

Generative Programming Techniques for Java Library Migration^{*}

Victor L. Winter Azamat Mametjanov

University of Nebraska at Omaha
{vwinter,amametjanov}@mail.unomaha.edu

Abstract

Embedded systems can be viewed as scaled-down versions of their stand-alone counterparts. In many cases, the software abstractions and libraries for embedded systems can be derived from libraries for stand-alone systems. One such example is the Java library for Java Virtual Machines. An embedded system does not always support all features as in the case of an embedded JVM that does not support floating-point operations. In such cases, an existing library needs to be migrated to the embedded platform. Libraries are large collections of code and manual migration is a daunting task. In this paper, we provide an automated approach to the library migration problem using program transformations. The solution developed in this paper enables rapid adaptation and re-targeting of Java libraries in the presence of evolving libraries and evolving embedded platforms.

Categories and Subject Descriptors D.2.2 [Software Engineering]: Design Tools and Techniques—Software libraries

General Terms Algorithms, Languages

Keywords Java Libraries, Program Transformation, Strategic Programming, HATS, TL

1. Introduction

Virtually all microprocessors built today are targeted for use in embedded systems. In the year 2000, approximately eight billion microprocessors were built, of which 98% were used in embedded systems [8]. Due to the advances in computing power, the functionality expected from microprocessors in embedded contexts continues to increase. At present, embedded systems span a broad spectrum of increasingly complex applications ranging from medical applications, weapons systems, space borne systems to flight control systems. This rise in application complexity has created an attendant demand for high-level language support for embedded systems programming.

^{*}This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPCE '07 October 1–3, 2007, Salzburg, Austria.
Copyright © 2007 ACM 978-1-59593-855-8/07/0010...\$5.00

A hardware realization of the Java Virtual Machine (JVM) such as the Java Card technology [10] alleviates the complexity burden of embedded systems programming by enabling high-level Java programs to run on embedded systems. Developers can now build programs using standard software development tools and convert them into a form acceptable by an embedded JVM implementation (e.g. smart cards). The converted program is then executed by an embedded JVM natively.

Hardware realizations of the JVM, however, only partially address the high-level language support problem associated with embedded systems programming. While it is true that programmers can now develop embedded applications directly in Java rather than in some low-level language, this shift only constitutes a part of the high-level programming paradigm. A fundamental component of high-level Java programming is provided by the abstractions found in Java libraries. Java libraries provide a widely ranging API including standard data structures, graphics, and I/O. Through the use of libraries, Java programmers can avoid the need to re-create commonly used data structures and algorithms. For example, the system library `java.lang` provides `Object` and `String` classes, while the 'data structures' library – `java.util` – provides implementations for commonly used data structures such as vectors, queues, and linked lists. As a result, it is highly desirable to make Java libraries available to the embedded systems programmer.

It is not uncommon for a Java library make full use (or near full use) of the capabilities of the JVM. The problem that arises in the context of embedded systems programming is that typical hardware implementations of the JVM realize only a subset of the JVM. In this paper, we will use the term `jVM` to refer to a virtual machine that is a restricted version of the JVM. Specifically, due to constraints on hardware resources, expensive string manipulations, floating-point arithmetic, and support for threads are frequently excluded from hardware implementations of a `jVM`. This gives rise to an incompatibility between the `jVM` and computational requirements of the Java library. More specifically, this incompatibility creates a question regarding what role should be played by Java libraries in embedded systems programming. One answer to this question is to provide the embedded systems programmer with the subset of a particular library whose feature set is supported by the restricted `jVM` which they are targeting. Fundamental follow-on questions include: "How should such a library subset be created?" "Can the creation of such libraries be automated?"

There are two types of approaches that one can consider in the context of library migration: *removal* and *re-implementation*. Removal entails strict deletion of code that is not supported by a `jVM`. Re-implementation adapts the libraries by replacing (when possible) unsupported code fragments with equivalent code fragments that are expressed in terms of computations supported by the `jVM`. It should be noted that re-implementation is not always possible. Furthermore, re-implementation of libraries is a potentially

error-prone approach because it involves the development of new code. Depending on the nature of the re-implementation (i.e., its scope and complexity), re-implementation can entail a potentially unacceptable risk of introducing bugs into the resulting adapted library. Standard libraries typically undergo a significant number of tests to ensure their correctness before official release by Sun. Re-implementation would jeopardize assurance arguments based on these tests. In addition, standard libraries mature based on bug feedback from a large group of library users. Re-implemented libraries used in embedded applications generally will not be subjected to the maturation process associated with a large user base.

Because of the concerns raised in the previous paragraph, library adaptation based on the *removal* (of unsupported code) is more attractive than adaptation based on *re-implementation*. At least from the perspective of assurance. Abstractly, *removal* can be understood simply as a process of deleting code fragments from a library containing computations that are unsupported by the given JVM. Such a perspective reveals that removal shares similarities with program slicing. More specifically, removal can be seen as the complement of slicing. In a more theoretical setting, removal can be described as a relation between unsupported computational elements and code fragments. It is the calculation of the transitive closure of this relation that gives rise to the complexity of removal-based library adaptation.

Java libraries are evolving entities. The major updates (e.g. v. 1.5 to 1.6) are released approximately once a year. It is desirable to provide embedded systems programmers with up-to-date versions of the libraries they need.

Contribution In this paper, a transformation-based approach for migration of Java libraries is presented. This approach is currently being developed to create a library migration tool that is able to generate instances of Java libraries (particularly `java.lang` and `java.util`) suitable for use with the SCORE processor – a JVM developed at Sandia National Laboratories.

At present, our library migration tool is in its second stage of development. In the first stage, Java parser and pretty-printer were created. The Java parser is able to successfully parse the source code of the following libraries: `java.lang`, `java.util`, `java.io`, `java.nio`, and `java.swing`. In order to facilitate oversight and traceability, our parser preserves all comments rather than filtering them during the lexical analysis phase. The pretty-printer developed in the first stage is able to output Java code, which is stored and manipulated internally as Java parse trees, in a format that satisfies the Code Conventions for the Java Programming Language as defined by Sun.

The second stage of the development of the library migration tool has now been completed. In particular, syntactic dependency removal – removal of unsupported class members and references to them based on syntactic/verbatim type signature matching. What this means is that, libraries can be migrated provided that the migration can be achieved without accounting for semantic dependencies such as method signature matching with implicit parameter type casting. We expect that our theoretical results can be lifted to semantic dependency removal in a relatively seamless fashion. The problem that we will encounter is one of scale. Particularly, transforming very large tree structures.

2. An Overview of Java Library Migration

Java library migration can be seen as a generative process that takes as input a library L_1 , suitable for use with a standard JVM, and produces as output a library L_2 suitable for use with a restricted implementation of the JVM. More formally, we say a JVM is a restricted implementation of the JVM if the set of bytecodes (and native methods) it supports is a strict subset of the bytecodes (and

native methods) supported by the standard JVM. A library L is suitable for use with a given JVM if the bytecodes in the compiled version of L (i.e., its classfiles) belong to the set of bytecodes implemented by the JVM.

2.1 Restrictions

In this paper, library migration is a process that consists strictly of removing elements from a library based upon whether the elements are either directly or indirectly dependent upon features that are not supported by the computing platform being targeted. In particular, at this time we do not consider general-purpose generative techniques that attempt to re-implement library components in such a manner that the dependence upon unsupported features is avoided.

2.2 Constraints

Given these restrictions, the library migration problem can be described in a more formal setting as removing *terms* (i.e., components) from a term structure C corresponding to a class. The removal of terms is subject to certain syntactic and semantic constraints. We write $t[\dots t_i \dots]$ to denote that the term t contains t_i as a sub-term. Even though the notation $t[\dots t_i \dots]$ contains ellipsis and is only informally defined, we believe that in the context of a transformation-based discussion it is conceptually clearer than either writing $t_i \in t$ or using the positional notation that is often used in a more traditional term rewriting setting. Using the notation $t[\dots t_i \dots]$, the migration of a class C belonging to a library can now be expressed as the selective application of transformation steps having the form:

$$\mathcal{T}_i \stackrel{def}{=} C[\dots t_i \dots] \rightarrow C[\dots \epsilon \dots] \quad (1)$$

In this framework, control over the application of transformation steps like \mathcal{T}_i is subject to four primary constraints:

1. Each transformation step is required to produce a term that is syntactically well-formed;
2. When applied to a term $C_{initial}$ the transitive closure of transformation steps must yield a resulting term C_{final} such that $C_{final} \sqsubseteq C_{initial}$, where \sqsubseteq denotes refinement with respect to a given semantic function \mathcal{M} (e.g., the JVM);
3. The removal of terms t_i must be minimal with respect to certain assumptions (to be discussed shortly);
4. The term C_{final} must consist solely of elements that are supported by the targeted computing platform.

The syntactic well-formedness constraint suggests that only terms having particular shapes be considered as candidates for removal. For example, if the objective is to remove dependencies on floating point values and operations then naively removing all references to floating point values will yield results that are not syntactically well-formed. This is demonstrated by the following example.

$$int\ x = (int)1.0 \rightarrow int\ x = (int) \quad (2)$$

The semantic refinement constraint (also) suggests that only terms having particular shapes be considered as candidates for removal. In particular, syntactic well-formedness alone does not form a sufficient criterion for the basis of removal. The following example shows a transformation that results in a term that is syntactically well-formed but does not satisfy the semantic well-formedness constraint defined by the relation \sqsubseteq .

M	$::=$	$x \mid mb$
x	$::=$	inner-classes
mb	$::=$	$m \mid b$
m	$::=$	field-declaration method-declaration constructor-declaration
b	$::=$	instance-initializer static-initializer
instance-initializer	$::=$...
static-initializer	$::=$...
field-declaration	$::=$...
method-declaration	$::=$...
constructor-declaration	$::=$...
inner-classes	$::=$...

Figure 1. Syntactic Categories for Class Members

$$\begin{array}{l}
\text{int } x = 0; \\
\text{int } f()\{\text{double } x = 1.0; \text{return } (int)x;\} \\
\implies \\
\text{int } x = 0; \\
\text{int } f()\{\text{return } (int)x;\}
\end{array}
\tag{3}$$

And finally, the minimality constraint requires that a good-faith attempt be made to retain as much of the original library as possible. Otherwise, the empty library could be used to trivially satisfy the constraints 1, 2, and 4. We address the minimality constraint by removing code at the level of class members (fields, methods, etc.). This is based on the assumption that code within class members is tightly-coupled, while class members within a class are loosely-coupled. In other words, removal of a statement within a member will invalidate the entire member, while removal of a member within a class will not invalidate the entire class. Based on the inspection of the libraries, we have found this to be a reasonable assumption.

2.3 A Deductive Description of Java Library Migration

The deductive description presented here is based on a conceptual model in which there are four term structures of primary interest: unsupported features (U), classes (C), members (M), and keys (K). Terms corresponding to unsupported features U describe (at the syntactic level) the primary set of unsupported features of the target JVM. Terms corresponding to classes C are list-like structures whose elements are members M . Based on their structural properties, we partition member terms into two kinds: (1) atomic members and (2) composite members. The set of *atomic members* consists of field declarations, method declarations, constructor declarations, as well as (anonymous) initializer blocks (i.e., instance initializers and static initializers). The set of *composite members* consists of inner class declarations. An abstract syntax defining the syntactic categories of members is shown in Figure 1.

Our adaptation algorithm treats atomic and composite members differently. In particular, based on the results of dependency analysis, an atomic member is either removed completely or remains fully intact (e.g., a field/method/constructor declaration is either in

or out). The decision to subject atomic members to such a removal policy represents an approximation by our removal algorithm.

In contrast, adaptation selectively removes portions (i.e., atomic (sub)members) from classes as well as composite members occurring as members of classes.

With the exception of class blocks, all atomic member terms are associated with a key K by which they can be referenced. In the case of fields, keys denote the fully qualified field identifier. In the case of methods (and constructors), keys denote the fully qualified method (constructor) identifier and their signature.

A reference to a member can only occur under certain environmental conditions (i.e., from within certain scopes). We write $ref\text{-}to(m)$ to denote a function that, when given a member m , returns the key by which m can be referenced: e.g. m , $this.m$, $DefiningClass.m$. Since our approach is strictly syntactic at this moment, we do not track dependencies that span type hierarchies such as $super.m$, $DeclaringAbstractClassOrInterface.m$, $InheritingClass.m$.

Let $t = ref\text{-}to(m)$. We write $x[\dots \hat{t} \dots]$ to denote an occurrence of t in an environmental context where \hat{t} is interpreted as a reference to m : e.g. $Math.min(1.0, x)$ for static members, and $new Object().hashCode()$ for non-static members.

Our goal is to develop a transformation that removes a member term t from a class C when t directly (or indirectly) depends upon a feature that is unsupported on the target computing platform (e.g., the SCORE processor). It is important to note that this removal of members from classes is fully extended to inner classes, but does not extend to inner classes nested within inner classes. Empirical evidence suggests that nested inner classes are rare. In such corner cases, a comment is logged and treatment of the particular case is left to manual adaptation.

Removal of terms referencing unsupported features needs to account for transitive dependencies. To this end, we define a term set R_C that contains the closure of transitively dependent member terms. Membership in R_C is defined by the set of axioms and rules, presented in a natural deduction-style syntax, shown in Figure 2. In the axioms, the terms $FpLiteral$, $Modifier$ [[strictfp]], $BasicType$ [[float]], and $BasicType$ [[double]] denote terms corresponding to Java token-level floating point dependencies. The set R_C is the smallest set that is closed under the rules given in Figure 2.

$$\begin{array}{c}
\frac{}{FpLiteral \in R_C} \text{ (Axiom-lit)} \\
\frac{}{Modifier[[strictfp]] \in R_C} \text{ (Axiom-strictfp)} \\
\frac{}{BasicType[[float]] \in R_C} \text{ (Axiom-float)} \\
\frac{}{BasicType[[double]] \in R_C} \text{ (Axiom-double)} \\
\frac{C[\dots m \dots] \quad t \in R_C \quad m[\dots \hat{t} \dots]}{ref\text{-}to(m) \in R_C} \text{ (T-transitive)}
\end{array}$$

Figure 2. Dependency rules for floating point bytecodes

Using the dependency set R_C , the removal of members from a class can be concisely stated by the rules T-migrate1 and T-migrate2.

$$\begin{array}{c}
\frac{C[\dots mb \dots] \quad t \in R_C \quad mb[\dots \hat{t} \dots]}{C[\dots mb \dots] \rightarrow C[\dots \epsilon \dots]} \text{ (T-migrate1)} \\
\frac{C[\dots x \dots] \quad t \in R_C \quad x[\dots \hat{t} \dots]}{x[\dots \hat{t} \dots] \rightarrow x[\dots \epsilon \dots]} \text{ (T-migrate2)}
\end{array}$$

Example In Figure 3 there are two declarations of the field `innocent`. References to the field `innocent` corresponding to the first declaration are indirectly dependent upon a value of type `float` and should be removed from the class `Cafe`, while references to the field `innocent` corresponding to the second declaration are not and should remain.

```

class Cafe {
    float fabulous = 0xCAFEBADE;
    int innocent = (int) fabulous;

    { // initializer block
      innocent = 0xDD;
    }

    { // initializer block
      int innocent;
      innocent = 0xAA;
    }
}

```

Figure 3. Intra-class dependencies

3. Program Transformation Using the HATS System

HATS [16] is an IDE that provides a variety of capabilities germane to transformation-based software development. These capabilities include: (1) an engine where transformation can be performed through the execution of programs written in a special purpose language called TL, (2) a parser generator having GLR-like capabilities accepting as input extended BNF grammars together with precedence and associativity rules, (3) an abstract pretty printer, (4) graphical display facilities for viewing the structure of parse trees, (5) text editors, (6) a display showing various metrics associated with TL program execution (e.g., number of rewrites applied, etc.) and (7) some rudimentary tracing capabilities to assist in debugging transformations [17].

3.1 The Strategic Programming Language TL

TL is a language that has been developed exclusively for describing transformation-based computation [15, 18]. The principle artifacts manipulated during the execution of a TL program are *parse trees*, which we will also refer to as *terms*. TL provides a notation for describing parse tree structures relative to a given (assumed) grammar G . Trees expressed using this notation are referred to as *patterns*.

For example, suppose we are given a grammar where the derivation $stmt \xrightarrow{\alpha} id = 5$ is possible. The pattern $stmt[id_1 = 5]$ describes a tree corresponding to this derivation. In this context, the subscripted variable id_1 denotes a typed variable quantified over the syntactic category of all trees having the nonterminal id as their root node.

In general, a pattern of the form $A[\alpha']$ is well-formed if and only if the derivation $A \xrightarrow{\alpha} \alpha$ is possible according to the grammar and α' is obtained from α by subscripting all nonterminals occurring in α . Note that $\alpha' = \alpha$ when α' consists entirely of terminal symbols.

In TL, transformation is accomplished by the application of rewrite rules to terms (i.e., parse trees). In TL, a rewrite rule has the following syntactic structure:

$$lhs \rightarrow rhs [\text{if condition}]$$

where $[\text{and}]$ are syntactic meta-symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional. In order for a rule to be well-formed it is necessary that lhs be a *pattern*, that rhs be a strategic expression, and that *condition* be an expression consisting of one or more *strategic applications* or *match expressions* combined using the Boolean connectives: *and*, *or*, *not*.

A *strategic application* is simply an expression consisting of the application of a strategy to a term. The evaluation of a strategic application is either successful (which in this context is interpreted as the boolean value *true*) or unsuccessful (which in this context is interpreted as the boolean value *false*).

A *match expression* is a first-order match between two patterns. Let t_1 denote a pattern, possibly non-ground, and let t_2 denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to *true* if and only if a substitution σ can be constructed so that $\sigma(t_1) = t_2$.

In TL, labelled rules can be written using the traditional rewrite notation, in curried functional form, or a mixture of both. The following shows three different syntactic styles defining the label r . All definitions shown are semantically equivalent.

$$\begin{array}{ll}
 r & : \quad lhs_2 \rightarrow lhs_1 \rightarrow rhs_1 \\
 r \quad lhs_2 & : \quad lhs_1 \rightarrow rhs_1 \\
 r \quad lhs_2 \quad lhs_1 & : \quad rhs_1
 \end{array}$$

Which syntactic form to use is largely a matter of preference.

3.2 Standard Control Mechanisms

In contrast to a pure rewriting system, TL requires that the application of rewrite rules to terms (i.e., the transformation process) be explicitly controlled. A specification of such control is called a *strategy*. The presence of strategies classifies TL as a *strategic programming* language [7]. Other examples of strategic programming systems include Stratego [13] and Strafunski [6].

In TL, a strategy is an expression composed of various elements including *rewrite rules*, *rewrite rule abstractions* (i.e., rule labels), *combinators*, and *traversals*. The application of a strategy s to a tree t is expressed using the traditional function application syntax: $s(t)$. Within a strategy, control is expressed using the following standard control mechanisms.

1. The application of rules to a single term is controlled as follows:
 - At the rule level control is exercised through *first-order matching* and optional rule *conditions* (i.e., conditional rewrite rules).
 - At the strategy level control is exercised through *combinators*. The set of standard binary strategic combinators includes the sequential composition ($<;$) and left-biased choice ($<+$) combinators. Let s_1 and s_2 denote two strategies. The composition $s_1 <; s_2$ is a strategy that when applied to a term t will first apply s_1 and then apply s_2 . In contrast, the composition $s_1 <+ s_2$ is a strategy that when applied to a term t will first apply s_1 to t and only apply s_2 (to t) if the application of s_1 to t fails. TL also has a combinator ($<*$) which can be used to enforce a successful sequence of applications. The composition $s_1 <* s_2$ is a strategy that when applied to a term t will first apply s_1 and if the application is successful then s_2 will be applied. If the application of either s_1 or s_2 is unsuccessful then the system will behave as if the application had never been attempted.
2. The application of a strategy to a term structure is controlled by *traversals*. TL provides a rich framework for defining traversals

as well as a library of standard generic first-order traversals. Traversals specify the order in which the sub-terms of a given term are visited. Standard first-order traversals include top-down left-to-right (*TDL*) and bottom-up left-to-right (*BUL*). Let s denote a strategy. In TL, the expression $TDL\{s\}$ denotes a strategy that will traverse the term to which it is applied in a top-down left-to-right fashion and apply the strategy s to every (sub)term visited.

3.3 Special Control Mechanisms

In addition to the standard strategic mechanisms described, TL also provides several mechanisms unique to TL. These mechanisms include a number of unary combinators and constructs that enable generic traversals to be generalized to higher-order strategies.

3.3.1 The *transient* Combinator

An important unary combinator in TL is the *transient* combinator. This combinator restricts a strategy so that it may be applied *at most once*. The “at most once” property is the hallmark of the *transient* combinator. Theoretically, a *transient* can be understood as follows. Let s denote a transient-free strategy enclosed in the context $transient(\dots s \dots)$, where the ellipsis denote a transient-free strategic context. Furthermore, suppose that s has just been successfully applied to a term. Under these conditions, the following strategy reduction occurs:

$$transient(\dots s \dots) \rightarrow SKIP \quad (4)$$

where *SKIP* is a strategic constant whose application to a term is never successful. Intuitively, *SKIP* behaves as if it is not present. That is, a strategy that is reduced to *SKIP* is simply removed from the transformational process. Formally, the behavior of *SKIP* is defined as follows:

$$\begin{aligned} SKIP(t) &\equiv t \\ (SKIP \lt+ s)(t) &\equiv s(t) \end{aligned} \quad (5)$$

Transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, it becomes possible to apply a different strategy to every term encountered during a traversal. It should be noted that the *transient* combinator becomes particularly useful in a framework (like TL) where strategies can be created dynamically. Especially, when higher-order traversals are used to create strategies [15].

3.3.2 The *hide* Combinator

It is widely recognized that generalized explicit control is the primary distinction between (pure) rewriting and rewrite-based transformation (e.g., strategic programming). In this section, we describe the non-standard unary combinator *hide*.

In a strategic setting, control can be considered from the perspective of an observer. The role of an observer in such a setting is to “observe” the successful application of rules and strategies. This metaphor underlies the semantics of TL. We say that in TL, control is *observer-based*.

In a standard strategic framework, conditional composition, of the kind realized by $\lt+$, is a primary mechanism for controlling rule and strategy application. In TL, the control exercised by $\lt+$ is realized by an (implicit) observer function f_{choice} that monitors successful strategy application. The *hide* combinator is a unary combinator that when applied to a strategy s , blinds the function f_{choice} from observing whether the application of s to a given term is successful from the perspective of the $\lt+$ combinator. The semantics of *hide* is formally described as follows:

$$hide(s_1) \lt+ s_2 \equiv s_1 \lt; s_2 \quad (6)$$

The equivalence shown in Equation 6 generally raises an initial reaction prompting several questions: (1) Does *hide* do anything new? The answer is yes. (2) Is the *hide* combinator simply a derived form? The answer is no. A full discussion of this issue lies beyond the scope of this paper. However, we would like to point out several characteristics of TL that are germane to such a discussion: First, in TL, the application of strategies to terms is *identity-based* [15, 18]. What this means is that application failure is treated as an identity on terms. This is in stark contrast to traditional strategic frameworks that are failure-based and treat application failure as an operation that produces the strategic constant FAIL [13, 6]. Second, TL generalizes generic traversal to higher-order strategies. It is in this setting that the *hide* combinator becomes particularly useful. And third, TL supports a variety of non-standard combinators such as the *transient*. The interplay between these non-standard combinators and dynamic strategy creation provides a rich environment for expressing control. An example of a strategy using the *hide* combinator is given in Appendix A. A non-trivial example can be found in [16].

3.3.3 Observing the Application of Traversals

In a strategic programming idiom, a typical first-order strategy is one that traverses a term in a bottom-up left-to-right fashion and applies some strategy s to every term encountered. In TL, such a strategy would be expressed $BUL\{s\}$. Due to the heterogeneous nature of term structures, the strategy s can typically only apply to a few select sub-terms during the course of a traversal. This is the power of the generic traversal.

There is a tension between the generality of generic traversal and the unforgiving nature of failure-based semantics. Specifically, in order for a generic traversal like $BUL\{s\}$ to succeed in a failure-based system, the strategy s will need to successfully apply to every sub-term visited during the traversal. This tension is typically resolved by conditionally extending s with the strategic constant *ID* (i.e., $BUL\{s \lt+ ID\}$), where the constant *ID* denotes a strategy that can be successfully applied to any term and whose application to a term leaves the term unchanged.

In TL, there is no tension between generic traversal and its identity-based semantics. In particular, it is unnecessary to extend strategies with *ID* in order to achieve a reasonable traversal semantics. This enables the meaningful observation of whether the application of $BUL\{s\}$ to a term t succeeds. In particular, the application $BUL\{s\}$ succeeds if and only if there exists a sub-term t_1 in t to which the application $s(t_1)$ can be observed by choice combinators. Because of this, choice combinators such as $\lt+$ can be used to compose traversals in a meaningful fashion (e.g., $BUL\{s_1\} \lt+ BUL\{s_2\}$). Similarly, strategies (e.g., traversals and such) may form the conditional portion of a rule. In TL, for example, one can write:

$$lhs \rightarrow rhs \text{ if } BUL\{s\}(lhs) \quad (7)$$

Such conditions are useful for checking term structures for a variety of syntactic properties.

3.3.4 Higher-Order Generic Traversal

In addition to unary combinators, TL also lifts the notion of generic traversal to higher-order strategies. In TL, a higher-order strategy is a strategy that when applied to a term returns a strategy as a result instead of returning a term. An abstract example of a rule having order $k + 1$ is the following:

$$pattern_{k+1} \rightarrow pattern_k \rightarrow \dots \rightarrow pattern_0 \quad (8)$$

The successful application of this strategy to a term t yields the result:

$$pattern'_k \rightarrow \dots \rightarrow pattern'_0 \quad (9)$$

where $pattern'_i$ is an instance of $pattern_i$ that has been instantiated with the bindings obtained from $pattern_{k+1} \ll t$. Thus, higher-order rules are simply rules whose parameters are carried. However, their interplay with generic traversal is particularly interesting.

Let s^{n+1} denote a strategy of order $n + 1$. If s^{n+1} is applied to a term using a traversal, the strategy sequence $\langle s_1^m, \dots, s_m^n \rangle$ will be produced (assuming that s^{n+1} applies successfully m times during this traversal). This sequence can be turned into a strategy by composing the s_i^n using a binary combinator such as $\langle\leftarrow$ or $\langle;$. To generate these kinds of strategies, TL provides a library of higher-order traversals that includes $lcond_tdl$ and $lseq_tdl$. The expression $lcond_tdl\{s^{n+1}\}[t]$ will traverse the term t in a top-down left-to-right (tdl) fashion and compose the resulting strategies using the combinator $\langle\leftarrow$. The expression $lseq_tdl\{s^{n+1}\}[t]$ is similar and composes the results using the combinator $\langle;$. TL also provides a set of primitives that enable the specification of user-defined higher-order traversals.

4. Implementation

In this section we discuss TL transformations implementing the deductive rules given in Section 2.3. The fragment of the Java BNF grammar underlying this implementation is shown in Figure 4.

CompilationUnit	::=	... TypeDeclList
TypeDecl	::=	ClassDecl InterfaceDecl
ClassDecl	::=	Modifiers "class" QualifiedId ... ClassBody
ClassBody	::=	"{" Members "}"
Members	::=	Comments Member Members ϵ
Member	::=	StaticOpt Block Field Method Constructor TypeDecl
Field	::=	Modifiers Type Fields ";"
Fields	::=	FieldRest [";" Fields]
FieldRest	::=	Id FieldDeclRest
FieldDeclRest	::=	Brackets ["=" VarInitializer]
...		
VarDeclarator	::=	Id VarDeclaratorRest
VarDeclaratorRest	::=	Brackets ["=" VarInitializer]
...		
Literal	::=	IntLiteral FpLiteral ...
...		
Type	::=	QualifiedType Brackets BasicType Brackets
Brackets	::=	"[" "]" Brackets ϵ
BasicType	::=	"int" "float" "double" ...
...		
Modifiers	::=	Modifier Modifiers ϵ
Modifier	::=	"public" "private" "strictfp" ...
...		
BaseExpression	::=	[BaseExpression "."] Primary ...
Primary	::=	TypeArgsOpt "this" ArgumentsOpt Id ...

Figure 4. Java BNF grammar fragment

Figure 5 shows the overall strategies for controlling library migration as well as strategies implementing the declarative rules given in Section 2.3. In this figure, the top-level strategy is called *migration*. When applied to a *CompilationUnit* (i.e., a class)

the strategy will remove all members from the class that are dependent upon floating point operations and values. Appendix A discusses the strategies that implement the decision procedure for recognizing references to fields (e.g., \hat{t}) when these references belong to the closure of $t \in R_C$.

Due to space limitations only a cross-section of the TL implementation is shown and the discussion corresponding to this cross-section is high-level. For more specific details about the semantics of TL the reader is referred to Section 3.

The general approach for translating the deductive rules given in Section 2.3 into TL strategies is as follows. The deductive aspect of a deductive rule is implemented in TL as conditional rewrite rule with the premise of deductive rule being mapped to the conditional portion of a TL rule. Generic traversal is used to simulate the sub-term selection capabilities implied by the ellipsis in terms such as $t[\dots t_i \dots]$, and semantic analysis (e.g., the recognition of \hat{t} in a given context) is performed in the conditional portion of rules. The distinction between the syntactic categories x and m is made through first-order matching.

The premise $t \in R_C \wedge w[\dots \hat{t} \dots]$ where $w = m$ or $w = x$ found in the deductive rules T-transitive and T-migrate is implemented in a non-traditional manner. Specifically, this compound premise is implemented as a single dynamic strategy which in Figure 5 is named *s_rc*. Initially, *s_rc* consists of the strategy *base* which simply contains the axioms of R_C . However, *base* is also embedded in the following strategic expression:

$$fold \langle\leftarrow (base \langle\leftarrow add) \quad (10)$$

It is this strategic expression that we refer to as *s_rc*.

The strategy *s_rc* contains a description of how the strategy *base* can dynamically grow in response to deductions corresponding to T-transitive. Specifically, whenever a deductive step corresponding to T-transitive is performed, a strategy s_i asserting that $ref\text{-}to(t_i)$ constitutes a floating point dependency is created. This strategy s_i is then dynamically folded into *s_rc* on the left using the $\langle\leftarrow$ combinator. The result after this fold is a strategy of the form:

$$s_i \langle\leftarrow s_rc \quad (11)$$

The closure of *s_rc* has the form:

$$s_1 \langle\leftarrow s_2 \langle\leftarrow \dots \langle\leftarrow base \quad (12)$$

A brief summary of the library migration transformation is as follows. The migration strategy will only apply to terms of type *CompilationUnit* (e.g., classes). When applied to a class, the strategy $closure\{s_rc\}$ will repeatedly traverse the class and apply the strategy $process_member\{s_rc\}$ to each class member encountered. When applied to a member x , the strategy $process_member\{s_rc\}$ will check if x contains a reference to a term belonging to *s_rc* (e.g., $x[\dots \hat{t} \dots]$). If such an occurrence is found, then a deductive step is taken in which (1) x is removed from the class, and (2) $ref\text{-}to(x)$ is added to *s_rc*; otherwise x is left unaltered and processing goes on to the next member in the class via the strategy $process_next_member\{s_rc\}$.

4.1 An Example

Figure 6 shows an example of library migration whose goal is to remove floating point dependencies from the class *Cafe*. Though the example illustrates just the removal of fields it does highlight the various kinds of analysis that must be performed. In the example, the class instance field *innocent* has a floating point dependency and all class members containing references to this field should be removed. There are three primary analysis issues that arise:

migration	:	CompilationUnit _{in} → closure{fold <+ (base <+ add)}(CompilationUnit _{in})
def closure s_rc	=	FIX{ first_TDL{class_processor{s_rc}} }
def first_TDL s_rc	=	s_rc <+ mapL(first_TDL{s_rc})
def class_processor s_rc	=	TypeDecl ₁ → TypeDecl ₂ <i>(* for each class in this CompilationUnit *)</i> if extractClassName(TypeDecl ₁) <i>(* extract class id from the term and store it *)</i> ∧ TypeDecl ₂ << first_TDL{process_member{s_rc}}(TypeDecl ₁)
def process_member s_rc	=	(occurs_t_hat{s_rc} <*< (deductive_step{s_rc} <; process_member{s_rc})) <+ process_next_member{s_rc}
def deductive_step s_rc	=	s_rc <*< s_rc <i>(* Apply T-migrate and T-transitive *)</i>
def occurs_t_hat s_rc	=	Members[Comments ₁ Member ₁ Members ₁] → Members[Comments ₁ Member ₁ Members ₁] if not (Member ₁ << Member[TypeDecl ₁]) and TDL{s_rc}(Member ₁) <+ Members[Comments ₁ TypeDecl ₁ Members ₁] → Members[Comments ₁ TypeDecl ₂ Members ₁] if updateClassName(TypeDecl ₁) <i>(* extract inner class id and store it *)</i> ∧ TypeDecl ₂ << first_TDL{process_member{s_rc}}(TypeDecl ₁) ∧ not (TypeDecl ₂ << TypeDecl ₁) ∧ restoreClassName() <i>(* restore outer class id *)</i> <+ <i>(* similar rules for interface members *)</i>
def process_next_member s_rc	=	Members[Comments ₁ Member ₁ Members ₁] → Members[Comments ₁ Member ₁ Members ₂] if Members ₂ << process_member{s_rc}(Members ₁) <+ <i>(* similar rules for interface members *)</i>
axiom_lit:	FpLiteral ₁ → FpLiteral ₁	
axiom_strictfp:	Modifier[strictfp] → Modifier[strictfp]	
axiom_float:	BasicType[float] → BasicType[float]	
axiom_double:	BasicType[double] → BasicType[double]	
axiom_supported:	QualifiedType[Id ₁ TypeArgsOpt ₁] → QualifiedType[Id ₁ TypeArgsOpt ₁] if TypeList ₁ << TypeList[Object, System, Integer, ...] <i>(* list of supported classes *)</i> and not (TDL{ Id ₁ → Id ₁ }(TypeList ₁))	
base:	axiom_lit <+ axiom_strictfp <+ axiom_float <+ axiom_double <+ axiom_supported	
add:	remove_field_and_add_dependency <+ remove_method_and_add_dependency <+ remove_constructor_and_add_dependency <+ remove_init_block	
remove_field_and_add_dependency:	Members[Comments ₁ Field ₁ Members ₁] → (transient(Members[Comments ₁ Field ₁ Members ₁] → Members ₁) <i>(* T-migrate: field removal rule *)</i> <+ gen_field_reference_recognizer[Field ₁] <i>(* T-transitive: adding the reference Field₁ to s_rc *)</i>)	
remove_method_and_add_dependency: ...		
remove_constructor_and_add_dependency: ...		
remove_init_block: ...		

Figure 5. A cross-section of the strategies realizing library migration

1. A re-declaration of the integer field `innocent` within class members (e.g. methods). References to the field `innocent` occurring in these scopes do not constitute a floating point dependency (unless the reference occurs before the re-declaration).
2. Scopes containing a re-declaration of the integer field `innocent` but where a reference to the class instance field `innocent` occurs before the re-declaration. Such a reference constitutes a floating point dependency.
3. References to the class instance field `innocent` via an expression of the form: `this.innocent`. Such references can occur from within scopes where the integer field `innocent` has been re-declared. Nevertheless, the expression `this.innocent` constitutes a floating point dependency.

5. Application

At Sandia National Laboratories, a subset of the Java Virtual Machine (JVM) has been developed in hardware for use in high-consequence embedded applications. The implementation is called SCORE (Scaleable CORE) [9, 14] and supports a large subset of Java opcodes as its native instruction set. Among the unsupported features are (1) multi-threading, (2) floating-point operations, and (3) multi-dimensional arrays. Thus, for example, instructions (i.e., JVM opcodes) such as `monitorenter`, `fdiv`, and `multianewarray` are not supported. The SCORE processor also does not support a variety of methods which the standard JVM assumes are implemented natively.

SCORE runs at about 25MHz and provides 64KB for each of the program, heap, and stack memory. The memory size allows to create embedded applications of around 12,000 source lines of

```

class Cafe {

    float fabulous = 0xCAFEBABE; // catch me if you can
    int innocent = (int) fabulous; // masquerade as int

    Cafe(int innocent) {
        this.fabulous = 0xDD; // explicit ref to FP field
        this.innocent = innocent;
    }

    int environment(int innocent) { // re-decl in params
        innocent = 0xFACADE;
        return innocent;
    }

    int environment() {
        int harmless = innocent; // use comes before decl
        int innocent = harmless;
        return innocent;
    }

    int order(int choice){
        int waitress;
        if (choice == 0xCOFFEE) {
            int innocent = 0xBABE; // re-decl in local if-block
            waitress = innocent;
        } else {
            waitress = innocent; // indirect ref to FP field
        }
        return waitress;
    }

    {
        // initializer block
        int innocent;
        innocent = 0xAA;
    }
}

```

T-migrate →

```

class Cafe {
    int environment(int innocent) {
        // re-decl in params
        innocent = 0xFACADE;
        return innocent;
    }
    {
        // initializer block
        int innocent = 0xAA;
        innocent = innocent;
    }
}

```

Figure 6. An example highlighting the kind of dependency analysis involved in recognizing a reference to a field.

code reachable from the main method of an application. Thus, the platform can be used for fairly large embedded applications.

5.1 Results

In this section we report on the results of applying program transformations to the system package of the standard Java libraries

– `java.lang` (version 1.6.0). The target platform is the SCORE processor. We summarize JVM restrictions below:

synchronized Multi-threading and mutex features of the standard JVM are not supported – JVM opcodes `monitorenter`, `monitorexit`. These opcodes are tied to the ‘synchronized’ Java source code keyword. In most cases, the keyword can be deleted without affecting execution semantics on a single-threaded platform. The only case that is treated differently is when a block of code is qualified by the keyword: e.g.

```

synchronized(mutexExpr){ blockBody } →
{Object freshVar = mutexExpr ; blockBody }

```

Here, a new variable with a unique identifier w.r.t. surrounding code is introduced to capture any possible side effects within the expression ‘mutexExpr’.

volatile/transient/assert Keyword ‘transient’ annotates fields that should not persist during I/O processes. Since our JVM does not support file, network, etc. I/O, this keyword can be deleted without affecting overall semantics. Keyword ‘volatile’ annotates fields for coherent access by multiple threads and can also be deleted with no semantic effects. Usage of the keyword ‘assert’ can insert reflection-dependent features into bytecodes and is therefore not supported. It is removed using the following scenario:

```

assert(expression) →
if (!(expression)) { System.out.println(“
Warning: The assertion evaluated to false”);}

```

floating-point operations Floating-point (FP) arithmetic is not supported. FP computations arise whenever float/double variables or methods are declared in the source code. We use the decision procedure summarized in Section 2 to remove FP dependencies.

reflection/unsupported classes The reflection features of the standard JVM such as lookup of the run-time type of an object are not supported. This feature is related to the `Class` class and members of the ‘`java.lang.reflect`’ library. In addition, libraries can reference other libraries, which have not yet been migrated to the JVM: e.g. class `Pattern` in `java.util.regex`. References to unsupported classes are removed using the FP dependency routine: if a class is not in the list of supported classes, then it’s classified as an FP dependency – `axiom_supported` in Figure 5.

native Native methods of the libraries are implemented in platform-dependent manner; such methods are declared in library code and are implemented in non-Java code. The set of supported native methods is `{Object.clone(), System.identityHashCode(Object), System.arraycopy(Object,int,Object,int,int)}`. All other native methods are not supported and are removed similar to FPs.

As can be seen from the list of unsupported features we used in our library migration, it is straightforward to extend the ideas concerning floating point dependency removal to a wide variety of unsupported features by adding appropriate axioms similar to the ones in Figure 5. As long as an unsupported feature can be tied to a particular source code construct, our technique can recognize such constructs and remove them using the FP dependency technique.

We now look at quantitative results. Size-wise, `java.lang` package consists of 41620 source lines of code in 107 source files including: (i) 7 interfaces, (ii) 26 exception classes, (iii) 22 error classes, (iv) 3 annotation classes, and (v) 49 classes.

	SLOC	Orig	Auto	FP	UC	NM	Post	End	SLOC'
AbsStringBuild	1423	59	6	4	2	0	0	53	1361
AssertionError	134	9	2	2	0	0	0	7	111
Boolean	247	18	2	2	0	0	0	16	210
Byte	442	26	3	2	1	0	0	23	415
Character	5465	158	33	0	33	0	0	125	2141
Enum	200	13	3	3	0	0	0	10	132
EnumCNotPEX	55	5	1	1	0	0	0	4	44
Integer	1163	50	6	2	4	0	0	44	1014
Long	1145	46	6	2	4	0	0	40	1011
Math	1513	61	52	52	0	0	0	9	179
Number	95	7	2	2	0	0	0	5	79
Object	536	13	7	0	0	7	0	6	250
Readable	38	1	1	0	1	0	0	0	20
Short	457	27	3	2	1	0	0	24	430
String	3003	93	28	2	25	1	0	65	1834
StringBuilder	463	43	9	4	5	0	-1	35	351
System	1150	46	39	0	25	14	0	7	323
Throwable	651	26	16	0	5	11	-4	14	356
Void	31	2	1	0	1	0	0	1	26
Total	18211	703	220	80	107	33	-5	488	10287

Figure 7. Transformation results – java.lang

Transformations have removed either (1) unsupported classes, (2) unsupported keywords, or (3) unsupported members that (a) contained floating-point operations, (b) referenced unsupported native methods, and (c) referenced unsupported classes and members. At the class level, transformations removed 35 unsupported classes: e.g. `ClassLoader`, `Float`, `Double`, `Thread`. 53 classes were untouched, i.e. they did not contain any unsupported features. The results of transformations of the remaining 19 classes are summarized in Figure 7. The first two columns show the original number of lines of code and number of class members. The column ‘Auto’ displays the number of methods removed using the automated transformations. The number of automatically removed class members is broken down into (1) members removed because of dependency on floating-point calculations (column FP), (2) members dependent on unsupported classes and unsupported methods (column UC), and (3) members dependent on native methods (column NM). Due to syntactic limitations described above, some code had to be manually adjusted to pass compilation; the number of members manually adjusted is shown in column ‘Post’. The resulting number of members and lines of code is listed in the last two columns.

To avoid overly limited functionality of the resulting library, some methods were manually re-inserted to the library after performing the transformations. For example, four constructors of ‘`Throwable`’ were removed by the transformations because the constructors referenced the unsupported native method `fillInStackTrace()`. Since this class is the super-class of all exception and error classes, removal of the constructors resulted in a large number of compiler errors. We have manually re-inserted the 4 constructors but removed references to the unsupported native method. Similar manual modification was performed on ‘`StringBuilder`’.

Regarding usefulness of the resulting library, we can only provide a qualitative assessment. It is non-trivial to define usefulness of transformed libraries in a quantitative fashion, because the nature, functionality, and therefore key fields and methods are different from one class to another. An attempt could be made to qualitatively assign to all members of a class a numeric weight based upon usefulness of that class member for overall functionality of a class. For example, for a data structure class like ‘`java.util.Stack`’ all of its 5 members {`empty()`, `peek()`, `pop()`, `push()`, and `search(Object)`} can be considered as primary and thus

a weight system {0.05, 0.3, 0.3, 0.3, 0.05} could be assigned. The sum of weights of transformed classes could be used to quantitatively judge its usefulness. However, there are other classes like ‘`java.lang.Math`’ that are simple utility collections of various methods and thus a weight assignment approach is not clear.

Continuing with key features of a class, we note that, whenever an unsupported feature was key for a class, the entire class was removed. Examples of this are classes `Float`, `Double` due to unsupported float/double primitive types, classes `Thread`, `Runnable`, `ThreadGroup`, `ThreadLocal` due to unsupported multi-threading, class `ClassLoader` due to unsupported dynamic class loading and others. In all other cases, removal was performed at class member level and removed features were not primary. For example, consider classes in the `Number` class hierarchy: `Byte`, `Integer`, `Long`, `Short`. In these classes, removed members are floating-point members (e.g. `floatValue()`) and members encoding locale-dependent (e.g. dot vs. comma digit separator) string representations of internal state and are not primary features. The resulting classes still contained MIN/MAX constants, constructors, and query methods such as `intValue()`, `longValue()`, etc. and were still useful. This was the case in the majority of transformed classes. Therefore, the resulting libraries are still functioning and retained usefulness.

Coverage-wise, as can be seen from Figure 7, transformations removed 100% of unsupported members and only 5 out of 220 removed methods had to be manually adjusted (2.33% overhead). The number of source lines of code was reduced by 43%; the number of class members went down by 30%. Overall, automation performed 97.67% of all the work!

Time-wise, the effort spent on building the infrastructure amounted to about 2 man/months of work: 132 SLOC in lexer, 277 SLOC in BNF grammar for parser, and 2655 SLOC in pretty-printer. Development of transformation code resulted 1133 SLOC in about 1 man/months. As far as execution time of transformations, the entire transformation of the ‘`lang`’ library took about a day with 3 hours on class `Character` (5465 SLOC), around 5 minutes on average for the remaining 18 transformed classes, and less than 1 minute each on the 53 unaffected classes. The initial investment into the infrastructure is well worth the effort since transformations can now be used on other libraries (e.g. `java.util`). The size of the code to be transformed places a slight limitation to our transformations: large input code generates large parse trees and, since the HATS transformation engine is implemented in ML and executes

interpretively instead of natively, execution time of transformations grows more than linearly with the size of parse trees. However, we do not perceive this as a problem because the input code can be broken into pieces and transformed in a piecewise manner. Therefore, the approach described in this paper is scalable to any other Java library and Java code.

In summary, our experience with `java.lang` suggests that automated transformations provide a significant help. The manual inspection of the code resulting from transformations took only a few hours, instead of possibly weeks of un-aided manual library migration. The industrial rule-of-thumb, as suggested by Akers et al. [1], states that 75% migration rate using automated techniques produces significant cost savings over the manual approach. Using only the limited, syntactic dependency based transformations, we have achieved a 97.67% migration rate.

6. Related and Future Work

Majority of the work on transformation of Java to Java code utilizes behavior-preserving refactoring techniques [11] [2][3]. Refactoring is a transformation that alters structure of existing code without affecting its observable behavior. Refactoring can range from simple modifications like formatting code with proper indentations to more complex modifications like converting blocks of code into sub-routines or methods. Refactoring has been used to customize and optimize Java code including Java libraries. Sutter et al. [11] profile libraries and based on their execution speed and memory footprint create optimized versions of the standard library classes. Refactoring has also been used to capture developers' manual modifications of libraries with the intent of replaying the captured actions on client software that uses the modified libraries [5].

Recent developments in the use of refactoring to migrate Java code use type information and type constraints. Here, the typing rules of Java are captured as type constraints that have to be satisfied by refactored code. The impressive work by Tip et al. [12] uses type constraints to preserve behavior in extraction of interfaces and pull-up of members into a super-class. Refactoring with type constraints has also been used to remove deprecated code [2] and convert legacy code with downcasts into generic code that eliminates downcasts [4].

When it comes to Java, Eclipse is one of the most developed IDEs around. For example, it provides definition/use and use/definition variable analysis. This allows one to highlight a variable in the Eclipse UI and to ask Eclipse via a keystroke to locate all references to the variable. This approach could be used to remove floating-point dependencies, for example, by searching for keywords `float/double`, removing corresponding class member declarations, and searching for transitive dependencies on the removed class members. However, this is still a manual technique (at least removing is). The quantitative analysis of transformation results of a single library `java.lang` shows that 161 class members were removed automatically. This illustrates that manual removal of floating-point dependencies does not scale well for migration of multiple libraries with tens of classes and hundreds of KSLOC in each library.

Eclipse with its abstract syntax infrastructure could be used as a *transformation system*, however the advantage of Eclipse over HATS in transformations is not convincing. Eclipse is first and foremost a Java IDE and not a transformation system. Therefore, such transformation primitives as normal form computation (HATS operator FIX), generic traversals of term structures (HATS traversals TDL, TD, etc.), and strategy composition operators (HATS combinators `transient`, `<`, `<+`, etc.) do not have straightforward equivalents in Eclipse. A potential gain from parsing/prettyping that is already present in Eclipse might be easily offset by the complexity of expressing transformations in Eclipse.

The work presented here differs from other work related to Java libraries in that we modify the libraries directly instead of the client software that uses the libraries. In addition, whereas others focus on preserving the API exposed by the libraries, we reduce the functionality of the libraries because of restrictions of the (SCORE) target platform. To the best of our knowledge this has not been done elsewhere. Even Sun Microsystems does not provide a migrated version of the standard libraries for use on their Java Card platform [10].

The chief limitation of this work is absence of support for full semantic type information; we track syntactic dependencies only. Future work entails inclusion of type information that can be gathered from class library hierarchies and class members. Essentially, this involves scanning of class hierarchies in a top-down manner collecting member type signatures into a set of filters. Then, the fully scoped dependency on unsupported features can be calculated precisely eliminating the possible need for post-transformation manual revision of code entirely.

As was briefly mentioned in Section 2, member removal based solely on syntactic analysis is not sufficient to assure that the semantic constraints of migration are met (i.e., that the resulting library is a refinement of the original library). Figure 8 gives an example of a type-hierarchy semantic dependency.

<code>class Babe</code>	<code>{ int innocent = 32; }</code>
<code>class Fabulous extends Babe</code>	<code>{ int innocent = (int) 34.0; }</code>

Figure 8. The need for inter-class analysis

7. Conclusion

Library migration is an essential part of providing high-level language support for embedded systems programmers. Libraries such as `java.lang` are evolving artifacts and periodically new versions of these libraries are released. As a result, in order to remain current the migration of a particular library must be repeated in response to a new release of the library. In this paper, we presented a transformation-based approach for Java library migration. The scope of the migration effort is currently limited to the removal of portions of the library dependent upon features that are unsupported by a target JVM (e.g., the SCORE processor). Syntactic and semantic well-formedness constraints influence the shape of terms that are candidates for removal. In particular, these constraints lead to a simplifying generalization that limits the scope of removal to a class' member terms (e.g., field declarations, method declarations, constructor declarations, and initializer blocks).

Declarative-style rules are given describing the migration process with respect to individual classes. The declarative rules *T-transitive*, *T-migrate1*, and *T-migrate2* have premise conditions that require semantic analysis (e.g., the recognition of \hat{t} in a particular context). This analysis goes beyond syntactic matching capabilities and can be summarized as a decision procedure for recognizing references to unsupported elements within a class (e.g., floating point dependencies). The declarative-style rules are next implemented in the strategic programming language TL. The implementation of the declarative rules is fairly direct (modulo the syntax of TL). However, a noteworthy aspect of this implementation is that the decision procedure for recognizing floating point dependencies is encoded as a strategy. The outcome (success/failure) of the application of this strategy to a term captures the result of the decision procedure.

A. A Decision Procedure for Field References

The strategies in Figure 10 form a decision procedure that, in the case when t is a field reference, can recognize \hat{t} when $t \in R_C$.

```

TD{ Block0 → Block0
  if TDL{transient(
    hide(
      VarDeclarator[ Id0 VarDeclaratorRest1 ] →
      VarDeclarator[ Id0 VarDeclaratorRest1 ]
    )
    <+ (Id0 → Id0)
  )(Block0)
}

```

Figure 9. An instantiation of a race-condition strategy

Though not shown in this paper, similar decision procedures have also been developed for other class members.

The strategy `gen_field_reference_recognizer` is a second-order strategy that when applied to a field declaration matching the pattern

$$\text{Field}[\text{Modifiers}_1 \text{ Type}_1 \text{ Fields}_1 ;]$$

will return a strategy that can be used to check for references to any fields declared in this declaration. The following gives a pseudo-concrete example of the strategy returned when the strategy `gen_field_reference_recognizer` is applied to a declaration.

$$\begin{array}{l}
\text{float } x, y, z; \\
\implies \\
s_x <+ s_y <+ s_z
\end{array} \tag{13}$$

where s_x is a strategy that checks for references to the field x in (1) methods, (2) constructors, (3) declarations of other fields, (4) initializer blocks, and (5) direct (scope transcending) references to x via the keyword `this`. The strategies s_y and s_z have similar descriptions.

Conceptually speaking, the primary strategic theme in the recognition of a reference to a field id is a race condition between finding a benign re-declaration of id and an occurrence of id constituting a reference. Whichever case wins the race determines the outcome of the analysis (i.e., a reference to id did or did not occur). Furthermore, this race condition occurs along every path between the root/leaf nodes of a tree.

This sort of analysis arises in many program analysis settings. Examples include use-def analysis and various compiler optimizations, as well as more exotic analysis arising in aspect-oriented programming. In TL we have developed a template (of sorts) for this kind of behavior. The (manual) instantiation of this template to the problem at hand yields the strategy shown in Figure 9.

In Figure 9, the rewrite rule enclosed in the `hide` combinator checks for the occurrence of a declaration of Id_0 . If such an occurrence is encountered the entire contents enclosed by the `transient` combinator will be reduced to SKIP; otherwise if a use of Id_0 is encountered first, again the contents of the `transient` will be reduced to SKIP. However, the presence/absence of the `hide` combinator will be seen to have opposite behavior from the perspective of the success/failure of the application of $TD\{\dots\}(Block_0)$.

We now consider the strategy `ref_to_field_in_methods`. When applied to an identifier Id_0 (i.e., a field identifier) this strategy returns a rule that can recognize a reference to Id_0 in the context of a method declaration. The rule itself is an identity whose successful (or unsuccessful) application is determined by its conditional component. In particular, the condition first checks to make sure that Id_0 has not been re-declared as a formal parameter to the method. If this check succeeds, then race-condition strategy shown in Figure 9 is applied to the body of the method.

References

- [1] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49:275–291, 2007.
- [2] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of OOPSLA 2005*, pages 265–279, San Diego, California, United States, 2005. ACM.
- [3] Danny Dig and Ralph Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM’05)*, pages 389 – 398. IEEE, 2005.
- [4] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java Programs to Use Generic Libraries. In *Proceedings of OOPSLA 2004*, pages 15 – 34, Vancouver, BC, Canada, 2004.
- [5] Johannes Henkel and Amer Diwan. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*, St.Louis, Missouri, USA, 2005.
- [6] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCSE*, pages 137–154. Springer-Verlag, January 2002.
- [7] Ralf Lämmel, Eelco Visser, and Joost Visser. *The Essence of Strategic Programming*. 2002.
- [8] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. On the Design and Development of a Customizable Embedded Operating System. In *Proceedings of the International Workshop on Dependable Embedded Systems, 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, October 2004.
- [9] J. A. McCoy. An Embedded System for Safe, Secure and Reliable Execution of High Consequence Software. In *HASE 2004: The 5th IEEE International Symposium on High Assurance Systems Engineering*, pages 107–114, Albuquerque, New Mexico, United States, 2004. IEEE.
- [10] Sun Microsystems. Java card technology. <http://java.sun.com/products/javacard/>.
- [11] Bjorn De Sutter, Frank Tip, and Julian Dolby. Customization of Java Library Classes using Type Constraints and Profile Information. In *Proceedings of ECOOP 2004*, pages 585 – 609, Oslo, Norway, 2004.
- [12] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of OOPSLA 2003*, pages 13–26, Anaheim, California, USA, 2003.
- [13] E. Visser, Z. e. A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *third ACM SIGPLAN International Conference on Functional Programming (ICFP’98)*, pages 13–26. ACM Press, September 1998.
- [14] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. The SSP: An Example of High-Assurance System Engineering. In *HASE 2004: The 8th IEEE International Symposium on High Assurance Systems Engineering*, pages 167–177, Tampa, Florida, United States, 2004. IEEE.
- [15] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
- [16] Victor Winter and Jason Beranek. Program Transformation Using HATS 1.84. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCSE*, pages 378–396, 2006.
- [17] Victor L. Winter, Christopher Scalzo, Arpit Jain, Brent Kucera, and Azamatbek Mametjanov. Comprehension of generative techniques. In *Software Transformation Systems Workshop (STS)*, 2006.
- [18] V.L. Winter. Strategy Construction in the Higher-Order Framework

gen_field_reference_recognizer:

```

Field[[ Modifiers1 Type1 Fields1 ; ]] →
  lcond_tdl{ FieldRest[[ Id1 FieldDeclRest1 ]] → (* for every field in this decl stmt *)
    ( ref_to_field_in_methods[QualifiedId1] <+
      ref_to_field_in_constrs[QualifiedId1] <+
      ref_to_field_in_fields[QualifiedId1] <+
      ref_to_field_in_init_blocks[QualifiedId1] <+
      BaseExpression[[this.Id1]] → BaseExpression[[this.Id1]]
    )
  if IdcurrentClassName := lookupCurrentClass() (* build current class id term *)
  ^ QualifiedId1 << QualifiedId[[ IdcurrentClassName . Id1 ]] (* build qualified field id term *)
  ^ {print("field:"); output_leaves(QualifiedId1)}
  }[Fields1]

```

ref_to_field_in_methods QualifiedId₀:

```

Method[[ Modifiers1 TypeParameterOpt1 Type1 Id1 FormalParams1 Brackets1 Throws1 MethodBody1 ]] →
Method[[ Modifiers1 TypeParameterOpt1 Type1 Id1 FormalParams1 Brackets1 Throws1 MethodBody1 ]]
  if QualifiedId0 << QualifiedId[[ Idclass . Idfield ]] (* extract class id and field id *)
  and not (TDL{Idfield → Idfield}(FormalParams1)) (* field is not re-declared in parameters *)
  and MethodBody2 << TDL{lcond_tdl{ (* rewrite locally declared vars into their types *)
    (LocalVarDecl[[ FinalOpt1 Idclass Idx VarDeclaratorRest1 ; ]] →
      (Idx → Idclass)
      if not( Idx << Idfield ))
      }[MethodBody1]}(MethodBody1))
  and MethodBody3 << TDL{Primary[[ new TypeArgsOpt1 Idclass Arguments1 ]] → (* rewrite object allocators into types *)
    Primary[[ Idclass ]]
  }(MethodBody2)
  and (TD{ Block0 → Block0 (* for every block in this body broadcast the field filter *)
    if TDL{transient( (* reduce to SKIP if there exists a re-decl or a reference *)
      hide( (* hide re-declaration from TDL, i.e. filter fails to apply *)
        VarDeclarator[[ Idfield VarDeclaratorRest1 ]] → VarDeclarator[[ Idfield VarDeclaratorRest1 ]] (* field is re-declared locally *)
      )
    } <+
    ( BaseExpression[[ Idclass . Idfield ]] → BaseExpression[[ Idclass . Idfield ]])
    <+
    ( BaseExpression[[ Idfield ]] → BaseExpression[[ Idfield ]])
  }(Block0))
  }(MethodBody3))

```

ref_to_field_in_constrs Id₀: ... (* similar to ref_to_field_in_methods *)

ref_to_field_in_fields Id₀: ...

ref_to_field_in_init_blocks Id₀: ...

Figure 10. Strategies for recognizing \hat{t} when $t \in R_C$ and t is a reference to a field