at each iteration we first copy the class graph ($copy(\mathcal{CG}, k) = \mathcal{CG}^{\mathtt{k}}$) and we make a copy of the strategy nodes ($copy((V, \emptyset, \emptyset), k) = (V', \emptyset, \emptyset)$). We then use *allpaths* to obtain a labeled annotated graph $(V^{\mathtt{k}}, E^{\mathtt{k}}, L^{\mathtt{k}})$ of all path expansions in $\mathcal{CG}^{\mathtt{k}}$ for the strategy edge (C_1, C_2). We also provide the set of all strategy nodes, annotated with the current edge index V'. The set V' is the exclude set used with CPExpansion that captures our condition for WYSIWYG paths. We then update the annotations for $C_1^{\mathtt{k}}$ and $C_2^{\mathtt{k}}$ to be $C_1^{\mathtt{s}}$ and $C_2^{\mathtt{s}}$ in the accumulator node set, edge set, and, label set. Finally we update our accumulator, increment our counter and repeat the loop until all strategy edges have been processed.

We now return to the evaluation of constraints that takes place during the typing of adaptive methods. The rule for adaptive methods verifies that the constraint annotation on the adaptive method yields *tt* in the given class graph. The rules for constraint verification are given in Figure 4.9.

$$\boxed{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \mathcal{CC} : \mathbb{B}}$$

$$\frac{\begin{array}{c} gettg(\mathcal{SG}, \overline{\mathcal{D}}) = (V, E, L) \\ E = \emptyset \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{empty}(\mathcal{SG}) : tt} \qquad \frac{\begin{array}{c} gettg(\mathcal{SG}, \overline{\mathcal{D}}) = (V, E, L) \\ E \neq \emptyset \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{empty}(\mathcal{SG}) : ff}$$

$$\frac{\begin{array}{c} \mathcal{SG} = (V, E, n_1, n_2) \\ gettg(\mathcal{SG}, \overline{\mathcal{D}}) = \mathcal{TG} \\ \mid \mathcal{TG}.paths(n_1^{\mathrm{s}}, n_2^{\mathrm{s}}) \mid = 1 \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{unique}(\mathcal{SG}) : tt} \qquad \frac{\begin{array}{c} \mathcal{SG} = (V, E, n_1, n_2) \\ gettg(\mathcal{SG}, \overline{\mathcal{D}}) = \mathcal{TG} \\ \mid \mathcal{TG}.paths(n_1^{\mathrm{s}}, n_2^{\mathrm{s}}) \mid \neq 1 \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{unique}(\mathcal{SG}) : ff}$$

$$\frac{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} c : ff}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{not}(c) : tt} \qquad \frac{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} c : tt}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{not}(c) : ff}$$

$$\frac{\begin{array}{c} \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_1 : b \\ \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_2 : b' \\ tt \in \{b, b'\} \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{or}(c_1, c_2) : tt} \qquad \frac{\begin{array}{c} \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_1 : b \\ \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_2 : b' \\ tt \notin \{b, b'\} \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{or}(c_1, c_2) : ff}$$

$$\frac{\begin{array}{c} \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_1 : tt \\ \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_2 : tt \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{and}(c_1, c_2) : tt} \qquad \frac{\begin{array}{c} \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_1 : b \\ \overline{\mathcal{D}} \vdash_{\mathrm{sg}} c_2 : b' \\ ff \in \{b, b'\} \end{array}}{\overline{\mathcal{D}} \vdash_{\mathrm{sg}} \texttt{and}(c_1, c_2) : ff}$$

Figure 4.9: Rules for Constraints checking

The `empty` primitive constraint verifies that the traversal graph obtained from the constraint's strategy specification and the given class graph has no edges. The `unique` primitive constraint verifies that the resulting traversal graph contains one, and only one, path. The remaining composite constraint definitions define logical operators on constraints such as `or`, `and` and `not`.

Figures 4.10 and 4.11 give the typing rules for APCORE expressions. The rule for `new` (TNEW) checks that the type of each expression e_i given as argument to `new` is a subtype of the class' constructor argument type. TLET first type checks e_1 and then type checks the body of the let expression e_2 in an environment where the bound variable x is of the same type as e_1. The rule TVAR checks that the variable is bound under the type environment Γ and returns its type. The rule for method calls (TMCALL) first typechecks the receiver e_0 and uses *mtype* (Figure 4.14) to search for the type signature of the method m starting from the type of e_0. Finally the rule for method calls checks that the type of each expression given as argument e_i is a subtype of the corresponding method's argument type. For a field get expression (TFGET) we first type the receiver e and we search for the type of field f starting from e's type using *fieldType* (Figure 4.12). Typing of a field set expression (TFSET) goes through the same steps as for a field get expression but also verifies that the expression that we are assigning to the field is a subtype of the field's declared type. Typing of a super call expression (TSUPERCALL) obtains the type for `this` using the current type environment and then searches for the method m starting from the immediate supertype of the type given to `this`. The rule for super calls checks that the type of each argument is a subtype of the corresponding method's argument types. A sequence expression (TSEQ) types both expressions e_1 and e_2 and the sequence expression is given the same type as the type of expression e_2. Finally a cast expression (TCAST) verifies that the expression's type and the type we are casting to are in a subtype relationship.

Figure 4.11 gives the definitions for our subsumption rule (TSUB), for a well defined class type (TCDEF) and for a well defined visitor type (TVDEF).

4.2.3 Translation to CLASSICJAVA

Our translation takes a well typed APCORE program and generates a CLASSICJAVA program. For each adaptive method definition in the input our translation generates a series of mangled CLASSICJAVA methods (names with a # subscript) that encode all valid paths in the input program. Informally we can decompose our translation into three steps.

- First all adaptive methods are translated to CLASSICJAVA methods.

$$\boxed{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t}$$

$$\text{TNew}\ \frac{\overline{\mathcal{D}} \vdash_{\mathrm{t}} C \quad \mathit{ftypes}(C) = \overline{t} \qquad \mathit{ktype}(C) = \overline{t} \to C \qquad \overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e_i : t_i}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} \texttt{new}\ C(\overline{e}) : C} \qquad \text{TLet}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_1 : t_1 \qquad \overline{\mathcal{D}}, \Gamma[x : t_1] \vdash_{\mathrm{e}} e : t}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} \texttt{let}\ x = e_1\ \texttt{in}\ e : t}$$

$$\text{TVar}\ \frac{x \in \mathit{domain}(\Gamma)}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} x \Rightarrow x : \Gamma(x)} \qquad \text{TMCall}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_0 : t_0 \qquad \mathit{mtype}(t_0, m) = (\overline{t} \to t) \qquad \overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e_i : t_i}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_0.m(\overline{e}) : t}$$

$$\text{TFGet}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t' \qquad \mathit{fieldType}(t', f_i) = t_i}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e.f_i : t_i} \qquad \text{TFSet}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t' \qquad \mathit{fieldType}(t', f_i) = t_i \qquad \overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e_1 : t_i}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e.f_i := e_1 : t_i}$$

$$\text{TSuper}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} \texttt{this} : t_0 \quad t_0 \prec: t_0' \qquad \mathit{mtype}(t_0', m) = (\overline{t} \to t) \qquad \overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e_i : t_i}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} \texttt{super}.m(\overline{e}) : t} \qquad \text{TSeq}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_1 : t_1 \qquad \overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_2 : t_2}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_1; e_2 : t_2}$$

$$\text{TCast}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e : t}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} (t)e : t}$$

Figure 4.10: Type Rules for Expressions Part I

$$\boxed{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e : t} \qquad \boxed{\overline{\mathcal{D}} \vdash_{\mathrm{t}} t}$$

$$\text{TSub}\ \frac{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t' \qquad t' \leq_p^c t}{\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e : t} \qquad \text{TCDef}\ \frac{t \in \mathit{domain}(<_p^c) \cup \{\texttt{Object}\}}{\overline{\mathcal{D}} \vdash_{\mathrm{t}} t}$$

$$\boxed{\overline{\mathcal{D}} \vdash_{\mathrm{V}} t}$$

$$\text{TVDef}\ \frac{t \in \mathit{domain}(<_p^v) \cup \{\texttt{Visitor}\}}{\overline{\mathcal{D}} \vdash_{\mathrm{V}} t}$$

Figure 4.11: Type Rules for Expressions Part II

All class fields $\boxed{\mathit{fields}(\mathcal{D}) = \overline{\mathcal{D}\ f}}$

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \mathit{fields}(\texttt{Visitor}) = \bullet$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}}}{\mathit{fields}(C) = \mathit{fields}(D) \cdot \overline{C\ f}} \qquad \frac{\texttt{visitor}\ V\ \texttt{extends}\ V'\ \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}}}{\mathit{fields}(V) = \mathit{fields}(V') \cdot \overline{C\ f}}$$

All class field types $\boxed{\mathit{ftypes}(\mathcal{D}) = \overline{\mathcal{D}}}$ *All class field names* $\boxed{\mathit{fnames}(\mathcal{D}) = \overline{\mathcal{D}}}$

$$\frac{\mathit{fields}(C) = \overline{\mathcal{D}\ f}}{\mathit{ftypes}(C) = \overline{\mathcal{D}}} \qquad \frac{\mathit{fields}(C) = \overline{\mathcal{D}\ f}}{\mathit{fnames}(C) = \overline{f}}$$

Field Type Lookup $\boxed{\mathit{fieldType}(\mathcal{D}, f) = t}$

$$\mathit{fieldType}(\texttt{Object}, \_) = \bullet \qquad \mathit{fieldType}(\texttt{Visitor}, \_) = \bullet$$

$$\frac{\texttt{class}\ C \cdots \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}} \quad f_i \in \overline{f}}{\mathit{fieldType}(C, f_i) = C_i} \qquad \frac{\texttt{visitor}\ V \cdots \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}} \quad f_i \in \overline{f}}{\mathit{fieldType}(V, f_i) = C_i}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}} \quad f_i \notin \overline{f}}{\mathit{fieldType}(C, f_i) = \mathit{fieldType}(D, f_i)} \qquad \frac{\texttt{visitor}\ V\ \texttt{extends}\ V'\ \{\overline{C\ f;} \cdots\} \in \overline{\mathcal{D}} \quad f_i \notin \overline{f}}{\mathit{fieldType}(V, f_i) = \mathit{fieldType}(V', f_i)}$$

Figure 4.12: Field Lookup functions

The translation takes each adaptive method and its strategy definition and calculates its traversal graph. The traversal graph and the visitor attached to the adaptive method are used to generate methods in all the classes that are in the traversal graph. There are two kinds of classes, classes that are mentioned in the strategy (annotated with an s) and classes that are not mentioned in the strategy (annotated with an integer). For classes that are mentioned in the strategy our translation checks to see if the attached visitor contains any before or after visit methods and generates method calls to the visitor methods.[4] For all classes that are part of a path that satisfies the strategy we gen-

[4]Visitor methods are translated to CLASSICJAVA methods with specific mangled names

Class constructor method lookup $\boxed{\mathit{cktype}(C) = \overline{C} \rightarrow C}$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots \mathcal{K} \cdots\} \in \overline{\mathcal{D}} \quad \mathcal{K} = C(\overline{D\ x}, \overline{D'\ y})\{\texttt{super}(\overline{x}), \overline{\texttt{this}.f := y;}\}}{\mathit{cktype}(C) = \overline{D} \cdot \overline{D'} \rightarrow C}$$

Visitor constructor method lookup $\boxed{\mathit{cktype}(C) = \overline{C} \rightarrow C}$

$$\frac{\texttt{visitor}\ V\ \texttt{extends}\ V'\ \{\cdots \mathcal{K} \cdots\} \in \overline{\mathcal{D}} \quad \mathcal{K} = V(\overline{C\ x}, \overline{C'\ y})\{\texttt{super}(\overline{x}), \overline{\texttt{this}.f := y;}\}}{\mathit{vktype}(V) = \overline{C} \cdot \overline{C'} \rightarrow V}$$

Constructor method lookup $\boxed{\mathit{ktype}(\mathcal{D}) = \overline{C} \rightarrow \mathcal{D}}$

$$\frac{\vdash_{\mathrm{t}} \mathcal{D}}{\mathit{ktype}(\mathcal{D}) = \mathit{cktype}(\mathcal{D})} \qquad \frac{\vdash_{\mathrm{V}} \mathcal{D}}{\mathit{ktype}(\mathcal{D}) = \mathit{vktype}(\mathcal{D})}$$

Figure 4.13: Constructor method type lookup

erate ClassicJava methods that perform the traversal. A field edge becomes a call to the generated method on the class' field, a $\diamond\!\uparrow$ edge becomes a super call and a $\diamond\!\downarrow$ generates a super call in the subclass (the target of the $\diamond\!\downarrow$ edge).

- Second, all visitor definitions are translated into ClassicJava classes. The `Visitor` becomes an empty visitor that extends `Object`. For each type-expression pair in a visitor definition, our translation generated a ClassicJava method with one argument. The naming convention used for these methods mangles the concatenation of the visit method name (before or after) an underscore and the type name *e.g.*, `before`$\_C_{\#}$.

- Third, all constructor methods (and calls to constructor methods) are rewritten. Constructor methods are rewritten into two methods, a

Class method type lookup $\boxed{cmtype(m, C) = \overline{C} \rightarrow C}$

$$\frac{\texttt{class}\ C \cdots \{\cdots \overline{\mathcal{M}} \cdots\} \in \overline{\mathcal{D}} \quad t\ m(\overline{D\ x})\{e\} \in \overline{\mathcal{M}}}{cmtype(m, C) = \overline{D} \rightarrow t}
\qquad
\frac{\texttt{class}\ C \cdots \{\cdots\ \overline{\mathcal{A}}\} \in \overline{\mathcal{D}} \quad @\{\mathcal{CC}\}\ t\ m(\overline{D\ x})\ \texttt{with}\ \mathcal{SG}[V] \in \overline{\mathcal{A}}}{cmtype(m, C) = \overline{D} \rightarrow t}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots \overline{\mathcal{M}} \cdots\} \in \overline{\mathcal{D}} \quad t\ m(\overline{D\ x})\{e\} \notin \overline{\mathcal{M}}}{cmtype(m, C) = cmtype(m, C')}
\qquad
\frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots \overline{\mathcal{A}}\} \in \overline{\mathcal{D}} \quad @\{\mathcal{CC}\}\ t\ m(\overline{D\ x})\ \texttt{with}\ \mathcal{SG}[V] \notin \overline{\mathcal{A}}}{cmtype(m, C) = cmtype(m, C')}$$

Visitor method type lookup $\boxed{vmtype(m, V) = \overline{C} \rightarrow C}$

$$\frac{\texttt{visitor}\ V \cdots \{\cdots \overline{\mathcal{M}} \cdots\} \in \overline{\mathcal{D}} \quad t\ m(\overline{C\ x})\{e\} \in \overline{\mathcal{M}}}{vmtype(m, V) = \overline{C} \rightarrow t}
\qquad
\frac{\texttt{visitor}\ V\ \texttt{extends}\ V'\ \{\cdots \overline{\mathcal{M}} \cdots\} \in \overline{\mathcal{D}} \quad t\ m(\overline{C\ x})\{e\} \notin \overline{\mathcal{M}}}{vmtype(m, V) = vmtype(m, V')}$$

Method type lookup $\boxed{mtype(m, \mathcal{D}) = \overline{C} \rightarrow C}$

$$\frac{\vdash_{\mathsf{t}} \mathcal{D}}{mtype(m, \mathcal{D}) = cmtype(m, \mathcal{D})}
\qquad
\frac{\vdash_{\mathsf{V}} \mathcal{D}}{mtype(m, \mathcal{D}) = vmtype(m, \mathcal{D})}$$

Figure 4.14: Method type lookup

no argument constructor and an initializer method that has the same signature as the original APCORE constructor and initializes all class fields. Calls to `new` in the APCORE program are translated into a let expression that first calls the empty constructor, binds the result to a fresh temporary variable name and then calls the generated initializer method.

The translation of APCORE programs to CLASSICJAVA programs is defined in Figures 4.15,to 4.21.

The first two rules in Figure 4.15 define the start of our translation; $\vdash \overline{\mathcal{D}}\ e\ \rightharpoonup_P\ \mathsf{defn}^*\ \mathsf{e}$ translates an APCORE program with a set of class and visitor definitions $\overline{\mathcal{D}}$ and the main expression e to a list of CLASSICJAVA definitions defn^* and a CLASSICJAVA expression e. The second rule $\vdash \overline{\mathcal{D}} \rightharpoonup_{\overline{D}}$

$$\boxed{\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}} \qquad \boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_{\overline{D}} \mathsf{defn}^*}$$

$$\frac{\begin{array}{c}\vdash \overline{\mathcal{D}} \rightharpoonup_{\overline{D}} \mathsf{defn}^* \\ \vdash e \rightharpoonup_e \mathsf{e}\end{array}}{\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}} \qquad \frac{\begin{array}{c}\vdash \overline{\mathcal{D}} \rightharpoonup_A \overline{\mathcal{D}'} \quad \vdash \overline{\mathcal{D}'} \rightharpoonup_V \overline{\mathcal{D}''} \quad \vdash \overline{\mathcal{D}''} \rightharpoonup_K \mathsf{defn}^* \\ m_1 = V\ \mathtt{init\_V_\#}()\{\mathtt{this}\} \\ m_2 = \mathtt{Object}\ \mathtt{return}\ \ ()\{\mathtt{new\ Object}()\} \\ \mathsf{d} = \mathtt{class\ Visitor\ extends\ Object}\{m_1 m_2\}\end{array}}{\vdash \overline{\mathcal{D}} \rightharpoonup_{\overline{D}} \mathsf{d} \cdot \mathsf{defn}^*}$$

$$\boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_A \overline{\mathcal{D}}} \qquad \boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_D \overline{\mathcal{D}}}$$

$$\frac{\begin{array}{c}\vdash \overline{\mathcal{D}} \rightharpoonup_D \overline{\mathcal{D}''} \\ \vdash \overline{\mathcal{D}''} \rightharpoonup_{tg} \overline{\mathcal{D}'}\end{array}}{\vdash \overline{\mathcal{D}} \rightharpoonup_A \overline{\mathcal{D}'}} \qquad \frac{\vdash \mathcal{D}_i \rightsquigarrow_D \mathcal{D}'_i \qquad i \in [1, n]}{\vdash \mathcal{D}_1, \ldots, \mathcal{D}_n \rightharpoonup_D \mathcal{D}'_1, \ldots, \mathcal{D}'_n}$$

$$\boxed{\vdash \mathcal{D} \rightsquigarrow_D \mathcal{D}}$$

$$\frac{\begin{array}{c}\vdash \mathcal{M}_j \rightsquigarrow_m \mathcal{M}'_j \qquad j \in [1, k] \\ C \vdash \mathcal{A}_i \rightsquigarrow_M \overline{\mathcal{M}}_i \qquad i \in [1, n]\end{array}}{\begin{array}{l}\vdash \mathtt{class}\ C \cdots \{\cdots \mathcal{M}_1, \ldots, \mathcal{M}_k \mathcal{A}_1, \ldots, \mathcal{A}_n \cdots\} \quad \rightsquigarrow_D \\ \quad \mathtt{class}\ C \cdots \{\cdots \mathcal{M}'_1, \ldots, \mathcal{M}'_k, \overline{\mathcal{M}}_1, \ldots, \overline{\mathcal{M}}_n \cdots\}\end{array}}$$

$$\boxed{\vdash \mathcal{M} \rightsquigarrow_m \mathcal{M}} \qquad \boxed{C \vdash \mathcal{A} \rightsquigarrow_M \overline{\mathcal{M}}}$$

$$\frac{\vdash e \rightharpoonup_e \mathsf{e}}{\begin{array}{l}\vdash t\ m(\overline{C\ x})\{e\} \rightsquigarrow_m \\ \quad t\ m(\overline{C\ x})\{\mathsf{e}\}\end{array}} \qquad \frac{\begin{array}{l}\mathcal{M}_1 = t\ m\_C\_\mathtt{return}_\#(V\ \mathtt{v}_\#)\{\mathtt{v}_\#.\mathtt{return}\ ()\} \\ \mathcal{M}_2 = t\ m(\overline{C\ x})\{ \\ \qquad \mathtt{let}\ \mathtt{v}_\# = \mathtt{new}\ V().\mathtt{init\_V_\#}(\overline{x})\ \mathtt{in} \\ \qquad \mathtt{let}\ \mathtt{tmp}_\# = \mathtt{this}.m\_C\_s\_\mathtt{trv}_\#(\mathtt{v}_\#)\ \mathtt{in} \\ \qquad \mathtt{this}.m\_C\_\mathtt{return}_\#(\mathtt{v}_\#)\}\end{array}}{\begin{array}{l}C \vdash @\{\mathcal{CC}\}\ t\ m(\overline{C\ x})\ \mathtt{with}\ \mathcal{SG}[V] \rightsquigarrow_M \\ \qquad \mathcal{M}_1 \mathcal{M}_2\end{array}}$$

$$\boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_{tg} \overline{\mathcal{D}}}$$

$$\frac{\begin{array}{c}\overline{\mathcal{D}} = \mathcal{D}_1, \ldots, \mathcal{D}_n \\ \overline{\mathcal{D}} \vdash \mathcal{D}_1; \overline{\mathcal{D}} \rightsquigarrow_{tg} \overline{\mathcal{D}}_1 \\ \overline{\mathcal{D}} \vdash \mathcal{D}_2; \overline{\mathcal{D}}_1 \rightsquigarrow_{tg} \overline{\mathcal{D}}_2 \\ \vdots \\ \overline{\mathcal{D}} \vdash \mathcal{D}_n; \overline{\mathcal{D}}_{n-1} \rightsquigarrow_{tg} \overline{\mathcal{D}}_n\end{array}}{\vdash \overline{\mathcal{D}} \rightharpoonup_{tg} \overline{\mathcal{D}}_n}$$

Figure 4.15: Compilation of AP methods

$\mathsf{d} \cdot \mathsf{defn}^*$ performs three translations that correspond to the three stages of our translation. First we translate all classes and their adaptive methods ($\vdash \overline{\mathcal{D}} \rightharpoonup_A \overline{\mathcal{D}'}$) to obtain a new set of APCORE definitions $\overline{\mathcal{D}'}$. The new set of APCORE classes is then translated further by $\vdash \overline{\mathcal{D}'} \rightharpoonup_V \overline{\mathcal{D}''}$ that rewrites visitor definitions into class definitions yielding a second set of APCORE definitions $\overline{\mathcal{D}''}$. The third rewrite $\vdash \overline{\mathcal{D}''} \rightharpoonup_K \mathsf{defn}^*$ rewrites $\overline{\mathcal{D}''}$ into a set of CLASSICJAVA class definitions by rewriting all constructor methods and all calls to constructors. The last step in the $\rightharpoonup_{\overline{D}}$ generates the `Visitor` class that contains default implementations for a constructor, an initializer and a return method.

Translation of APCORE class definitions ($\rightharpoonup_A$) translates first all class definitions and methods with $\rightharpoonup_D$. The translation of an adaptive method creates a CLASSICJAVA method that delegates to traversal methods generated by $\rightharpoonup_{tg}$.

The translation of class definitions relies on $\rightsquigarrow_D$ which iterates over all APCORE class definitions and generates CLASSICJAVA methods from APCORE methods. For normal APCORE methods we translate the method body and leave the method signature intact ($\rightsquigarrow_m$). For APCORE adaptive methods we generate two CLASSICJAVA methods ($\rightsquigarrow_M$); $m\_C\_\mathtt{return}_{\#}(V\ \mathtt{v}_{\#})$ that wraps a call to the visitor's return method and a CLASSICJAVA method with the same type as as the APCORE adaptive method that:

1. creates a new instance of the visitor V using the arguments passed to the adaptive method.

2. calls the traversal method $m\_C\_s\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})$ to start the traversal passing the newly created visitor object

3. and the last expression in the method's body calls $m\_C\_\mathtt{return}_{\#}$.

Our $\rightharpoonup_{tg}$ (Figure 4.15) rule deals with the generation of all methods responsible for traversing strategy selected paths (*e.g.*, $m\_C\_s\_\mathtt{trv}_{\#}$). We use

$\rightsquigarrow_{tg}$ to process the adaptive methods in each APCORE definition. Each application of $\rightsquigarrow_{tg}$ generates methods inside all CLASSICJAVA class definitions that take part in a traversal. Each application of $\rightsquigarrow_{tg}$ takes as input the resulting program of the previous application of $\rightsquigarrow_{tg}$. The first application of $\rightsquigarrow_{tg}$ takes the original program as input.

The generation of traversal methods for each adaptive method ($\rightsquigarrow_{tg}$, Figure 4.16) follows a similar pattern as our $\rightharpoonup_{tg}$ rule. Given the current APCORE class definition that we are processing, $\mathcal{D}$, we iterate over all adaptive methods $\mathcal{A}_i$ and apply our generation rule for an adaptive method ($\rightsquigarrow_A$). The application of $\rightsquigarrow_A$ generates a new program that includes all the necessary traversal methods for an adaptive method $\mathcal{A}$. Each application of $\rightsquigarrow_A$ takes as input the resulting program of the previous applications of $\rightsquigarrow_A$.

The heart of our translation phase deals with the generation of CLASSICJAVA methods that perform object traversals guided by the strategy specification of an adaptive method. The generation of traversal methods starts with the definition of $\rightsquigarrow_A$. Given the APCORE program text $\overline{\mathcal{D}}$ the current APCORE class name under translation C the adaptive method to translate $\mathcal{A}$ and the current translated program $\overline{\mathcal{D}'}$, $\rightsquigarrow_A$ produces an new translated program $\overline{\mathcal{D}''}$. The rule uses the strategy specification $\mathcal{SG}$ given in $\mathcal{A}$ to construct the traversal graph ($gettg(\mathcal{SG}, \overline{\mathcal{D}}) = \mathcal{TG}$). To start the generation of CLASSICJAVA code we use our method generation rule

$$P, C^{\mathrm{s}}, \mathcal{TG}, m\_C, V \vdash_m (\overline{\mathcal{D}'}; \emptyset) \Rightarrow (\overline{\mathcal{D}''}; S)$$

where we are providing

- the APCORE program text P,
- C^{s}, the current class is the source of our strategy (Figure 4.6),
- the traversal graph $\mathcal{TG}$,
- the prefix of the method name to use m_C,

$$\boxed{\overline{\mathcal{D}} \vdash \mathcal{D}; \overline{\mathcal{D}} \rightsquigarrow_{tg} \overline{\mathcal{D}}}$$

$$\frac{\begin{array}{c}\overline{\mathcal{D}}, C \vdash \mathcal{A}_1; \overline{\mathcal{D}} \rightsquigarrow_A \overline{\mathcal{D}}_1 \\ \overline{\mathcal{D}}, C \vdash \mathcal{A}_2; \overline{\mathcal{D}}_1 \rightsquigarrow_A \overline{\mathcal{D}}_2 \\ \vdots \\ \overline{\mathcal{D}}, C \vdash \mathcal{A}_n; \overline{\mathcal{D}}_{n-1} \rightsquigarrow_A \overline{\mathcal{D}'}\end{array}}{\overline{\mathcal{D}} \vdash \texttt{class}\ C \cdots \{\cdots \mathcal{A}_1 \ldots \mathcal{A}_n \cdots\}; \overline{\mathcal{D}} \rightsquigarrow_{tg} \overline{\mathcal{D}'}}$$

$$\boxed{\overline{\mathcal{D}}, C \vdash \mathcal{A}; \overline{\mathcal{D}} \rightsquigarrow_A \overline{\mathcal{D}}}$$

$$\frac{\begin{array}{c}\mathcal{SG} = (V, E, C, n_t) \quad gettg(\mathcal{SG}, \overline{\mathcal{D}}) = \mathcal{TG} \\ \overline{\mathcal{D}}, C^{\mathrm{s}}, \mathcal{TG}, m\_C, V \vdash_m (\overline{\mathcal{D}'}; \emptyset) \Rightarrow (\overline{\mathcal{D}''}; S)\end{array}}{\overline{\mathcal{D}}, C \vdash @\{\mathcal{CC}\}\ t\ m(\overline{C\ x})\ \texttt{with}\ \mathcal{SG}[V]; \overline{\mathcal{D}'} \rightsquigarrow_A \overline{\mathcal{D}''}}$$

$$\boxed{\overline{\mathcal{D}}, C^{\mathrm{k}}, \mathbb{TG}, m, C \vdash_m (\overline{\mathcal{D}}; \mathbb{C}^{\mathrm{k}}) \Rightarrow (\overline{\mathcal{D}}; \mathbb{C}^{\mathrm{k}})}$$

$$\frac{\mathcal{TG} = (V', E', L') \quad S = V'}{P, C^{\mathrm{k}}, \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}; S) \Rightarrow (\overline{\mathcal{D}}; S)} \qquad \frac{\mathcal{TG} = (V', E', L') \quad S \subset V' \quad C^{\mathrm{k}} \in S}{P, C^{\mathrm{k}}, \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}; S) \Rightarrow (\overline{\mathcal{D}}; S)}$$

$$\frac{\begin{array}{c}C^{\mathrm{k}} \notin S \quad \mathcal{TG} = (V', E', L') \quad S \subset V' \\ \mathcal{TG}.outgoing(C^{\mathrm{k}}) = E_1, \ldots, E_n \\ \mathcal{TG}.incoming(C^{\mathrm{k}}) = \overline{I} \\ P, m, C^{\mathrm{k}}, V \vdash (E_1, \ldots, E_n; \overline{I}) \rightharpoonup \mathcal{M}' \\ \overline{\mathcal{D}} = \texttt{class}\ C \cdots \{\cdots \overline{\mathcal{M}} \cdots\} \cdot \overline{\mathcal{D}''} \\ \overline{\mathcal{D}}_0 = \texttt{class}\ C \cdots \{\cdots\ \mathcal{M}' \cdot \overline{\mathcal{M}} \cdots\} \cdot \overline{\mathcal{D}''} \\ S_0 = S \cup \{C^{\mathrm{k}}\} \\ P, target(E_1), \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}_0; S_0) \Rightarrow (\overline{\mathcal{D}}_1; S_1) \\ P, target(E_2), \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}_1; S_1) \Rightarrow (\overline{\mathcal{D}}_2; S_2) \\ \vdots \\ P, target(E_n), \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}_{n-1}; S_{n-1}) \Rightarrow (\overline{\mathcal{D}}_n; S_n)\end{array}}{P, C^{\mathrm{k}}, \mathcal{TG}, m, V \vdash_m (\overline{\mathcal{D}}; S) \Rightarrow (\overline{\mathcal{D}}_n; S_n)}$$

Figure 4.16: Compilation of AP methods (Part I)

- the type of the Visitor V,
- the translated program thus far $\overline{\mathcal{D}'}$,
- the classes that we have already processed, the *seen set* (initialized to the empty set),

and the rule produces

- a new translated program $\overline{\mathcal{D}''}$,
- an updated seen set S,

Our method generation rule walks the traversal graph and generates traversal methods accordingly. Method generation maintains a seen set of traversal graph nodes skipping cycles and terminates when all the $\mathcal{TG}$ nodes have been processed, *i.e.*, $\mathcal{TG}.nodes = S$.

Generation proceeds for a traversal graph node $C^{\mathtt{k}}$ that is not in our seen set S by first obtaining the outgoing and incoming edge set for $C^{\mathtt{k}}$. We then use $P, m, C^{\mathtt{k}}, V \vdash (E_1, \ldots, E_n; \overline{I}) \rightharpoonup \overline{\mathcal{M}'}$ to generate traversal methods $\overline{\mathcal{M}'}$ inside class C, we update the definition of C in the translated program $\overline{\mathcal{D}_0}$ and we include $C^{\mathtt{k}}$ in our seen set. Finally, we recursively call the same rule passing along the target node of each outgoing edge and weave through the resulting translated program and seen set.

Code generation for a traversal method is described by $P, m, C^{\mathtt{k}}, C \vdash (\overline{E}; \overline{E}) \rightharpoonup \overline{\mathcal{M}}$ (Figure 4.17). The rule consumes the prefix of the method name to generate m, the current traversal graph node $C^{\mathtt{k}}$, the visitor V, and the outgoing and incoming edge set of $C^{\mathtt{k}}$. The rule produces a list of methods (one or two methods to be exact).

For a node $C^{\mathtt{k}}$ we always generate a method $m\_k\_\mathtt{trv}_{\#}$ in class C (Figure 4.17, rule $\overline{\mathcal{D}}, m, C^{\mathtt{k}}, C \vdash_{\mathtt{m}} (\overline{E}; \overline{E}) \rightharpoonup \mathcal{M}$). We generate an extra method $m\_k''\_\mathtt{trv}_{\#}$ in C, that delegates to $m\_k\_\mathtt{trv}_{\#}$, when we have an edge $(D^{\mathtt{k}''}, \diamond\!\downarrow^{\mathtt{k}'}, C^{\mathtt{k}})$ in our incoming edge set and the annotations on D and C differ (Fig-

$\boxed{\overline{\mathcal{D}}, m, C^{\mathtt{k}}, C \vdash (\overline{E};\overline{E}) \rightharpoonup \overline{\mathcal{M}}}$

$$\frac{\begin{array}{c}\exists e : e \in \overline{I} \quad e = (D^{\mathtt{k}''}, \diamond\!\downarrow^{\mathtt{k}'}, C^{\mathtt{k}}) \quad \mathtt{k} \neq \mathtt{k}'' \\ m, C^{\mathtt{k}}, V \vdash_{\mathtt{o}} e; \overline{O} \rightharpoonup \mathcal{M}_1; \overline{O'} \\ P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O'}; \overline{I}) \rightharpoonup \mathcal{M}_2\end{array}}{P, m, C^{\mathtt{k}}, V \vdash (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}_1 \mathcal{M}_2} \qquad \frac{\begin{array}{c}\nexists e : e \in I \quad e = (D^{\mathtt{k}''}, \diamond\!\downarrow^{\mathtt{k}'}, C^{\mathtt{k}}) \quad \mathtt{k} \neq \mathtt{k}'' \\ P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}\end{array}}{P, m, C^{\mathtt{k}}, V \vdash (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}}$$

$\boxed{m, C^{\mathtt{k}}, C \vdash_{\mathtt{o}} E; \overline{E} \rightharpoonup \mathcal{M}; \overline{E}}$

$$\frac{\begin{array}{c} e' = (D^{\mathtt{k}''}, \diamond\!\downarrow^{\mathtt{a}}, C^{\mathtt{k}}) \\ \nexists e : e \in \overline{O} \qquad e = (C^{\mathtt{k}}, \diamond\!\uparrow^{\mathtt{k}'}, D^{\mathtt{k}''}) \\ \mathcal{M} = \mathtt{Object}\ \ m\_k''\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{this}.m\_k\_\mathtt{trv}_{\#}(\mathtt{v}_{\#});\}\end{array}}{m, C^{\mathtt{k}}, V \vdash_{\mathtt{o}} e'; \overline{O} \rightharpoonup \mathcal{M}; \overline{O}}$$

$$\frac{\begin{array}{c} e' = (D^{\mathtt{k}''}, \diamond\!\downarrow^{\mathtt{a}}, C^{\mathtt{k}}) \\ \exists e : \overline{O} = \overline{O'} \cdot e \qquad e = (C^{\mathtt{k}}, \diamond\!\uparrow^{\mathtt{k}'}, D^{\mathtt{k}''}) \\ \mathcal{M} = \mathtt{Object}\ \ m\_k''\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{this}.m\_k\_\mathtt{trv}_{\#}(\mathtt{v}_{\#});\}\end{array}}{m, C^{\mathtt{k}}, V \vdash_{\mathtt{o}} e'; \overline{O} \rightharpoonup \mathcal{M}; \overline{O'}}$$

$\boxed{\overline{\mathcal{D}}, m, C^{\mathtt{k}}, C \vdash_{\mathtt{m}} (\overline{E};\overline{E}) \rightharpoonup \mathcal{M}}$

$$\frac{\begin{array}{c} \mathtt{k} \neq \mathtt{s} \\ \forall \mathtt{k}' \in \mathbb{Z} \quad (\_, \diamond\!\downarrow^{\mathtt{k}'}, C^{\mathtt{k}}) \notin \overline{I} \\ \mathtt{v}_{\#}, m, \mathtt{this} \vdash_E \overline{O} \rightharpoonup \mathtt{e} \\ \mathcal{M}_1 = \mathtt{Object}\ \ m\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}\}\end{array}}{P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}_1} \qquad \frac{\begin{array}{c} \mathtt{k} = \mathtt{s} \\ \forall \mathtt{k}' \in \mathbb{Z} \quad (\_, \diamond\!\downarrow^{\mathtt{k}'}, C^{\mathtt{k}}) \notin \overline{I} \\ \mathtt{v}_{\#}, P, \mathtt{this}, \mathtt{after}\ \ \vdash V; C \rightharpoonup_v \mathtt{e}_1 \\ \mathtt{v}_{\#}, m, \mathtt{e}_1 \vdash_E \overline{O} \rightharpoonup \mathtt{e}_2 \\ \mathtt{v}_{\#}, P, \mathtt{e}_2, \mathtt{before}\ \ \vdash V; C \rightharpoonup_v \mathtt{e} \\ \mathcal{M}_1 = \mathtt{Object}\ \ m\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}\}\end{array}}{P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}_1}$$

$$\frac{\begin{array}{c} \mathtt{k} \neq \mathtt{s} \\ \exists!(C_1^{\mathtt{k}'}, \diamond\!\downarrow^{\mathtt{k}''}, C^{\mathtt{k}}) \in \overline{I} \quad \mathtt{k}', \mathtt{k}'' \in \mathbb{Z} \cup \{\mathtt{s}\} \\ \mathtt{v}_{\#}, m, \mathtt{this} \vdash_E \overline{E} \rightharpoonup \mathtt{e}' \\ \mathit{fresh}(x) \\ \mathtt{e} = \mathtt{let}\ x = \mathtt{super}.m\_k'\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})\ \mathtt{in}\ \ \mathtt{e}' \\ \mathcal{M}_1 = \mathtt{Object}\ \ m\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}\}\end{array}}{P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}_1} \qquad \frac{\begin{array}{c} \mathtt{k} = \mathtt{s} \\ \exists!(C_1^{\mathtt{k}'}, \diamond\!\downarrow^{\mathtt{k}''}, C^{\mathtt{k}}) \in \overline{I} \quad \mathtt{k}', \mathtt{k}'' \in \mathbb{Z} \cup \{\mathtt{s}\} \\ \mathtt{v}_{\#}, P, \mathtt{this}, \mathtt{after}\ \ \vdash V; C \rightharpoonup_v \mathtt{e}_1 \\ \mathtt{v}_{\#}, m, \mathtt{e}_1 \vdash_E \overline{O} \rightharpoonup \mathtt{e}_2 \\ \mathtt{v}_{\#}, P, \mathtt{e}_2, \mathtt{before}\ \ \vdash V; C \rightharpoonup_v \mathtt{e}' \\ \mathit{fresh}(x) \\ \mathtt{e} = \mathtt{let}\ x = \mathtt{super}.m\_k'\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})\ \mathtt{in}\ \ \mathtt{e}' \\ \mathcal{M}_1 = \mathtt{Object}\ \ m\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}\}\end{array}}{P, m, C^{\mathtt{k}}, V \vdash_{\mathtt{m}} (\overline{O}; \overline{I}) \rightharpoonup \mathcal{M}_1}$$

Figure 4.17: Compilation of AP methods (Part II)

$\boxed{x, m, \mathtt{e} \vdash_E \overline{E} \rightharpoonup \mathtt{e}}$

$$\frac{}{x, m, \mathtt{e} \vdash_E \bullet \rightharpoonup \mathtt{e}} \qquad \frac{l^{\mathtt{k}} \notin \{\diamond\uparrow^{\mathtt{k}}, \diamond\downarrow^{\mathtt{k}}\} \quad \mathit{fresh}(y) \\ \mathtt{e}_1 = \mathtt{let}\ y = \mathtt{this}.l.m\_k'\_\mathtt{trv}_{\#}(x)\ \mathtt{in}\ \mathtt{e} \\ x, m, \mathtt{e}_1 \vdash_E \overline{E} \rightharpoonup \mathtt{e}'}{x, m, \mathtt{e} \vdash_E (\_, l^{\mathtt{k}}, C^{\mathtt{k}'}) \cdot \overline{E} \rightharpoonup \mathtt{e}'}$$

$$\frac{l^{\mathtt{k}} = \diamond\downarrow^{\mathtt{k}} \\ x, m, \mathtt{e} \vdash_E \overline{E} \rightharpoonup \mathtt{e}'}{x, m, \mathtt{e} \vdash_E (\_, l^{\mathtt{k}}, C^{\mathtt{k}'}) \cdot \overline{E} \rightharpoonup \mathtt{e}'} \qquad \frac{l^{\mathtt{k}} = \diamond\uparrow^{\mathtt{k}} \quad \mathit{fresh}(y) \\ \mathtt{e}_1 = \mathtt{let}\ y = \mathtt{super}.m\_k'\_\mathtt{trv}_{\#}(x)\ \mathtt{in}\ \mathtt{e} \\ x, m, \mathtt{e}_1 \vdash_E \overline{E} \rightharpoonup \mathtt{e}'}{x, m, \mathtt{e} \vdash_E (\_, l^{\mathtt{k}}, C^{\mathtt{k}'}) \cdot \overline{E} \rightharpoonup \mathtt{e}'}$$

Figure 4.18: Generation of expressions from a set of edges

ure 4.17, rule $m, C^{\mathtt{k}}, C \vdash_{\mathtt{o}} E; \overline{E} \rightharpoonup \mathcal{M}; \overline{E}$). When we generate an override method for a class C we also remove any outgoing edges with $\diamond\uparrow$ label. [5]

The generation of the traversal method $m\_k\_\mathtt{trv}_{\#}$ for traversal graph node $C^{\mathtt{k}}$ is described by $P, m, C^{\mathtt{k}}, C \vdash_{\mathtt{m}} (\overline{E}; \overline{E}) \rightharpoonup \mathcal{M}$ (Figure 4.17). For a non-strategy node (*i.e.,* nodes whose annotation is not s) the traversal method contains a sequence of method calls one for each outgoing edge that is either a field edge or a $\diamond\uparrow$ edge. (Figure 4.18). A field edge $(C^{\mathtt{k}}, f^{\mathtt{k}''}, C^{\mathtt{k}'})$ generates a $\mathtt{this}.m\_k'\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})$ expression and an upward inheritance edge $(C^{\mathtt{k}}, \diamond\uparrow^{\mathtt{k}''}, C^{\mathtt{k}'})$ generates a $\mathtt{super}.m\_k'\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})$ expression. Code generation gives priority to super calls and behaves differently when we have an incoming edge with a $\diamond\downarrow$ label, *i.e.,* $(C_1^{\mathtt{k}'}, \diamond\downarrow^{\mathtt{k}''}, C^{\mathtt{k}}) \in \overline{I}$; we wrap the method body with a super call to $m\_k'\_\mathtt{trv}_{\#}$.

For a strategy node we follow a similar pattern but we also include calls to the appropriate visitor methods. Before methods execute before we traverse any of the edges of the current node and after methods execute after we have traversed all the edges of the current node. The rule for generating calls to before and after methods ($x, \overline{\mathcal{D}}, \mathtt{e}, m \vdash C; C \rightharpoonup_v \mathtt{e}$) is given in Figure 4.19. We use the function $\mathit{match}(m, C, V, P)$ (Figure 4.22) that traverses

[5]In a single inheritance language there is only one $\diamond\uparrow$ edge.

$$\boxed{x, \overline{\mathcal{D}}, \texttt{e}, m \vdash C; C \rightharpoonup_v \texttt{e}}$$

$$\frac{match(m, C, V, P) = \bullet}{x, P, \texttt{e}, m \vdash V; C \rightharpoonup_v \texttt{e}} \qquad \frac{\begin{array}{c} match(m, C, V, P) = D_1, \ldots, D_n \\ \exists! D' : \quad D' \in D_1, \ldots, D_n \quad D' \leq^c_p D_i \\ fresh(y) \end{array}}{\begin{array}{l} x, P, \texttt{e}, m \vdash V; C \rightharpoonup_v \\ \qquad\qquad \texttt{let}\ y = x.m\_D'_{\#}(\texttt{this})\ \texttt{in e} \end{array}}$$

Figure 4.19: Compilation of AP methods (Part III)

the visitor type hierarchy in P starting from V and collects all subtypes of C that appear in a type-expression pair in a visitor method m. From the result of *match*, $D_1, \ldots, D_n$ we then select D' that is a subtype of all D_i and prepend a method call to the visitor method for D'.[6]

The second phase of our translation deals with APCORE visitor definitions. Figure 4.20 give the definition of $\vdash \overline{\mathcal{D}} \rightharpoonup_V \overline{\mathcal{D}}$ and $\vdash \mathcal{V} \rightsquigarrow_V \overline{\mathcal{M}}$. The translation turns each visitor definition to a CLASSICJAVA class definition. Before and after methods are expanded into a list of CLASSICJAVA methods; for each type-expression pair we generate a new CLASSICJAVA method. CLASSICJAVA does not support method overloading so for before and after methods we encode method overloading by using a naming convention for each type-expression pair; we append the argument type to the method name *e.g.*, `after`$\_C$. The translation of visitor return methods is straightforward.

The third phase of our translation deals with APCORE constructor methods. CLASSICJAVA only supports the default constructor for each class and fields are initialized through field set expressions. For each APCORE class C our translation generates a special mangled method `init_C` that has the same arguments as the APCORE constructor. The body of `init_C` consists

[6]CLASSICJAVA does not support method overloading; we encode method overloading for a method m by generating a new method for each method implementation. We generate the new method's name by appending the argument's type, *e.g.*, m_D'.

of a sequence of field initializations and returns `this`.

Finally, Figure 4.21 gives the translation of APCORE expressions to CLASSICJAVA expressions. The translation is straightforward with some exceptions. All `new` expressions are translated into two calls, one to create the CLASSICJAVA object using the default CLASSICJAVA constructor and then a call to our generated initialization method to initialize all fields; calls to `new Object()` are unaffected. Cast expressions in APCORE have different syntax than CLASSICJAVA and the sequence expression in APCORE is translated into a CLASSICJAVA let expression.

4.2.4 Program translation preserves the program's type

In this subsection we state and prove our type preservation theorem. Given a well typed APCORE program as input of type t we show that our translation generates a CLASSICJAVA program whose type is also t.

Theorem 4 (Type Preservation)**.** *Given a well typed* APCORE *program* $\mathcal{P}$ *with type t and its translated* CLASSICJAVA *program* $\vdash \mathcal{P} \rightharpoonup_P \mathsf{P}$ *then* P *is a well typed* CLASSICJAVA *program with type t.*

Proof. The proof proceeds in three steps. We prove that each of the CLASSICJAVA predicates on the translated program hold and that all generated classes are well typed. We then prove that the translation of the program's main expression preserves its type.

CLASSESONCE($\overline{\mathsf{defn}}\ \mathsf{e}$) We know that in the original APCORE program class names are unique (ClassesOnce($\overline{\mathcal{D}}$)), visitor names are unique (VisitorsOnce($\overline{\mathcal{D}}$)), and that visitor names and class names are distinct (DistinctVCNames($\overline{\mathcal{D}}$)). The translation leaves class names unaffected, rewrites all visitor definitions as class definitions ($\rightharpoonup_V$), and adds the reserved visitor `Visitor` as a new class definition ($\rightharpoonup_{\overline{D}}$).

$\boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_K \mathsf{defn}^*}$ $\boxed{\vdash \mathcal{D} \leadsto_K \mathsf{defn}}$

$$\frac{\mathcal{D}_i \leadsto_K \mathsf{defn}_i \quad i \in [1,n]}{\vdash \mathcal{D}_1, \ldots, \mathcal{D}_n \rightharpoonup_K \mathsf{defn}_1, \ldots, \mathsf{defn}_n}$$

$$\frac{\begin{array}{c} \mathcal{K} = \mathtt{C}(\overline{C\ x}, \overline{C\ y})\{\mathtt{super}(\overline{x}); \overline{\mathtt{this}.f := y;}\} \\ \mathit{fnames}(C) = f_1, \ldots, f_n \quad |\overline{x}| = p \quad |\overline{y}| = q \\ \mathit{fresh}(z_1) \ldots \mathit{fresh}(z_n) \\ \mathcal{M}' = C\ \mathtt{init\_C_\#}(\overline{C\ x}, \overline{C\ y})\{ \\ \quad \mathtt{let}\ z_1 = \mathtt{this}.f_1 = x_1\ \mathtt{in} \\ \quad \mathtt{let}\ z_2 = \mathtt{this}.f_2 = x_2\ \mathtt{in} \\ \vdots \\ \quad \mathtt{let}\ z_p = \mathtt{this}.f_p = x_p\ \mathtt{in} \\ \quad \mathtt{let}\ z_{p+1} = \mathtt{this}.f_{p+1} = y_1\ \mathtt{in} \\ \vdots \\ \quad \mathtt{let}\ z_n = \mathtt{this}.f_n = y_q\ \mathtt{in} \\ \mathtt{this}\} \end{array}}{\begin{array}{l} \vdash \mathtt{class}\ C\ \mathtt{extends}\ C'\{\overline{C\ f}; \mathcal{K}\ \overline{\mathcal{M}}\} \quad \leadsto_K \\ \quad \mathtt{class}\ C\ \mathtt{extends}\ C'\{\overline{C\ f}; \mathcal{M}' \cdot \overline{\mathcal{M}}\} \end{array}}$$

$\boxed{\vdash \overline{\mathcal{D}} \rightharpoonup_V \overline{\mathcal{D}}}$

$$\frac{}{\begin{array}{l} \vdash \mathtt{class}\ C \cdots \{\cdots\} \rightharpoonup_V \\ \quad \mathtt{class}\ C \cdots \{\cdots\} \end{array}}$$

$$\frac{\begin{array}{c} \vdash \mathcal{M}_j \leadsto_m \mathcal{M}''_j \quad j \in [1,k] \\ \vdash \mathcal{V}_i \leadsto_V \overline{\mathcal{M}'_i} \quad i \in [1,n] \end{array}}{\begin{array}{l} \vdash \mathtt{visitor}\ V\ \mathtt{extends}\ V'\{\cdots \mathcal{M}_1, \ldots, \mathcal{M}_k\ \mathcal{V}_1 \ldots \mathcal{V}_n\} \quad \rightharpoonup_V \\ \quad \mathtt{class}\ V\ \mathtt{extends}\ V'\{\cdots\ \mathcal{M}''_1 \ldots \mathcal{M}''_k\ \overline{\mathcal{M}'}_1 \ldots \overline{\mathcal{M}'}_n\} \end{array}}$$

$\boxed{\vdash \mathcal{V} \leadsto_V \overline{\mathcal{M}}}$

$$\frac{\begin{array}{c} \alpha \in \{\mathtt{before}\ , \mathtt{after}\ \} \\ \vdash e_i \rightharpoonup_e \mathsf{e}_i \end{array}}{\begin{array}{l} \vdash \alpha\{\overline{C \rightarrow \{e\}}\} \quad \leadsto_V \\ \quad \overline{\mathtt{Object}\ \alpha\_C_\#(C\ \mathtt{host})\{\mathsf{e}\}} \end{array}} \qquad \frac{\vdash e \rightharpoonup_e \mathsf{e}}{\begin{array}{l} \vdash t\ \mathtt{return}\ \ \{e\} \leadsto_V \\ \quad t\ \mathtt{return}\ ()\{\mathsf{e}\} \end{array}}$$

Figure 4.20: Translation of constructors methods and Visitors

$$\boxed{\vdash e \rightharpoonup_e \mathtt{e}}$$

$$\frac{}{\vdash x \rightharpoonup_e x} \qquad \frac{}{\vdash \mathtt{new\ Object}() \rightharpoonup_e \mathtt{new\ Object}()} \qquad \frac{\vdash e_i \rightharpoonup_e \mathtt{e}_i \quad i \in [1,n]}{\vdash \mathtt{new}\ C(e_1,\ldots,e_n) \rightharpoonup_e \mathtt{new}\ C().\mathtt{init\_C_\#}(\mathtt{e}_1,\ldots,\mathtt{e}_n)}$$

$$\frac{\vdash e \rightharpoonup_e \mathtt{e} \qquad \vdash e_i \rightharpoonup_e \mathtt{e}_i \quad i \in [1,n]}{\vdash e.m(e_1,\ldots,e_n) \rightharpoonup_e \mathtt{e}.m(\mathtt{e}_1,\ldots,\mathtt{e}_n)} \qquad \frac{\vdash e \rightharpoonup_e \mathtt{e}}{\vdash e.f \rightharpoonup_e \mathtt{e}.f} \qquad \frac{\vdash e \rightharpoonup_e \mathtt{e} \quad \vdash e' \rightharpoonup_e \mathtt{e}'}{\vdash e.f := e' \rightharpoonup_e \mathtt{e}.f = \mathtt{e}'}$$

$$\frac{\vdash e_i \rightharpoonup_e \mathtt{e}_i \quad i \in [1,n]}{\vdash \mathtt{super}.m(e_1,\ldots,e_n) \rightharpoonup_e \mathtt{super}.m(\mathtt{e}_1,\ldots,\mathtt{e}_n)} \qquad \frac{\vdash e \rightharpoonup_e \mathtt{e}}{\vdash (C)e \rightharpoonup_e \mathtt{view}\ C\ \mathtt{e}}$$

$$\frac{\vdash e \rightharpoonup_e \mathtt{e} \quad \vdash e' \rightharpoonup_e \mathtt{e}'}{\vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' \rightharpoonup_e \mathtt{let}\ x = \mathtt{e}\ \mathtt{in}\ \mathtt{e}'} \qquad \frac{\vdash e_1 \rightharpoonup_e \mathtt{e}_1 \quad \vdash e_2 \rightharpoonup_e \mathtt{e}_2 \quad \mathit{fresh}(x)}{\vdash e_1; e_2 \rightharpoonup_e \mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2}$$

Figure 4.21: Translation of expressions

List of visited types $\boxed{\mathit{visitTypes}(C) = \overline{C}}$

$$\frac{}{\mathit{visitTypes}(\mathtt{Visitor}) = \bullet}$$

$$\frac{V = \mathtt{visitor}\ V\ \mathtt{extends}\ V'\ \{\ \overline{C\ f}; \mathcal{K}\ \overline{\mathcal{M}}\ \overline{\mathcal{V}}\} \qquad \overline{\mathcal{V}} = \mathtt{before}\ \ \{\overline{A\ \rightarrow \{e\}}\}\quad \mathtt{after}\ \ \{\overline{B\ \rightarrow \{e\}}\}}{\mathit{visitTypes}(V) = \overline{A} \cdot \overline{B} \cdot \mathit{visitTypes}(V')}$$

All matching types inside visitor's before and after methods $\boxed{\mathit{match}(m, C, C, \overline{\mathcal{D}}) = \overline{C}}$

$$\frac{}{\mathit{match}(m, C, \mathtt{Visitor}, P) = \emptyset}$$

$$\frac{\begin{array}{c}\mathtt{visitor}\ V\ \mathtt{extends}\ V'\{\cdots\ m\{D_1\ \rightarrow \{e_1\}\ldots D_n\ \rightarrow \{e_n\}\}\cdots\} \in P \\ \overline{D'} = D_1 \ldots D_q \qquad q \leq n \qquad C \leq^c_p D_i \qquad i \in [1,q] \\ \overline{D''} = \mathit{match}(m, C, V', P)\end{array}}{\mathit{match}(m, C, V, P) = \overline{D'} \cdot \overline{D''}}$$

Figure 4.22: Visit types and matching for before and after methods

FIELDONCEPERCLASS($\overline{\mathsf{defn}}$ e) In the original APCORE program fields inside a class are unique (1FieldPerClass($\overline{\mathcal{D}}$)) and fields inside a visitor are unique (1FieldPerVisitor($\overline{\mathcal{D}}$)). The translation does not inject or modify field definitions.

COMPLETECLASSES($\overline{\mathsf{defn}}$ e) From the original APCORE program we know that classes that are extended are defined (CompleteClasses($\overline{\mathcal{D}}$)) and visitors that are extended are defined (CompleteVisitors($\overline{\mathcal{D}}$)). The translation does not alter any inheritance relationships between user defined classes and visitors. The extra `Visitor` class generated by the translation extends `Object`.

WELLFOUNDEDCLASSES($\overline{\mathsf{defn}}$ e) In the original APCORE program we the class hierarchy is an order (WFClasses($\overline{\mathcal{D}}$)) and that the visitor hierarchy is an order (WFVisitors($\overline{\mathcal{D}}$)). The two hierarchies are separate (DistinctVCNames($\overline{\mathcal{D}}$)) and the translation rewrites visitor definitions as class definitions without altering any inheritance relationships. The generated `Visitor` class only extends `Object`. The resulting class hierarchy is an order.

METHODONCEPERCLASS($\overline{\mathsf{defn}}$ e) From the original APCORE program we know that methods inside a class are unique (1MethodPerClass($\overline{\mathcal{D}}$)) and methods inside a visitor are unique (1MethodPerVisitor($\overline{\mathcal{D}}$)). The translation replaces visitor methods, constructors and adaptive methods with new generated method definitions. We examine each case separately.

Visitor Methods The $\rightharpoonup_V$ rule translates visitor definitions in APCORE to class definitions in CLASSICJAVA and generates a new method definition for each type expression pair found inside a `before` or `after` definition and a new method definition for a `return` definition ($\rightsquigarrow_V$). We know that types inside a `before` method are unique (WFBeforeMethod($\overline{\mathcal{D}}$)). Translation generates a method with the name pattern `before_`$C_{\texttt{#}}$ for each type C in a before's type expression pair. Mangling the name ensures that the generated methods are unique within a translated visitor definition. A similar argument applies to the generation of after methods. For the `return` method we know that it is unique inside the APCORE visitor definition (OneReturnMethod($\overline{\mathcal{D}}$)). The translation generates a CLASSICJAVA method with the same name.

Constructors For every class and visitor definition in APCORE the translation replaces constructor definitions with a new method. The new method 's name is created from mangling the concatenation of the string `init` and the name of the class or visitor. The resulting mangled name is always unique inside a class.

Adaptive Methods For an adaptive method m inside a class C the translation generates four new CLASSICJAVA methods

1. m_C_$\texttt{return}_{\texttt{#}}$,
2. a new CLASSICJAVA method m to replace the original adaptive method, and
3. a series of methods with the name pattern m_C_k_$\texttt{trv}_{\texttt{#}}$ are introduced inside classes (including C) that perform the appropriate traversal for the adaptive method's strategy.

The method name m is unique inside C (1MethodPerClass($\overline{\mathcal{D}}$)) and so is the mangled method name m_C_$\texttt{return}_{\texttt{#}}$. For all classes affected by the generation of traversal code the method names generated are composed of the adaptive method name m, the class C within which m is defined, a number `k` which is the annotation from the traversal graph node corresponding to class C, or the special symbol `s`, and the mangled string $\texttt{trv}_{\texttt{#}}$. The generated method name does not clash with any of the programmer defined method names and translation avoids duplicate method definitions (Figure 4.16). Incorporating the name of the adaptive method's defining class avoids name clashes due to overriding.[7]

The second step in our proof checks that all generated class definitions are well typed. We check the translation of APCORE classes and adaptive methods, then APCORE visitors with before, after and return methods, and finally class and visitor constructors.

Class Definitions An APCORE class definition is rewritten into a CLASSICJAVA class definition by rewriting APCORE method and adaptive method definitions.

Method Definitions Given a normal method definition $t\ m(\overline{C\ x})\{e\}$ inside an APCORE class C, the translation generates a method with the same method signature and rewrites the method body expression to $\mathtt{e}$, $t\ m(\overline{C\ x})\{\mathtt{e}\}$. We know that $\overline{\mathcal{D}}, [\mathtt{this} : C, \overline{x : C}] \vdash_s e : t$, by Lemma 6 we can deduce that $\mathsf{defn}^*, [\mathtt{this} : C, \overline{x : C}] \vdash_s \mathtt{e} \Rightarrow \mathtt{e}' : t$.

Adaptive Method Definitions Given an adaptive method definition $@\{\mathcal{CC}\}\ t\ m(\overline{C\ x})\ \mathtt{with}\ \mathcal{SG}[V]$ in an APCORE class C the translation generates

1. a CLASSICJAVA method $m\_C\_\mathtt{return}_{\#}$,
2. a CLASSICJAVA method m, and
3. a series of methods injected into the set of classes calculated by the traversal graph.

We examine each of the preceding cases.

Case 1 The generated method $t\ m\_C\_\mathtt{return}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{v}_{\#}.\mathtt{return}\ ()\}$ has the same return type as the adaptive method's return

[7]An overriding adaptive method cannot use super .

type. We know that $\overline{\mathcal{D}}, C \vdash_{\text{A}} @\{\mathcal{CC}\}\ t\ m(\overline{C\ x})$ `with` $\mathcal{SG}[V]$: OK, from which we can deduce that $\mathit{mtype}(\texttt{return}, V) = \overline{t'} \rightarrow t$ where $|\,\overline{t}\,| = 0$.

Case 2 The generated CLASSICJAVA method m has the same method signature as the adaptive method and its method body performs the following operations

1. instantiates and initializes an object of class V. We know that the adaptive method is well typed and thus we also know that $\mathit{ktype}(V) = \overline{C} \rightarrow V$. We also know from our translation that V is now a class and that $\texttt{init\_V}_{\#}$ is defined inside V ($\rightsquigarrow_K$) with signature $\overline{C} \rightarrow V$.
2. performs the traversal based on the adaptive method's strategy by calling $\texttt{this.}m\_C\_s\_\texttt{trv}_{\#}(\texttt{v}_{\#})$. **Case 3** shows that the expression $\texttt{this.}m\_C\_s\_\texttt{trv}_{\#}(\texttt{v}_{\#})$ is well typed.
3. call $m\_\texttt{C\_return}_{\#}(\texttt{v}_{\#})$. **Case 1** shows that the expression has type t.

Case 3 From the typing rule for adaptive methods we know that the current class C (Figure 4.16) is the source of $\mathcal{SG}$ and thus C^{s} is a member of $\mathit{gettg}(\mathcal{SG}, \overline{\mathcal{D}}).\mathit{nodes}$. Using Lemma 2 we can deduce that the method `Object` $m\_C\_s\_\texttt{trv}_{\#}(V\ \texttt{v}_{\#})$ is defined in C. Using Lemma 3 we can deduce that the method definition is well typed. Translation uses the traversal graph to inject methods inside classes that appear on a path to a strategy's target node. Lemma 2 and Lemma 3 also show that each generated method `Object` $m\_C\_k\_\texttt{trv}_{\#}(V\ \texttt{v}_{\#})\{\texttt{e}\}$ inside a class C' is well typed.

Visitor Definitions An APCORE visitor definition is rewritten into a CLASSICJAVA class definition by rewriting APCORE method definitions and visitor method definitions.

Method Definitions This case is similar to the case for method definitions inside an APCORE class.

Visitor Definitions There are two cases here

- for `before` and `after` visitor methods the translation generates a method for each type expression pair inside `before` and `after` method definitions. For a type expression pair $C \rightarrow \{e\}$ inside a visitor V we know that $\overline{\mathcal{D}}, [\texttt{this} : V, \texttt{host} : C] \vdash_{\mathrm{s}} e : \texttt{Object}$. The translation of a type expression pair generates the method $\texttt{Object}\ \alpha\_C_{\#}(C\ \texttt{host})\{\texttt{e}\}$ where α is either the string `before` or `after` and $\vdash\ e \rightharpoonup_e \texttt{e}$. Using Lemma 6 we can deduce that $\mathsf{defn}^*, [\texttt{this} : V, \texttt{host} : C] \vdash_s \texttt{e} \Rightarrow \texttt{e}' : \texttt{Object}$.
- `return` visitor methods take no arguments and have a single expression as the method body. We know that $\overline{\mathcal{D}}, [\texttt{this} : V] \vdash_{\mathrm{s}} e : t$ and that $\vdash\ e \rightharpoonup_e \texttt{e}$. Using Lemma 6 we can deduce that $\mathsf{defn}^*, [\texttt{this} : V] \vdash_s \texttt{e} \Rightarrow \texttt{e}' : t$.

Constructor Methods We know that a well typed constructor method definition consumes a list of values whose size is equal to all the fields, inherited and defined, inside a visitor or class. The constructor rewrite rule ($\rightsquigarrow_K$) is used after all visitor definitions have been rewritten as classes. For each class definition B the rewrite rule $\rightsquigarrow_K$ generates a new method ($\texttt{init\_B}_{\#}$) with type ($\bar{t} \rightarrow B$). This type signature is identical to the type signature of the constructor method. The body of the generated method ($\texttt{e}$) assigns, in order, each value passed as an argument to the class' fields and returns the instance.

Using CLASSICJAVA type rule for methods we can show that $\mathsf{defn}^*, [\texttt{this} : B, x_1 : t_1, \ldots, x_n : t_n] \vdash_{\mathsf{e}} \texttt{e} \Rightarrow \texttt{e}' : B$ where $x_1, \ldots, x_n$ are the method's formal argument names, $t_1, \ldots, t_n$ are the formal argument types, and B is the name of the definition (class or visitor) under translation.

The final step in our proof deals with the APCORE program's main expression e. We know that $\overline{\mathcal{D}}, [\,] \vdash_{\mathrm{e}} e : t$ and that $\vdash e \rightharpoonup_e \mathsf{e}$. Using Lemma 6 we can deduce that $\mathsf{defn}^*, [\,] \vdash_e \mathsf{e} \Rightarrow \mathsf{e}' : t$. □

Lemma 2 shows that for each adaptive method definition in the original APCORE program our translation generates appropriate methods inside all the relevant class definitions. The generated method names follow a specific pattern $m\_D\_k\_\mathtt{trv}_{\#}$ where m is the name of the original adaptive method, D is the class where m is defined, k is the annotation of the traversal graph node corresponding to the current class and $trv_{\#}$ is a mangled string. The method's type signature is always $V \rightarrow \mathtt{Object}$ where V is the visitor used in the adaptive method's definition.

Lemma 2. *Given a well typed* APCORE *program* $\vdash_{\mathrm{P}} \overline{\mathcal{D}}\ e : t$ *and the translated* CLASSICJAVA *program* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}$, *then for every adaptive method* $@\{\mathcal{CC}\}\ t'\ m(\overline{C\ x})$ `with` $\mathcal{SG}[V]$ *defined in a class* $D \in \overline{\mathcal{D}}$

$$\begin{aligned} \forall C^{\mathrm{k}} \;:\; & C^{\mathrm{k}} \in \mathit{gettg}(\mathcal{SG}, \overline{\mathcal{D}}).\mathit{nodes} \iff \\ & \mathtt{class}\ C\ \cdots \{\cdots \mathsf{meth}^*\} \in \mathsf{defn}^* \wedge \\ & \mathtt{Object}\ m\_D\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\cdots\} \in \mathsf{meth}^* \end{aligned}$$

Proof. The proof inspects the translation rules for adaptive methods in Figures 4.16 and 4.17 to show that the appropriate method definition is generated for all nodes found in an adaptive method's traversal graph.

$\Rightarrow$ The generation algorithm defined in Figure 4.16 begins at the strategy's start node, generates a method for the strategy's start node and then recursively proceeds to all outgoing nodes. At each node C^{k} the algorithm generates a method $\mathtt{Object}\ m\_D\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathsf{e}\}$ (Figure 4.16).

$\Leftarrow$ A method definition with a mangled name implies that the method was not part of the original APCORE program but a result of our translation. A method name of the form $m\_D\_k\_\mathtt{trv}_{\#}$ can only be created by the traversal method generation rule $P, m, C^{\mathtt{k}}, C \vdash_{\mathtt{m}} (\overline{E}; \overline{E}) \rightharpoonup \mathcal{M}$ (Figure 4.17). The translation only uses the traversal method generation rule in $\overline{\mathcal{D}}, C^{\mathtt{k}}, \mathcal{TG}, m, C \vdash_m (\overline{\mathcal{D}}; \mathbb{C}^{\mathtt{k}}) \Rightarrow (\overline{\mathcal{D}}; \mathbb{C}^{\mathtt{k}})$ (Figure 4.16). In Figure 4.16 the traversal method generation rule is used for each $C^{\mathtt{k}} \in \mathcal{TG}$.

□

Lemma 3 shows that all generated methods of the form
$t\ m\_D\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}\}$ are well typed.

Lemma 3. *Given a well typed* APCORE *program* $\vdash_{\mathrm{P}} \overline{\mathcal{D}}\ e : t$ *and the translated* CLASSICJAVA *program* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathtt{e}$, *for all methods*
$\mathtt{Object}\ m\_D\_k\_\mathtt{trv}_{\#}(V\ \mathtt{v}_{\#})\{\mathtt{e}_1\}$ *in a class definition* $\mathtt{class}\ C\ \cdots\{\cdots\} \in \mathsf{defn}^*$ *then*

$$\mathsf{defn}^*, [\mathtt{this} : C, \mathtt{v}_{\#} : V] \vdash_s \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \mathtt{Object}$$

Proof. The proof proceeds by a case analysis on the definition of $\overline{\mathcal{D}}, m, C^{\mathtt{k}}, C \vdash_{\mathtt{m}} (\overline{E}; \overline{E}) \rightharpoonup \overline{\mathcal{M}}$ (Figure 4.17).

There are four cases in total to consider, two cases for copy nodes $C^{\mathtt{i}}$ where $\mathtt{i} \in \mathbb{Z}$ and two cases for strategy nodes $C^{\mathtt{s}}$.

Case $C^{\mathtt{i}}$ Both cases start with the CLASSICJAVA expression `this` and create a new CLASSICJAVA expression $\mathtt{e}_1$ using $\mathtt{v}_{\#}, m, \mathtt{this} \vdash_E \overline{E} \rightharpoonup \mathtt{e}_1'$. Using Lemma 5 with $\Gamma = [\mathtt{this} : C, \mathtt{v}_{\#} : V]$ we can deduce that $\mathsf{defn}^*, \Gamma \vdash_s \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \mathtt{Object}$. The two cases differ on how they manipulate $\mathtt{e}_1$.

1. $\mathtt{e}_1' = \mathtt{e}$.
   Immediate.
2. $\mathtt{e} = \mathtt{let}\ x = \mathtt{super}.m\_k'\_\mathtt{trv}_{\#}(\mathtt{v}_{\#})\ \mathtt{in}\ \mathtt{e}_1$
   We know that $\mathsf{defn}^*, \Gamma \vdash_s \mathtt{e}_1 \Rightarrow \mathtt{e}_1' : \mathtt{Object}$ and we know that x is a fresh variable name. We also know that there is an incoming edge $(C_1^{\mathtt{k}'}, \diamond\!\downarrow^{\mathtt{k}''}, C^{\mathtt{k}})$ and we can deduce that $C <_p^c C_1$. By

Lemma 2 we also know that the method $m\_k'\_\texttt{trv}_{\#}$ is defined in C_1 with method signature $V \rightarrow \texttt{Object}$. Using Lemma 8 and CLASSICJAVA 's type rule for `let` we can deduce that $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e} \Rightarrow \mathsf{e}' : \texttt{Object}$

Case C^{s} Both cases start with the CLASSICJAVA expression `this` and create a new CLASSICJAVA expression e_1 using

1. $\texttt{v}_{\#}, P, \texttt{this}, \texttt{after} \ \vdash V; C \rightharpoonup_v \mathsf{e}_3$
2. $\texttt{v}_{\#}, m, \mathsf{e}_3 \vdash_E \overline{E} \rightharpoonup \mathsf{e}_2$
3. $\texttt{v}_{\#}, P, \mathsf{e}_2, \texttt{before} \ \vdash V; C \rightharpoonup_v \mathsf{e}_1$

in this order. Each rule generates a new CLASSICJAVA expression which is used as the starting expression for the next rule in the sequence. We first show that each rule in the sequence generates a CLASSICJAVA expression of type `Object` . Then we examine how the two cases manipulate the resulting expression e_1 to obtain the final CLASSICJAVA expression used as the method's body.

1. We can show that $\texttt{v}_{\#}, P, \texttt{this}, \texttt{after} \ \vdash V; C \rightharpoonup_v \mathsf{e}_3$ returns an expression e_3 such that $\mathsf{defn}^*, [\texttt{this} : C, \texttt{v}_{\#} : V] \vdash_s \mathsf{e}_3 \Rightarrow \mathsf{e}_3' : \texttt{Object}$ by Lemma 4.
2. Given $\texttt{v}_{\#}, m, \mathsf{e}_3 \vdash_E \overline{E} \rightharpoonup \mathsf{e}_2$ we can show that $\mathsf{defn}^*, [\texttt{this} : C, \texttt{v}_{\#} : V] \vdash_s \mathsf{e}_2 \Rightarrow \mathsf{e}_2' : \texttt{Object}$ by Lemma 5.
3. Finally given that $\texttt{v}_{\#}, P, \mathsf{e}_2, \texttt{before} \ \vdash V; C \rightharpoonup_v \mathsf{e}_1$ we can show that $\mathsf{defn}^*, [\texttt{this} : C, \texttt{v}_{\#} : V] \vdash_s \mathsf{e}_1 \Rightarrow \mathsf{e}_1' : \texttt{Object}$ by Lemma 4.

Given e_1 the two cases manipulate e_1 in the same way as in the case of copy nodes. The proof proceeds in the same way as in the preceding case.

$\square$

Lemma 4 shows that the generated calls to all the necessary before and after methods of a visitor are well typed.

Lemma 4. *Given a well typed* APCORE *program* $\vdash_{\text{P}} \overline{\mathcal{D}}\ e : t$ *and the translated* CLASSICJAVA *program* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}$

$$\begin{aligned}\forall C, C' \quad : \quad & C \in \mathsf{defn}^* \quad \wedge \\ & C' \in \mathsf{defn}^* \quad \wedge \\ & \overline{\mathcal{D}} \vdash_{\text{V}} C'\end{aligned}$$

such that

$$\begin{aligned}\exists \Gamma, \mathsf{e}_1, x \quad : \quad & \Gamma(x) = C' && \wedge \\ & \Gamma(\mathtt{this}) = C && \wedge \\ & \mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}_1 \Rightarrow \mathsf{e}_1' : t && \wedge \\ & x, \overline{\mathcal{D}}, \mathsf{e}_1, m \vdash C'; C \rightharpoonup_v \mathsf{e}_1''\end{aligned}$$

then

$$\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}_1'' \Rightarrow \mathsf{e}_1''' : t$$

Proof. We examine each case for $\rightharpoonup_v$.

1. $match(m, C, C', \overline{\mathcal{D}}) = \bullet$.

   There are no applicable methods for C in C'; e remains unchanged.

2. $match(m, C, C', \overline{\mathcal{D}}) = D_1 \ldots D_n$.

   There is a list of applicable methods in C' for class C. Applicable methods have an argument D_i that is a supertype of C. The resulting expression e'' is a let expressions with e as then body of the let expression. The let expression adds a method call to x's m method $x.m\_D'_{\#}(\mathtt{this})$ for $D' \in D_1 \ldots D_n$ and $D' \leq_p^c D_i, i \in [1, n]$. By $\rightsquigarrow_V$ we know that $m\_D'_{\#}$ is either defined by C' or inherited from a superclass of C' with signature $D' \rightarrow \mathtt{Object}$ and that $C \leq_p^c D'$. Translation preserves the inheritance relation and thus $C \leq_P^c D'$. The result of this method call is bound to a fresh variable y, by Lemma 9 we can deduce that $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}'' \Rightarrow \mathsf{e}''' : t$.

□

Lemma 5. *Given a well typed* APCORE *program* $\vdash_P \overline{\mathcal{D}}\ e : t$ *and the translated* CLASSICJAVA *program* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}$, *for any class* $D \in \overline{\mathcal{D}}$ *and any adaptive method* $@\{\mathcal{CC}\}\ t\ m(\overline{C\ y})$ `with` $\mathcal{SG}[V] \in D$

$$\begin{array}{lllll} \textit{and} & \overline{E} = \textit{gettg}(\mathcal{SG}, \overline{\mathcal{D}}).\textit{outgoing}(D^{\mathtt{k}})\ \textit{where}\ \mathtt{k} \in \mathbb{Z} \cup \{\mathtt{s}\} \\ \textit{and} & \exists \Gamma, \mathsf{e}_1, x & : & \Gamma(x) = V & \wedge \\ & & & \Gamma(\mathtt{this}) = D & \wedge \\ & & & \mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}_1 \Rightarrow \mathsf{e}_1' : t' & \wedge \\ & & & x, m\_D, \mathsf{e}_1 \vdash_E \overline{E} \rightharpoonup \mathsf{e}_1'' \\ \textit{then} & \mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}_1'' \Rightarrow \mathsf{e}_1''' : t' \end{array}$$

Proof. The proof proceeds by induction on the size of $\overline{E}$.

Case $|\overline{E}| = 0$ Immediate.

Case $|\overline{E}| = n + 1$ There are three cases based on the definition of $x, m, \mathsf{e} \vdash_E \overline{E} \rightharpoonup \mathsf{e}''$ (Figure 4.17).

1. $\overline{E} = (\_, \diamond\downarrow^{\mathtt{k}}, \_) \cup \overline{E'}$. The rule skips all edges with label $\diamond\downarrow^{\mathtt{k}}$, thus by the induction hypothesis we can deduce $x, m, \mathsf{e} \vdash_E \overline{E'} \rightharpoonup \mathsf{e}''$ and $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}'' \Rightarrow \mathsf{e}''' : t$.
2. $\overline{E} = (C_1^{\mathtt{k}}, \diamond\uparrow^{\mathtt{k}'}, C^{\mathtt{k}''}) \cup \overline{E'}$. The rule creates a new expression $\mathsf{e}_1 =$ `let` y $=$ `super.`m_k''`_trv`$_{\#}(x)$ `in` `e`. From Lemma 2 we know that class C contains the method m_k''`_trv`$_{\#}$ with signature $V \rightarrow$ `Object`. By Lemma 9 we can deduce that $\mathsf{defn}^*, \Gamma[y : \mathtt{Object}] \vdash_s \mathsf{e} : t$. By the induction hypothesis we can deduce that $x, m, \mathsf{e}_1 \vdash_E \overline{E'} \rightharpoonup \mathsf{e}''$ and $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}'' \Rightarrow \mathsf{e}''' : t$.

3. $\overline{E} = (C_1^{\mathrm{k}}, l^{\mathrm{k}'}, C^{\mathrm{k}''}) \cup \overline{E'}$ and $l^{\mathrm{k}} \notin \{\diamond\downarrow^{\mathrm{k}}, \diamond\uparrow^{\mathrm{k}}\}$. The rule creates a new expression $\mathsf{e}_1 = \texttt{let}\ y = \texttt{this.}l.m\_k''\_\texttt{trv}_\#(x)\ \texttt{in}\ \mathsf{e}$. From Lemma 2 we know that class C contains the method $m\_k''\_\texttt{trv}_\#$ with signature $V \to \texttt{Object}$. By Lemma 9 we can deduce that $\mathsf{defn}^*, \Gamma[y : \texttt{Object}] \vdash_s \mathsf{e} : t$. By the induction hypothesis we can deduce that $x, m, \mathsf{e}_1 \vdash_E \overline{E'} \rightharpoonup \mathsf{e}''$ and $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e}'' \Rightarrow \mathsf{e}''' : t$.

□

Lemma 6. *Given* $\vdash_{\mathrm{P}} \overline{\mathcal{D}}\ e : t$ *and* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}$*, for any* APCORE *expression* e *and type environment* Γ *such that* $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e : t$ *and* $\vdash e \rightharpoonup_e \mathsf{e}$ *then* $\mathsf{defn}^*, \Gamma \vdash_s \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Proof. We know that $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{s}} e : t$ and by the definition of $\vdash_{\mathrm{s}}$ we know that $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t'$ for some t' and $\overline{\mathcal{D}} \vdash_{\mathrm{t}} t'$ and $t' \leq_p^c t$. By Lemma 7 we can deduce that $\mathsf{defn}^*, \Gamma \vdash_{\mathrm{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t'$. Translation does not alter the inheritance relation defined by APCORE classes and thus we can deduce that $\mathsf{defn}^* \vdash_t t'$ and $t' \leq_P^c t$. □

Lemma 7 shows that our translation of APCORE expressions preserves their type.

Lemma 7. *Given* $\vdash_{\mathrm{P}} \overline{\mathcal{D}}\ e : t$ *and* $\vdash \overline{\mathcal{D}}\ e \rightharpoonup_P \mathsf{defn}^*\ \mathsf{e}$*, for any* APCORE *expression* e *and type environment* Γ *such that* $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t$ *and* $\vdash e \rightharpoonup_e \mathsf{e}$ *then* $\mathsf{defn}^*, \Gamma \vdash_{\mathrm{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Proof. The proof proceeds by induction on the height of the derivation tree of $\vdash e' \rightharpoonup_e \mathsf{e}$. We show all the cases for e at the last derivation step.

Case $e \equiv x$**.** We know that $\mathsf{e} \equiv x$ and that $x \in domain(\Gamma)$ thus $\mathsf{defn}^*, \Gamma \vdash_{\mathrm{e}} x \Rightarrow x : \Gamma(x)$

Case $e \equiv \texttt{let}\ x = e_1\ \texttt{in}\ e_2$**.** We know that $\mathsf{e} \equiv \texttt{let}\ x = \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2$ and that $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e : t$. We also know that $\overline{\mathcal{D}}, \Gamma \vdash_{\mathrm{e}} e_1 : t_1$ and $\overline{\mathcal{D}}, \Gamma[x : t_1] \vdash_{\mathrm{e}}$

$e_2 : t$. By the induction hypothesis we know that $\vdash e_1 \rightharpoonup_e \mathsf{e}_1$ and that $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_1 : t_1$. Similarly $\vdash e_2 \rightharpoonup_e \mathsf{e}_2$ and $\mathsf{defn}^*, \Gamma[x : t_1] \vdash_{\mathsf{e}} \mathsf{e}_2 : t$, thus $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv$ `new Object()`**.** The expression remains unaffected by the translation. The expressions type, `Object`, does not depend on Γ and remains the same.

Case $e \equiv$ `new` $C(e_1, \ldots, e_n)$**.** We know that $\mathsf{e} \equiv$ `new` $C()$.`init_C#`$(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ where the type of `new` $C()$ is C and does not depend on Γ. Furthermore, we know that $ktype(C) = \overline{t} \rightarrow C$ and from the translation ($\rightharpoonup_K$) we also know that the translated class C contains the method `init_C#` with type $\overline{t} \rightarrow C$. By the induction hypothesis we can deduce that $\vdash e_i \rightharpoonup_e \mathsf{e}_i$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_i \Rightarrow \mathsf{e}'_i : t_i$ for all $i \in [1, n]$. Using CLASSICJAVA 's type rule for method calls we can deduce that e has type C.

Case $e \equiv e_1.f_i$**.** We know that $\mathsf{e} \equiv \mathsf{e}_1.f_i$, by the induction hypothesis we can deduce $\vdash e_1 \rightharpoonup_e \mathsf{e}_1$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_1 \Rightarrow \mathsf{e}'_1 : t_1$. We know that $\mathit{fieldType}(t_1, f_i) = t$ and that *fieldType* returns the field type from the declaration found at the minimum (*i.e.*, furthest from the root) superclass. The same behavior as CLASSICJAVA 's $\in^c_P$ relation. Using CLASSICJAVA 's type rule for field access we can deduce that $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv e_1.f_i := e_2$**.** We know that $\mathsf{e} \equiv \mathsf{e}_1.f_i = \mathsf{e}_2$. By the induction hypothesis we know that $\vdash e_1 \rightharpoonup_e \mathsf{e}_1$ and $\vdash e_2 \rightharpoonup_e \mathsf{e}_2$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_1 \Rightarrow \mathsf{e}'_1 : t_1$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_2 \Rightarrow \mathsf{e}'_2 : t$. Also, we know that $\mathit{fieldType}(t_1, f_i) = t$ and that CLASSICJAVA 's $\in^c_P$ relation agrees with *fieldType*. The type rule for field set in both APCORE and CLASSICJAVA allows for subsumption and thus we can deduce $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv (C)\ e_1$**.** We know that $\mathsf{e} \equiv \texttt{view}\ C\ \mathsf{e}_1$. By the induction hypothesis we know that $\vdash e_1 \rightharpoonup_e \mathsf{e}_1$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_1 \Rightarrow \mathsf{e}'_1 : t_1$. We also know that $t_1 \leq_p^c C$ thus we can deduce $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv e_1; e_2$**.** We know that $\mathsf{e} \equiv \texttt{let}\ x = \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2$ where x is a fresh variable name. By the induction hypothesis we have that $\vdash e_1 \rightharpoonup_e \mathsf{e}_1$ and $\vdash e_2 \rightharpoonup_e \mathsf{e}_2$ and that $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_1 \Rightarrow \mathsf{e}'_1 : t_1$ $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_2 \Rightarrow \mathsf{e}'_2 : t_2$. Using Lemma 8 and the type rule for let from CLASSICJAVA we can deduce $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv e_0.m(e_1, \ldots, e_n)$**.** We know that $\mathsf{e} \equiv \mathsf{e}_0.m(\mathsf{e}_1, \ldots, \mathsf{e}_n)$. By the induction hypothesis we can deduce that $\vdash e_i \rightharpoonup_e \mathsf{e}_i$ and $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e}_i \Rightarrow \mathsf{e}'_i : t_i$ for $i \in [0, n]$. We also know that $mtype(m, t_0) = t_1, \ldots, t_n \rightarrow t$ and by its definition *mtype* returns the method type signature found at the minimum (*i.e.,* furthest from the root) superclass. Translation maintains the visitor hierarchy and connects it to the class hierarchy at `Object` , thus *mtype* and CLASSICJAVA 's $\in_p^c$ relation agree on the types returned for methods defined inside visitors. Using CLASSICJAVA 's type rule for method calls we can deduce $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Case $e \equiv \texttt{super}.m(e_1, \ldots, e_n)$**.** The reasoning is similar to the previous case.

□

Lemma 8. *If* $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$ *and* $x \notin fv(\mathsf{e})$ *then* $\mathsf{defn}^*, \Gamma[x : t'] \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$

Proof. There are two cases to consider

Case $x \in domain(\Gamma)$ Since $x \notin fv(\mathsf{e})$ then x is either

- bound inside e. In this case $[x : t']$ shadows x in Γ and the binding inside e shadows x for a second time in $\Gamma[x : t']$. Since $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$ we can conclude that $\mathsf{defn}^*, \Gamma[x : t'] \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$

- not bound inside e. Since $x \notin \mathit{fv}(\mathsf{e})$ then it does not contribute to the type derivation of e.

Case $x \notin \mathit{domain}(\Gamma)$ By Lemma 9.

□

Lemma 9 (Free[8]). *If* $\mathsf{defn}^*, \Gamma \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$ *and* $x \notin \mathit{domain}(\Gamma)$ *then* $\mathsf{defn}^*, \Gamma[x : t'] \vdash_{\mathsf{e}} \mathsf{e} \Rightarrow \mathsf{e}' : t$.

Proof. The claim follows by reasoning about the shape of the derivation. □

[8]Similar to Lemma 14 from [14]

CHAPTER 5

Demeter Interfaces

The first half of this chapter informally describes the extensions to DAJ to accommodate Demeter Interfaces (DI). Our extension is implemented as a rewrite from the extended DAJ syntax for DIs to the original DAJ system with WYSIWYG and constraints. The second half of this section discusses the application of Demeter Interfaces in the development of a design by contract system for Java showing some of the benefits of DIs. The chapter concludes with a discussion on some of the shortcomings of DIs.

Demeter Interfaces provide a mechanism to encapsulate adaptive code while at the same time abstract over class dictionaries. Typically adaptive code deals with a subset of the class dictionary and not the whole of the class dictionary. Currently there is no mechanism to encapsulate the relevant parts of a class dictionary used by adaptive code. Developing adaptive code over a large class dictionary, such as the class dictionary that defines a grammar for Java programs, becomes difficult.

Furthermore, in situations where the structure of different subparts of the class dictionary are similar but class names differ, there is no mechanism to abstract over class names in adaptive code. To deal with these similarities programmers tend to either

- have similar strategies and visitors that differ only in the class names used inside strategy and visitor definitions, or,

- develop very general strategies (using the $*$ pattern) and either have visitor definitions that perform multiple tasks (if possible) using a large number of before methods, or separate visitors for each task.

Maintaining similar adaptive code that differs only in the class names used inside strategies is cumbersome and error prone. Overly general strategies with "fat" visitor implementations decreases separation of concerns, induces unnecessary dependencies between visitors and the class dictionary, and makes reasoning about visitor behavior difficult. Furthermore, general strategies capture a large set of valid paths which need to be considered by developers making the design of visitors more challenging.

Demeter Interfaces aim at providing an abstraction mechanism that allows programmers to capture only the important–to the adaptive code–subparts of the class graph. The abstraction provided by DIs defines the boundaries within which adaptive code operates leading to modular adaptive code.

At the same time DIs provide a mechanism to abstract over class dictionary class names and the means to provide an "instantiation" for each abstract name using a mapping from a DI's abstract names to a class dictionary class names. The ability to abstract over class dictionary names and to instantiate (possibly multiple times within the same class graph) a DI leads to higher code reuse.

5.1 Support for Demeter Interfaces

Our extension to DAJ[1] to support Demeter Interfaces is implemented as a rewrite from the extended DAJ syntax for DIs to the original DAJ system with WYSIWYG and constraints. We extend DAJ's syntax to allow for the definition of DIs, and extend class dictionary definitions to allow for naming the class dictionary and allow for mapping(s) and instantiations of DIs.

[1]DIs and normal traversal files can co-exist.

We use our list example (Figure 5.1) to assist with our description of the syntax and translation of Demeter Interfaces.

5.1.1 Defining a Demeter Interface

A Demeter Interface [39] (DI) consist of:

- an Interface Class Graph [33, 30] (ICG). The ICG is a class dictionary that acts as an interface between adaptive code (strategies, traversal declarations, and, visitor definitions) and a concrete class dictionary. The ICG captures only the relevant–to the adaptive code–classes and their relationships.
- a set of strategy and traversal declarations. Strategies define interesting sets of paths on the ICG. Traversals bind visitors to these strategies.
- constraint annotations on adaptive methods.

Figure 5.1 gives the definition of the `FlatList` DI for flat lists. The interface class graph defines four types, `FList`, two types that extend `FList`, `Mt` and `Cons` and a type `E` that represents list elements.

The `FlatList` DI also defines seven adaptive methods:

1. `getFirst`, defined in `FList`, returns the first element in a flat list,
2. `getRest`, defined in `FList`, returns the tail of a flat list,
3. `getLastElement`, defined in `FList`, returns the last element in a flat list,
4. `size`, defined in `FList`, returns the total number of elements in a flat list,
5. `asString`, defined in `FList`, takes a `StringBuffer` as an argument a returns a string representation of a flat list,

```
di FlatList {
  // ICG for flat lists
  FList : Mt | Cons.
  Mt = .
  Cons = <f> E <r> FList.
  E = .

  // strategy specifications
  declare strategy: toFirst : from FList via Cons to E.
  declare strategy: toRest : from FList to Cons.
  declare strategy: toAll : from FList to ∗.
  declare strategy: cons2rest : from Cons via (→∗,r,∗) to FList.
  declare strategy: toLast : source:FList → FList
                                        FList → target:E.

  // traversal specifications
  @constraint {unique(toFirst)}
  declare traversal: public E getFirst() : toFirst(GetEV);
  @constraint {unique(toRest)}
  declare traversal: public FList getRest() : toRest(GetRestV);
  declare traversal: public E getLastElement() : toLast(GetEV);
  declare traversal: public int size() : toAll(CountV);
  declare traversal: public String asString(StringBuffer sb) : toAll(AsStringV);
  declare traversal: public String asCommaSepString(): toAll(AsCommaSepStringV);
  declare traversal: public E atIndex(int i) : toAll(AtIndexV);
  @constraint {unique(cons2rest)}
  declare traversal: public FList getR() : cons2rest(GetRV);
}
```

Figure 5.1: The Demeter Interface for flat lists.

6. `asCommaSepString`, defined in `FList`, is similar to `asString` but separates list elements with a comma,

7. `atIndex`, defined in `FList`, takes an index i as an argument and returns the element found at index i in a flat list, and,

8. `getR`, defined in `Cons`, returns the member `r` of a `Cons` type.[2]

The strategy `toLast` uses the graph based notation where each line defines an edge in the strategy graph with the keywords `source` and `target` specifying the strategy source and target node respectively. The `toLast` strategy graph contains a self loop on `FList` and an edge from `FList` to `E`.

[2]Helper method for `getRest`

```
class AsCommaSepStringV extends AsStringV {

  AsCommaSepStringV(StringBuffer sb){ super(sb); }

  public void before(E host){
    this.sb = sb.append(host.toString()).append(", ");
  }

  public String return(){
    return this.sb.reverse().delete(0,2).reverse().toString();
  }
}
```

```
class AsStringV {
  protected StringBuffer sb;

  AsStringV(StringBuffer sb){ this.sb = sb; }

  public void before(E host){
    this.sb = sb.append(host.toString()).append(" ");
  }

  public String return(){ return this.sb.toString(); }
}
```

Figure 5.2: Visitor implementations used with the `FlatList` DI, part I.

The self loop on `FList` allows the strategy to select paths that contain one or more `FList` objects on the way to an `E` object.

The visitors used by each adaptive method in `FlatList` are given in Figures 5.2 and 5.3.

Figure 5.2 gives the definitions of `AsStringV` and `AsCommaSepStringV`. The `AsStringV` visitor uses a class field `sb` and accumulates the string representation of each element `E` along a traversal. The `AsCommaSepStringV` visitor extends `AsStringV` and appends a comma after each element `E`. At the end of its traversal the `AsCommaSepStringV` visitor removes the last comma added in its local variable `sb` and returns `sb`'s string representation.

Figure 5.3 gives the definitions of the remaining five visitors used inside the `FlatList` DI. The `AtIndexV` visitor maintains a running counter of `E` objects encountered during traversal and stores the element found at index *i*. At the end of the traversal the visitor returns the stored `E` object or throws a

```
class AtIndexV {

  private int index;
  private int count;
  private E res;

  AtIndexV(int index){
    this.index = index;
    this.count = 0;
    this.res = null;
  }

  public void before(E host){
    if (count == index) res = host;
    count = count + 1;
  }

  public E return(){
    if (res == null)
      throw new RuntimeException("Index " + this.index + " not found");
    else return res;
  }

}
```

```
class CountV {
  private int res;

  CountV(){ this.res = 0; }

  public void before(E host){
    this.res = res + 1;
  }
  public int return() { return this.res; }
}
```

```
class GetEV {
  private E res;

  GetEV(){ this.res = null; }

  public void before(E host){
    this.res = host;
  }
  public E return() { return this.res; }
}
```

```
class GetRV{
 private FList res;

 GetRV(){ this.res = null; }

 public void before(FList host){
   this.res = host;
 }
 public FList return() { return this.res; }
}
```

```
class GetRestV {
  private FList res;

  GetRestV(){ this.res = null; }

  public void before(Cons host){
    this.res = host.getR();
  }
  public FList return() { return this.res; }
}
```

Figure 5.3: Visitor implementations used with the `FlatList` DI, part II.

runtime exception if the visitor has not encountered *i* `E` objects. The `CountV` visitor counts all `E` objects encountered during a traversal, the `GetEV` visitor returns the last `E` object encountered while the `GetRV` visitor returns the last `FList` object encountered during a traversal. Finally, the `GetRestV` visitor uses the adaptive method `getR` on the last `Cons` object encountered during a traversal.

The `FlatList` DI attaches constraints to adaptive methods that capture the properties expected by a flat list implementation and assists in defining the appropriate behavior for each adaptive method. For example, consider the behavior of the visitor `GetEV` and the behavior of the `getFirst` adaptive method. The implementation of `GetEV` visitor returns the last `E` object encountered in a traversal. Our `getFirst` adaptive method has to return the *first* `E` object in a flat list. The constraint given on `getFirst` ensures that any traversal starting at `Flist` going through a `Cons` object and terminating at an `E` object has one path, leading to one `E` object. Similarly, the definitions of the `getRest` and `getR` adaptive methods enforce a unique path to their targets.

This uniqueness restriction might appear redundant at first sight, it is not. The DI is going to be mapped on a concrete class graph where the mapping might introduce more than one path to a strategy's targets. Also, the DI might be correctly mapped to a concrete class graph but evolutions of the class graph might introduce multiple paths to a strategy's targets. Constraints guard against these situations.

5.1.2 Mapping a Demeter Interface

To use the functionality defined in a Demeter Interface we need to map the Demeter Interface's ICG to a concrete class dictionary. We first use a straightforward mapping for our `FlatList` DI to explain the extensions to class dictionaries and their mappings to DIs. We then provide more com-

```
cd LoIntegers {
//Class Dictionary
LoI : MtLoI | ConsLoI.
MtLoI = .
ConsLoI = Integer LoI.

//Mappings
for FlatList (prefix_all loi_){
  use (=>, FList, Mt)
    as LoI → MtLoI.
  use (=>, FList, Cons)
    as LoI → ConsLoI.
  use (→, Cons,f, E)
    as ConsLoI → Integer.
  use (→, Cons,r, FList)
    as ConsLoI → LoI.
  }
}
```

Figure 5.4: An implementation for a list of integers that implements the `FlatList` DI.

plex mappings of `FlatList` to further explain their behavior.

First we consider a flat list of integers as our concrete class dictionary. Figure 5.4 gives the definition of `LoIntegers` that implements `FlatList`.

The body of a class dictionary consists of the definition of classes and their relationships, as in DAJ, and a list of mapping definitions. A valid class dictionary can provide mapping definitions for each DI implemented by the class dictionary.

A mapping for a DI has to map each ICG edge once, and each ICG node has to be mapped to one and only one class dictionary class. Figure 5.4 shows the mapping for the `FlatList` DI. A mapping specification starts with the keyword `for` followed by the name of the DI for which we are providing a mapping. Following the DI's name we can optionally define a renaming for the adaptive methods defined in the DI. In Figure 5.4 we prefix the seven adaptive methods in `FlatList` with the string `loi_`.

A renaming specification consists of a list of renaming directives. A renaming directive is either a pair of method names (strings), for example, `(search,songSearch)` (Figure 2.26) or a renaming recipe such as `prefix_all` or `suffix_all`. Given a renaming directive (s_1, s_2) where s_1 and s_2 are method names (strings), the adaptive method s_1 is renamed to s_2 in the class dictionary. The renaming recipe (`prefix_all` s) prefixes all adaptive methods in the DI with the string s. Similarly the renaming recipe (`suffix_all` s) adds the suffix s to each adaptive method name defined in the DI.

The mapping consists of a list of mapping directives. A mapping directive can be either a `use-as` statement that maps an ICG edge to a path in the concrete class dictionary, or, an application of a mapping function (Figure 5.5).

An ICG edge is given using the syntax (*edge-type*,*source*,*label*,*target*), where *label* is optional. An edge-type is either an inheritance edge (=>) or a has-a edge ($\rightarrow$), source and target are ICG classes, and, labels are field names as defined in the ICG. The path can be any graph-based WYSIWYG strategy that contains a single source and single target, *i.e.*, strategies cannot use the star pattern, `*`. to select all classes or the optional pattern { } as their source or target nodes. The paths used in mappings are selectors over the class graph. Restricting paths to be WYSIWYG strategies allows for the mappings to be general and share the same benefits as with strategy definitions. We also maintain consistency in the way selectors for paths in class graphs are used throughout the DI.

A mapping function serves as a syntactic abstraction over mapping specifications. We can define a mapping function by using the keyword `mdef` followed by the name of the mapping function, a list of arguments and the mapping function's body which is a mapping definition. For example, in Figure 5.5 the mapping function `oneElement` takes one argument, an `Element`. We can apply a mapping function *m* by writing the mapping function's name followed by a list of arguments that can be either class dictionary class names and/or strategies over the class dictionary as arguments to *m*. The application of a mapping function results in an "expansion" of the mapping function's body, with the appropriate substitution for each of the mapping function arguments. For example in Figure 5.5 the first application of `oneElement(Integer)` gives:

```
use (=>, FList, Mt) as LoX → MtLoX.
use (=>, FList, Cons) as LoX → ConsLoX.
use (→, Cons,f, E) as ConsLoX → Integer.
use (→, Cons,r, FList) as ConsLoX → LoX.
```

Given a mapped DI, DAJ generates the necessary traversal code and specialized visitors for each adaptive method using the strategies and the mapping(s) provided.

Given a DI D and a class graph C that implements D, DAJ first calculates the traversal graph t for each strategy s in D using D's ICG as the class graph. DAJ validates any constraints given on adaptive methods in D that use strategy s. Once all strategies in the DI have been processed DAJ turns to class dictionaries that implement the DI and verifies that their mappings are valid, *i.e.*, for each mapping given in class dictionary C each ICG edge is mapped exactly once and each ICG node is mapped to one and only once class dictionary class.

From each of the mapping definitions DAJ creates two maps; M contains mappings of ICG nodes to class dictionary nodes, and E contains mappings of ICG edges to class dictionary strategies (Figure 5.6).

DAJ uses the two maps M and E in order to expand strategy specifications of the DI D in the class graph C. For each ICG edge $e_1 \in E$ with mapped strategy s_1, DAJ calculates the traversal graph t_1 under the class graph C. DAJ restricts mapping definitions so that each ICG edge is mapped once, and each ICG node is mapped once. In order to calculate the valid paths in C that satisfy the mapped strategy s in D, DAJ uses the mapping E to rewrite the strategy s. The rewriting of s replaces each edge $e \in s$ with its mapped strategy from the map E, *i.e.*, $E(e)$. Because of the restrictions on mappings imposed by DAJ (map each ICG edge and each ICG node once) the rewrite of strategy s using a the mapping from class graph C yields a

```
cd LoX {
  //Class Dictionary
  LoX : MtLoX | ConsLoX.
  MtLoX = .
  ConsLoX = <x> X <y> Y LoX.
  X = Integer.
  Y = String.

  //Mappings
  mdef oneElementList(Element) {
    use (=>, FList, Mt) as LoX → MtLoX.
    use (=>, FList, Cons) as LoX → ConsLoX.
    use (→, Cons,f, E) as ConsLoX → Element.
    use (→, Cons,r, FList) as ConsLoX → LoX.
  }

  //Instantiations
  for FlatList (prefix_all loi_){
    oneElementList(Integer).
  }

  for FlatList (prefix_all los_){
    oneElementList(String).
  }
}
```

Figure 5.5: Example of a mapping function. The `FlatList` is mapped twice, the first mapping allows programs to view an `LoX` list as a list of integers, the second mapping allows programs to view an `LoX` list as a list of strings.

$$
\begin{array}{lllll}
M & := & \texttt{FList} & \mapsto & \texttt{LoI} \\
 & & \texttt{Mt} & \mapsto & \texttt{MtLoI} \\
 & & \texttt{Cons} & \mapsto & \texttt{ConsLoI} \\
 & & \texttt{E} & \mapsto & \texttt{Integer}
\end{array}
$$

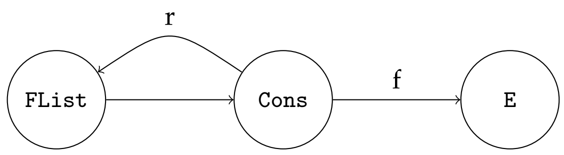
$$
\begin{array}{lllll}
E & := & (\Rightarrow,\texttt{FList},\texttt{Mt}) & \mapsto & \texttt{LoI} \rightarrow \texttt{MtLoI} \\
 & & (\Rightarrow,\texttt{FList},\texttt{Cons}) & \mapsto & \texttt{LoI} \rightarrow \texttt{ConsLoI} \\
 & & (\rightarrow,\texttt{Cons},\texttt{f},\texttt{E}) & \mapsto & \texttt{ConsLoI} \rightarrow \texttt{Integer} \\
 & & (\rightarrow,\texttt{Cons},\texttt{r},\texttt{FList}) & \mapsto & \texttt{ConsLoI} \rightarrow \texttt{LoI}
\end{array}
$$

Figure 5.6: The maps M and E for the list of integers example.

valid (single source single target) strategy graph for C.

Observe that for a valid strategy s in a Demeter Interface D will have a single source and a single target. Also, for each node $v \in s$, v is reachable from the strategy's source node and the strategy's target node is reachable from v (Definition 2, page 80). Since a mapping m has a one to one correspondence between nodes and edges in the DI and the class dictionary C, our substitution will yield a valid strategy s' for C.

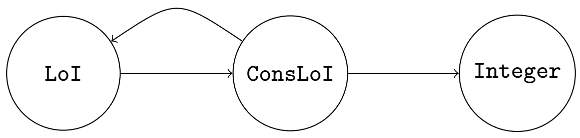
Turning to our list of integers example and to the `toLast` strategy in the `FlatList` DI the traversal graph for `toLast` using the DI's ICG contains three edges,



with `FList` as the source and `E` as the target. There is one edge from `Cons` to `FList` and that is the edge labeled `r`. The upwards inheritance edge from `Cons` to `FList` is *not* in the traversal graph because that would result in a path with two consecutive edges being $\diamond\downarrow, \diamond\uparrow$ and this is not allowed (Figure 4.7, page 89).

The three edges are mapped in E to the strategies `LoI` $\rightarrow$ `ConsLoI`, `ConsLoI` $\rightarrow$ `LoI` and `ConsLoI` $\rightarrow$ `Integer` respectively.

Replacing the edges in `toLast`'s strategy graph with the three preceding edges we obtain the strategy graph for the mapped `toLast` strategy over the concrete class graph `LoIntegers`.



DAJ the calculates all valid paths in the concrete class graph using the mapped strategy for `toLast`. In this example the traversal graph for the mapped strategy `toLast` is equal to its traversal graph.

```
class $GetEV$ {
  private Integer res;
  $GetEV$(){ this.res = null; }
  public void before(Integer host){ this.res = host; }
  public Integer return() { return this.res; }
}
```

Figure 5.7: The generated visitor for `GetEV` using the mapping in `LoIntegers`. Names enclosed in $ $ stand for unique mangled names in the generated program.

DAJ uses the same approach in order to rewrite strategies that appear in constraint annotations on adaptive methods. Once all strategies in a Demeter interface have been rewritten, DAJ verifies all constraints in adaptive methods. Each constraint contains now a rewritten strategy over the class dictionary and the constraint is evaluated against the class dictionary.

DAJ generates specialized versions of the visitors used inside a mapped DI. Visitor definitions use the ICG class names in their method implementations. For each mapping in the class dictionary DAJ generates a new visitor, with a fresh unique name, by rewriting the original visitor implementation replacing ICG class names with their mapped class dictionary class names. Figure 5.7 shows the generated visitor for `GetEV` using the mapping in `LoIntegers`.

Finally, DAJ generates the traversal code with appropriate calls to the newly generated visitors. Adaptive method renaming takes place only if a renaming map is given with the DI's mapping definition. The final program is then compiled using the original DAJ compiler.

Consider a more interesting example where we use the `FlatList` DI with a class dictionary for doubly linked lists. Figure 5.8 shows the class dictionary with two mappings for `FlatList`.

The mappings define two sets of methods, one set prefixed with `forward_` traverse the doubly linked list in the forward direction using the `next` filed in `ConsDLList`. The second send of methods prefixed with `backward_` tra-

```
cd DLList {
//Class Dictionary
DLList : MtDLList | ConsDLList.
MtDLList = .
ConsDLList = <prev> DLList <val> Integer <next> DLList.

//Mappings
mdef direction(dir) {
  use (=>, FList, Mt)
    as DLList → MtDLList.
  use (=>, FList, Cons)
    as DLList → ConsDLList.
  use (→, Cons, f, E)
    as ConsDLList → Integer.
  use (→, Cons, r, FList)
    as ConsDLList → DLList via (→, ConsDLList, dir, DLList).
}

//Instantiations
for FlatList (prefix_all forward_){
 direction(next).
}

for FlatList (prefix_all backward_){
 direction(prev).
}
}
```

Figure 5.8: The `DLList` class dictionary with the mapping for `FlatList`.

verse the doubly linked list in the backward (reverse) direction by using the `prev` field in `ConsDLList`.

Observe that if we remove the bypassing directive from the mapping in Figure 5.8, *i.e.*,

```
use (→, Cons, r, FList) as ConsDLList → DLList
```

we violate the constraint annotation on the `getR` adaptive method resulting to a compile time error.

Constraints on Demeter Interfaces can safeguard against some evolutions/mappings of the DI but cannot capture all possible evolutions/mappings of a DI that result in undesired behavior. For example, consider extending our original class dictionary for lists of integers to allow for nested lists (Figure 5.9).

```
cd LoIntegers {
//Class Dictionary
LoI : MtLoI | ConsLoI.
MtLoI = .
ConsLoI = IorList LoI.
IorList : WrappedInteger | LoI.
WrappedInteger = Integer.

//Mappings
for FlatList {
  use (=>, FList, Mt)
    as LoI → MtLoI.
  use (=>, FList, Cons)
    as LoI → ConsLoI.
  use (→, Cons,f, E)
    as ConsLoI → Integer.
  use (→, Cons,r, FList)
    as ConsLoI → LoI.
  }
}
```

Figure 5.9: Extended `LoIntegers` to allow for nested lists.

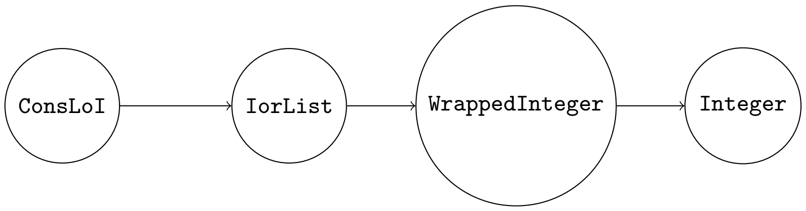
The new class dictionary defines an abstract type `IorList` that is either `WrappedInteger`[3] or `LoI`. Note that Figure 5.9 does not alter any of the original mappings from Figure 5.4. The extended class dictionary does not violate any of the DI constraints; there is still a unique path from `ConsLoI` to `Integer` and from `ConsLoI` to `LoI` through the field `r`. However, the behavior of the adaptive methods defined in `FlatList` does not take into account any element of a list that is itself a list. For example, a call to the adaptive method `size` on an instance of a list of integers that *only* contains lists as elements returns 0. The reason for this behavior is due to the mapping of the edge `Cons,E`. The mapping in Figure 5.9 maps (→,`Cons,f,E`) to the strategy `ConsLoI` → `Integer`. Calculating the valid paths in the concrete class graph for the strategy `ConsLoI` → `Integer` selects the subgraph

[3]We cannot alter the definition of Java's `Integer` class to extend `IorList`.

```
cd LoIntegers {
//Class Dictionary
LoI : MtLoI | ConsLoI.
MtLoI = .
ConsLoI = IorList LoI.
IorList : WrappedInteger | LoI.
WrappedInteger = Integer.

//Mappings
for FlatList {
  use (=>, FList, Mt)
    as LoI → MtLoI.
  use (=>, FList, Cons)
    as LoI → ConsLoI.
  use (→, Cons,f, E)
    as source:ConsLoI → ConsLoI
               ConsLoI → target:Integer.
  use (→, Cons,r, FList)
    as ConsLoI → LoI.
  }
}
```

Figure 5.10: Naive extension to the mapping in order to accommodate for nested lists causes a compile time error.



The WYSIWYG condition forces the paths selected to have only one occurrence of `ConsLoI` and `Integer`.

Notice however that a naive modification to the mapping of `FlatList` (Figure 5.10) in `LoIntegers` that allows for `E` to be mapped to all reachable `Integers` from `ConsLoI` violates the uniqueness constraint on the `getFirst` adaptive method.

The strategy mapped to the edge `Cons,f,E` contains a self loop on `ConsLoI` and therefore selects all `Integer` objects reachable through one or more `ConsLoI` objects. Calculating the valid paths in the concrete class graph for this strategy selects the subgraph

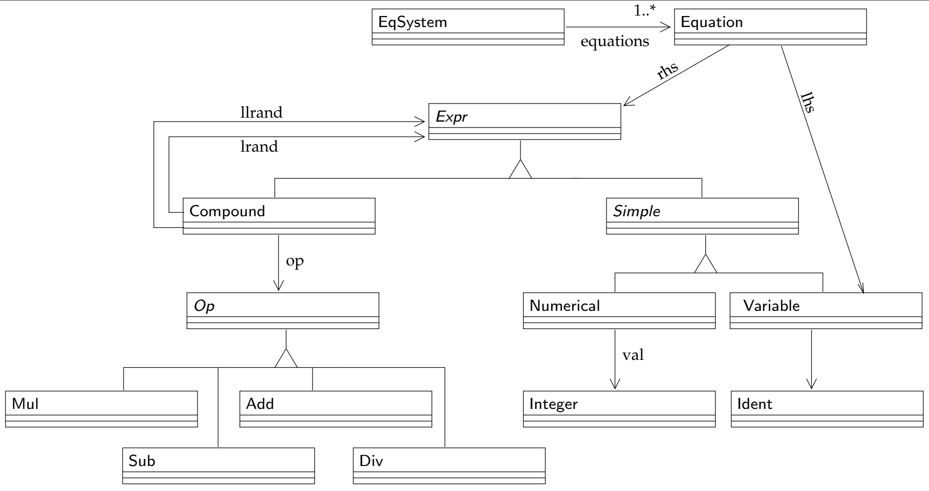
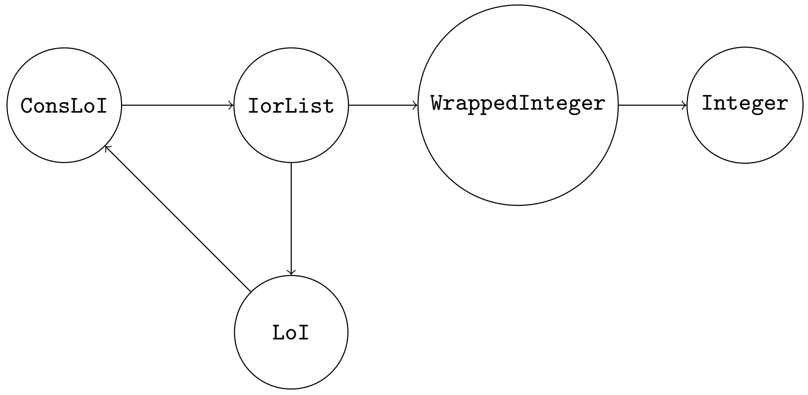


Figure 5.11: The UML equivalent of Simple Equations Class Graph.



The constraint on `getFirst` in `FlatList` requires that there is a unique path from `FList` to `E` via a `Cons` object. But with the mapping definition in Figure 5.10 we have mapped `Cons` (`ConsLoI`) to `E` (`Integer`) in a way that allows for more than one distinct path to reach `E` (`Integer`) from `Cons` (`ConsLoI`) thus violating the DI constraint.

Demeter Interfaces and constraints assist with disallowing the application of adaptive code on class dictionaries that violate the DI's constraints. The behavior of a correctly mapped DI still needs to be validated by programmers. Programmers have to rely on testing to ensure that the mapped adapted code behaves as expected.

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> Variable "=" <rhs> Expr.
Expr : Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr")".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".
```

Figure 5.12: Class dictionary in DAJ for simple equations.

5.1.3 A simple equation system with Demeter Interfaces

Our example in this subsection is about systems of equations in which we want to check that all used variables are defined (we call this a *semantic checker*). We first discuss a solution in DAJ without Demeter Interfaces and then proceed to provide a solution using Demeter Interfaces and compare the two solutions.

We define a simple equation system where each equation introduces a new variable binding and bindings have global scope, *e.g.*, $x = 5; y = 9; z = x + y;$. Figure 5.12 shows the class dictionary for the simple equation system and Figure 5.14 shows the traversal, visitor and main class that implements the semantic checker. A cd file is a textual representation of the object oriented structure of the program which specifies classes and their members. Figure 5.11 provides the UML representation of the class dictionary in Figure 5.12. DAJ uses a class dictionary as a grammar definition, providing a language that can parse in sentences and create the appropriate object instances. Tokens in the class dictionary surrounded in quotes define the generated language's syntax tokens (Figure 5.13). Parametrized classes are defined in class dictionaries using a tilde ("~") operator, *e.g.*, `BList(S)` defines a list enclosed in parentheses of one (or more) elements of type `S`

```
(x = 5;
 y = (x − 2);
 z = ((y−x) + (y+9)))
```

Figure 5.13: An instance of a simple equations system given as input to DAJ.

each element separated by a semicolon.

In our simple equation system the strategy `defined` visits all `Variable` objects starting form an `EqSystem` object and bypassing any edge with the name `rhs` along the way. This strategy selects all bindings in a set of equations. The traversal method `getDefined` uses the strategy `defined` and the visitor `CollectDef` to collect all bindings in a set of equations.

In a similar manner the strategy `used` selects all `Variable` objects from each equation in `EqSystem` by bypassing any edge with the name `lhs`. The traversal method `getUsed` uses the strategy `used` and the same visitor class `CollectDef` to collect all uses of bindings in a set of equations. Both traversal methods return return a set of bindings and the method `check` checks that the set of used bindings is a subset, or equal to, the set of defined bindings. The method `check` is introduced in the class `EqSystem`.

With the completed AP implementation of the semantic checker in place we can now evaluate our solution and verify the claims made, both in favor and against, AP. For the simple equation system example, modifying the system so that equations are now in prefix notation does not affect the program's behavior. Doing so requires a single modification to the class dictionary,

```
Compound = ‘‘(’’ <op> Op <lrand> Expr <rrand> Expr ‘‘)’’.
```

No other changes are needed to the traversal file or the visitor. The modification simply changed the order between the `Op` data member and the first `Expr` data member of the `Compound` class. This is not surprising since even in plain Java, switching the order of member definitions does not change a

```
// SemanticChecker.trv
aspect SemanticChecker {
  declare strategy : defined: from EqSystem bypassing (→ ∗,rhs,∗) to Variable;
  declare traversal: public Set getDefined(): defined(CollectDef);
  declare strategy : used : from EqSystem bypassing (→ ∗,lhs,∗) to Variable;
  declare traversal: public Set getUsed():used(CollectDef);

  public boolean EqSystem.check(){
    return this.getUsed().subseteq(this.getDefined());
  }
}
// CollectDef.java Visitor
class CollectDef{
  Set res;

  public CollectDef() { this.set = new Set(); }
  public void before (Variable v) { this.res.add(v); }
  public Set return () { return this.res; }
}
```

Figure 5.14: The traversal file (`SemanticChecker`), visitor class (`CollectDef`) for the system of simple equations in DAJ.

program's behavior. Lets consider a more drastic extension, lets add exponent operations to our system but also impose precedence between operators. Listing 5.15 shows the complete class dictionary file, the definitions have been factored to accommodate for operator precedence. Again, no other changes are need to either the traversal file or the visitor. The semantic checker still functions correctly.

Why is the semantic checker unaffected by these changes? In both cases the modifications to the cd file did not falsify the strategy (*i.e.*, there is still a path from source to the target) and it did not affect the way by which variables are defined and used in the equation system (*i.e.*, there is no other way of binding a variable to equations other than `‘‘=’’` and variables still have global scope). Any modification to the class dictionary that does not falsify the strategy and does not alter the assumptions about variable definition and usage within the equation system will not affect the semantic checker's code.

However any alteration that either

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> Variable "=" <rhs> Expr.
Expr : AddExp | SubExp | Term.
AddExp = Add Expr Term.
SubExp = Sub Expr Term.
Term : MulTerm | DivTerm | Expo.
MulTerm = Mul Term Expo.
DivTerm = Div Term Expo.
Expo : Raised | Factor.
Raised = "**" Expo Factor.
Factor : Simple | BExpr.
BExpr = "(" Expr ")".
Simple : Variable | Numerical.
Variable = Ident.
Numerical = <val> Integer.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ S {";" S}.
```

Figure 5.15: Extended class graph accommodating exponents and operator precedence.

- modifies class and/or class member variable names that are explicitly referenced by traversals and/or visitors,
- or, breaks an assumption about the system on which adaptive code depends on (*e.g.*, adding a new variable binding construct to the equation system like `let` for local bindings or functions with arguments).

will alter the program's behavior.

For example, altering the equation system to allow for function definitions with arguments causes no compile time error, but results in erroneous program behavior. This modification breaks two assumptions:

1. There is only one new `Variable` defined at each equation.
2. All variables have global scope and thus can be used anywhere.

Adaptive methods, as well as the visitor, depend on these assumptions. However these assumptions are not explicitly captured in AP programs.

There is no tool support to stop such modifications. In fact naively extending the equation system to accommodate for functions parameters, as in Listing 5.16, will generate a valid AP program that will provide the wrong results for the semantic checker.

With larger AP programs, it becomes nearly impossible to find all these implicit assumptions and even harder to predict which modifications will cause erroneous behavior. Programmers have to rely on exhaustive testing in order to increase their confidence that the program still behaves according to its specification. This in turn limits the effectiveness of AP and its application in iterative development since modifications to the data structure due to an iteration can introduce bugs in parts of the code developed in previous iterations.

These dependencies impede parallel development and decrease productivity. Addressing these issues requires

- The ability to define the assumptions made by adaptive code about the underlying data structure,
- Tool support to allow for the verification of these assumptions,
- Decrease the dependency on class and class member variable names,
- The modularization of only the relevant data structure information for each adaptive behavior instead of the whole class dictionary.

A Demeter Interface resides between a class graph and the *implementation* of adaptive behavior, *i.e.*, adaptive methods and visitor implementations. Figure 5.17 shows the Demeter Interface for the simple equation system with strategies, traversals and constraints, as well as with its visitor implementation. Observe that the ICG given in Figure 5.17 does not contain any lists, but rather, it is a simple, flat (no lists) representation of an equation system. The ICG defines an `ESystem` as having a `Definition`. A `Definition` contains in turn an equation that has a left hand side and a

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> VarOrFunc "=" <rhs> Expr.
VarOrFunc : Variable | Function.
Function = "fun" <fname> Variable
            "(" <args> CList(Variable)")".
Expr : FunCall | Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
FunCall = <fname> Variable "(" <fargs> CList(Simple) ")".
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr")".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".
CList(S) ~ S {"," S}.
```

Figure 5.16: Class Graph for equation systems with functions of one argument.

```
// ExprICG.di File
di ExprICG {
  //ICG
  ESystem = Definition.
  Definition = <def> DThing "=" <body> Body.
  Body = UThing.
  UThing = Thing.
  DThing = Thing.
  Thing = .

  // Strategies
  declare strategy: gdefinedIdents: from ESystem via DThing to Thing.
  declare strategy: gusedIdents: from ESystem via UThing to Thing.
  declare strategy: definedIdent: from Definition via DThing to Thing.

  // Traversals
  @constraints{unique(definedIdent) &&
               (unique(gdefinedIdents) || nonempty(gdefinedIdents))}
  declare traversal: void printDefined(): gdefinedIdents(DVisitor);
  @constraints{unique(gusedIdents) || nonempty(gusedIdents)}
  declare traversal: void printUsed(): gusedIdents(DVisitor);
}
// DVisitor.java File
class DVisitor {
  public void before(Thing t){ System.out.println(t.toString()); }
}
```

Figure 5.17: The Demeter Interface for the simple equations system.

right hand side. A binding is found on the left hand side of the definition and a use of a binding is found on the right hand side of the definition.

The DI also defines three strategies, `gdefinedIdents` finds a binding starting from `ESystem` and navigating via a `DThing` to a `Thing`. The strategy `gusedIdents`, in a similar way, finds all binding uses in an `ESystem` by navigating via a `UThing` to a `Thing`. The third strategy `definedIdent` starts from a `Definition` and navigates via `DThing` to `Thing`.

The DI also defines two traversal methods, `printDef` uses the strategy `gdefinedIdents` and the `DVisitor` to print all binding definitions. [4] Similarly, the traversal method `printUsed` prints all uses of a binding.

Both traversal methods are annotated with constraints. The constraints on `printDefined` ensure that there is one, and only one, path starting from `Definition` via `DThing` to `Thing` and expect that there is one or more paths that lead to a `Thing` starting from `ESystem` and navigating via a `DThing`. The pattern `unique(s) || nonempty(s)` can be thought of as specifying a one to many relationship on the paths selected by strategy `s`. This is similar to the UML notation for associations where we specify `1..*` on UML diagrams to the multiplicity of an association as one or more. The annotation on `printUsed` only requires that there is one or more paths that satisfy the strategy `gusedIdents`. This annotation on `gusedIdents` specifies that we can use this DI with class graphs that allow for a one or more uses of a binding to occur in a definition's right hand side.

Figure 5.18 gives an example implementation of the `ExprICG` Demeter Interface (on the right) along with a driver class (on the left). The concrete class dictionary `InfixEQSystem` (Figure 5.18) provides a definition of its equation system and a mapping M between the classes in its class graph and all the classes in `ExprICG`'s interface class graph. The mapping in Figure 5.18 maps an `ESystem` to `EquationSystem`, a `Definition` to `Equation`.

[4] Altering the DI code to collect all binding definition and binding uses as sets, just like the preceding DAJ implementation requires that we use the same visitor as before, `CollectDef` but replace all occurrences of `Variable` with `Thing`.

```
import java.io.*;

class Main {
  public static void main(String[] args){
    try {
      InfixEQSystem ieqs = InfixEQSystem.
            parse(new File(args[0]));
      System.out.println("IDs in def:");
      ieqs.printDefined();
      System.out.println("IDs in use:");
      ieqs.printUsed();
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

```
cd LetEQSystem{
  EquationSystem = <eqs> List(Equation).
  List(S) ~ "(" {S} ")".
  Equation = <lhs> Variable "=" <rhs>
        Expr.
  Expr : Simple | Compound.
  Simple : Variable | Numerical.
  Variable = Ident.
  Numerical = <v> Integer.
  Compound = <lrand>List(Expr) <op>
        Op <rrand>List(Expr).
  Op : Add | Sub | Mul | Div.
  Add = "+".
  Sub = "-".
  Mul = "*".
  Div = "/".

  //Mappings
  for ExprICG (
    use (→, ESystem, Definition)
      as EquationSystem to Equation.
    use (→, Definition, DThing)
      as (→, Equation,lhs,*) to Variable.
    use (→, Definition, Body)
      as Equation to Expr.
    use (→, Body, UThing)
      as (→, Equation,rhs,*) to Variable.
    use (→, UThing, Thing)
      as Variable to Ident.
    use (→, DThing, Thing)
      as Variable to Ident.
  )
}
```

Figure 5.18: `InfixEQSystem` defines a class graph and a mapping of the entities in the class graph to the interface class graph of `ExprICG`. The driver class `Main` uses the adaptive methods introduced by `ExprICG`.

The mapping also maps a DThing to all reachable Variable objects reachable from Equation via an edge labeled lhs. The ICG class Body is mapped to Expr and a UThing is mapped to all the Variable objects reachable from Equation via an edge labeled lhs. The last two mapping directives map UThing and DThing to Variable and Thing to Ident.

With the simple equation system implemented using Demeter Interfaces we now extend the system by performing the same sequence of program evolutions as with the DAJ solution to the semantic checker. As a first evolution step we want to change from infix notation to prefix notation. This is a modification that does not alter the program's behavior even in the original DAJ solution. Moving to a prefix notation requires to change the definition of Compound in InfixEQSystem to

```
Compound = <op> Op <lrand> List(Expr) <rrand> List(Expr).
```

This change does not affect the Demeter Interface at all. We update the equation system class graph while keeping the original mapping M. All constraints of the DI are still satisfied after they are mapped into the actual interface class graph and the adaptive methods function correctly.

It is important to note that during this evolution step, only the DI and the concrete implementation of the interface class graph was needed. Under the assumption that the DI's constraints capture the semantic checker's intend, the static assurances provided by the tool because of the DI, suffice to show that the strategies pick the correct paths and that the semantic checker still operates as expected. The Demeter Interface allows in this case for separate development and ease of evolution. The concrete class graph and its mapping can be a maintained separately while adaptive code can be developed based on the publicly available DI. Alterations made to the concrete class graph do not need to be visible to adaptive code maintainers unless it affects the mapping to an implemented DI. This form of data hiding through the Demeter Interface also provides for easier maintainability

```
// ParamExprICG.di File
di ParamExprICG {
  ESystem = Definition.
  Definition = <def> DThing <fnc> DFThing <body> Body.
  DThing = Thing.
  DFThing = <fname> DThing <fparam> DThing.
  Body = <fc> UFThing UThing.
  UFThing = <name> UThing <aparam> UThing.
  UThing = Thing.
  Thing =.

  // Strategies
  declare strategy: definedIdents : from ESystem to DThing.
  declare strategy: usedIdents : from ESystem to UThing.
  declare strategy: dName : from DFThing via (→, ∗,fparam,∗) to Thing.
  declare strategy: uName : from UFThing via (→, ∗,aparam,∗) to Thing.
  declare strategy: dFName : from DFThing via (→, ∗,fname,∗) to Thing.
  declare strategy: uFName : from UFThing via (→, ∗,name,∗) to Thing.

  // Traversals
  @constraints{unique(definedIdents) || nonempty(definedIdents)}
  declare traversal: LinkedList getDefined(): definedIdents(PVisitor);
  @constraints{unique(usedIdents) || nonempty(usedIdents)}
  declare traversal: LinkedList getUsed(): usedIdents(PVisitor);
  @constraints{unique(dName)}
  declare traversal: LinkedList getDefName(): dName(PVisitor);
  @constraints{unique(uName)}
  declare traversal: LinkedList getUsedName(): uName(PVisitor);
  @constraints{unique(dFName)}
  declare traversal: LinkedList getDefArg(): dName(PVisitor);
  @constraints{unique(uFName)}
  declare traversal: LinkedList getUsedArg(): uName(PVisitor);

  //Introduction of a helper method
  public boolean ESystem.checkBindings(LinkedList l1, LinkedList l2){
    // checks appropriate variable usage (elided)
  }
}
```

Figure 5.19: The evolved Demeter Interface that deals with one argument functions.

and higher system modularity.

As our next evolution step we extend the set of operators to include exponents and add operator precedence. Keeping the headers and mapping definition the same as in `InfixEQSystem` and replacing the data structure definition by that of Figure 5.15 gives us a working AP system. The modifications made to the data structure to accommodate for exponents and operator precedence do not invalidate any of the DI's constraints and the resulting AP program behaves as expected.

In the next evolution step we want to add functions with one argument to the equation system. This evolution step affects information that is relevant to the semantic checker. The semantic checker has to deal with parameter names on each function definition but also uses of function definitions that may appear on the right-hand side of equations. Unlike definitions so far function parameters do not have global scope, their scope is local to the function definition. A naive approach would be to alter the class dictionary as in Figure 5.20.[5] Altering the data structure and only the mapping to `DThing` results in a compile time error. The reason for this error is the predicate `unique(definedIdent)` from `ExprICG`, no longer holds. The modification to allow functions with one parameter breaks one of the assumptions of the interface, in particular the fact that we can reach more than one variable through the left hand side of the equal sign. With one argument functions the meaning of what is defined and what is its scope has changed and these changes have to be reflected in the Demeter Interface.

It is important to note that for this evolution step that the interface has to change (Figure 5.19). With a new interface class graph `ParamExprICG` we can abstractly reason about semantically checking systems with one argument functions. The two strategies `definedIdent` and `usedIdents` are used to navigate to definitions and references of variable names, both function names as well as simple variables. The strategies `dName` and `uName` are then used to collect arguments (at function definition) and actual arguments (at function invocation) respectively. Similarly `dFName` and `uFName` collect function names at function definitions and function usage respectively. The traversal declarations use the strategies to collect `Thing` objects. The implementation of the method `checkBindings` is introduced into `ESystem` and

[5]To keep the example simple we do not allow the usage of function calls as arguments to other functions, *i.e.*, $f(f(3))$

```
cd ParamEquations {
  EqSystem = <equations> BList(Equation).
  Equation = <lhs> VarOrFunc "=" <rhs> Expr.
  VarOrFunc : Variable | Function.
  Function = "fun" <fname>Variable "("<args>Variable")".
  Expr : FunCall | Simple | Compound .
  Simple : Variable | Numerical.
  Variable = Ident.
  FunCall = <fname> Variable "(" <fargs> Simple ")".
  Numerical = <val> Integer.
  Compound = "("<lrand>Expr <op>Op <rrand>Expr")".
  Op : Add | Sub | Mul | Div.
  Add = "+".
  Sub = "-".
  Mul = "*".
  Div = "/".
  BList(S) ~ "(" S {";" S} ")".

  // Mappings
  for ExprICG (
    use (→, ESystem, Definition)
      as EquationSystem to Equation.
    use (→, Definition, DThing)
      as (→, Equation,lhs,∗) to Variable.
    use (→, Definition, Body)
      as Equation to Expr.
    use (→, Body, UThing)
      as (→, Equation,rhs,∗) to Variable.
    use (→, UThing, Thing)
      as Variable to Ident.
    use (→, DThing, Thing)
      as Variable to Ident.
  )
}
```

Figure 5.20: Extending the class dictionary to accommodate function definitions using the `ExprICG` DI.

```
cd ParamEquations {
EqSystem = <equations> BList(Equation).
Equation = <lhs> VarOrFunc "=" <rhs> Expr.
VarOrFunc : Variable | Function.
Function = "fun" <fname> Variable "(" <args> Variable")".
Expr : FunCall | Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
FunCall = <fname> Variable "(" <fargs> Simple ")".
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr")".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".


for ParamExprICG(
    use (→, ESystem, Definition) as EqSystem to Equation.
    use (→, Definition, DFThing) as Equation to Function.
    use (→, Definition, Body) as Equation to Exp.
    use (→, Definition, DThing) as (→, Equation, lhs,∗) to Variable.
    use (→, DFThing, fname, DThing) as (→, Function, fname, ∗) to Variable.
    use (→, DFThing, fparam, DThing) as (→, Function, args, ∗) to Variable.
    use (→, Body, fc, UFThing) as Exp to FunCall.
    use (→, Body, UThing) as Exp bypassing FunCall to Variable.
    use (→, UFThing, name, UThing) as (→, FunCall, fname, ∗) to Variable.
    use (→, UFThing, aparam, UThing) as (→, FunCall, fargs, ∗) to Variable.
    use (→, UThing, Thing) as Variable to Ident.
    use (→, DThing, Thing) as Variable to Ident.
    )
}
```

Figure 5.21: Modifications to the concrete class dictionary to accommodate single argument functions.

it is used to check the correct usage of variable and function definitions. The inputs to this function are two lists where the first represents variable and function definition names at different scopes and the second represents names of variables and functions references at their corresponding scope. Figure 5.21 shows the class graph that implements `ParamExprICG`. The mapping in Figure 5.21 maps `ESystem` to `EqSystem`, `Definition` to `Equation`, `DFThing` to `Function` and `Body` to `Expr`. The `ICG` class `DThing` is mapped to all reachable `Variable` objects via an edge with source `Equation` and label

```
import java.util.LinkedList;

class PVisitor {
  LinkedList env;

  PVisitor(){ this.env = new LinkedList(); }
  public void before(Thing t) { env.add(t); }
  public void before(UFThing ud) {
    LinkedList rib = getUsedName();
    rib.addAll(getUsedArg());
    env.add(rib);
  }
  public void before(DFThing ud) {
    LinkedList rib = getDefName();
    ribaddAll(getDefArg());
    env.add(rib);
  }
  public LinkedList return(){ return env; }
}
```

```
import java.io.*;

class Main {
  public static void main(String[] args){
    boolean codeOk ;

    PVisitor defV = new PVisitor();
    PVisitor useV = new PVisitor();
    ParamEquations pe = ParamEquations.
         parse(new File(args[0]));
    codeOk = pe.checkBindings( pe.getDefined(
         defV), pe.getUsed(useV));
    if (!codeOk)
      System.out.println(" Variables used
           before they where defined");
  }
}
```

Figure 5.22: Changes to the interface affect `Main`. The definition of `PVisitor` is used to check for the local parameter names in parametric equations.

`lhs`. The ICG edge labeled `fname` with source `DFThing` and target `DThing` is mapped to all reachable `Variable` object starting at an edge in the class graph with source `Function` and label `fname`. Similarly, the ICG edge labeled `fparam` with source `DFThing` and target `DThing` is mapped to all reachable `Variable` objects starting at the class graph edge labeled `args` with `Function` as the edge source. The mapping also maps `UFThing` to `FunCall` and the edge (`Body`, `UThing`) to all reachable `Variable` objects starting at `Exp` and bypassing `FunCall`. The two edges in ICG from `UFThing` to `UThing` labeled `name` and `aparam` are mapped to all reachable `Variable` objects starting at `FunCall` and navigating via the edge labeled `fname` (for `name`) and `fargs` (for `aparam`). The last two mapping directives in Figure 5.21 map `UThing` and `DThing` to `Variable` and `Thing` to `Ident`. Figure 5.22 shows the visitor implementation and the driver class.

In this evolution step, the Demeter Interface helped by disallowing a naive extension that would violate the intended behavior of the original

Demeter Interface. The nature of the evolution required an extension of the interface and that resulted to changes in the driver class and a new concrete class dictionary. It is important to note how the Demeter Interface exposed the erroneous usage of the `ExprICG` interface for this evolution step and assisted in updating all the dependent components due to the definition of `ParamExprICG`.

Although Demeter Interfaces are a big improvement over traditional AP, their usage does not completely remove the need for testing adaptive code after modifications are made to the class graph. The mechanisms behind Demeter Interfaces rely on the appropriate constraints and mapping between the ICG and the class dictionary. Constraints in the interface could be too permissive allowing modifications to a class dictionary that lead to unintended behavior. Through testing we can verify the program's behavior and strengthen the program's constraints accordingly. The abstraction provided by the ICG assist programmers in this task allowing them to focus on the relevant subset of their application.

5.2 Case Study

In this section we discuss our case study, a re-implementation of a Design by Contract (DbC) system for Java originally implemented in DAJ. The original implementation CONAJ [38, 32] uses the original DAJ system, the re-implementation DCONAJ uses our extended DAJ implementation that supports WYSIWYG strategies and Demeter Interfaces.

Both systems extend a subset of Java's[6] syntax to accommodate for preconditions, post-conditions and invariant specifications in a class definition. Both systems, CONAJ and DCONAJ, rewrite all input files by commenting out all contract definitions and create AspectJ aspect definitions that are responsible for the runtime evaluation of contracts. The runtime evaluation

[6]Java version 1.3, without packages, arrays and nested class/interface definitions.

of contracts (pre and post-conditions) takes into account the type hierarchy and follows the work of Findler et. al. [12, 13].

Figure 5.23 shows the implementation of a stack in our extended Java syntax with contracts. Contract definitions contain a boolean Java expression that allows for the keywords `old` and `result`. The keyword `old` is used in post-condition definitions to refer to the state of the object at the time when the method was called. The keyword `result` is also used inside a post-condition definition and refers to the value returned by the method's execution. Both systems expect contract definitions to be side effect free but do not enforce this restriction. To support the usage of `old` the generation phase introduces a `clone` method inside every class that uses `old` in one (or more) of its contract definitions. The implementation of the clone method implements a shallow clone on the class' fields.

The rewrite process is split into two steps. First we collect information about classes and their supertypes, classes and their invariants, methods and their signatures and contracts. The tool verifies that each contract definition is valid, *i.e.*, pre-conditions refer to method arguments and methods that are visible within the class. The second step walks over the abstract syntax tree (AST) and generates Java classes and AspectJ aspects.

Informally CONAJ generates one aspect for each class that contains contract definitions. Aspect generation goes through the following steps:

1. Generate pointcuts that capture calls to methods for which we have contracts defined. For each method m in type t we generate two pointcuts; one pointcut to capture calls to method m on an instance whose static and runtime type is t, and one pointcut to capture calls to method m on an instance whose static type is t' and its runtime type is t where $t \neq t'$ and t is a subtype of t'.

2. Generate before and after advice for each generated pointcut. Before advice validates pre-conditions and after advice validates post-

```
class MyStack {
  @invariant{0<=size() && size()<=maxSize}
  protected Vector elements;
  protected int maxSize;
  protected int size;

  MyStack(int n){
    elements = new Vector(n);
    maxSize = n;
    size = 0;
  }
  public int size() { return size; }
  public void push(int i){
    @pre{!full()}
    @post{!empty() && top()==i && size()==old.size() + 1}
    Integer val = new Integer(i);
    elements.add(val);
    size=elements.size();
  }
  public int pop(){
    @pre{!empty()}
    @post{!full() && size()==old.size() − 1}
    Integer result = (Integer) elements.lastElement();
    elements.remove(elements.lastIndexOf(result));
    size = elements.size();
    return result.intValue();
  }
  public int top(){
    @post{size() == old.size()}
    Integer result = (Integer) elements.lastElement();
    return result.intValue();
  }
  public boolean full(){return (size() == this.maxSize);}
  public boolean empty(){return elements.isEmpty();}
}
```

Figure 5.23: Stack implementation with contracts. The `Vector` class is an implementation that has the same methods as `java.util.Vector`.

conditions. The body of before and after advice is a modified version of the contract definition provided by the programmer. The pointcuts bind the running instance of the object (the target instance to a method call) and update the contract definition so that any calls to the instance are redirected appropriately. Before advice calls the `clone` method if `old` is used inside the method's post-condition. After advice binds the value returned by the method if `result` is used inside the method's post-condition.

3. Generate a `clone` method if necessary.

4. Generate helper methods to carry out the necessary hierarchical checks on pre and post-conditions.

We discuss parts of the implementations of DConAJ that differ from ConAJ. We then evaluate the advantages and disadvantages of DIs.

Central to the generation phase is the manipulation of methods and specifically their formal argument list. Method arguments are used in contract definitions *and* are essential for pointcut generation. The generated pointcut definitions use both the types and the names given to formal arguments in the Java program for two purposes:

1. formal argument types assist with disambiguation between overloaded versions of a method in the input program,

2. formal argument types and formal argument names are used to bind values passed as arguments to a method call in the original program, to their appropriate name within aspect definitions.

AspectJ provides the pointcut descriptor `args` which takes a list of arguments that can either be a list of types, or a list of variable names. When a list of types is given to `args` then a call pointcut matches method calls whose argument list has the same types (in order) as those given to `args`.

```
di List {

  //ICG
  List : Mt | Cons.
  Mt = .
  Cons = E List.
  E = .

  // strategy specifications
  declare strategy: toFirst : from List via Cons to E.
  declare strategy: toRest : from List to Cons.
  declare strategy: toAll : from List to *.
  declare strategy: cons2rest : from Cons via (→*,r,*) to List.

  // traversal specifications
  @constraint {unique(toFirst)}
  declare traversal: public E getFirst() : toFirst(GetEV);
  @constraint {unique(toRest)}
  declare traversal: public FList getRest() : toRest(GetRestV);
  declare traversal: public int size() : toAll(CountV);
  declare traversal: public String asString(StringBuffer sb) : toAll(AsStringV);
  declare traversal: public E atIndex(int i) : toAll(AtIndexV);
  @constraint {unique(cons2rest)}
  declare traversal: public FList getR() : cons2rest(GetRV);

  // extra adaptive methods not in FlatList.di
  declare traversal: public String asSepString(String s, StringBuffer sb): toAll(AsSepStringV);
  declare traversal: public int getIndex(E e) :toAll(GetIndexV);
  declare traversal: public String selectedIndexes(java.util.List indxs, String s, StringBuffer sb
      ) :toAll(SelectedIndexV);
}
```

Figure 5.24: The `List` DI used inside DCONAJ.

When a list of variable names is given[7] `args` binds the values passed as arguments to a method call to these names. AspectJ's `args` pointcut designator allows `*` to appear at positions that the type (or name) is not to be used for matching (or binding) arguments.

We use the `List` DI to introduce behavior for generating the `arg` pointcut designators inside pointcut definitions. The `List` DI (Figure 5.24) is similar to `FlatList` (Figure 5.1) and uses the same `CountV`, `AsStringV` and `AtIndex` visitors. Figure 5.25 shows the implementation of the three extra visitors `AsSepStringV`, `GetIndexV`, and `SelectIndexV`.

[7]These are the names of the pointcut's formal arguments.

```
class AsSepStringV extends AsStringV {
  protected String sep;
  AsSepStringV(String sep, StringBuffer sb){
    super(sb);
    this.sep = sep;
  }
  public void before(E host){ this.sb = sb.append(host.toString()).append(sep); }
  public String return(){ return this.sb.reverse().delete(0,2).reverse().toString(); }
}
```

```
class GetIndexV{
  private E e;
  private int count;
  private boolean found;
  private int res;

  GetIndexV(E e){
    this.e = e;
    this.count = 0;
    this.res = 0;
    this.found = false;
  }
  public void before (E host) {
    if (host.equals(e)) { this.found = true; }
    this.count++;
  }
  public int return(){
    if (found) { return res;}
    else {
      throw new RuntimeException("Element "+ e.toString() + " not found");
    }
  }
}
```

```
import java.util.List;
class SelectedIndexV extends AsSepStringV {
  private int count;
  private List indxs;
  private String s;

  SelectedIndexV(List indxs, String s, StringBuffer sb){
    super(",", sb);
    this.count = 0;
    this.s = s;
    this.indxs = indxs;
  }
  public void before (E host) {
    if (indxs.contains(this.count)) { super.before(host); }
    else { this.sb = sb.append(s).append(this.sep);}
    this.count++;
  }
}
```

Figure 5.25: The three extra visitors used inside `List` DI.

The adaptive methods defined in `List` DI are straightforward with the exception of `selectedIndexes`. The adaptive method `selectedIndexes` takes in a list (a standard Java list) `indxs`, a string `s` and a `StringBuffer`. The first argument is a list of integers that designate the index of method arguments that are also used in the definition of a contract. The `SelectedIndexV` visitor generates the argument list for `args` replacing all positions that are *not* in `indxs` with `s` (typically `*`).

DI (Figure 5.1) is mapped three times (Figure 5.26) on the subtree of the AST that represents formal arguments. Each mapping gives a different view of the formal arguments list, one view maps the elements of the list to the formal argument (first mapping in Figure 5.26), another maps elements to the formal argument types (second mapping in Figure 5.26) and the last view maps elements to formal argument names (third mapping in Figure 5.26).

We use the three mappings on formal arguments in the generation of method pointcuts in order to distinguish between overloaded versions of the method in the input program and bind the appropriate values for method arguments inside advice. In the case where a contract specification does not refer to any of the method's arguments the `args` pointcut designator lists the method's argument types in order, *e.g.*, `args(int,bool,String)` selects calls to a method whose formal argument list has three arguments with types `int`, `bool` and `String` in exactly this order.

In the case where some of the method's arguments are used in the contract the pointcut generated contains two `arg` pointcut designators; the first contains all argument types, *e.g.*, `args(int,bool,String)`, the second provides names to bind values passed as arguments, *e.g.*, `args(index,*,name)` binds the first argument to `index`, does not bind the second argument, and binds the third argument to `name`. The names given to the `args` pointcut designator are the names of the pointcut's formal arguments. For example, given a method `m` in class `C` with formal arguments `int i` and `String s` and

```
cd Conaj
{
  //elided parts ...
  //related cd definitions with parsing directives removed
  FormalParameterList : FormalParameterMt | FormalParameterCons.
  FormalParameterMt = .
  FormalParameterCons = FormalParameter "," FormalParameterList.
  FormalParameter  = [ CFinal ] Type VariableDeclaratorId .
  VariableDeclaratorId = Identifier .
  //elided parts ...
  //mappings

  mdef formalArgs(T) {
    use (=>, List, Mt)
      as FormalParameterList → FormalParameterMt.
    use (=>, List, Cons)
      as FormalParameterList → FormalParameterCons.
    use (→, Cons, E)
      as FormalParameterCons → T.
    use (→, Cons, List)
      as FormalParameterCons → FormalParameterList.
  }

  for List (prefix_all fargs_){ formalArgs(FormalParameter). }

  for List (prefix_all ftypes_){ formalArgs(Type). }

  for List (prefix_all fname_){ formalArgs(VariableDeclaratorId). }
}
```

Figure 5.26: Mappings for the `List` DI with the relevant section from the class dictionary.

both `i` and `s` are used inside `m`'s precondition, the generated pointcut will be

```
pointcut m_pcd(int i, String s) :
    //call pointcut designator elided
    && args(int,String)
    && args(i,s);
```

Implementing the same behavior in CONAJ without DIs and WYSIWYG strategies pauses a design dilemma

1. design the traversal file with a general strategy using * as the target and use specialized visitors that contain specific before and after operations for types of interest (Figure 5.27), or,

2. design the traversal file with more specific strategies that navigate to types of interest along with visitors that can be reused between strategies (Figure 5.29).

Figure 5.27 gives the definition of `FormalParams` traversal file which uses the general strategy from `FormalParameterList` to *. Traversal declarations mimic the operations introduced by `List` DI in Figure 5.24. With the exception of `asString`, for each traversal declaration in `List` DI, the `FormalParams` traversal file defines one method for each AST node that we operate on, *e.g.*, `countFP`, `countTypes` and `CountIDV` count the number of `FormalParameter` objects, `Type` objects, and `VariableDeclaratorId` objects respectively. The visitors for the count operations `CountFormalParameterV`, `CountTypeV` and `CountIDV` are similar and differ only on the argument type of their return method.[8] The visitors for the remaining operations follow a similar pattern.

Figure 5.29 gives the definition of `FormalParametersSpecific` traversal file which uses four strategies; the same general strategy as in Figure 5.27, a strategy that navigates to all `FormalParameter` objects, a strategy that navigates to all `Type` objects and a strategy that navigates to all objects of type `VariableDeclaratorId`. The use of specific strategies allows for some traversal specifications to share the same visitor implementation. For traversal declarations that return the same type and operate on `Type` and `VariableDeclaratorId` objects, *e.g.*, `countTypes` and `countNames` their respective strategies ensure that `VariableDeclaratorId` objects cannot be traversed by the strategy `toFP`, and `Type` objects cannot be traversed by the strategy `toID`. The visitor `CountTypeOrIDV` (Figure 5.30) takes advantage of

[8]DAJ does not support Generics. Subsection 5.2.1 discusses the use of Generics to implement the same adaptive behavior.

```
aspect FormalParams {
  declare strategy: toAll : from FormalParameterList to ∗.

  // traversal specifications
  // counters
  declare traversal: public int countFP() : toAll(CountFormalParameterV);
  declare traversal: public int countTypes() : toAll(CountTypeV);
  declare traversal: public int countNames() : toAll(CountIDV);

  declare traversal: public String asString(StringBuffer sb) : toAll(AsStringFormalParamsV);

  // atIndex
  declare traversal: public FormalParameter atIndex(int i) : toAll(AtIndexFPV);
  declare traversal: public Type atIndex(int i) : toAll(AtIndexTV);
  declare traversal: public VariableDeclaratorId atIndex(int i) : toAll(AtIndexIDV);

  // asSepString
  declare traversal: public String asSepStringFP(String s, StringBuffer sb): toAll(
        AsSepStringFPV);
  declare traversal: public String asSepStringT(String s, StringBuffer sb): toAll(
        AsSepStringTV);
  declare traversal: public String asSepStringID(String s, StringBuffer sb): toAll(
        AsSepStringIDV);

  // getIndex
  declare traversal: public int getIndex(FormalParameter fp) :toAll(GetIndexFPV);
  declare traversal: public int getIndex(Type t) :toAll(GetIndexTV);
  declare traversal: public int getIndex(VariableDeclaratorId id) :toAll(GetIndexIDV);

  // selectIndex
  declare traversal: public String selectedIndexesFP(java.util.List indxs, String s, StringBuffer
        sb) :toAll(SelectedIndexFPV);
  declare traversal: public String selectedIndexesT(java.util.List indxs, String s, StringBuffer
        sb) :toAll(SelectedIndexTV);
  declare traversal: public String selectedIndexesID(java.util.List indxs, String s, StringBuffer
        sb) :toAll(SelectedIndexIDV);

  }
```

Figure 5.27: Using a general strategy.

```
abstract class CountV {

  protected int count;

  public int return() {
    return this.count;
  }
}
```

```
class CountFormalParameterV extends
    CountV {

  public CountFormalParameterV(int count){
    this.count = count;
  }

  public void before(FormalParameter host){
    this.count++;
  }
}
```

```
class CountTypeV extends CountV {
  public CountTypeV(int count) {
    this.count = count;
  }

  public void before(Type host){
    this.count++;
  }
}
```

```
class CountIDV extends CountV {
  public CountIDV(int count){
    this.count = count;
  }

  public void before(VariableDeclaratorId
      host){
    this.count++;
  }
}
```

Figure 5.28: Count Visitors.

this fact and defines two before methods. The visitor is used for both the `countFP` traversal specification and the `countID` traversal specification. The visitors for the remaining operations follow a similar pattern.

Comparing the two approaches (Figure 5.27 and Figure 5.29) we can observe that we increase strategy reuse with a general strategy. A general strategy however requires that each visitor used is specifically designed for each traversal declaration. It is more difficult to use visitors that can perform multiple tasks (*i.e.*, `CountTypeOrIDV` in Figure 5.30) with general strategies. More specialized visitors decreases reuse leading to multiple similar visitor implementations. [9] The ability to define visitors that can contain before/after methods on any type in the class graph makes visitor implementations brittle. Evolutions/modifications to AST can easily yield

[9]If DAJ supported generics the number of visitors could be significantly reduced (Section 5.2.1).

```
aspect FormalParametersSpecific {
  declare strategy: toAll : from FormalParameterList to ∗.
  declare strategy: toFP : from FormalParameterList to FormalParameter.
  declare strategy: toType : from FormalParameterList to Type.
  declare strategy: toID : from FormalParameterList to VariableDeclaratorId.


  // traversal specifications
  // counters
  declare traversal: public int countFP(): toFP(CountFormalParameterV);
  declare traversal: public int countTypes(): toType(CountTypeOrIDV);
  declare traversal: public int countNames(): toID(CountTypeOrIDV);

  declare traversal: public String asString(StringBuffer sb):toAll(AsStringFormalParamsV);

  // atIndex
  declare traversal: public FormalParameter atIndex(int i): toFP(AtIndexFPV);
  declare traversal: public Type atIndex(int i): toType(AtIndexTV);
  declare traversal: public VariableDeclaratorId atIndex(int i): toID(AtIndexIDV);

  // asSepString
  declare traversal: public String asSepStringFP(String s, StringBuffer sb): toFP(
       AsSepStringFPV);
  declare traversal: public String asSepStringT(String s, StringBuffer sb): toType(
       AsSepStringTOrIDV);
  declare traversal: public String asSepStringID(String s, StringBuffer sb): toID(
       AsSepStringTOrIDV);

  // getIndex
  declare traversal: public int getIndex(FormalParameter fp): toFP(GetIndexFPV);
  declare traversal: public int getIndex(Type t):toType(GetIndexTOrIDV);
  declare traversal: public int getIndex(VariableDeclaratorId id):toID(GetIndexTOrIDV);

  // selectIndex
  declare traversal: public String selectedIndexesFP(java.util.List indxs, String s, StringBuffer
       sb) :toFP(SelectedIndexFPV);
  declare traversal: public String selectedIndexesT(java.util.List indxs, String s, StringBuffer
       sb) :toType(SelectedIndexTOrIDV);
  declare traversal: public String selectedIndexesID(java.util.List indxs, String s, StringBuffer
       sb) :toID(SelectedIndexTOrIDV);

  }
```

Figure 5.29: Using specific strategies.

```
class CountTypeOrIDV extends CountV {
  public CountTypeOrIDV(int count) {
    this.count = count;
  }

  public void before (VariableDeclaratorId host){
    this.count++;
  }

  public void before(Type host){
    this.count++;
  }
}
```

Figure 5.30: Visitor for counting `Type` and `VariableDeclaratorId` objects.

valid adaptive programs with unexpected behavior.

On the other hand, specific strategies make their intend explicit and for traversal declarations that define generic behavior (*i.e.*, `count`) we can implement more reusable visitors. Visitor reuse in this case depends on the set of types operated on during traversal; operations that depends on non-overlapping types during traversal can be grouped together into one visitor. The implementation of CONAJ uses both general and specific strategies.

As part of the rewrite operation both systems build internal data structures to keep information about classes/interfaces along with their fields and supertypes, and methods along with their types and contracts. Figure 5.31 shows the DI used to obtain information about methods. We maintain a map from class name to a list of method records one for each method defined inside the class. A method record contains the method's name, the method's signature (the list of formal argument types and the return type), the list of formal argument names and the method's pre and post-condition.

The ICG in `Methods` captures methods declarations, but only the parts that we are interested in; method name, return type, formal arguments and any pre or post-condition definitions. The DI defines strategies and adaptive methods that retrieve the necessary information to create a method

```
import java.util.List;

di Methods {

  //ICG
  MethodDef = ReturnType MName LFArgs Pre Post.
  ReturnType : Type | Void.
  MName = Ident.
  Pre = .
  Post = .
  Type = Ident.
  Void = .
  LFArgs : LFArgsMt | LFArgsCons.
  LFArgsMt = .
  LFArgsCons = FArg LFArgs.
  FArg = Type Name.
  Name = Ident.

  // strategy specifications
  declare strategy: toMName : from MethodDef via MName to Ident.
  declare strategy: toPre : from MethodDef to Pre.
  declare strategy: toPost : from MethodDef to Post.
  declare strategy: toRType : from MethodDef via ReturnType to {Ident,Void}.
  declare strategy: toFArgTypes : from MethodDef via FArg via Type to Ident.
  declare strategy: toFArgNames : from MethodDef via FArg via Name to Ident.

  // traversal specifications
  @constraint{unique(toMName)}
  declare traversal: public String mnameAsString() : toMName(IdentV);
  @constraint{unique(toRType)}
  declare traversal: public String rtypeAsString() : toRType(RTypeV);
  @constraint{unique(from FArg to Type)}
  declare traversal: public List fargsTypesAsList() : toFArgTypes(IdentListV);
  @constraint{unique(from FArg to Name)}
  declare traversal: public List fargsNamesAsList() : toFArgNames(IdentListV);
  declare traversal: public Pre getPre() : toPre(PreV);
  declare traversal: public Post getPost() : toPost(PostV);
  declare traversal: public MakeMTableRecordV.MTableRecord buildMTableRecord() :
      toMName(MakeMTableRecordV);
}
```

Figure 5.31: The Demeter Interface for Java method definitions with contracts.

```
class PreV {
  private Pre p;

  PreV() { p = new Pre(); }
  public void before (Pre host) { this.p = host;}
  public Pre return() { return this.p; }
}
```

```
class PostV {
  private Post p;

  PostV() { p = new Post(); }
  public void before (Post host) { this.p = host;}
  public Post return() { return this.p; }
}
```

Figure 5.32: Visitors for retrieving pre and post-condition definitions from a method's definition.

record. Figures 5.32 and 5.33 contain the definitions for the visitors that retrieve pre and post-conditions, a single identifier (used to retrieve a method's name), a special visitor to retrieve a method's return type (including `void`), and a list of identifiers (used to retrieve formal argument names or types).

The last adaptive method relies on the previous adaptive methods to build an instance of `MTableRecord`. `MTableRecord` is defined as an inner class of the `MakeMTableRecordV` visitor (Figure 5.36). In `MakeMTableRecordV` we use inner classes to define helper classes that we want to use *and* return as results. These classes are not part of the ICG and therefore do not need to be mapped. Members of `MTableRecord` however can be ICG classes, *e.g.*, `Pre` and `Post` appear as members to `MTableRecord`. By defining `MTableRecord` as an inner class of `MakeMTableRecordV` the contents of `MTableRecord` is also rewritten for class names that are in the ICG. Uses of `Pre` and `Post` inside `MTableRecord` get rewritten to their mapped classes.

The relevant parts of the class dictionary for the `Methods` DI are given in Figure 5.34 and the mapping for the `Methods` DI is given in Figure 5.35. Even though the `Methods` DI is mapped once it is a good example that shows how DIs can be used to provide a smaller interface to the class dictionary.

```
import edu.neu.ccs.demeter.∗;
class IdentV {
  protected StringBuffer sb;

  IdentV() { this.sb = new StringBuffer(); }
  public void before(Ident host) { sb = sb.append(host); }
  public String return() { return this.sb.toString(); }
}
```

```
import edu.neu.ccs.demeter.∗;
class RTypeV extends IndentV {
  RTypeV() { super(); }
  public void before(Void host) { sb = sb.append("void"); }
}
```

```
import java.util.List;
import java.util.ArrayList;
import edu.neu.ccs.demeter.∗;

class IdentListV {
  List l;
  IdentListV() { this.l = new ArrayList(); }
  public void before(Ident host){ l.add(host.toString()); }
  public List return() { return l; }
}
```

Figure 5.33: Visitors for retrieving an identifier and a list of identifiers. `RTypeV` retrieves a method's return type as a string and extends `IdentV` to deal with `void`.

The implementation of DCONAJ uses some other smaller DIs for smaller tasks,

- the `HasA` DI is used to check for the existence of a `old` or `result` inside post-condition definitions,
- the `HashMap` creates a map between a key and a list of values. `HashMap` is used to build a map of class names to a list of their supertypes and a map of class names to their list of fields.
- the `Getter` DI is used to generate getter methods that need to traverse to reach their targets.

In the case of DCONAJ DIs help to abstract certain operations on the

```
cd Conaj
{
 //elided parts ...
 //related cd definitions with parsing directives removed
 MethodDeclaration = MethodModifiers MethodSignature AnyBlock .
 MethodSignature = ResultType MethodDeclarator [ "throws" NameList ] .
 AnyBlock  : A_Block | A_SemiColon .
 A_Block = BlockHead BlockStatements BlockTail.
 BlockHead = OpenBrace [PreCondition] [ PostCondition] .
 BlockTail = [PreCondition] [PostCondition] CloseBrace .
 A_SemiColon  = SemiColon [PreCondition] [PostCondition] .
 MethodDeclarator = MethodName FormalParameterList .
 MethodName = Identifier.
 FormalParameterList : FormalParameterMt | FormalParameterCons.
 FormalParameterMt = .
 FormalParameterCons = FormalParameter "," FormalParameterList.
 FormalParameter  = [ CFinal ] Type VariableDeclaratorId .
 VariableDeclaratorId = Identifier.
 ResultType  : Void | Type .
 Void = "void".
 Type = Identifier.
 //elided parts ...
 //mappings are given in Figure 5.35
}
```

Figure 5.34: The relevant parts of the class dictionary for the `Methods` DI. The mapping definition is given in Figure 5.35

class dictionary. The abstraction provided by DIs allows for more code reuse than in the case of CONAJ. Also, the ability to carve out the subparts of the class dictionary allows for smaller, cleaner interfaces that make adaptive code development easier.

5.2.1 Java Generics and DAJ

DAJ does not support generics. Figure 5.27 and Figure 5.29 are prime examples where generics can be used to abstract over similar visitor implementations. For example, we could reimplement the counting visitors in Figure 5.28 into one generic visitor (Figure 5.37).

The use of generics significantly reduces the number of visitor definitions needed in order to achieve the same behavior as `List` DI without Demeter Interfaces.

```
for Methods{
  use (→, MethodDef, ReturnType)
    as MethodDeclaration → ResultType.
  use (→, MethodDef, MName)
    as MethodDeclaration → MethodName.
  use (→, MName, Ident)
    as MethodName → Identifier.
  use (→, MethodDef, LFArgs)
    as MethodDeclaration → FormalParameterList.
  use (→, MethodDef, Pre)
    as MethodDeclaration → PreCondition.
  use (→, MethodDef, Post)
    as MethodDeclaration → PostCondition.
  use (=>, ReturnType, Void)
    as ResultType → Void.
  use (=>, ReturnType, Type)
    as ResultType → Type.
  use (→, Type, Ident)
    as Type → Identifier.
  use (=>, LFArgs, LFArgsMt)
    as FormalParameterList → FormalParameterMt.
  use (=>, LFArgs, LFArgsCons)
    as FormalParameterList → FormalParameterCons.
  use (→, LFArgsCons, FArg)
    as FormalParameterCons → FormalParameter.
  use (→, LFArgsCons, LFArgs)
    as FormalParameterCons → FormalParameterList.
  use (→, FArg, Name)
    as FormalParameter → VariableDeclaratorId.
  use (→, FArg, Type)
    as FormalParameter → Type.
  use (→, Name, Ident)
    as VariableDeclaratorId → Identifier. }
```

Figure 5.35: Mappings for the `Methods` DI.

Generics can also be used in the implementation of DIs. Demeter interfaces are implemented as a rewrite that that generates specialized visitor implementations. For each mapping provided to an ICG node that also appears in a visitor implementation our generation creates a new visitor where all occurrences of the ICG node are replaced with its mapped name. We can skip this generation step if we define generic visitors using the ICG node(s) as type parameter(s). Figure 5.38 gives the implementation of the same visitors as Figure 5.25.

```
import java.util.List;

class MakeMTableRecordV {
  private MTableRecord res;
  MakeMTableRecordV() { this.res = new MTableRecord(); }

  public void before(MethodDef host) {
    this.res.setMname(host.mnameAsString());
    this.res.setMsig(
        new MType(host.rtypeAsString(),host.fargsTypesAsList()));
    this.res.setFargsNames(host.fargsNamesAsList());
    this.res.setPre(host.getPre());
    this.res.setPost(host.getPost());
  }
  public MTableRecord return() { return this.res; }

  public static class MTableRecord {
    private String mname;
    private MType msig;
    private List fargNames;
    private Pre pre;
    private Post post;
    // constructors setter and getter methods omitted
    public boolean equals(MTableRecord mtRec){
      return this.mname.equals(mtRec.getMname()) &&
        this.msig.equals(mtRec.getMsig());
    }
  }

  public static class MType {
    private String rtype;
    private List fargTypes;
    // constuctors setters and getters omitted.
    public boolean equals(MType mt){
      return this.rtype.equals(mt.getRtype()) &&
        this.fargTypes.equals(mt.getFargsTypes());
    }
  }
}
```

Figure 5.36: `MakeMTableRecordV` relies on the other adaptive methods defined in `Methods` DI and builds a record that contains the method's name, the method's signature with the method's return type and the types of all its arguments, a list of the methods formal argument names and any pre and post-conditions.

```
class CountV<X> {
  protected int count;

  public CountV(int count){
    this.count = count;
  }

  public void before(X host){
    count++;
  }

  public int return(){
    return this.count;
  }
}
```

Figure 5.37: Generic count visitor.

5.2.2 Comparing DCONAJ with CONAJ

DCONAJ has a smaller code base than CONAJ (approximately 400 lines of code less). We have also compared parts of the DCONAJ implementation (DI implementations) with corresponding code in DAJ using Java generics. The Demeter Interface implementation tends to contain more lines of code than its corresponding DAJ implementation with generics. For Demeter Interfaces with a small ICG and multiple mappings in the class graph, the code size of the DI implementation was comparable to the DAJ implementation with generics (*e.g.*, `List` DI implementation is 123 lines of code, and the corresponding implementation in DAJ with generics is 112 lines of code). However in the cases where the Demeter interface contains a large ICG and is only mapped once (*e.g.*, `Methods` in Figure 5.31, DCONAJ implementation is 329 lines of code, DAJ implementation with generics is 294 lines of code), the DAJ implementation is smaller. The DI implementation contains extra lines compared to the DAJ implementation with generics for the ICG, and the mapping(s).

Demeter interfaces provide a clear separation during development of adaptive code and the class dictionary. This separation makes development

```
class AsSepStringV<E> extends AsStringV {
  protected String sep;
  AsSepStringV(String sep, StringBuffer sb){
    super(sb);
    this.sep = sep;
  }
  public void before(E host){ this.sb = sb.append(host.toString()).append(sep); }
  public String return(){ return this.sb.reverse().delete(0,2).reverse().toString(); }
}
```

```
class GetIndexV<E>{
  private E e;
  private int count;
  private boolean found;
  private int res;

  GetIndexV(E e){
    this.e = e;
    this.count = 0;
    this.res = 0;
    this.found = false;
  }
  public void before (E host) {
    if (host.equals(e)) { this.found = true; }
    this.count++;
  }
  public int return(){
    if (found) { return res;}
    else {
      throw new RuntimeException("Element "+ e.toString() + " not found");
    }
  }
}
```

```
import java.util.List;
class SelectedIndexV<E> extends AsSepStringV<E> {
  private int count;
  private List indxs;
  private String s;

  SelectedIndexV(List indxs, String s, StringBuffer sb){
    super(" , ", sb);
    this.count = 0;
    this.s = s;
    this.indxs = indxs;
  }
  public void before (E host) {
    if (indxs.contains(this.count)) { super.before(host); }
    else { this.sb = sb.append(s).append(this.sep);}
    this.count++;
  }
}
```

Figure 5.38: The three extra visitors used inside `List` DI with generics.

easier as we only have to consider ICG(s) and their behavior. Furthermore, Demeter interfaces make testing of adaptive code easier. Taking the abstract names in the ICG DAJ can generate stub classes and independently test adaptive code before we instantiate the DI on a concrete class graph.

Also, Demeter interfaces minimize dependencies between visitors and the class graph. Even in situations were a Demeter interface uses a general strategy (*e.g.*, `toAll` from Figure 5.24) the mapping restricts the applicability of adaptive code to specific parts of the class graph. Consider for example the mapping of `List` (Figure 5.26) and the implementations without DIs in Figures 5.27 and 5.29. The `toAll` strategy, common to all three implementations, in Figures 5.27 and 5.29 allows for a visitor implementation to define before or after methods on *any* of the classes reachable from `FormalParameterList`. The DI implementation, for the same strategy allows visitor implementations to define before and after advice only on subgraphs rooted at `FormalParameterList`. For example the first mapping in Figure 5.26 none of the methods prefixed with `fargs_` can operate on `Type` or `VariableDeclaratorId` through a before or after method defined in their visitors.

The use of Demeter interfaces admits some limitations. Demeter interfaces and their mappings impose an extra redirection that programmers need to keep in mind. This becomes important when behavior added through one DI is being used by another DI. Programmers need to read and understand the mappings in order to ensure which adaptive methods are available in a class definition. Also, a Demeter interface that contains a large ICG introduces more complexity since mapping the Demeter interface requires that programmers map each edge in the ICG.

CHAPTER 6

Related Work

This chapter presents a brief survey on research areas and tools that share similarities with Adaptive programming. The chapter covers Adaptive programming tools (Demeter Tools), Strategic Programming, Scrap your Boilerplate, Haskell Type classes and views. We conclude this chapter with a section on related work in the area of XML processing and related technologies, in terms of their features, to AP.

6.1 The Demeter Tools

The set of Demeter Tools, DemeterJ, DJ, and DAJ, are incarnations of AP tools that rely on different techniques and technologies but share the same algorithm and semantics. DemeterJ and DAJ are generative in nature, DJ is a pure Java library that employs Java's reflection API. DemeterJ takes as input

1. a class dictionary; a textual representation of a class graph that also serves as the grammar definition for an input language, and,
2. a set of behavior files; each behavior file defines methods (adaptive or simple Java methods) grouped by class name.

DemeterJ generates a Java program that includes a lexer and parser for the input language, a set of classes that mimic the class graph defined in the class dictionary and injects methods defined in behavior files into each class. Adaptive method definitions are expanded into a series of Java methods.

DAJ follows a similar setup as DemeterJ but can also obtain the class graph from a given set of Java source code. DAJ optionally accepts a class dictionary as input and can generate a parser and lexer for the input language defined in the class dictionary. DAJ takes a set of traversal files (§ 2) and generates a mixture of Java and AspectJ source code. DAJ uses Java to generate class definitions and AspectJ aspects, inter-type and advice definitions to implement the behavior of adaptive methods.

DJ is a pure Java library that relies on Java's reflection API in order to infer the program's class graph and to perform traversals. Adaptive methods in DJ reflectively traverse an object's structure and dispatch to visitor before and after methods. All three Demeter Tools rely on the AP library to calculate valid paths in a class graph given a strategy specification. We focus on the AP library and specifically on the semantics of strategy definitions and the computation of the traversal graph.

6.1.1 Strategies

All Demeter tools use *general strategies* and share the same semantics for strategy specifications. The semantics for general strategies define a path expansion of a strategy edge $A \rightarrow B$ over a class graph $\mathcal{CG}$ as all paths from A to B in $\mathcal{CG}$. There is not restriction on the nodes that appear in the expansion. The notion of *valid* path under general strategies is similar to the notion of a valid path for WYSIWYG strategies; given a strategy $\mathcal{SG}$ and a class graph $\mathcal{CG}$ then for a path in q in $\mathcal{SG}$, a path p in $\mathcal{CG}$ is valid path if and only if $expansion(p, q, \mathcal{CG}.nodes)$.

Even though the semantics between the general strategies and WYSIWYG strategies differ, WYSIWYG strategies are a subset of the AP library semantics. Given strategy $\mathcal{SG}$ and a class graph $\mathcal{CG}$ such that under WYSIWYG semantics the strategy selects a path set P in $\mathcal{CG}$, we can systematically rewrite $\mathcal{SG}$ to $\mathcal{SG}'$ such that $\mathcal{SG}'$ and $\mathcal{CG}$ under general strategy semantics the strategy $\mathcal{SG}'$ selects the same path set P in $\mathcal{CG}$.

Informally the transformation of $\mathcal{SG}$ takes each edge (A, B) in $\mathcal{SG}$ and generates a new strategy for the edge bypassing all nodes in $\mathcal{SG}$ except B and all outgoing edges of B. We construct the complete, rewritten strategy $\mathcal{SG}'$ by taking the union of all the generated strategies, *e.g.*, using the edge notation of DemeterJ, given strategy $A \to B \to C$ we obtain

$$
\begin{array}{ll}
\{\ \mathsf{A} \to \mathsf{B}\ \textbf{bypassing}\ \{ & \to\mathsf{B},*,*, \\
 & \Rightarrow\mathsf{B},*, \\
 & \mathcal{SG}.nodes \setminus \{B\}\ \} \\
\ \ \mathsf{B} \to \mathsf{C}\ \textbf{bypassing}\ \{ & \to\mathsf{C},*,*, \\
 & \Rightarrow\mathsf{C},*, \\
 & \mathcal{SG}.nodes \setminus \{\mathsf{C}\}\}\ \}
\end{array}
$$

The directive $\to B, *, *$ denotes any field edge with source B that can have any label and any target node; $\Rightarrow B, *$ denotes any inheritance edge with source B and any target node. We use $\mathcal{SG}.nodes \setminus \{C\}$ to denote the set of all strategy nodes except C. The rewrite enforces the WYSIWYG condition that no strategy node appears in the expansion of a strategy edge. The rewrite generates longer strategies; the length of the generated strategy is bounded by $|\mathcal{SG}.nodes|$.

The reverse argument cannot be made for WYSIWYG strategies since our definition of strategies does not allow for parallel edges or optional edges in a strategy graph. Given a strategy graph $\mathcal{SG}$ and a class graph $\mathcal{CG}$ such that $\mathcal{SG}$ under the general strategy semantics selects a path set P in class graph $\mathcal{CG}$ we cannot always generate a new strategy $\mathcal{SG}'$ such that the path set selected by $\mathcal{SG}'$ under WYSIWYG semantics in $\mathcal{CG}$ is P.

6.1.2 Traversal Graph

The notion of a traversal graph was first introduced in [28]. The authors in [28] provide an algorithm that given a class graph and a strategy graph returns the traversal graph. The algorithm is a variation on the algorithm for calculating the cross product of the two automata, the class graph and the strategy graph. The algorithm in Chapter 4 is inspired by [28].

Informally the algorithm in [28] given a strategy graph $\mathcal{SG} = (V, E)$ and a class graph $\mathcal{CG} = (V', E')$ where $\mid E \mid = k$ creates k copies of the class graph. For each edge $e_i \in E$ the algorithm prunes the ith class graph copy $\mathcal{CG}_i$ retaining only nodes and edges that lead from e_i*.source* to e_i*.target*. The algorithm then introduces *intercopy edges* between class graph copies for all strategy nodes. The source node of these intercopy edges is then changed to point to the incoming nodes of v in each class graph copy. The intercopy edge is annotated with the same label as the label on the incoming edge.

The traversal graph is used to guide traversal of object graphs at runtime in the same manner as in Chapter 3, and for generating code that responsible for performing traversals on object graphs. The construction of the traversal graph, however, yields a graph that is non-deterministic in nature. A second algorithm is defined in [28] to generate methods responsible for performing object graph traversals. The generation algorithm maintains a set of tokens during traversal in order to keep track of the legal traversal possibilities. The algorithm is similar to the NFA simulation technique from [7].

Our algorithm for calculating and generating code given in Chapter 4 is inspired by the algorithm given in [28]. The use of WYSIWYG strategies ensures that every time we reach a strategy node then this node is not part of an expansion. If we reach a strategy node in the traversal graph then we know for sure that we have reached a milestone that is part of the strategy. Code generation becomes simpler and there is no longer the need for tokens to keep track of traversal possibilities.

Our new algorithm *can* generate static traversal code that does not rely on runtime information resulting in faster traversal execution. However, in the case where both algorithm's implement traversals by generating methods, our new algorithm generates more method definitions and thus larger programs.

The idea of abstracting over a class graph in adaptive programs using an interface class graph was first introduced with adaptive plug-and-play-components (APPCs) [33]. The mapping of interface class graphs to class dictionaries allowed for the mapping of a class name to a class name and for an edge to an edge or strategy. APPCs have no provision for further constraints on interface class graphs or on concrete class dictionaries. Furthermore, APPCs rely on the AP library and share the same issues with all of the Demeter Tools.

The JAsCo system [40] is a dynamic aspect oriented (AOP) approach that aims to combine the ideas of AOP and component based software development. JAsCo extends the notion of a Java bean and defines aspect beans and connectors. An aspect bean contains Java methods and *hooks*. Hooks define pointcuts–points in the program's execution during which the aspect bean intervenes–and code to execute when a step in the program's execution satisfies a given pointcut. An aspect bean can contain abstract pointcuts that can be extended by other aspect beans. Furthermore, aspect beans can define a special method called `isApplicable` that serves as an extra triggering condition for a hook that decides at runtime whether the pointcut is valid or not. Aspect beans need to be instantiated before they can be used. JAsCo provides *connectors* to instantiate an aspect bean.

An extension to the JAsCo [45] system accommodates for AP by allowing pointcuts to be given by strategy definitions and aspect beans to act as visitors. JAsCo's extension introduces a new connector for aspect beans that contain strategy specifications. The AP extension to JAsCo borrows strategy specifications and the AP library implementation as is and thus inherits non-WYSIWYG strategies. The implementation of AP ideas in JAsCo share a lot with DJ where the class graph and the traversal graph are computed at runtime. Aspect beans used as visitors can advice any object encountered during traversal.

The AP extension to JAsCo does not alter the meaning of strategies and does not enforce any interface between the strategy and its attached aspect beans. Modifications to the class graph can lead to correct programs with unexpected behavior as in the case of DAJ (§ 2). Furthermore, errors in strategy specifications are discovered at runtime and there is no mechanism that parallels constraints that allows programmer's to state and statically verify path properties. One could however encode path properties at runtime by using `isApplicable` methods, however this is cumbersome to encode and incurs a runtime overhead.

6.2 Strategic programming

Strategic Programming (SP) [21] is a generic programming idiom for processing compound data such as graphs, trees etc. Even though SP has its roots in term rewriting it has been successfully applied to other programming paradigms such as functional programming [20] and object oriented programming [19].

SP relies on two concepts basic data processing computations and traversal schemas. Traversals are composed by passing data processing computations as arguments to traversal schemas. Using a combinator style, traversal schemas are defined from a set of primitive traversal schemas. For example, any SP incarnation should provide the following strategy combinators

- `id` - the identity strategy combinator applied to a datum simply returns that datum.

- `fail` - the fail strategy combinator fails and returns immediately.
- `seq(s,s')` - the sequence combinator takes two strategy combinators as arguments and applies first `s` and then `s'`.
- `choice(s,s')` - the choice combinator attempts `s` first and returns its result on success, on failure it applies `s'`.
- `adhoc(s,a)` - the adhoc combinator consumes a strategy combinator and an action (a computation that deals with data of a specific type *t*). If the datum has the same type as the one dealt by `a` then apply `a` to the datum else proceed with `s`.
- `all(s)` - process all immediate subcomponents of a composite datum.
- `one(s)` - process one of the immediate subcomponents of a composite datum. If all subcomponents fail then return failure, else select one that succeeds.

This set of combinators is enough to define an AP specific language [25]. The OO incarnation of SP [19] generalizes visitor [15] objects to model strategies. Combinators are encoded as a type hierarchy rooted at a generalized `Visitor` type with a `visit` method. To enable generic access to immediate sub-objects data structures traversed must support a `Visitable` interface. Failure is encoded through a specialized Java exception. The double dispatch protocol of ordinary visitors is complemented by a visitor combinator that forwards any class specific visit method to a generic visit method. The combinator style of first class visitors relies on parametrized constructors for visitors. JJForester is a generative tool that introduces the `Visitable` interface, the generic `Visitor` and the forwarding visitor combinator. Figure 6.1 shows the implementation of a topdown traversal over a binary tree with types `Node` and `Leaf`.

The object oriented incarnation of SP provides programmable, reusable, generic traversal schemas that allow the encoding a variety of traversal

```
class Sequence implements Visitor {
  Visitor first;
  Visitor then;

  public Sequence(Visitor first, Visitor then){
    this.first = first;
    this.then = then;
  }

  public void visit_Node(Node node){
    node.accept(first);
    node.accept(then);
  }
  public void visit_leaf(Leaf leaf){
    leaf.accept(first);
    leaf.accept(then);
  }
}
```

```
class All implements Visitor {
  Visitor v;

  public All(Visitor v){
    this.v = v;
  }

  public void visit_Leag(Leaf leaf) { }
  public void visit_Node(Node node){
    node.left.accept(v);
    node.right.accept(v);
  }
}
```

```
class TopDown implements Sequence {
  public TopDown(Visitor v){
    super(v,null);
    then = new All(this);
  }
}
```

Figure 6.1: An SP implementation for a top down traversal over a binary tree

variations. AP on the other hand does not allow the same flexibility on traversal specification. The definition of where to go and what to do in SP is encapsulated inside visitors and relies on object composition blurring the distinction of traversal code and behavior. SP's approach to traversal generation is dynamic; there is no notion of static verification between the traversal and the data to be traversed. While AP and WYSIWYG strategies use static meta-information that make it easier to optimize traversal code, SP takes a more dynamic approach making difficult to optimize traversal code.

6.3 Scrap your boilerplate

The Scrap your Boilerplate (SyB) series of papers [22, 23, 24] present a lightweight approach to generic programming in Haskell. The goal of SyB is to write code that traverses data structures and performs operations on them. Functions take a traversal combinator which specifies which nodes in the data structure it should apply its argument to. The traversal combinator's argument is itself a function, a type-extender, that takes as argument the function to be called at each node. The type extender function behaves as its argument function when applied to nodes of interest and as the identity function when applied to uninteresting nodes. Interesting nodes are the nodes whose types match the argument type of the function given to the type-extender. There is a direct correspondence between SyB code and AP. Traversal combinators correspond to strategies, the application of a type-extender to a function corresponds to adaptive methods. Visitors in AP define the interesting nodes in a traversal.

Unlike AP, SyB inspects the data structure being traversed at runtinme and explores each immediate child of a recursive data structure. AP relies on meta information and optimizes traversals by pruning the search space and ignoring parts of the data structure for which there is no chance of reaching a target object. SyB traverses all data structure parts, the type-extender is responsible for performing computation at interesting parts of the data structure. The traversal combinator allows for different traversal scheme much like in SP, while AP does not provide the ability to alter the traversal scheme used to walk a data structure.

6.4 Haskell Type classes and Views

The Haskell programming language [4] provides type system constructs that support ad-hoc polymorphism called *type classes* [48]. Type classes provide a principled way to qualify polymorphic types by defining a type class and specifying which types are members or *instances* of it.

For example, one of Haskell's predefined type class, `Eq`, defines operations related to equality that members of the type class should implement.

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

The declaration of Eq can be read as, a type a is an instance of the class Eq only if there are operations == and /= that consume two arguments of type a and return Bool. The last two lines of the declaration are default implementations for the two operations.

We can specify which types are instances of a class through instance declarations. For example, using the primitive Haskell function IntegerEq, Haskell defines Integer as an instance of Eq with the following instance declaration

```
instance Eq Integer where
  x == y = IntegerEq x y
```

Haskell also supports a notion of *inheritance* between type classes. Inheritance between type classes allows one type class to "inherit" operations from its super class. Haskell also supports multiple inheritance allowing for a type class to inherit operations from more than one super class. The notion of inheritance allows for better decomposition and reuse of operations between type classes.

Instance declarations in Haskell can also contain constraints on the values manipulated by operations. A typical example is the definition of binary trees and equality between binary trees.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)


instance Eq a => Eq (Tree a) where
  Leaf a == Leaf b               = a == b
  Branch l1 r1 == Branch l2 r2   = l1 == l2 && r1 == r2
  _ == _                         = false
```

The instance declaration says that we can compare trees of a's if we know how to compare a's for equality.

The ability of DIs to define a set of operations (adaptive methods) that we can then introduce (instantiate) onto a concrete class graph, provides a similar notion of ad-hoc polymorphism as type classes. The set of adaptive methods defined inside a DI can be viewed as the operations defined in a type class declaration. The mapping of DIs is analogous to the instantiation declaration in Haskell in that it is the step that provides the final concrete implementation of the operation/method.

There are however difference, in DIs the implementation of the operation, *i.e.*, the strategy and Visitor implementation, is given at the same abstraction level as the ICG. The mapping of the DI "customizes" (makes concrete) the implementation to the given class graph. In the case of type classes, each type provides its own implementation for the operations defined in a type class.

Type classes in Haskell are part of the type system and extend the Hindley/Milner polymorphic type system while DIs have no support for type inference. There is no notion of inheritance, similar to the notion of inheritance for type classes, or any other means of sharing between DIs and this limits their reuse and the way we decompose adaptive code into DIs.

Even though DIs and type classes try to address ad-hoc polymorphism under two different paradigms with analogous functionality, type classes are better embedded into Haskell's type system and provide better support for reuse.

6.4.1 Views

Views [47] provide a mechanism that allows any type to be viewed as a free data type and be visible. Views were developed to resolve the mismatch between data abstraction and pattern matching in languages like Haskell. Consider for example the definition of exponentiation over a Peano representation of numbers.

```
data Peano ::= Zero | Succ Peano


power x Zero = 1
power x (Succ n) = x ∗ power x n
```

The definition of `power` uses pattern matching (left hand side) on the type `Peano`. It is easy to check that the definition of `power` deals with both cases for `Peano` numbers and that no case was omitted; a check that can (and is made) by compilers in languages like Haskell and ML. The definition however has defined the representation for numbers (at least the number used for the exponent) as the free data type `Peano` even though other representation might be preferred (*e.g.*, for performance reasons). Ideally we would like to abstract away the representation details, so that we could use any representation of numbers that we see fit, while keeping the definition of `power` the same.

Views are a mechanism that allows programs to "view" an arbitrary data type as a free data type, so for example we could view the build-in integer type as if it were of type `Peano`, thus allowing definitions like `power` while the underlying implementation manipulates numbers using the built-in integer type. The following view definition allows us to view built-in `int`s as `Peano`.

```
view int :: = Zero | Succ int
  in n = Zero, if n = 0
       = Succ ( n  1), if n > 0
  out Zero = 0
  out Succ n = n + 1
```

The view introduces the names `Zero` and `Succ` that can be used for pattern matching (on the left hand side of definitions) or in terms (on the right hand side of definitions). The `in` and `out` clauses are similar to function definitions and are used to map values from the `Peano` type to `int` and from `int` to `Peano`. These functions are used to map values from one type to the other inside definitions (like `power`). A view is well defined only if the functions defined by the `in` and `out` are inverses.

Another example taken from [47] deals with lists and how we can view lists backwards.

```
data List a ::= Nil | a Cons (List a)

view List a ::= Nil | (List a) Snoc a
 in (x Cons Nil) = (Nil Snoc x)
 in (x Cons (xs Snoc x′)) = (x Cons xs) Snoc x′
 out (Nil Snoc x) = x Cons Nil
 out ((x Cons xs) Snoc x′) = x Cons (xs Snoc x′)
```

The `List` view allows us to view `(1 Cons (2 Cons Nil))` as `((Nil Snoc 1) Snoc 2)`. Note that some left hand sides in the `in` close contain `Snoc`. Matching against these `Snoc` values will cause recursive invocations of `in`. The `out` clause is recursive in a similar way.

DIs can also be used to provide different ways to view/manipulate a concrete class graph. The example of the doubly linked list (Figure 5.8, page 138) provides two views, a forward and a backward view, of a doubly linked list. Using the `FLatList` DI (Figure 5.1, page 128) we can define a new DI to view a list in reverse order. We define an abstract visitor `ReverseV` that traverses to all elements of a `FlatList` and builds the list in reverse order storing the result as a field. We then create visitors that wrap the behavior of `ReverseV` and before a visitors return's we call the original methods from `FlatList` on the reversed version of the list.

Our new DI `ReverseList` defines the same `ICG` as `FlatList`, uses one strategy, the same as `toAll` from `FlatList` and defines the same adaptive method names but uses the `toAll` strategy for all of them and new visitors. Each visitor used in `ReverseList` extends `ReverseV` and contain a return method that calls the adaptive methods from `FlatList` on the inherited field from `ReverseV` that holds the list in reverse order.

Again DIs can provide analogous functionality as views in Haskell. With a DI we can create visitors that will perform the transformation from one representation to the other and provide the necessary operations on both representations. In the case of the reverse list, the `ReverseList` DI depends on the adaptive method names given during the mapping of the `FlatList` DI since all visitors used in `ReverseList` wrap calls to adaptive methods introduced by the mapping of `FlatList`.

Unlike views, there is no direct mapping with exact inverse functions between representations. With DIs the programmer is responsible for getting the mapping correct making the process susceptible to human error.

DIs compared to type classes and views from Haskell share partial functionality from each Haskell feature. DIs provide ad-hoc polymorphism for AP programs and at the same time provide different views to concrete class graphs. Unlike type classes and views DIs do not have the same reusability as type classes nor the type safety guarantees provided by Haskell's type system and by views.

6.5 XPath

Ideas in AP can be found in other technologies where the separation between navigation code and computation is necessary. According to the abstractions that strategies allow, the problems of surprise behavior are present in these systems as well. XML [1] and XPath [2] queries are technologies widely used today that share similar issues with AP. Specifically one can think of a DTD as a class dictionary and XPath expression as strategies [31]. The problems of surprise behavior are prominent in these technologies as well since modifications to the XML document might break assumptions that the XPath query depends upon. Consider the following DTD

```
<?xml version="1.0"?>
<!ELEMENT busRoute (bus*)>
<!ELEMENT bus (person*)>
<!ELEMENT person EMPTY>
```

```
<!ATTLIST bus number CDATA #REQUIRED>
<!ATTLIST person
        name CDATA #REQUIRED
            ticketprice CDATA #IMPLIED>
```

that defines `busRoute` containing a list of `buses`. Each `bus` contains a list of `person` which in turn contains a `name` for the person and a `ticketprice` for the bus fare. Consider the following XPath query

```
var nodes=xmlDoc.selectNodes(".//person")
```

that collects all `person` elements from a `busRoute`. We can think of a simple Java script that will iterate through `nodes` and calculate the total amount of money received by the current passengers riding the bus.

Making a correspondence between a DTD to class dictionary and an XPath query to a strategy it is straightforward to create a corresponding DAJ program. In fact, the two systems are so alike in this respect that they also share the same problems when it comes to modifications of their underlying data structure.

Leaving the JavaScript code and the XPath query the same we can extend the DTD (Figure 6.2) to accommodate for villages with bus stops along the bus route. The resulting amount this time is not the total of all passengers riding the bus, but instead the total amount for all passengers, both riding and waiting at bus stops. Similar problems are found in other XML technologies that use XPath like mechanisms, such as XLinq [5] and XQuery [3], to select elements from a graph like structure.

Ideas from Demeter Interfaces can help to stop this kind of situations. Just like any XML document can define the DTD to which it confronts to, DTDs can define the XPath interfaces that they support and a mapping between the DTD's elements and the elements of the XPath interfaces that they implement. For instance, if the current total of all passengers riding the bus is to be supported by the DTD representing bus routes then it should

```
<?xml version="1.0"?>
<!ELEMENT busRoute (bus∗,village∗)>
<!ELEMENT bus (person∗)>
<!ELEMENT village (busStop∗)>
<!ELEMENT busStop (person∗)>
<!ELEMENT person EMPTY>
<!ATTLIST bus number CDATA #REQUIRED>
<!ATTLIST person
   name CDATA #REQUIRED
   ticketprice CDATA #IMPLIED>
```

Figure 6.2: Extended DTD with BusStops

make available an interface with XPath queries and constraints on these queries. The constraints are the guarantees provided to programmers by DTDs. At the same time, the mapping between the interface and the DTD itself allows for changes to the DTD to both names of entities as well as structure within the bounds of the constraints without imposing modifications to client code.

The usage of Demeter Interfaces also provides a clear distinct separation of responsibilities. In the case where an modification to the DTD breaks one of the XPath constraints then the blame lies with the DTD maintainer for breaking an interface that the DTD claims to implement.

CHAPTER 7

Conclusion and future work

In this chapter we iterate over some possible future directions that will allow for even better support for iterative software development for AP followed by our concluding remarks.

7.1 Future directions

7.1.1 Parametrized Demeter Interfaces

Even though Demeter interfaces encapsulate adaptive code and allow for separate development and testing inter dependencies between Demeter interfaces remain ad-hoc. Consider a program with two Demeter interfaces *A* and *B* where a traversal in *A* depends on an adaptive method in class *C* defined in *B*. Demeter interfaces in their current incarnation do not provide any mechanism to capture this dependency. Currently the two Demeter interfaces will appear independent making static errors detectable only after translation to DAJ.

An extension to Demeter interfaces that will allow to explicitly define dependencies between Demeter interfaces and provide static checking at the interface level will yield more modular adaptive code.

7.1.2 Tool support

The required mapping between a Demeter interface and the class graph can be complex and tedious. Tool support to assist with assigning ICG edges to class graph paths would make the process smoother. A graphical representation of both the ICG and the class graph that will allow to indicate mappings of ICG edges interactively would help programmers to visualize and automatically generate mappings for Demeter interfaces.

7.2 Conclusion

We have introduced three extensions to Adaptive programming, WYSIWYG strategies, constraints and Demeter interfaces. WYSIWYG strategies are a subset of general strategies. We have defined the semantics of WYSIWYG strategies using a simpler and more intuitive model based on automata theory and we have also proved the correctness and soundness of our new approach. We have also given a simpler code generation algorithm for WYSIWYG strategies producing programs that do not rely on runtime information for traversing object structures. The generation algorithm uses a second algorithm, based on the notion of a traversal graph, providing a second way to calculate valid paths for WYSIWYG strategies and extending previous work on traversal graphs [28].

The addition of constraints provides a mechanism with which programmers can express structural invariants on the underlying data structure. Constraints are statically checked and provide early detection of inappropriate uses of adaptive code resulting to more resilient adaptive programs.

Demeter interfaces encapsulate adaptive code by providing an interface (the ICG) between the programs OO structure and the adaptive code (strategies and visitor definitions). We have shown that the encapsulation provided by Demeter interfaces allows for separate development and provides more opportunities for reuse.

Our three extensions provide for more modular, reusable and resilient Adaptive programs that better support iterative software development.

Bibliography

[1] Extensible Markup Language (XML) 1.1 (Second Edition). http://www.w3.org/TR/2006/REC-xml11-20060816/, August 2006.

[2] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007.

[3] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, January 2007.

[4] Haskell 2010 Language Report. http://www.haskell.org/onlinereport/haskell2010/, July 2010.

[5] XLINQ Overview. http://msdn.microsoft.com/VBasic/Future/XLinq%20Overview.doc, 2010.

[6] Ahmed Abdelmeged, Therapon Skotiniotis, Panagiotis Manolios, and Karl Lieberherr. Traversal graphs: Characterization and efficient implementations. Technical report, Northeastern University, 2008.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[10] JBoss Community. Hibernate, 2010.

[11] F. Brito e Abreu, M Goualou, and R. Esteves. Toward the design quality evaluation of object-oriented systems. In *Internationl Conference on Software Quality*, 1995.

[12] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *In Proceedings of Sixteenth International Conference Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–15. ACM, 2001.

[13] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *In Proceedings of ACM Conference Foundations of Software Engineering*, pages 229–236. ACM Press, 2001.

[14] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1994.

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[17] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley Professional, 1999.

[18] Karl Liebrherr. The law of demeter LoD website. http://www.ccs.neu.edu/research/demeter/demeter-method/lawofdemeter/general-formulation.html, 2005.

[19] Tobias Kuipers and Joost Visser. Object-oriented tree traversal with JJForester. *Sci. Comput. Program.*, 47(1):59–87, 2003.

[20] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[21] Ralf Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003. Also available as arXiv technical report cs.PL/0205018.

[22] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[23] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[24] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.

[25] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic Programming Meets Adaptive Programming. In *Proc. Aspect-Oriented Software Development (AOSD'03)*, pages 168–177. ACM Press, 2003.

[26] Craig Larman. *Applying UML and Patterns*. Prentice-Hall PTR, 2004.

[27] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. *SIGPLAN Not.*, 23(11):323–334, 1988.

[28] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[29] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, web book at www.ccs.neu.edu/research/demeter.

[30] Karl J. Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[31] Karl J. Lieberherr, Jeffrey Palm, and Ravi Sundaram. "expressiveness and complexity of crosscut languages". In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, March 2005.

[32] David H. Lorenz and Therapon Skotiniotis. Contracts and aspects. Technical Report NU-CCIS-03-13, College of Computer and Information Science, Northeastern University, Boston, MA 02115, 2003.

[33] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 10, pages 97–116, Vancouver, October 1998. ACM.

[34] Doug Orleans and Karl Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[35] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag.

[36] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[37] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2004.

[38] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 196–197, New York, NY, USA, 2004. ACM Press.

[39] Therapon Skotiniotis, Jeffrey Palm, and Karl Lieberherr. Demeter Interfaces: Adaptive programming without surprises. In *European Conference on Object Oriented Programming*, 2006.

[40] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[41] The Selenium Team. Selenium - WebDriver documentation, 2010.

[42] The AspectJ Team. AspectJ Development Tools, 2005. `http://www.eclipse.org/aspectj/`.

[43] The AspectJ Team. The AspectJ Programming Guide, 2008. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[44] The Demeter Group. The DAJ website. http://www.ccs.neu.edu/research/demeter/DAJ, 2005.

[45] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in jasco. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM.

[46] W3C. Web services description language, march 2001.

[47] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM.

[48] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[49] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

[50] The DemeterJ website. http://www.ccs.neu.edu/research/demeter.