

# 4

## Program Object Model Adapters

Software transformation systems provide powerful analysis and transformation frameworks with concise languages for language processing, but instantiating them for every subject language is an arduous task, often resulting in half-completed front-ends. A lot of mature front-ends with robust parsers and type checkers exist, but few of them expose good APIs to their internal program representations. Expressing language processing problems in general purpose languages without the benefit of transformation libraries is usually tedious. Reusing these front-ends with existing transformation systems is therefore attractive. However, for this reuse to be optimal, the functional logic found in the front-end should be exposed to the transformation system – simple data serialisation of the abstract syntax tree is not enough, as this fails to expose important compiler functionality such as import graphs, symbol tables and the type checker.

This chapter introduces a novel design for a *program object model adapter* that enables program transformation systems to rewrite directly on compiler program object models such as ASTs. The design is reusable across language front-ends and also across program transformation systems based on the term rewriting paradigm. It provides an efficient and serialisation-free interface between the language-general software transformation system and the language-specific front-end infrastructure.

Chapter 10 illustrates the applicability of this design using a prototype framework based on MetaStratego and the Eclipse Compiler for Java. The prototype allows scripts written in Stratego to perform framework and library-specific analyses and transformations.

Much of the content of this chapter has been presented in the paper “*Fusing a Transformation Language with an Open Compiler*” written with Eelco Visser [KV07a].

### 4.1 Introduction

Software transformation systems are attractive candidates for implementing program analyses and transformations because their high-level domain-specific languages and

their supporting infrastructure allow precise and concise formulations of transformation problems. Unfortunately, transformation systems rarely provide robust and mature parsers and type analysers for a given subject language. Open compilers are also attractive because they provide solid parsers and type analysers, but they are mostly implemented in general-purpose languages. This means that the analyses and transformations must also be implemented in a general-purpose language without the benefit of the transformation features covered in Chapter 2. A consequence of this is that even relatively simple transformation tasks may quickly become time-consuming to implement.

The design introduced in this chapter aims to obtain the best of both worlds by combining the expressive power provided by transformation languages with the maturity and robustness of open compilers using a program object model (POM) adapter. The POM adapter welds together the transformation system runtime and the abstract syntax tree (AST) of the compiler by translating rewriting operations on-the-fly to equivalent sequences of method calls on the AST API. This obviates the need for data serialisation. The technique can be applied to most tree-like APIs and is applicable to many term-based rewriting systems. Using this adapter, transformation languages become compiler scripting languages. Their powerful features for analysis and transformation, such as pattern matching, rewrite rules, tree traversals, and reusable libraries of generic transformation functions, are offered to developers. By instantiating this design with a concrete transformation language and a concrete compiler, as is shown in Chapter 10, a powerful platform for programming domain-specific analyses and transformations is obtained. Depending on the transformation language used, the combined system can be wielded by advanced developers and framework providers because large and interesting classes of domain-specific analyses and transformations may often be expressed by reusing the libraries provided with the transformation system.

The contribution of this chapter is a general technique for fusing domain-specific languages for language processing with open compilers without resorting to data serialisation. When instantiated, this design brings the analysis and transformation capabilities of modern compiler infrastructure into the hands of advanced developers through convenient and feature-rich transformation languages. The technique can help make transformation tools and techniques practical and reusable both by compiler designers and by framework developers since it directly integrates them with stable tools such as the Java compiler. Developers can write interesting classes of analyses and transformations easily and compiler designers can experiment with prototypes of analyses and transformations before committing to a final implementation. In Chapter 10, the system's applicability is validated through a series of examples taken from mature and well-designed applications and frameworks.

The rest of this chapter is organised as follows: In Section 4.2, the design of the POM adapter is explained. Section 4.3 discusses the implementation details of the

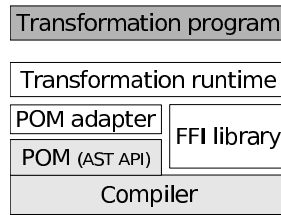


Figure 4.1: Program object model adapter architecture.

design. Section 4.4 discusses related work. Section 4.5 discusses tradeoffs related to the proposed technique. Section 4.6 summarises.

## 4.2 The Program Object Model Adapter

The program object model adapter fuses together a compiler and a software transformation language. The term *program object model* is used in this dissertation for referring to the object model representing a program inside the compiler. This is typically an AST with symbol tables and other auxiliary data structures such as import graphs. The POM adapter translates the primitive rewriting operations of the rewriting engine to function calls of the POM API.

### 4.2.1 Architecture Overview

Consider Figure 4.1 which shows the principal components of the design. There are three distinct layers in the figure, coded with different shades of grey. At the bottom, the compiler provides an API for modifying and inspecting its internal program object model. It may also provide additional functionality that should be exposed to the transformation programs such as the ability to manipulate its include paths and output directories. The language used to implement the compiler will be referred to as the *compiler language*. (The source language will, as usual, be the language of the input programs fed into the compiler.) At the top of the figure, transformation programs are written in the *transformation language*. The middle parts of the figure (white boxes) make out the runtime of the transformation language, also referred to as the transformation engine. This part is written in (potentially) another language: the implementation language of the transformation systems, referred to as the *runtime language*. This will in practise always will be the same as the compiler language.

The POM adapter design explained in this chapter is discussed using examples of ASTs implemented in a traditional object-oriented style, but this is not a requirement. Other styles can also be used, however, a large portion of modern language infrastructures are implemented in a object-oriented style. Refer to [Jon] for additional abstract

syntax tree implementation idioms. Irrespective of the implementation language, the essential requirement for the adapter to work is that there are operations on each node for obtaining its children, and, if modification is required, to construct new nodes from existing children.

In the object-oriented style, each node type in the AST, such as `CompilationUnit`, is represented by a concrete class. Children of a node can be retrieved using get-methods and replaced using set-methods. New nodes are typically constructed using factories such as the method `newCompilationUnit()` of an AST factory. Constructing nodes using a `new` operator is also supported by the adapter technique.

The runtime of the transformation system must execute on the same platform, and in the same process, as the compiler. Remote procedure calls, possibly across platforms, is an obvious and a relatively straightforward extension to POM adapter design, but its performance overhead will most likely be prohibitive. A requirement on the transformation runtime is that it has a clear interface to its term representation. Investigations of term rewriting systems suggests that this is the case for a good number of systems, including ASF+SDF [vdBvDH<sup>+</sup>01], Tom [MRV03] and Stratego [BKVV06]. Provided such a term interface, the task of the POM adapter is to translate operations on the term interface to equivalent operations on the AST API. Any data structure that can provide a suitable interface can be treated as terms and rewritten. This is done by wrapping a POM in the term interface required by the interpreter. The adapter translates term rewriting operations to POM API method calls. These are executed directly on the POM without any intermediate data serialisation.

The transformation runtime would also benefit from a facility for calling foreign functions, i.e. functions implemented outside the transformation language, but this is not strictly necessary. Most transformation systems seem to have such a facility. If a foreign function interface (FFI) facility exists, it may be used to expose native AST API functions as library functions in the transformation language. For example, type analysis, type lookup and import graph queries may be exposed to transformation programs through an FFI.

The design does not place any restrictions on the mode of operation of the transformation system. As was discussed in Chapter 2, several architectures exist for transformation systems, such as pipeline-based and incrementally updating. The POM adapter design does not dictate any one model.

### 4.2.2 Design Overview

In binding a transformation runtime to a compiler, two interfaces need to be connected: the term interface of the transformation system and the POM interface of the compiler. The algebraic concept of a signature is a very good tool for this task. Based on the signature, adapters can be generated that provide a term interface to the

POM implementation by translating term interface operations to POM operations.

### Signatures for AST Classes

Signatures are fundamental for describing formal languages and very appropriate formalisms for describing the abstract syntax of programming languages. The approach taken by the POM adapter design is to extract an exact signature from the AST class hierarchy. This is possible because the AST class hierarchy essentially expresses the signature of the abstract syntax of the language.

Consider the AST class hierarchy in Figure 4.2. The root of this type hierarchy is the abstract class `ASTNode`. The abstract classes `Expression` and `Type` derive from it. Concrete types like `ArrayType` and `BooleanType` exist under `Type`. Expressions like `ArrayAccess` exist under `Expression`. This design follows the typical AST implementation idiom found in many object oriented compilers [Jon].

A signature  $\Sigma = (S, \Omega)$  can be generated from a class hierarchy using the following algorithm. Assume the root node type of the AST class hierarchy to be  $r$ , the function  $s(c)$  which maps classes  $c$  to sorts and the function  $c(c)$  which maps classes  $c$  to constructor names.

1. For every abstract class  $c_a$ , add a sort  $s(c_a)$  to  $S$ .
2. If  $c'_a$  is a direct subclass of  $c_a$ , then  $s(c'_a)$  is a direct subsort of  $s(c_a)$ . Alternatively, injections may be used: Given  $c'_a$ , a direct subclass of  $c_a$ , add an injection  $s(c'_a) \rightarrow s(c_a)$  to  $\Omega$ .
3. For every concrete class  $c_c$ , add a constructor with name  $c(c_c)$  to  $\Omega$ . For every parameterless method in  $c_c$  returning a subtype  $c''$  of  $r$ , add an argument of sort  $s(c'')$  (corresponding to a class  $c''$ ) to the parameter list of  $c(c_c)$ . The result sort of  $c(c_c)$  is the sort of the direct superclass of  $c_c$ .

Applying this algorithm to Figure 4.2 gives the following excerpt of a signature with injections at the bottom.

**signature** EclipseJava

**sorts**

`ASTNode`, `Expression`, `Annotation`, `Type`

**constructors**

`MarkerAnnotation` : `Name`  $\rightarrow$  `Annotation`

`ArrayAccess` : `Expression`  $\times$  `Expression`  $\rightarrow$  `Expression`

`ArrayCreation` : `ArrayType`  $\times$  `List(Expression)`  $\times$  `ArrayInitializer`  $\rightarrow$  `Expr'n`

`CastExpression` : `Type`  $\times$  `Expression`  $\rightarrow$  `Expression`

`PostfixExpression` : `Operator`  $\times$  `Expression`  $\rightarrow$  `Expression`

`PrefixExpression` : `Operator`  $\times$  `Expression`  $\rightarrow$  `Expression`

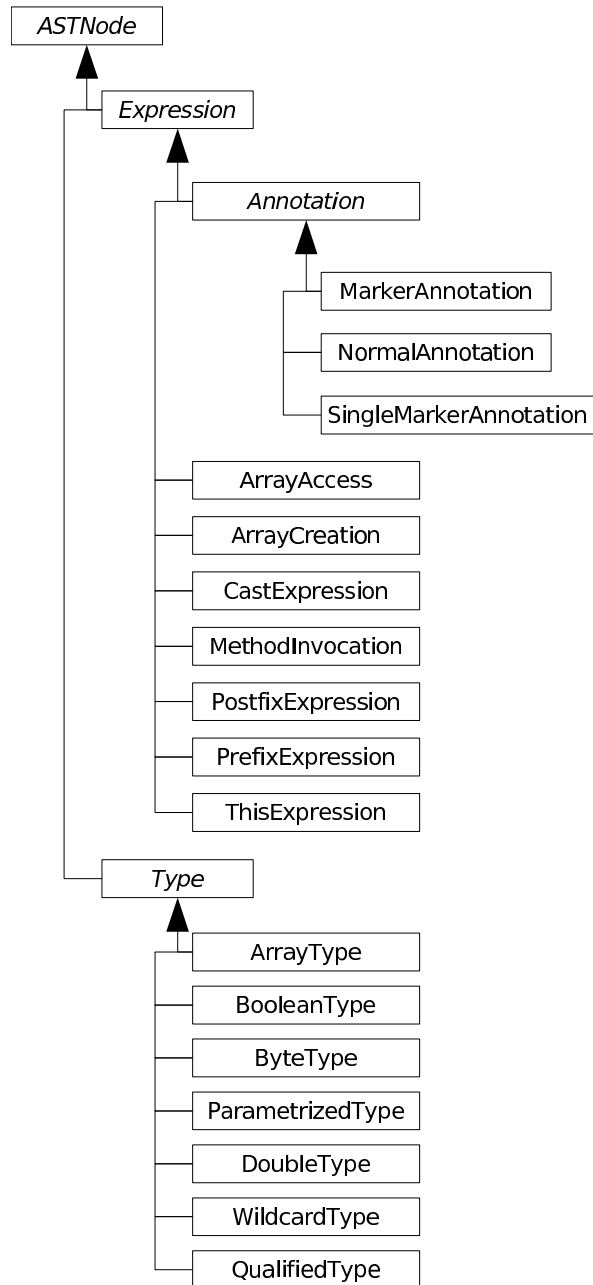


Figure 4.2: Excerpt from the AST type hierarchy of the Eclipse Compiler for Java. Other Java compilers, such as Polyglot and Sun's javac, have structurally similar ASTs. Filled arrows indicate inheritance. Names in *italics* indicate abstract classes.

```

ThisExpression : Name → Expression
ArrayType      : Type × Int × Type → Type
BooleanType   : Type
ParametrizedType : Type × List(Type) → Type
WildcardType  : Type → Type
QualifiedType  : SimpleName × Type → Type
                : Expression → ASTNode
                : Annotation → Expression
                : Type → ASTNode

```

Consider the class `MarkerAnnotation` in Figure 4.2 and its corresponding constructor definition in the signature above. `MarkerAnnotation` derives from the abstract class `Annotation`, making `Annotation` the result sort of the `MarkerAnnotation` constructor. Further, the class `MarkerAnnotation` has one method returning a subclass of `ASTNode`, `Name getName()` (not shown), and this gives rise to the single constructor argument of sort `Name`.

By processing the source code of the POM, a starting signature is automatically generated. While immediately usable, the initial result is only a proposal. The order of the argument list for each constructor may need manual tuning for consistency among the various constructors. The extracted signature will remain stable as long as the POM implementation changes only rarely. For many mature compilers, the AST designs seem to change rather slowly and mostly in response to changes in the subject language.

Using algebraic signatures, as shown above, is sufficient for capturing precisely and concisely the relationships between the AST node types (as sorts) and their allowed subnodes (in the constructor declarations). In principle, other abstract syntax description languages, such as the Zephyr [WAKS97] abstract grammar language, may be used to represent the abstract syntax. Grammatical formalisms may offer additional expressiveness, but for the scheme illustrated above, this is not necessary. For some readers, grammars may present a more familiar notation than signatures, however.

### Term Hierarchy

Terms are recursively built from constructor applications, but as Chapter 2 showed, term rewriting systems frequently have additional primitive term types used for expressing transformation algorithms such as integers and real numbers, lists and tuples, and strings. The terms are available as primitive types in the transformation language, but the machinery behind the terms is written in the runtime language. That is, the term library is implemented in the runtime language.

Below is given a signature for term manipulation. It is a generalisation of the `ATerm` interface which is used by ASF+SDF, Tom, Stratego and other term rewriting

systems. The remainder of this chapter will explain how to map operations from this signature to AST method calls. The mapping goes from functions on objects in the runtime language to functions on objects in the compiler language. (The compiler and runtime languages are often the same.)

The term interface is separated into two complementary parts: the inspection interface and the generation interface. The former provides operations for traversing and decomposing terms and is a read-only interface to the underlying POM. The generation interface provides operations for constructing POM objects from the ground up, i.e. from the leaves up. The clear separation into two interfaces is very useful because not all POM implementations allow modification. Some front-ends only support inspection of their POM and, using the inspection interface, these front-ends may be reused for expression analysis, but not for transformations. When implementing a POM adapter, one can therefore decide whether read-only access to the POM is sufficient for the problems at hand, or if a full read/write solution must be instantiated.

**Inspection Interface** Figure 4.3 shows the type hierarchy of all primitive term types. The operations defined on these types comprise the term inspection interface. It illustrates that numerous subsorts of *Term* may exist. It is not required that the term rewriting engine supports all these subtypes. A minimal, but still useful, set would include *TermAppl*, *TermInt*, *TermList* and *TermString*. This is sufficient for representing ASTs of many, if not most, compilers.

At the root of Figure 4.3 is the sort *Term* which has the following operations defined for it:

```
signature TermInspection
sorts Term Integer TermCtor
ops
  get-primitive-type : Term → Integer
  get-constructor   : Term → TermCtor
  get-subterm-count : Term → Integer
  get-subterm       : Term × Integer → ITerm
  is-equal          : Term × Term → Term
  hash-code         : Term → Integer
```

The *get-primitive-type* operation returns an integer enumerating which primitive type a given term is, i.e. whether it is an application term, a string, a list, a tuple, an integer, a real or a constructor. The *get-constructor* returns the constructor for its given term. Constructor sorts are described later. The *subterm-count* and *get-subterm* operations are used to inspect and decompose a term. The *is-equal* and *hash-code* operations are used to compare terms. They are also used when terms are placed in collections such as sets and queues.

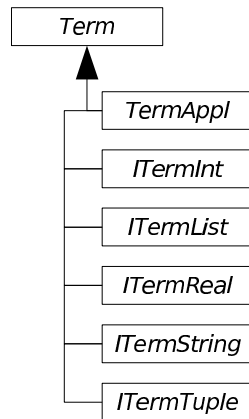


Figure 4.3: Hierarchy of sorts for the low-level term interface.

The various specific term types, such as string, integer and real have a *get-value* operation which returns a string, integer or real number, respectively, using the appropriate type in the runtime language. List term types have operations for obtaining the head and tail of the list, and a predicate signalling if a list is empty.

Objects of the sort *TermCtor* describe constructors of a signature. An *TermCtor* object has a name and an arity. The arity can be a list of sorts, or a number. Since many term rewriting systems are single-sorted, a number (of argument sorts) suffices to describe the arity of a constructor. This “one-typedness” presents some problems when rewriting on fully typed structures such as ASTs implemented in strongly typed languages. It necessitates some form of translation between the type system of the compiler language and that of the transformation language. This topic will be returned to later.

**Generation Interface** A separate generation interface exists which complements the inspection interface described previously. It consists of the following operations:

**signature** TermGeneration

**sorts**

String Int List(s)

{ TermList TermAppl TermInt TermString } < Term // *subsorts of Term*

**ops**

make-appl : TermCtor  $\times$  List(Term)  $\rightarrow$  TermAppl

make-int : Integer  $\rightarrow$  TermInt

make-list : List(Term)  $\rightarrow$  TermList

make-string : String  $\rightarrow$  TermString

make-real : Real  $\rightarrow$  TermReal

```
make-tuple : List(Term) → TermTuple
```

The *make-appl* operation is used to instantiate application terms, e.g. `Plus( $t_0, t_1$ )`, from a term constructor object (of sort `TermCTor`) and a list of terms, i.e. `[ $t_0, t_1$ ]`. The operations *make-int*, *make-real* and *make-string* are used to create terms which will be available as values in the transformation language from integers, reals and strings, respectively, in the runtime language. The parametrised sort *List(s)* is assumed to be a builtin or library type of the runtime language.

For a given POM, some of the term subsorts may be unused. As a result, not all of the generator operations need to be defined. For example, typical ASTs are constructed from named nodes (constructor applications), lists, strings and sometimes integers. In these cases, the operations *make-real* and *make-tuple* are irrelevant.

### Translating Operations

Given a formal signature for the POM and the term interfaces described previously, a POM adapter may be generated. Based on the previously extracted signature, code generation templates may be used to instantiate the necessary adapter code.

**Adapting Inspection** The crux of the inspection interface is the *get-subterm* method. In an object-oriented setting, this method will dynamically dispatch on its first argument. Assume a class `AdaptedCompilationUnit` with a field `actualCompilationUnit` of type `CompilationUnit` (from the AST implementation). The *get-subterm* method amounts to a switch:

```
meth get-subterm(this : AdaptedCompilationUnit, i : integer) =
  switch i :
    case 0: adapt(this.actualCompilationUnit.get-package())
    case 1: adapt(this.actualCompilationUnit.get-imports())
    case 2: adapt(this.actualCompilationUnit.get-types())
    default: raise ArrayIndexOutOfBoundsException
```

The *adapt* method is overloaded on `ASTNode` types. For each subclass `C` of `ASTNode`, it instantiates a term adapter object of type `AdaptedC`. The type of this new object has *get-subterm* defined on it similar to the one just shown. The remaining methods of the term interface are automatically generated using the code templates; for each `AdaptedC` class, the corresponding constructor `C` defines the return values of *get-subterm-count*, *get-constructor* and *get-primitive-type*. These functions can be generated automatically from the constructor definitions. The two remaining operations, *is-equal* and *hash-code* are discussed in the next section.

**Adapting Generation** The *make-appl* is the core of the generation interface. It forwards calls to the relevant factory methods of the AST, or instantiates subclasses

of `ASTNode` itself, say, via `new`, if the POM does not provide a node factory.

Based on the extracted signature, a map is constructed which goes from constructor names (and arity) to constructor methods. When the *make-appl* receives a request to construct `CompilationUnit`, the map is consulted and the request is forwarded to the (generated) method *make-compilation-unit*:

```
fun make-compilation-unit(t : ITermConstructor, kids : List(ITerm)) =
  if (is-package(kids[0])
    and is-import-list(kids[1])
    and is-type-list(kids[2]))
    astFactory.newCompilationUnit(as-package(kids[0]),
                                  as-import-list(kids[1]),
                                  as-type-list(kids[2]))
  else
    raise InvalidArguments
```

The responsibility of *make-compilation-unit* is to ensure that the term arguments are type correct before invoking the relevant factory method (or new expression) in the POM interface.

With this generation scheme in place, practically all of the adapter is boilerplate code that can be automatically generated based on two artifacts: the signature declaration and the compiler-specific code templates. Only the order of signature sorts must be verified and potentially fixed up by hand.

## 4.3 Implementation

The POM adapter design has been instantiated for the `MetaStratego` runtime (`Stratego/J`). This section describes the details of this implementation and how it fuses `MetaStratego` with the Eclipse Compiler for Java (ECJ).

### 4.3.1 Term Interface

The term interface described in the previous section has been implemented rather straightforwardly in Java. A basic, extensible implementation of the various interfaces is provided by `Stratego/J`. Its classes and their corresponding sorts (abstract classes) are shown in Figure 4.4. This implementation provides *basic* terms: the `Basic`-terms in the figure. By deriving from the basic implementation, POM adapters may supplement the primitive term types provided by a POM with the full range of primitive term types supported by `Stratego`. This allows reusing generic analysis transformation algorithms which assume the presence of certain term types, such as tuples or reals, even though the POM itself does not provide one. This is made possible because the basic terms can be mixed with the adapted POM terms. More on this later.

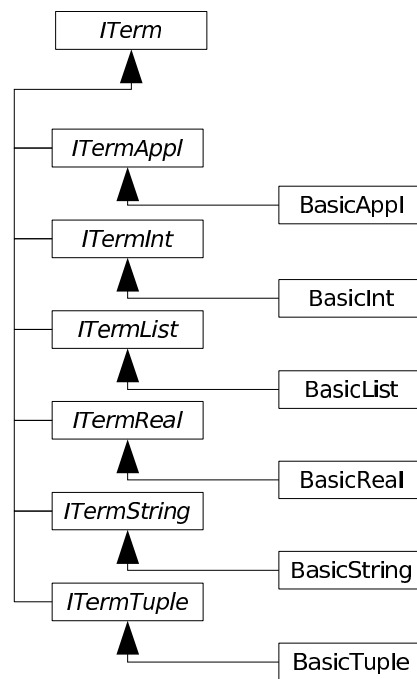


Figure 4.4: The Stratego/J runtime provides a default implementation for the sort hierarchy in Figure 4.3.

**Inspection Interface** The Java interface of the `ITerm` type is given below:

```
public int getPrimitiveTermType();
public ITermConstructor getConstructor();
public int getSubtermCount();
public ITerm getSubterm(int index);
public boolean isEqual(ITerm rhs);
public int hashCode();
```

The basic term implementation always forwards the `equals()` method to `isEqual()`. Because of this, terms may be freely used in Java collection classes. As an optimisation, the method `public ITerm[] getAllSubterms()` was added to the `ITerm` interface. In some POM implementations, children are kept in linked lists. If `getSubterm()` is used to traverse these lists, the traversal time will be quadratic in the number of children. This presents a significant slowdown. The problem occurs whenever the `MetaStratego` interpreter needs to traverse all children of a term, for example when it evaluates a `one` or an `all`. By extracting all children at the beginning of the traversal, this extra cost is avoided.

The following is another excerpt of the signature extracted from the AST class hierarchy of the Eclipse Java compiler.

```
signature EclipseJava
sorts Annotation Javadoc Name
constructors
...
PackageDeclaration : Javadoc × List(Annotation) × Name → ASTNode
...
```

The following shows the final code for the corresponding `PackageDeclaration` adapter:

```
class AdaptedPackageDeclaration implements AdaptedECJAppl {
    private final PackageDeclaration adaptee;
    private static final IStrategoConstructor CTOR =
        new ASTCtor("PackageDeclaration", 3);

    protected WrappedPackageDeclaration(PackageDeclaration adaptee) {
        super(CTOR);
        this.adaptee = adaptee;
    }

    public ITerm getSubterm(int index) {
        switch(index) {
            case 0: return ECJFactory.adapt(adaptee.getPackage());
            case 1: return ECJFactory.adapt(adaptee.imports());
        }
    }
}
```

```

    case 2: return ECJFactory.adapt(adaptee.types());
    } throw new ArrayIndexOutOfBoundsException();
}

public PackageDeclaration getAdaptee() {
    return adaptee;
}
}

```

In the current implementation, AST nodes are wrapped lazily, thus wrapping only occurs when needed. When AST nodes are traversed by the rewriting engine, the AST node children are wrapped progressively, as terms are unfolded.

The `isEqual()` method performs a deep equality check, but will not result in a recursive adaptation of child objects. Recall that Stratego allows pattern matching with variables. All the code for handling variable bindings is kept inside the interpreter implementation. This keeps the POM adapter interface minimal.

**Generation Interface** The POM adapter technique does not require an implementation of the generation interface. If one is not provided, only analysis can be done. For rewriting to be possible, the following factory methods must be available.

```

public interface ITermFactory { ...
    public ITerm makeAppl(ITermConstructor ctor, ITerm[] args);
    public ITerm makeString(String s);
    public ITerm makeInt(int i);
    public ITerm makeList(ITerm[] args);
}

```

Default implementations exist for strings, lists and integers, provided by a term factory for basic terms called `BasicTermFactory`. Only the `makeAppl` method must be supplied by hand. In the prototype, this method forwards constructor requests to the appropriate factory methods of the ECJ AST. When a request for constructing, say, a `PackageDeclaration` node is seen, the request is forwarded to `newPackageDeclaration()` of the ECJ AST factory.

```

1 public class ECJFactory implements ITermFactory {
2     ...
3     public ITerm makeAppl(ITermConstructor ctor, ITerm[] args) {
4         switch(constructorMap.get(ctor.getName())) {
5             ...
6             case PACKAGE_DECLARATION:
7                 return makePackageDeclaration(args);
8             ...

```

```
9     }
10  }
11
12  private ITermAppl makePackageDeclaration(ITerm[] args) {
13      if((!isJavadoc(kids[0]) && !isNone(kids[0]))
14          || !isAnnotations(kids[1])
15          || !isName(kids[2]))
16          return null;
17
18      PackageDeclaration pd = ast.newPackageDeclaration();
19      if(isNone(kids[0]))
20          pd.setJavadoc(null);
21      else
22          pd.setJavadoc(getJavadoc(kids[0]));
23      pd.annotations().addAll(getAnnotations(kids[1]));
24      pd.setName(getName(kids[2]));
25      return adapt(pd);
26  }
27 }
```

The method `makePackageDeclaration` first ensures that `args` contains the correct number and types of arguments, on lines 13–16. Next, it calls `AST.newPackageDeclaration()` on line 18, uses set methods on the resulting `PackageDeclaration` object on lines 19–24 to complete the construction. Line 4 is a performance trick for mapping constructor names to constructor methods. Java does not support function pointers directly. A lookup table (`constructorMap`) is statically initialised when the factory is created. This map allows rapid dispatch to the corresponding construction logic for a given constructor.

### Using the Adapter

The `Stratego/J` interpreter is a Java object of type `Interpreter`. When it is instantiated, one of its constructors allows the user to specify which term representations should be used for its programs (the compiled `Stratego` scripts) and which factory to use for the data (the terms which it will process). The following code snippet initialises an interpreter instance that will accept compiled scripts as `ATerms` and can rewrite on the Eclipse compiler ASTs.

```
ITermFactory data = new ECJFactory();
ITermFactory program = new WrappedATermFactory();
Interpreter intp = new Interpreter(data, program);
intp.addOperatorRegistry(new ECJLibrary());
```

The Eclipse Compiler FFI (discussed below) is added to the registry of known foreign functions on the last line.

### 4.3.2 Design Considerations

*Functional Integration* – The POM adapter for ECJ provides an FFI library that provides type checking support for Java subject code. This library exposes type checking strategies, such as `type-of`, to the transformation programs written in Stratego. These strategies will call the ECJ type checker, through the FFI mechanisms provided by Stratego/J. Invoking `type-of` on, say, an `InfixExpression` term, will result in a call to `resolveTypeBinding()` on the `InfixExpression` object wrapped by this term. Stratego is a single-sorted rewriting language typed, and only the arity of terms is statically guaranteed. If, say, a `SimpleName` term is passed to `type-of`, the FFI stub for `type-of` will detect this and fail, just like any expression in Stratego can fail.

*Imperative and Functional Data Structures* – The rewriting engine assumes a functional data structure. In-place updates to existing terms are not allowed. The generation interface is designed so that existing terms will never be modified – there simply are no operations for modifying existing terms. This makes wrapping imperative data structures in such a functional dress relatively straightforward – compilers need not provide one. The only restriction is that AST nodes must not change behind the scenes; the rewriting engine must have exclusive access while rewriting. For in-place rewriting systems, e.g. TXL [Cor04], a slight modification of the `ITerm` interface would be necessary so that subterms of existing terms can be modified in place. Instead of a `makeApp1` method, one would need a `ITerm replaceApp1(ITerm t, int index, ITerm newKid)` method. (Support for this already exists in the `BasicTerm` implementation, but is not used by Stratego/J.)

In imperative data structures, the state of the system may change during matching because traversal over the data may change the state of the traversed objects – data may for example be loaded from the disk during traversal. State change always occurs when the program object model is wrapped lazily: new POM adapter objects will (in general) be instantiated during matching. Strictly speaking, matching therefore has “side-effects”. While the POM adapter may technically speaking force a state change, it will never result in observable differences of terms: all previously bound terms will remain unchanged (so  $\varepsilon$  remains unchanged). Since it is extremely rare for visitor interfaces to have harmful state changing behaviour, potential side-effects are of little practical concern for building and matching.

*Efficiency Considerations* – Using a functional data structure provides some appealing properties for term comparison and copying. As described in [BV00], maximal sharing of subterms (i.e. always representing two structurally identical terms by the same object) offers constant-time term copying and structural equality checks as these reduce to pointer copying and comparisons, respectively. This is important for

efficient pattern matching because term equivalence is deep structural equality, not object (identifier) equality. The ECJ AST interface provides deep structural matching, but this is not constant-time.

Quick, deep structural matching can be provided in the POM adapter, but then lazy wrapping must be given up. In this case, hash codes must be computed deeply, because the hash must reflect the entire structure of the term and not just the object identity of the AST node. Ideally, only two objects that are structurally equal should have the same hash code. Once a hash code has been computed, it can be memoized since the subterms will never change.

*Efficiency* – The memory footprint of the wrapper objects is small. Each object has only two fields. By keeping a (weak) hash table of the AST nodes already wrapped, the overhead is reduced even further. The current implementation takes just over four minutes on a 1.4GHz laptop with 1.5GB of RAM to run a simple bounds checking idiom analysis described in Chapter 10 on the entire Eclipse code base (about 2.7 million lines of code). Complicated transformations are limited by the efficiency of the current Stratego interpreter, not the adapter. Compiling the scripts to Java byte code, instead of the abstract Stratego machine, should significantly improve performance for complicated scripts.

### Type System Interaction

The type system of Stratego is significantly more dynamic than that of Java. Many of the usual caveats of integrating a dynamically typed scripting language with a statically typed “host” language apply. However, a few additional considerations specific to the current context warrants further discussion.

**Strongly vs Weakly Typed ASTs** The ECJ AST is strongly typed and the term rewriting system needs to respect this. Stratego is dynamically typed and would normally allow the term `InfixExpression(1, BooleanLiteral(True), 3)` to be constructed, even though the subterms must be `String` and `Expression` as declared previously (making 1, 3 invalid subterms). `ECJFactory` has two modes for dealing with this. In strict mode, the factory bars invalid `EclipseJava` terms from being built. As a result, the build expression `!InfixExpression(1, BooleanLiteral(True), 3)` fails. Terms without any `EclipseJava` terms, such as `(1, 2, 3)`, can be built freely. These will not be represented as `EclipseJava` terms, but by the default internal term library of the interpreter. Terms without `EclipseJava` constructors are referred to as basic terms.

In lenient mode, mixed terms consisting of basic and `EclipseJava` terms are allowed, such as `InfixExpression(1, BooleanLiteral(True), 3)`. The `BooleanLiteral` subterm remains an `EclipseJava` term, but 1 and 3 are basic terms. The root term, `InfixExpression`, becomes a mixed term and is also handled by the basic term library. Since all terms are constructed from their leaves up (`ITermFactory` forces

this), `ECJFactory` can determine inside its `makeApp1()` method when it can build an `EclipseJava` term: if and only if all subterms are `EclipseJava` terms and are compatible with the requested constructor, an `EclipseJava` term is built. Otherwise, a mixed term must be constructed. ECJ FFI functions will fail if they are passed mixed terms. Java programs, such as Eclipse plugins, may embed the Stratego/J interpreter for rewriting ASTs. The embedding Java code will receive an `ITerm` as the result from the interpreter. The actual runtime type of this object can be any subtype of `ITerm`. Therefore, if the embedding Java expects an `AdaptedASTNode`, it must perform a dynamic type check to ensure this before proceeding.

**Subject Language Semantics** Rewritings can result in structurally valid but semantically invalid ASTs, for example, by removing from a class a method which is called elsewhere. Neither Stratego nor the ECJ AST API checks for this. However, a subsequent type reanalysis will uncover the problem. If the type analysis functions are used as transformation post-condition checks, one can ensure that a transformation is always type correct.

## 4.4 Related Work

Language processing is what software transformation systems like Tom [MRV03], TXL [Cor04], ASF+SDF [vdBvdH<sup>+</sup>01] Stratego [BKVV06] were designed for. Except for Tom, these systems were not designed to work with more than one term representation. Retrofitting the POM adapter into existing implementations should not be too difficult provided that there is a clean interface to the terms. In the cases where the contract of the term interface is similar to that described in Section 4.2, many details of the implementation used for Stratego/J should be reusable. This is the case for the `ATerm`-based approaches such as ASF+SDF and Tom.

Open compilers such as SUIF [WFW<sup>+</sup>94], OpenJava [TCIK00], OpenC++ [Chi95] and Polyglot [NCM03] are natural candidates for integration. They have well-defined APIs to many parts of their pipeline, often including the backend. It is not necessary for the compiler to be designed as an open platform, however. As long as the AST API is accessible, a POM adapter can be generated for it. If one accepts greybox reuse, this is possible for most compilers.

A key strength of Stratego is generic traversals (built with `one`, `some` and `all`) that cleanly separate the code for tree navigation from the actual operations (such as rewrite rules) performed on each node. The `JJTraveler` visitor combinator framework is a Java library described by van Deursen and Visser [vV02] that also provides generic traversals. Generic traversals and visitor combinators go far beyond traditional object-oriented visitors. The core term interface required by both approaches is very similar. Comparing the `Visitable` interface of `JJTraveler`, the `ATerm` inter-

face found in ASF+SDF and the Stratego C runtime, suggests that the POM adapter should be reusable for all of these systems, implementation language issues notwithstanding (C for ASF+SDF, and Java for JJTraveler and the MetaStratego runtime).

A related approach to rewriting on class hierarchies is provided by Tom [MRV03]. Tom is a language extension for Java. It provides features for rewriting and matching on existing class hierarchies. This is done by using a declaration language called Gom to describe existing classes as abstract data types. Using these descriptions, a pre-processor will expand matching operations in the Tom language into the appropriate method calls according to the Gom declaration. Recent versions of Tom also support generic traversals in the style of JJTraveler, but its library of analyses is still rather small. Gom and the POM adapter technique are both based on the idea of obtaining an abstract declaration of specific class hierarchy and adapting a term rewriting program to operate on the class hierarchy. The approach described in this chapter can extract these descriptions automatically. The POM adapters enable the plugging into any program model at runtime – the binding between a given transformation runtime and a given program object model may be deferred until it is needed. The Tom program is specialised for a given class hierarchy at compile-time. The POM approach makes very few assumptions about the rewriting language; the term interface provided by the POM adapter can form the basis for most rewriting languages, including Tom.

## 4.5 Discussion

Recent research has provided pluggable type systems, style checkers and static analysis with scripting support. This indicates that there is demand for high-level languages for expressing both analyses and transformations. The languages should be directly usable by software developers. The experiences gained in the field of program transformation, and that have gone into the language design for software transformation systems, are directly applicable for problems of this kind. The tradeoff with using a domain-specific language for expressing transformations and analysis is that of any high-level domain-specific language: the same language features that make the language powerful and domain-specific also make it more difficult to learn. This can be offset in part by good documentation and a sizable corpus of similar code to learn from. In conjunction with the Spoofox development environment, discussed in Chapter 9, the fusion of Stratego and ECJ described in this chapter becomes easily accessible to developers. This will become more apparent through the case studies presented in Part V of this dissertation.

The POM adapter implementation discussed in this chapter, and which is the basis for Chapter 10, was generated using a custom Stratego program and a collection of hand-written Java templates. Careful inspection of the AST implementations

of the Sun Java Compiler, the prototype Fortress compiler, the extensible Java compiler Polyglot [NCM03] and the JastAdd [HM03] compiler compiler, suggests that the POM adapter technique is applicable to a wide number of compilers written in the object-oriented style. Additionally, implementation platform notwithstanding, investigation of the GNU Compiler Collection (GCC) tree representation API indicates that this technique should also be applicable to GCC. Furthermore, it seems feasible to write a more general POM adapter generator based on the current prototype program which developers with basic knowledge of Stratego should be able to adapt this generator to process new AST hierarchies – at least those implemented in Java.

## 4.6 Summary

This chapter introduced a novel design of a program object model adapter and demonstrated how this design can fuse rewriting language systems with existing compilers and front-ends. This fusion enables language independence through large-scale reuse: entire transformation systems may be plugged onto existing language infrastructures, such as compiler front-ends. The stability and robustness of mainstream front-ends is thereby immediately available to transformation programmer who may express their analysis and transformation problems using high-level transformation languages which support precise and concise formulations.

This chapter demonstrated that even a relatively small degree of extensibility on the part of the compiler is sufficient for plugging in a rewriting system. It motivated that the POM adapter can be reused for other, tree-like data structures and that its design is also applicable to other rewriting engines. In Chapter 10, the applicability of the design will be demonstrated through a series of analysis and transformation problems taken from mature and well-designed frameworks.