– Eventually, you'll be famous enough to ramble about stuff you don't know.
– I do that all the time, but the compiler is my audience…

– Øyvind Kolås replies.

10 Extending Compilers with Transformation and Analysis Scripts

Efficient and robust tool support for domain abstractions is crucial for effective software development, but implementing such domain-aware tools using current language infrastructures is very difficult. Expressing framework and library-specific analyses for design rules, bug pattern finding or protocol checking, as well as transformations for library-specific optimisation, is a goal strived for by implementers of pluggable type systems, defect checkers, code smell detectors and framework migration tools. Domain-specific transformation languages hold the promise that language processing problems may be expressed succinctly and precisely, but this depends on the availability of robust language front-ends that can parse and type check large amounts of source code robustly.

This chapter demonstrates how the general transformation language abstractions introduced in Part III may be applied to building a scriptable analysis and transformation framework. The framework consists of a compiler, a transformation language and a program object model adapter for the abstract syntax tree (AST) of the compiler. The adapter fuses the Eclipse Compiler for Java with the Stratego/J. This enables Stratego scripts to be written which rewrite directly on the compiler AST. The applicability of the system is illustrated with user-definable scripts that perform framework and library-specific analyses and transformations.

The case study found in this chapter is a significantly expanded version of the one found in the paper *"Fusing a Transformation Language with an Open Compiler"* written with Eelco Visser [KV07a].

10.1 Introduction

Stringent use of domain abstractions is key to efficient and maintainable software. The compiler cannot optimise, nor check the correct usage of, domain abstractions because the rules governing the abstractions are part of the application domain and

not the programming language. As a result, domain abstractions are often used incorrectly or inefficiently. Various techniques have been devised to combat this, such as typestate analysis [SY86, SY93], pluggable type systems [ANMM06], code smell detectors, defect analysers [Cop05] and other static analysis tools. For the most part, development and maintenance of these tools is so costly that their construction can only be afforded for the most used domains. For example, common static analysis tools for Java support only the standard library [HP04] and Enterprise JavaBeans [CNFP06]. The situation is similar for domain-specific optimisation: high-performance compilers may come with extensions and directives which improve performance for certain, general numerical computation problems [CDK+01]. Other domains receive little or no support.

The state-of-the-art is that existing analyses and optimisations only serve a very restricted set of domains and the needs of most other projects and frameworks are largely left unattended. The absence of solid and adaptable tools has led to the proliferation of ad-hoc techniques that are often brittle and text-based [DR97]. Fortunately, it has become more common to expose at least some API to the compiler internals in the recent years, in particular to the abstract syntax tree. This presents a significant opportunity for providing good domain support for a much wider range of domains. It is now possible to leverage the maturity and robustness of the parsers and type analysers available in mainstream compilers. Previously, such infrastructure could only be reused from selected, open research compilers [WFW+94, TCIK00, NCM03].

The framework presented in this chapter takes advantage of the recent opening of mainstream compilers. It is a fusion between the Stratego rewriting language and the Eclipse Compiler for Java (ECJ) based on the program object model adapter technique described in Chapter 4.

The composed system provides a powerful framework that allows framework developers to implement domain-specific transformations and analysis for Java frameworks in Stratego. Developers may take advantage of pattern matching, rewrite rules, generic tree and graph traversals as well as a reusable library of generic transformation strategies and data-flow analyses.

The contributions of this chapter include:

- Bringing the analysis and transformation capabilities of modern compiler infrastructure into the hands of advanced developers via a convenient and mature program transformation language.

- Making program transformation tools and techniques practical and reusable for framework developers by integrating directly with stable tools like the Java compiler. This lowers the entry barrier for developers wanting to write library-specific program analyses and transformations.

- A discussion of the design and implementation of a prototype tool for domain-specific analysis and transformation.
- A validation of its applicability through a series of examples taken from mature and well-designed applications and frameworks.

The remainder of this chapter is organised as follows: Section 10.2 motivates the need for domain-specific analysis and transformation and why scripting these with a language-independent transformation system is useful. Section 10.3 shows the practical applicability of the prototype on a series of commonly encountered framework-specific analysis and transformation problems. Section 10.4 discusses implementation details of the prototype. Section 10.5 covers related work. Section 10.6 discusses some tradeoffs related to the technique.

10.2 Scriptable Domain-Specific Analysis and Transformation

The main motivation for extending compilers with scripts is the lack of domain knowledge possessed by traditional compilers. This domain ignorance bars compilers from providing detailed errors and warnings about the usage of domain abstractions and from automatically optimising the usage of domain abstractions. For example, the compiler will not warn if a function is written so that a file may be read before it is opened nor will it remove a call to `close()` on a file that is known to be closed. This is reasonable, given that the semantics of library objects is, in general, not known to the compiler writer.

A compounding problem is the lack of any general facility of adding such knowledge by the user. As a consequence of the closedness of compilers, many domain-specific language processing tools are constructed from scratch, duplicating substantial parts of compiler infrastructure that could and should have been reused. Because implementing and maintaining robust language infrastructures for mainstream languages is very laborious, many stand-alone language processors are often brittle or incomplete. This is clearly an unfortunate situation. Finding good approaches to compiler infrastructure reuse of is a worthwhile topic of study, and relates closely to the topic of language-independent transformations explored in this dissertation.

All compilers have an internal program object model. For most compilers, the abstract syntax tree (AST) forms the core of this model and is supplemented with additional data structures (such as a symbol table) and functionality (such as type analysis, code searching, and pretty-printing). Some modern compilers, such as the Sun Java Compiler, expose AST programming interfaces that allow developers to implement custom language processing tools based on the compiler. Another example is the Eclipse compiler for Java, which has APIs that are used to implement the

interactive source code refactorings in the Eclipse Java development environment. Compared to implementations with general purpose languages like Java, implementing language processing with transformation languages usually results in smaller and more readable programs, which are quicker to change and maintain. The downside is that their infrastructure for processing mainstream subject language code is generally not as robust, optimised nor up to date as that available in mainstream compilers. This is not likely to change. The massive user base served by mainstream compiler serves to weed out bugs and easily justifies investments for hand-optimising core parts of the infrastructure.

Recent research and industry practise is rife with examples where domain-specific transformations and analyses play an important role, such as framework-specific refactoring, optimisation of library abstractions [GL00, Kal03], advanced style and type checking [ANMM06] and framework-specific code smells [vEM02]. These domain-specific language processing problems are different from their general counterparts in at least four ways.

1. They are often less performance-critical because they only apply to small amounts of code and they can be targeted and applied only to the relevant parts of the code, whereas the general compiler analyses and transformations are applied exhaustively.

2. They may have a much higher degree of variability. The conditions they check for, the locations they should be applied to and the point in the software life cycle they should be applied vary even between individual projects using the same domain abstractions.

3. They have a much higher rate of change. Whenever the framework changes or is rearchitected, the domain-specific analyses and transformations must follow suit.

4. They occur across programming languages, and any one system may involve the combination of many languages which have clear project-specific rules for how they should interoperate.

Being able to attack this problem with a high-level, language-general transformation system coupled with stable and robust language processing foundations is appealing because the combination is likely to provide an effective and expressive platform.

10.2.1 Architecture

The Java transformation framework proposed in this chapter is available as a standalone command-line application and as a reusable Eclipse plugin.

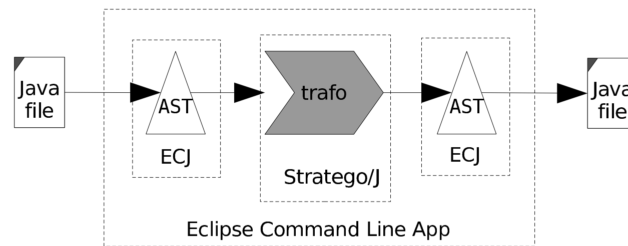

Figure 10.1: Command-line based transformation.

In stand-alone mode, as shown in Figure 10.1, the system performs fully automatic source-to-source transformation. The user supplies the path of a project and a script to execute, or the path to a single source file and a transformation script. The script may use a file API to traverse the project directories and to parse source files to obtain their AST. After rewriting, the script may use the file API to write modified ASTs back to disk in the form of formatted (pretty-printed) source code.

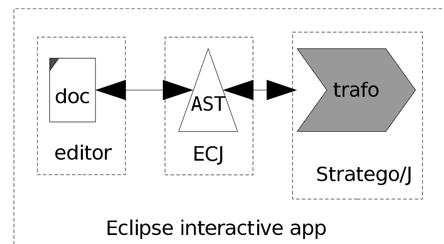

Figure 10.2: Interactive transformation.

In plugin mode, as shown in Figure 10.2, interpreter objects may be instantiated with arbitrary scripts. These objects may be handed individual ASTs obtained from live documents. Strategies may be invoked via the interpreter objects to perform AST rewrites. The editing framework provides logic for synchronising the text with the modified AST. This allows scripts to be used for very fine-grained source code queries and transformations on the source code alongside the programmer's manual editing of the source code.

10.2.2 MetaStratego as a Scripting Engine

The combination of the Stratego/J runtime and the transformlet lightweight component system is very handy for deploying interactive transformation scripts. The framework supports the loading and execution of transformation scripts in the form of transformlets. As each transformlet is loaded, it registers new actions with the framework. These actions become available through a separate menu in the user interface. The programmer may invoke the script actions from this menu. The

transformlets may be implemented with any of the language extensions introduced in previous chapters of this dissertation.

10.3 Examples of Domain Support Scripting

This section describes analyses and transformations that are specific to a given project or framework. Some analyses check that domain abstractions provided by the framework or library are used properly. They apply to the clients of the libraries, but not to the library code itself. Other analyses check for a consistent realisation of the domain abstractions. These apply to the implementation of the library. Additional examples will demonstrate how the results of these analyses may be used to perform source code transformations.

The examples have been chosen to demonstrate how the (extended) Stratego language can handle syntax-, type- and flow-based analyses, and what an advanced framework developer with a good working knowledge of language processing and Stratego could implement. However, Stratego is capable of performing significantly more advanced analyses and transformation than shown here. See [OV05, BKVV06, Kal03] for some examples. It is also important to note that with a program object model (POM) adapter for another front-end, for example a C or C++ compiler, the same techniques are directly applicable to libraries written in C or C++.

10.3.1 Project-Specific Code Style Checking

Software projects of non-trivial size always adopt some form of (moderately) consistent code style to aid maintenance and readability. Maintainers of such systems may be concerned with checking for proper implementation and proper use of domain abstractions. Consistency of implementation can be improved by encouraging systematic use of particular idioms. The following idioms are taken from the internal AST implementation of the Eclipse Compiler for Java and from the graphical user interface library Standard Widget Toolkit (SWT)[1].

Equality Test Idiom. A common idiom when implementing `equals()` methods on objects in Java is to start with an `instanceof` check. The internal AST classes of the Eclipse Java compiler follow this idiom. For the `CompilationUnit` class, the idiom looks like this:

```
public boolean equals(Object obj) {
  if(!(obj instanceof CompilationUnit))
    return false;
  ...
```

[1] Part of Eclipse.

```
}
```

The idiom is simple to implement, but consistently applying it requires a degree of fastidiousness best left to the computer. It is easy to miss a spot when performing maintenance, and new maintainers to an existing code base may not be aware of the idiom. The pattern matching capabilities of Stratego can be used to verify that the code for `equals()` is of the correct form:

```
1  check-equals-method =
2    ?MethodDeclaration(_,_,SimpleName("equals"),object(),_,Block(stmts))
3   ; where(not(<Hd> stmts ; ?IfStatement(e, _, _) ; <check-expr> e)
4         ; emit-warn(|"equals() does not start with instanceof check"))
5
6  check-expr =
7  ?PrefixExpression(
8        PrefixExpressionOperator("!")
9      , ParenthesizedExpression(InstanceofExpression(_, _)))
```

The `check-equals-method` strategy should be applied to the method declaration of an `equals()` method. It starts with a pattern match (on line 2) to verify this. In the process, it binds the variable `stmts` to the list of statements of the method body of `equals()`. The subterm `object` is an overlay that matches an argument list of in parameter of type `Object`. The first statement of the body is retrieved with `Hd` (on line 3) and matched against a pattern for `if`. On a successful match, the condition of the `if` is checked with `check-expr` which will only match expressions on the form `!(_ instanceof _)`, where `_` is any expression. Should the `if` be omitted, or not follow the required idiom, `emit-warn` (on line 4) will display a warning. When run in command-line mode, this warning will be displayed on the console. In interactive mode, it can appear in a window containing compile-time warnings. The code above is purely syntactical. It does not know anything about the Java type context in which it is applied. It would be reasonable to add additional context information so that the `instanceof` check was verified to check for an appropriate type, i.e. the type in which the `equals()` method is defined. To improve analysis accuracy and the expressive power, type checking can be employed, as shown next.

Field Type Restriction Idiom. The choice between `LinkedList`, `ArrayLists` or primitive arrays is often a source of contention. Once the selection has finally been made for a particular group of classes, it serves to be consistent. The internal AST of ECJ uses the primitive array type pervasively, e.g.:

```
class TypeDeclaration ... {
  ...
  public FieldDeclaration[] fields;
```

```
  ...
}
```

The following strategy can be applied to a type declaration and will verify that none of the fields are of the type `forbidden-type`, or any subtype thereof:

```
check-type-decl(|forbidden-type) =
   ?TypeDeclaration(_, _, _, _, _, body)
  ; where(<map(try(check-field(|forbidden-type)))> body)

check-field(|forbidden-type) =
    ?FieldDeclaration(_, field-tp, _)
  ; <type-of ; is-subtype-of(|forbidden-type)> field-tp
  ; emit-warn(|"Field is of illegal type!")
```

The strategy `check-type-decl` should be applied to a type declaration term, and will use `check-field` to iterate over its fields. The strategies `type-of` and `is-subtype-of` are used to retrieve the type of each field and test if these are subtypes of `forbidden-type`. Applying `check-type-decl(|"java.util.List")` to a type declaration with, say, a `List` field results in a warning.

Next, an example shows how generic traversals can control which AST nodes a strategy should apply to.

Context-Specific Visibility Idiom. Decomposing abstractions into namespaces may pose significant challenges for the visibility mechanism of the namespace system when constructing large libraries. Consider the placement of classes into Java packages for the graphical widgets found in SWT. Each graphical element, such as a button, text area, or check box, is encapsulated in its own subclass of `Widget`. Most widget classes in SWT are not intended to be subclassed by the user. However, for implementation purposes, the `final` keyword was not used and the subclassing prohibition is only mentioned in the source code documentation of the individual classes.

```
check-illegal-swt-subclass =
    ?TypeDeclaration(_, _, _, _, _, _)
  ; type-of ; supertype-of ; dotted-name-of ⇒ stp
  ; <list-contains(?stp)> restricted-swt-types
  ; emit-warn(|"Illegal subclass of org.eclipse.swt.widgets.Widget!")
```

This code snippet will check that a given type declaration is not a subtype of any of the “inheritance restricted” SWT classes. The list of these widgets is kept in the (global) variable `restricted-swt-widgets`. The strategy should be applied a type declaration term. If the initial pattern match succeeds, the dotted name (for example, `"java.lang.String"`) of the super type of this type declaration is computed and stored in `stp`. If the super type is contained in the `restricted-swt-types` list, the current type

declaration inherits a subclass-restricted widget and the warning at the bottom line is emitted. Arguably, with a sufficiently complicated regular expression, violations of the inheritance restriction may often be found using purely text-based approaches. Unfortunately, should programmers insert comments at unexpected places, this approach is sure to break.

Warnings will be emitted if the strategy is applied to the code of the SWT widget library. The strategy should therefore only be applied to code *using* SWT. It can be applied easily to a Java project using the following a two-level generic traversal scheme:

```
analyse-package(|project) =
  dir-topdown(parse-and-resolve(|project)
          ; topdown(try(check-illegal-swt-subclass)))
```

At the outer level of the traversal, `dir-topdown` will recurse through a directory structure and call `parse-and-resolve` to construct the corresponding compilation unit (AST) for each `.java` file. Once constructed, `topdown` will recurse over it, in preorder, and apply the `check-illegal-swt-subclass` strategy to all terms. This ensures that all contained top-level, local, anonymous and inner type declarations inside each compilation unit are visited and checked. Multiple checks can easily be composed into one pass using the `topdown` generic traversal:

```
topdown(try(check-illegal-swt-subclass) ; try(check-equals-method))
```

Bounds Checking Idiom. Consider the following code for iterating over `x`:

```
for(int i = 0; i < x.size(); i++) { ... }
```

If `x` is a value object of type `T`, i.e. happens to be immutable, then the `size()` method will be invoked needlessly for every iteration. The JIT may possibly inline this call but only if the code is executed frequently enough. One might like to encourage a coding style that is also efficient with the bytecode interpreter:

```
{ final sz = x.size(); for(int i = 0; i < sz; i++) { ... } }
```

This idiom is used throughout the implementation of the internal AST classes of ECJ and may be checked for using the following strategy:

```
check-for =
    ?ForStatement(_, e, _, _)
  ; <topdown(try(call-to-immutable))> e

call-to-immutable =
    ?MethodInvocation(_, _, _, _, _, [])
  ; binding-of ⇒ MethodBinding(class-name, _, _, _)
  ; <list-contains(?class-name)> immutable-classes
```

```
  ; emit-warn(|"Call to method on immutable object in loop iteration")
```

The strategy `check-for` should be applied to a `for`-statement. If any of the condition expressions are calls to methods without parameters of objects of an immutable type, a warning is emitted. The list of known non-mutating methods is given in the list `immutable-classes`.

Using data-flow analysis, method calls on objects which are not immutable could also be considered. As long as the body of the `for`-loop does not invoke any mutating operation and does not pass `x` as an argument to another method, immutability can be assumed. By keeping (*typename*,*methodname*) pairs in an `immutable-methods` list, the immutability property can be looked up, much like the subclass restriction property was looked up.

The field type restriction and the bounds checking idioms show how analyses requiring type information can be expressed. The type analysis functionality is provided by ECJ, and made available to scripts through the strategies `subtype-of`, `supertype-of`, `is-subtype-of`. These strategies connect to the compiler via the foreign function interface introduced in Chapter 4.

10.3.2 Custom Data-Flow Analysis

Totem propagation is a kind of data-flow analysis where variables in the source code are marked with annotations called totems [Kal03]. These assert properties on the variables which are later used by other analyses and transformations. A meta-program will perform data-flow analysis and propagate the asserted totems throughout the code, following the same principles as constant propagation.

Totem propagation is in many ways similar to typestate analysis, which is "a data-flow analysis for verifying the operations performed on variables obey the typestate rules of the language" [SY93]. Typestate analysis is mostly concerned with verifying protocols such as ensuring that files are opened before they are read. Totem propagation uses the same data-flow machinery to discover opportunities for optimising away unnecessary calls (such as a call to `sort()` on a sorted list) or replacing costly operations with cheaper ones (such as binary search instead of linear search on sorted lists). Meta-programs performing these forms of data-flow analyses must be aware of the propagation rules for each kind of totem.

A totem propagator could be useful for removing dynamic boundary checks in a library for matrix computations. Consider the following matrix interface defined in the Matrix Toolkits for Java (MTJ) library [mtj06]:

```
public interface Matrix {
  public Matrix add(Matrix B, Matrix C);
  public Matrix mult(Matrix B, Matrix C);
  public Matrix transpose();
```

```
  ...
}
```

These operations have certain well-defined requirements. Two matrices, A and B, may only be added if they have the same dimensions, i.e. A has same number of rows and columns as B. Two matrices, A and B, may be multiplied and placed into C if the number of columns of A equals the number of rows of B. The dimensions of C must be equal to the number of rows of A and the number of columns of B. Transposition of a matrix swaps the row and column dimensions. These rules are violated by the following code:

```
Matrix m = new DenseMatrix(5,4);
Matrix n = new DenseMatrix(4,6), z = new DenseMatrix(5,6),
      w = new DenseMatrix(3,5);
m.mult(n,z); z.transpose(); z.mult(m,w); // m and w incompatible
```

In the above example, all matrix dimensions are compatible with respect to the first two operations but not for the final expression `z.mult(m,w)`. The matrix operations in MTJ will verify dimensions before calculating and throw exceptions if the preconditions are not met. Performance-wise, this is costly and latent mismatches may lurk in seldom used code.

To alleviate this problem a totem propagator may be applied which knows how to propagate and verify the dimension of matrix operations. Initial dimensions can be picked up from programmer-supplied assertions (in the form of an assert statement, `assert Matrix.dimensions(m,4,3)`) or from the variable initialisation. Whenever a dimension is asserted for a variable in the code, a new, dynamic rule `Dimensions:` *name* -> *dim* is created that remembers the asserted dimensions *dim* for a variable *name*. If an existing rule for the variable already exists, it is updated. This rule can then be applied (and updated) when propagating the dimension totem across a transposition:

```
PropTotem =
   ?MethodInvocation(src, SimpleName("transpose"), _, [])
  ; <type-of ; dotted-name-of> src ⇒ "no.uib.cipr.matrix.Matrix"
  ; <Dimensions> src ⇒ [rows, columns]
  ; rules(Dimensions : src → [columns, rows])
```

Here, the old dimensions (if they are known) will be swapped and the `Dimensions` rule updated. There are other (overloaded) `PropTotem` rules which deal with addition and multiplication. The propagator core is based on the general constant propagation framework proposed by Olmos and Visser [OV05] but is adapted to allow propagating arbitrary data properties and not just constants:

```
prop-totem =
    PropTotem
```

```
<+ prop-totem-vardecl
<+ prop-totem-assign
...
<+ all(prop-totem)
```

The `prop-totem` strategy should be applied to a method body where it will recurse through the subterms. At each term, a series of strategies is tried in order. If all fail, the recursion continues into the children of the current term. The first strategy applied is `PropTotem`. This is a set of overloaded rules for the `add`, `mult` and `transpose` cases. The rule with the matching pattern will be applied. If none of the rules succeed, the current term is not method call to `add`, `mult` or `transpose`. In this case, the `prop-totem` strategy continues by calling the `prop-totem-vardecl` strategy. This will try to infer totems from variable declaration terms. If the current term is an assignment ($v = e$), the totem of e is inherited by v. This is handled by `prop-totem-assign`. Additional cases deal with control flow constructs like `if` and `while`, as described in [OV05]. These extra cases may retroactively be added to the algorithm using the aspect mechanism described in Chapter 5.

Once the correctness of the dimensions can be guaranteed, based on the user assertions and propagation, the runtime dimension checks can be removed by source code transformation.

10.3.3 Domain-specific Source Code Transformations

Results of analyses can be used to perform source code transformations either as part of the compilation process or as refactorings on the source code. Such code transformations could aid in framework migration and may perform pervasive style changes or remove code smells.

Optimising Matrix Dimension Checks Using totem propagation described previously, matrix operations can be rewritten to remove runtime dimension checks, provided that the matrix dimensions can be determined statically (at compile time) to be correct. In that case, the following substitution is applicable:

```
A.mult(B,C) -> A.uncheckedMult(B,C)
```

The following rewrite rule captures the necessary conditions and can be plugged directly into the totem propagator to achieve a correct substitution:

```
PropTotem:
   MethodInvocation(src1, SimpleName("mult"), x, [src2, dst])
 → MethodInvocation(src1, SimpleName("uncheckedMult"), x, [src2, dst])
 where
   <type-of ; name-of> dst ⇒ "no.uib.cipr.matrix.Matrix"
 ; <Dimensions> src1 ⇒ (s1r, s1c) ; <Dimensions> src2 ⇒ (s2r, s2c)
```

```
; <Dimensions> dst ⇒ (dr, dc); !s1c ⇒ s2r; !s2c ⇒ dc; !s1r ⇒ dr
```

The `where` clause is a rewriting condition which ensures that the `mult` call is on the correct data type and that the dimensions are compatible.

Optimising Loop Boundary Checks The bounds checking idiom from the previous section can also be turned into a code transformation:

```
OptimizeFor:
   ForStatement(init, cond, incr, body)
 → Block(<concat> [vdecls, [ ForStatement(init, cond', incr, body) ]])
 where
    <collect(is-immutable-call) ; new-names> cond ⇒ call-var-pairs
  ; <map(\(e, v) → vardecl(<type-of> e, v, e)\)> call-var-pairs
      ⇒ vdecls
  ; <bottomup(try(RewriteImmutable(|vars)))> cond ⇒ cond'
```

The generic `collect` strategy is used with `is-immutable` to find all invocation of get-like methods in the condition expression. For each expression, a new uniquely named variable is created (by `new-names`) and a variable declaration for it is created that gets added before the `for` loop. Each expression is replaced with its corresponding, freshly named, temporary variable using the `RewriteImmutable` strategy. This avoids name capture in the generated code.

10.4 Implementation

The analysis framework presented in this chapter reuses the Eclipse Compiler for Java via a POM adapter. The compiler is available as a plugin for the Eclipse development platform [Ecl]. Although most users encounter Eclipse as a graphical application, it is possible to create so-called headless applications using the Eclipse infrastructure that have no graphical interface. The command line application depicted in Figure 10.1 is an example of a headless application.

10.4.1 Analysis Architecture

The principal components of the transformation framework are shown in Figure 10.3 The plugin `org.spoofax.eclipsetrafo` contains both the application which can be invoked from the command-line and a plugin class which may be used as part of a graphical application.

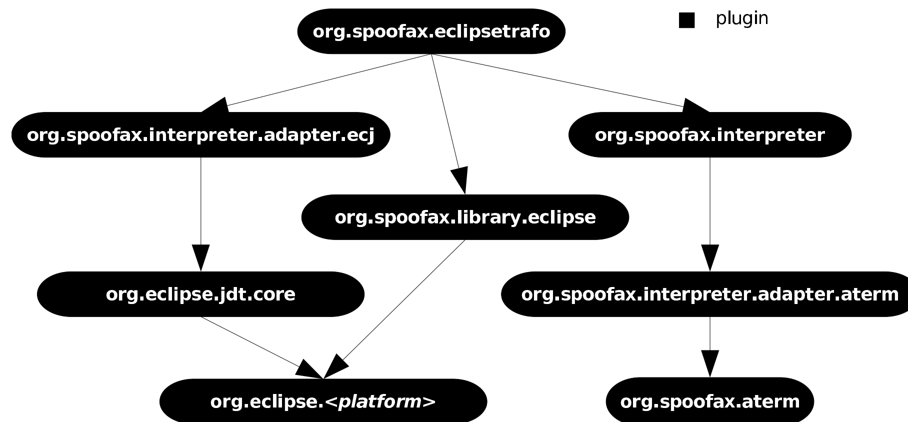

Figure 10.3: Principal components of the analysis and transformation framwork introduced in this chapter. The `org.eclipse.jdt.core` provides the Java compiler, and the `org.eclipse.<platform>` represents the collection of plugins that comprise the Eclipse runtime and necessary support plugins for the Java compiler.

10.4.2 Transformlet Repositories

The analysis and transformation scripts shown in the previous section can be compiled into standalone transformlets. This facilitates sharing of analyses and transformations between developers. Since transformlets are fully self-contained, it is easy to upload them into online repositories and distribute them to other users. However, there is currently no easy-to-use graphical interface for subscribing to repositories and downloading transformlets, but simple command-line tools are available.

10.5 Related work

Programmable static analysis tools, such as CodeQuest [HVdMdV05], PQL [MLL05], and CodeSurfer [AT01], all support writing various kinds of flow- and/or context-sensitive program analyses, in addition to (sometimes limited) queries on the AST. Pluggable type systems, an implementation of which is described by Andreae et al [ANMM06], also offer static analysis capabilities. Developers can express custom type checking rules on the AST that are executed at compile-time so as to extend the compiler type checking. Programmable analysis frameworks like PMD [Cop05] provides a good collection of standard analyses that are specific to a given programming language, in this case Java, and where the user can implement additional ones using an analysis API. Neither programmable static analysis tools nor pluggable type systems support source code transformations, however.

Languages for refactoring such as JunGL [VEdM06] and ConTraCT [KK04] provide both program analysis and rewriting capabilities. JunGL is hybrid between an

ML-like language (for rewriting) and Datalog (for data-flow queries) whereas ConTraCT is based on Prolog. JunGL supports rewriting on both trees and graphs, but is a young language and does not (yet) support user-defined data types. Stratego is a comparatively mature program transformation language with sizable libraries and built-in language constructs for data- and control-flow analysis, handling scoping and variable bindings, and pattern matching with concrete syntax (not demonstrated in this chapter). It comes with both a compiler and interpreter and has been applied to processing various other mainstream languages such as C and C++ [BDD06].

Open compilers, such as the SUIF [WFW+94] project, Polyglot [NCM03], OpenJava [TCIK00], and OpenC++ [Chi95], offer extensible language processing platforms. In many open compilers, the entire compiler pipeline, including the backend, is extensible. Constructing and maintaining such an open architecture is an arduous task. As demonstrated in this chapter, many interesting classes of domain-specific analyses and transformations require only the front-end to be open. Exposing just the front-end is less demanding than maintaining a fully open compiler pipeline. Transformation systems may be plugged into either of these open compilers using the POM adapter technique.

The research on active libraries [VG98] has largely focused on performance optimisation for example using pre-processors and library annotations [GL00, Kal03]. Compiler scripts are useful for exploring other topics of library design such implementation consistency, contract checking and sensible idiom or pattern usage.

10.6 Discussion

Program analysis tools have a long history, preceding even the venerable `lint` [Joh78]. Recent research has to a certain extent focused on scriptable frameworks for expressing extensible analysis tools. Scripts allow developers to adapt, for example, pluggable type systems, style checkers and static analysis to their specific frameworks or libraries.

The appealing feature of the system described in this chapter, and that of JunGL and ConTraCT, is that in addition to scripting syntax-, type- and flow-based analyses, it also allows scripting of source code transformations based on the analysis results. Transformation languages are good candidates for compiler scripting languages. New transformations and analyses may be expressed quickly due to their high-level domain-specific language constructs. This makes them an appealing part of a testbed for prototyping language extensions as well as new compiler analyses and optimisations.

The plethora of custom analysis and transformation tools suggests that compiler writers should cater for potential extenders in their infrastructure design. As this chapter shows, even the rather simple and minimal inspection interface of the POM adapter is sufficient for expressing powerful program analyses. General code transfor-

mation can be scripted if functionality for building AST nodes is also exposed by the compiler.

A limitation of ECJ, and of many other compilers, is that rewriting the AST will invalidate the type information. Type analysis is often done during or just after parsing. The typical usage scenarios for ASTs inside compilers do not make incremental reanalysis of types necessary. Incremental reanalysis would be very useful for POM clients that perform rewriting. Without such support, complete type reanalysis must be performed to restore accurate type information. This often involves unparsing and subsequent reparsing of text.

Stratego does not have any fundamental limitations on the types of analyses and transformations it can express. The language is Turing-complete, and can express both imperative and functional algorithms for program analysis and transformation. Special support exists, in the form of reusable strategy libraries and language constructs such as dynamic rules, for performing control- and data-flow analysis over subject programs represented as terms, i.e. abstract syntax trees. Refer to [OV05] for more details on these features. In practise, the current performance of the interpreter may be a limiting factor for particularly resource-intensive analyses and transformations. In these cases, the C-based Stratego/XT infrastructure [BKVV06] may be an alternative. Certain whole-program analyses may require very efficient implementations of specific data structures, such as binary decision diagrams (BDDs). Stratego does not currently have a library providing BDDs.

10.7 Summary

This chapter presented a powerful framework for scripting domain-specific analyses and transformations for Java based on the Stratego rewriting language and the Eclipse Compiler for Java. The examples, all taken from what is considered to be mature and well-designed frameworks, illustrate the usefulness of the domain-specific abstractions for program analysis and transformation provided by the MetaStratego system. The framework was made possible in particular by the program object model adapter technique (described in Chapter 4) which enabled the quick and large scale reuse of the Eclipse compiler front-end. The fusion between the Stratego/J runtime and the ECJ compiler is very efficient: it can apply the bounds checking idiom to around 2.7 million lines of Java code in just over four minutes on a low-end laptop.