Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2008) 21–36
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Fusing a Transformation Language with an Open Compiler

Karl Trygve Kalleberg [1]

Department of Informatics, University of Bergen, P.O. Box 7800, N-5020 BERGEN, Norway

Eelco Visser [2]

Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, The Netherlands

Abstract

Program transformation systems provide powerful analysis and transformation frameworks as well as concise languages for language processing, but instantiating them for every subject language is an arduous task, most often resulting in half-completed frontends. Compilers provide mature frontends with robust parsers and type checkers, but solving language processing problems in general-purpose languages without transformation libraries is tedious. Reusing these frontends with existing transformation systems is therefore attractive. However, for this reuse to be optimal, the functional logic found in the frontend should be exposed to the transformation system – simple data serialization of the abstract syntax tree is not enough, since this fails to expose important compiler functionality, such as import graphs, symbol tables and the type checker.
In this paper, we introduce a novel and general technique for combining term-based transformation systems with existing language frontends. The technique is presented in the context of a scriptable analysis and transformation framework for Java built on top of the Eclipse Java compiler. The framework consists of an adapter automatically extracted from the abstract syntax tree of the compiler and an interpreter for the Stratego program transformation language. The adapter allows the Stratego interpreter to rewrite directly on the compiler AST. We illustrate the applicability of our system with scripts written in Stratego that perform framework and library-specific analyses and transformations.

Keywords: compiler scripting; strategic programming; program transformation

1 Introduction

Developing and maintaining frameworks and libraries is at the core of software development: all domain abstractions of software applications are invariably encoded into libraries of a given programming language. Maintenance of this code involves various language processing tools such as compilers, editors, source code navigators,

[1] Email: karltk@ii.uib.no

[2] Email: visser@acm.org

doi:10.1016/j.entcs.2008.03.042

documentation generators, style checkers and static analysis tools. Unfortunately, most of these tools only have a fixed repertoire of functionality which seldom covers all the needs of the developer of a given library or framework. Relatively few processing tools can quickly and easily be programmed, extended or adapted by the library developer. This often drives developers to implement many additional, text-based tools from scratch. A preferable solution would be for library developers to quickly write custom scripts in a suitable scripting language and thus implement analyses and transformations specific to their own code bases, such as style checking and library-specific optimizations. Domain-specific languages (DSLs) for program analysis and transformations are attractive candidates for expressing these scripts, since DSLs allow precise and concise formulations. However, the DSLs are rarely coupled with robust and mature parsers and type analyzers. Open compilers are also attractive because they provide solid parsers and type analyses, but implementing analyses and transformation in their general-purpose languages is often very time-consuming.

In this paper, we obtain the best of both worlds by combining Stratego, a DSL for program transformation and the open Eclipse Compiler for Java (ECJ), using a program object model (POM) adapter. The POM adapter welds together the Stratego runtime and the ECJ abstract syntax tree (AST) by translating Stratego rewriting operations on-the-fly to suitable method calls on the AST API. This obviates the need for data serialization. The technique can be applied to many tree-like APIs, and is reusable for other rewriting systems. Using the POM adapter, Stratego becomes a compiler scripting language, offering its powerful features for analysis and transformation such as pattern matching, rewrite rules, generic tree traversals, and a reusable library of generic transformation functions and data-flow analysis. This combination is a powerful platform for programming domain-specific analyses and transformations. We argue that the system can be wielded by advanced developers and framework providers because large and interesting classes of domain-specific analyses and transformations can be expressed by reusing the transformation libraries provided with Stratego.

The contributions of this paper include the fusing of a DSL for language processing with an open compiler without resorting to data serialization. This brings the analysis and transformation capabilities of modern compiler infrastructure into the hands of advanced developers through a convenient and feature-rich transformation language. The technique is reusable for other transformation languages. It may 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 like 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. We validate the system's applicability through a series of examples taken from mature and well-designed applications and frameworks.

The remainder of this paper is organized as follows: In Sec. 2, we discuss the POM adapter and how it connects Stratego with ECJ. In Sec. 3, we show the practical applicability of our prototype on a series of common, framework-specific analysis and transformation problems. In Sec. 4, we discuss the implementation details of our prototype. In Sec. 5, we cover related work. In Sec. 6, we discuss some trade-offs related to our technique before we conclude in Sec. 7.

2 The Program Object Model Adapter

The program object model adapter is the linchpin in the composition of the compiler and the program transformation language. A *program object model* (POM) is our name for the object model representing a program in 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 method calls on the POM API.

Consider Fig. 2 which shows the principal components of our system. At the bottom, the ECJ provides an AST API for modifying and inspecting its internal program object model. The AST is implemented in a traditional 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 can be constructed using methods, such as `newCompilationUnit()`, in the `AST` factory.

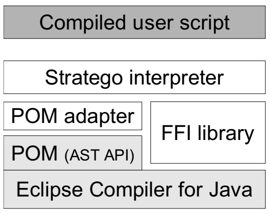

Fig 2: Architecture.

The Stratego interpreter is a rewriting engine, or runtime, for the Stratego term rewriting language (introduced below), written in Java. It executes scripts compiled to an abstract machine. The crucial feature of the interpreter is that it abstracts over the actual term implementation. Any data structure that can provide a suitable interface may be treated as terms and be rewritten. The job of the POM adapter is to adapt tree-like data structures so that they can be transformed with Stratego. 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 that are executed directly on the POM, without any intermediate data serialization.

The interpreter also has a facility for calling foreign functions, i.e. functions implemented in Java. The interface between Stratego and ECJ includes a small foreign function interface (FFI) that exposes parts of the native Eclipse AST API as Stratego library functions. These allow Stratego scripts to ask for the type of a suitable node using `type-of`, the supertype using `supertype-of`, and more.

Our prototype system is available as a stand-alone, command-line application based on Eclipse, and as a reusable Eclipse plugin. In stand-alone mode, the system performs source-to-source transformation. The user supplies the path of a project and a script to execute. The scripts use the FFI to traverse the project directories and to parse source files, to obtain their AST. After rewriting, the scripts use the FFI to write modified ASTs back to disk, as source code. In plugin mode, interpreter objects may be instantiated with arbitrary scripts. Scripts are executed directly on individual ASTs by calling execute methods on the interpreter object. This allows scripts to be used for very fine-grained source code queries and transformations inside the Eclipse environment

2.1 Scripting in Stratego

Stratego is a DSL for language processing based on the paradigm of strategic term rewriting. In our context, terms are essential equivalent to ASTs. Stratego has language constructs that make it well-suited for language processing, such as pattern matching, generic tree traversals, rewrite rules and powerful combinators for expressing strategies for rewriting and analysis. Essential constructs of Stratego are explained below.

Patterns are written using prefix notation on the form `SimpleName("b")`, and can contain variables, e.g. `SimpleName(n)`, where `n` is a term variable. A term is a pattern that does not contain variables. Lists are written as `[1,2,3]`. Terms are built (instantiated) from patterns using the build operator (`!`): `!MethodInvocation(obj, name, [], [])`, where `obj` and `name` are variables. Operators are applied to an implicit *current term*; a build replaces the current term. Patterns are matched against terms using the match operator (`?`): `?SimpleName(x)`, binds the term variable `x` to the subterm of `SimpleName`. Matching fails if the pattern does not match, i.e. the current term is not a `SimpleName` term.

Strategy expressions combine basic operations (such as match) into more complex transformations. Since match can fail, strategy expressions may fail as well. Combinators are used to compose expressions and handle failures. The choice combinator s_0 `<+` s_1 evaluates the expression s_0. If s_0 succeeds, the (potentially) new current term is kept, and s_1 is skipped. If s_0 fails, the current term is restored, and s_1 is evaluated instead. If s_1 fails, the combination fails. The sequence combinator s_0 `;` s_1 first evaluates s_0 then s_1. If either fails, the combination fails. The `fail` and `id` operators leave the current term untouched; `fail` always fails and `id` always succeeds.

The *primitive traversal operators* `one`, `some` and `all` are used to traverse terms by local navigation into subterms. `all(`s`)` applies the expression s to each subterm of the current term, potentially rewriting each. `all(`s`)` succeeds iff s succeeds for all subterms, e.g. `all(!1)` applied to the term `[1,2,3]` gives `[1,1,1]`. `one(`s`)` and `some` are similar, and applies s non-deterministically to exactly one or as many as possible but at least one subterm, respectively. Both fail iff s never succeeded.

Strategies may be named and parametrized, e.g. `try(s) = s <+ id` defines a strategy `try(s)` that attempts to apply s, and defaults to `id` if s fails. Generic traversal strategies can be built from the primitive traversal operations, e.g. `bottomup(s) = all(bottomup(s)); s` and `topdown(s) = s; all(topdown(s))`.

A rewrite rule `R:` $p_l(x)$ `->` $p_r(x)$ with name R, left-hand side pattern $p_l(x)$, and right-hand side pattern $p_r(x)$, x symbolizing term variables, is syntactic sugar for `R = ?`$p_l(x)$ `; !`$p_r(x)$. A `where(`s`)`-clause temporarily saves the current term, applies s to it, then restores the current term. The clause fails iff s fails. `wheres` are typically used to express rule conditions, as shown later.

Basic Stratego Constructs

Strategy Expression	Meaning
`!`$p(x)$	*(build)* Instantiate the term pattern $p(x)$ and make it the current term
`?`$p(x)$	*(match)* Match the term pattern $p(x)$ against the current term
s_0 `<+` s_1	*(left choice)* Apply s_0. If s_0 fails, roll back, then apply s_1
s_0 `;` s_1	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`some(`s`)`	Apply s to as many direct subterms of the current term as possible
`all(`s`)`	Apply s to all direct subterms of the current subterm

Syntactic Sugar

Strategy Expression	Meaning — *(syntacic sugar)*
`\`$p_l(x)$ `->` $p_r(x)$`\`	Anonymous rewrite rule from term pattern $p_l(x)$ to $p_r(x)$
`?`x`@`$p(y)$	Equivalent to `?`x `;` `?`$p(y)$; bind current term to x then match $p(y)$
`<`s`>` $p(x)$	Equivalent to `!`$p(x)$ `;` s; build $p(x)$ then apply s
s `=>` $p(x)$	Equivalent to s `;` `?`$p(x)$; match $p(x)$ on result of s

3 Domain-specific Analyses and Transformations

In this section, we motivate the applicability of our system by showing some framework-specific analyses and transformations. The examples in this section illustrate 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 [15,4,10] for some examples.

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. We are concerned with checking for proper implementation and proper use of domain abstractions. Consistency of implementation may be improved by encouraging systematic use of particular idioms. The following idiom is taken from the AST implementation in ECJ.

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

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

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

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

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

```
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")
```

`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 kept in the global variable `immutable-classes`[3].

With data-flow analysis, we could even consider method calls on objects which are not immutable; 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 function, we can assume immutability. By keeping (*typename*,*methodname*) pairs in an list, say `immutable-methods`, we can look up the immutability property.

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 [10]. 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 dataflow analysis for verifying the operations performed on variables obey the typestate rules of the language" [16]. 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 optimizing 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. The following interface is found in the Matrix Toolkits for Java (MTJ) library [1]:

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

[3] Technically, `immutable-classes` is a Stratego overlay, but this amounts to a global, immutable variable in our case.

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
```

All dimensions are compatible for the first two operations, but not for the final `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, we can apply a totem propagator which knows how to propagate and verify the dimension of matrix operations. Initial dimensions can be picked up from programmer-supplied assertions (on the form of a comment `// @dim(m,4,3)`) or from the variable initialization. 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*. Dynamic rules are like normal rewrite rules, except they can be introduced, updated and removed at runtime. If an existing `Dimensions` rule with the *name* left-hand side already exists, it is updated to a (potentially) new *dim*. 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` functions which deal with addition and multiplication. The propagator core is based on the general constant propagation framework proposed by Olmos and Visser [15], but is adapted to propagate arbitrary data properties, not just constants:

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

The `prop-totem` function should be applied to a method body where it will recurse through the subnodes. At each node, a series of functions is tried, in order. If all fail, the recursion continues into the children of the current node. The first function applied is `PropTotem`. This is a set of overloaded functions, for the `add`, `mult` and `transpose` cases; the one with the matching pattern will be executed. If none succeed, i.e., we are not at a method call to `add`, `mult` or `transpose`, we continue by trying the `prop-totem-vardecl` function (s_0 `<+` s_1 means evaluate s_0, then evaluate s_1 iff s_0 failed). This will try to infer totems from variable declaration nodes. If we are at an assignment node ($v = e$), 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 [15]. Once we can guarantee, based on the user assertions and propagation, that the dimensions are correct, we can remove the runtime dimension checks by source code transformation.

3.3 Domain-specific Source Code Transformations

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

Optimizing Matrix Dimension Checks

Using totem propagation described previously, we can rewrite matrix operations to remove runtime dimension checks when we can statically determine that the matrix dimensions are correct, e.g.:

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

The following rewrite rule can be plugged directly into the totem propagator to achieve such a transformation:

```
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. This rewrite rule is all that is needed to turn the analysis from Sec. 3.2 into an optimizing code transformation.

Optimizing 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` function 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` function, thus avoiding any name capture in the generated code.

4 Implementation

The ECJ AST is a class hierarchy consisting of abstract and concrete classes. For example, all expression nodes, such as `InfixExpression`, inherit from the abstract `Expression` class. The root node of the hierarchy is the abstract class `ASTNode`. The AST hierarchy is adapted to the term interface expected by the rewriting engine using the POM adapter.

4.1 Term Interface

The term interface is a generalization of the ATerm interface used by various term rewriting systems, such as ASF+SDF [18], Tom [13] and Stratego [4]. There are two levels to this interface, depending on whether read-only traversals or full rewriting is desired.

Inspection Interface

The inspection interface is a class hierarchy. At its root we find `ITerm`. There are four distinct *primitive term types* deriving from `ITerm`, for integers, strings, lists and applications. The essential methods of `ITerm` are given below.

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

The `getPrimitiveTermType()` method returns an integer specifying which primitive term type is represented by a given `ITerm` object. Most AST nodes are application nodes. An application C(t_0, ..., t_n) consists of a constructor name C and a list of subterms t_0 through t_n. The number and types of the subterms are given together with the constructor name in a signature, e.g.

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

This declares to Stratego that `InfixExpression` terms have three children, the first being a string and the remaining two being expressions. The declaration corresponds to the AST class `InfixExpression`. For each concrete AST node type, a constructor is generated. For each abstract AST node type, a sort is generated. The sets of constructors and sorts define the `EclipseJava` signature. Calling `getConstructor()` on an `InfixExpression` returns an object that can be queried for the constructor name (in this case `InfixExpression`), and arity (in this case three). Calling `getSubtermCount()` returns three and the method `getSubterm()` can be used to retrieve either of the subterms. The `isEqual()` method performs a deep equality check. Stratego allows pattern matching with variables. All the code for handling variable bindings is kept inside the interpreter, to keep the POM adapter interface minimal.

Concrete implementations of the `ITerm` inspection interface can be derived mostly automatically from the AST class hierarchy. Each concrete class in the ECJ AST requires a small adapter class, all of it generated boilerplate. The only place where human intervention is needed is to decide how the subtrees in the AST should map to an ordered set of terms, e.g:

```
class WrappedPackageDeclaration implements ITermAppl {
  private PackageDeclaration wrappee;
  ...
  public ITerm getSubterm(int index) {
    switch(index) {
    case 0: return ECJFactory.wrap(wrappee.getPackage());
    case 1: return ECJFactory.wrap(wrappee.imports());
    case 2: return ECJFactory.wrap(wrappee.types());
    } throw new ArrayIndexOutOfBoundsException();
  }
}
```

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.

Generation Interface

The POM adapter technique does not require an implementation of the generation interface, but if one is not provided, rewriting cannot be done (only analysis is possible). The following are the essential factory methods that must be provided.

```
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. Only the `makeAppl` method must provided. In our prototype, this method forwards constructor requests to the appropriate factory methods of the ECJ AST; when it sees a request for constructing, say, a `PackageDeclaration` node, the request is forwarded to `newPackageDeclaration()` of the ECJ `AST` factory.

```
class ECJFactory implements ITermFactory { ...
  public ITerm makeAppl(ITermConstructor ctor, ITerm[] args) {
    switch(constructorMap.get(ctor.getName())) {
      case PACKAGE_DECLARATION: {
        if((!isJavadoc(kids[0]) && !isNone(kids[0]))
            || !isAnnotations(kids[1])
            || !isName(kids[2]))
          return null;

        PackageDeclaration pd = ast.newPackageDeclaration();
        if(isNone(kids[0]))
          pd.setJavadoc(null);
        else
          pd.setJavadoc(getJavadoc(kids[0]));
        pd.annotations().addAll(getAnnotations(kids[1]));
        pd.setName(asName(kids[2]));
        return wrap(pd);
      }
      ...
    }
    ...
   }
23  }
```

Before the factory method is invoked (line 10), the type correctness of all children are checked (lines 5–7). If this fails, an error is signaled by returning `null` (line 8). The `EclipseJava` signature completely declares the structure of legal terms that `ECJFactory` should allow, so all of this code may be automatically generated. We discuss below how "incorrect" constructions are handled using mixed terms.

Once all children are verified as appropriate, a new `PackageDeclaration` AST node is created and initialized (lines 10–16). Finally, it is wrapped as a term and returned (line 17). The use of a constructor map for `switch` on line 3 is a performance trick for mapping constructor names to constructor methods.

4.2 Design Considerations

Functional Integration – The type analysis functions, such as `type-of`, are calls to the ECJ type checker, through the FFI library in Fig. 2. For example, invoking `type-of` on an `InfixExpression` term t results in a call to `resolveTypeBinding()` defined in the class `InfixExpression` on the object wrapped by t. Stratego is dynamically 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` detects this and fails, just like any expression in Stratego may 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 are never modified – there simply are no operations for modifying existing terms. This makes wrapping imperative data structures in such a functional dress relatively straightforward. The compiler need not provide one. The only restriction is that AST nodes must not change behind the scenes, i.e. the rewriting engine must have exclusive access while rewriting. For in-place rewriting systems, e.g. TXL [6], a slight modification of the `ITerm` interface would be necessary so that subterms of existing terms can be modified in place.

Efficiency Considerations – Using a functional data structure provides some appealing properties for term comparison and copying. As described in [7], 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 because 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. This may be provided in the POM adapter, but then lazy wrapping must be given up.

Hash codes should always be computed deeply. The hash code must be computed from the structure of the term – not the object identity of the AST node – since the equality is structural (two objects that are equal should have the same hash code). Once a hash code has been computed, it may be memoized, since the subterms can never change.

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 to run the bounds checking idiom analysis on the entire Eclipse code base (about 2.7 million lines of code), on a 1.4GHz laptop with 1.5GB of RAM. 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.

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(0),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(0),3)` fails. Terms without any `EclipseJava` terms, such as `(1,2,3)`, may be built freely. These are not represented as `EclipseJava` terms, but by the default internal term library of the interpreter. We call these terms without `EclipseJava` constructors *basic terms.*

In lenient mode, mixed terms consisting of basic and `EclipseJava` terms are allowed, such as `InfixExpression(1,BooleanLiteral(0),3)`. The subterm `BooleanLiteral` remains an `EclipseJava` term, but `0` 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 `makeAppl()` method when an `EclipseJava` term can be built: iff 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 fail if they are passed mixed terms. Java programs, such as Eclipse plugins, using the Stratego interpreter to rewrite ASTs receive an `ITerm` as the result from the interpreter. They should perform a dynamic type check to ensure that the `ITerm` is a wrapped ECJ AST node, and not a mixed or basic term.

Rewritings can result in structurally valid but semantically invalid ASTs, for example by removing a method which is called elsewhere from a class. 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 precondition checks, it is possible to ensure that well-written transformations always yield type-correct results. A post-condition check is computationally worse, since a full reparse is required – the ECJ type analyzer is designed to compute type information only during parsing, i.e. when constructing the initial AST from source code.

5 Related work

Language processing is what program transformation systems like Tom [13], TXL [6], ASF+SDF [18], Stratego [4] were designed for.

Programmable static analysis tools such as CodeQuest [9], CodeSurfer [2] and PQL [12], all support writing various kinds of flow- and/or context-sensitive program analyses, in addition to (often limited) queries on the AST. Pluggable type systems, an implementation of which is described by Andreae et al [3], also offer static analysis capabilities. Developers may express custom type checking rules on the AST. These are executed at compile-time so as to extend the compiler type checking. Neither programmable static analysis tools nor pluggable type systems support source code *transformations*, however.

Languages for refactoring such as JunGL [20] and ConTraCT [11] 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 paper) that comes with both a compiler and interpreter, and has been applied to processing various other mainstream languages such as C and C++ [4].

Open compilers such as SUIF [21], OpenJava [17], OpenC++ [5] and Polyglot [14] offer extensible language processing platforms, and in many open compilers, the entire compiler pipeline is extensible, including the backend. Constructing and maintaining such an open architecture is a laborious task. As we have shown, 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. In principle, we could have plugged Stratego into either of these 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 [19] that also provides generic traversals. Generic traversals and visitor combinators go far beyond traditional object-oriented visitors, and the core interface required by both approaches is very similar. Comparing the `Visitable` interface of JJTraveler, the ATerm interface 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 our interpreter).

A related approach to rewriting on existing class hierarchies is presented in Tom [13]. Tom is a language extension for Java that provides features for rewriting and matching on existing class hierarchies. Recent versions also support generic traversals in the style of JJTraveler, but its library of analyses is still rather small.

It works by adding a new match construct to the Java language that is expanded by the Tom pre-processor into Java method calls. A generator, Gom, is available for generating classes that implement term structures. These are specified algebraically, much like the signatures of Stratego.

High-level analyses are also provided by Engler et al [8], where a system for checking system-specific programming rules for the Linux kernel is described. These rules rules are implemented as extensions to an open compiler. Our system is different in that it can also perform arbitrary code transformation, and that the language we use to implement our rules is a feature-rich transformation language designed for language processing. For language processing problems, Stratego has the advantage of a sizable library of generic transformations, traversals and high-level data-flow analysis, in addition to its novel language features. The net result is that transformation code becomes both precise and concise.

6 Discussion

Recent research has provided pluggable type systems, style checkers and static analysis with scripting support. The appealing feature of our system, and that of JunGL and ConTraCT, is that we can also script source code transformations based on the analysis results. The tradeoff with using a domain-specific language for scripting is that the same language features that make the language powerful and domain-specific also make it more difficult to learn. This may be offset in part by good documentation, and a sizable corpus of similar code to learn from.

A compiler scripting language may also provide an appealing part of a testbed for prototyping language extensions, new compiler analyses and transformations because its high-level constructs support rapid prototyping. The plethora of custom analysis and transformation tools suggests that compiler writers should cater for potential extenders in their infrastructure design. As we have demonstrated, even a rather simple inspection interface is sufficient for read-only analysis. By adding functionality for building AST nodes, general rewriting may be scripted. The POM adapter presented in this paper was generated automatically using source code analysis techniques over the AST classes of existing frontends. We are currently experimenting with generalizing and improving our tools for automatic generation of POM adapters. It is being tested against other frontends such as the reference Java compiler from Sun, the Polyglot compiler [14] and various C/C++ frontends.

A limitation of ECJ is that rewriting the AST will invalidate the type information. After rewriting, complete type reanalysis must be performed to restore accurate type information. An open compiler with incremental type reanalysis would help in ensuring that the transformation is semantically correct, as “safe points” can defined in the transformation where the (intermediate) result is checked for type-correctness.

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. Please refer to [15] for more details on these features. In practice, 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 [4] may be an alternative. In the future, we anticipate a Java bytecode backend for the Stratego compiler. 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.

7 Conclusion

We have presented the design of the program object model adapter, a general technique for fusing program transformation systems with existing language infrastructures, such as compilers and frontends, without resorting to data serialization. This allows the transformation system to rewrite directly on the program object models (e.g. ASTs) of the language frontend. The applicability of the technique was illustrated through the discussion of a prototype framework composed of the Stratego rewriting language and the Eclipse Compiler for Java. The framework yields a powerful solution for scripting domain-specific analyses and transformations due to the stability of the Eclipse compiler and the features of Stratego – the analyses and transformations are expressed precisely and concisely. We have shown that even a relatively small degree of extensibility on the part of the compiler is sufficient for plugging in a rewriting system, 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. The usefulness of the framework was demonstrated through a series of analysis and transformation problems taken from mature and well-designed frameworks.

References

[1] Matrix Toolkits for Java. http://rs.cipr.uib.no/mtj/,2006.

[2] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of WISE'01 (Itl Workshop on Inspection in Software Engineering)*, 2001.

[3] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of OOPLSA'06: Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2006. ACM Press.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[5] S. Chiba. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[6] J. R. Cordy. TXL - a language for programming language tools and applications. *ENTCS*, 110:3–31, 2004.

[7] M. G. T. V. den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[8] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

[9] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. CodeQuest: querying source code with datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM Press.

[10] K. T. Kalleberg. User-configurable, high-level transformations with CodeBoost. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[11] G. Kniesel and H. Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[12] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proc. of the ACM SIGPLAN Conf. Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[13] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *12th International Conference on Compiler Construction*, LNCS, pages 61–76. Springer, 2003.

[14] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, LNCS, pages 138–152. Springer, 2003.

[15] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, April 2005.

[16] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Softw. Eng.*, 19(5):478–485, 1993.

[17] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag.

[18] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.

[19] A. van Deursen and J. Visser. Building program understanding tools using visitor combinators. In *Proceedings 10th Int. Workshop on Program Comprehension, IWPC 2002*, pages 137–146. IEEE Computer Society, 2002.

[20] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[21] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.