# Mixing Source and Bytecode

## A Case for Compilation by Normalization

Lennart C. L. Kats
Department of Software Technology
Delft University of Technology,
The Netherlands
l.c.l.kats@tudelft.nl

Martin Bravenboer
Department of Computer and
Information Science
University of Oregon, USA
martin.bravenboer@acm.org

Eelco Visser
Department of Software Technology
Delft University of Technology,
The Netherlands
visser@acm.org

## Abstract

Language extensions increase programmer productivity by providing concise, often domain-specific syntax, and support for static verification of correctness, security, and style constraints. Language extensions can often be realized through translation to the base language, supported by preprocessors and extensible compilers. However, various kinds of extensions require further adaptation of a base compiler's internal stages and components, for example to support separate compilation or to make use of low-level primitives of the platform (e.g., jump instructions or unbalanced synchronization). To allow for a more loosely coupled approach, we propose an open compiler model based on normalization steps from a high-level language to a subset of it, the core language. We developed such a compiler for a mixed Java and (core) bytecode language, and evaluate its effectiveness for composition mechanisms such as traits, as well as statement-level and expression-level language extensions.

***Categories and Subject Descriptors*** D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

***General Terms*** Languages, Design

***Keywords*** Dryad Compiler, Stratego, Java, Bytecode, SDF, Compilers, Meta Programming, Embedded Languages, Language Extensions, Domain-Specific Languages, Source Tracing, Traits, Iterators


*OOPSLA'08,* October 19–23, 2008, Nashville, Tennessee, USA.


## 1. Introduction

Programming languages should be designed for growth in order to evolve according to the needs of the user [34]. General-purpose programming languages offer numerous features that make them applicable to a wide range of application domains. However, such languages lack the high-level abstractions required to adequately cope with the increasing complexity of software. Through the introduction of *language extensions*, it is possible to increase the expressivity of a language, by turning programming idioms into linguistic constructs. Language extensions allow for static verification of correctness, security, and style constraints. Language extensions may be *domain-specific* in nature, such as embedded SQL, or may be *general purpose* in nature, such as traits or the enhanced for loop, enumerations, and other features added in Java 5.0.

The mechanisms available for realizing language extensions determine the quality of extension implementations and the effort needed for their construction. If language extensions can be realized with relative ease, then building new abstraction mechanisms can be used in the software development process to replace programming idioms and boilerplate code. The quality of a language extension comprises robustness (are error messages reported on the extended language? is code generation complete?), composability (does the extension combine with other extensions?), and the quality of the generated code. Finally, separate compilation, i.e. binary distribution of compiled, statically verified (library) components, is an important feature to reduce compilation time.

The creation of a language extension implies the reuse of an existing implementation of the base language. Generally, such reuse can be categorized as either black box reuse of a compiler in the form of a preprocessor, or white box reuse by means of a deep integration with the compiler implementation.

***Language Extension by Preprocessing*** A preprocessor transforms a program in an extended language into a program in the base language. Examples of preprocessors include annotation processors such as XDoclet [37] and Java's

APT [35], SQLJ [24] an embedding of SQL in Java, and StringBorg [5], a generic approach for embedding languages such as SQL. As they are employed as a separate tool, rather than requiring integration into a compiler, preprocessors are highly portable. By avoiding direct compiler extension, their implementation only requires basic knowledge of a language's structure and semantics, not of its compiler. Thus, the compiler is considered as a black box; only its published interface — the syntax of the programming language — is used, and the internal architecture of the compiler is hidden. This separation of concerns makes preprocessors well suited for rapid implementation or prototyping of language extensions.

While preprocessors are an attractive, lightweight solution to language extensibility, they are not considered a mature solution for production implementation of languages. Production of a parser that is exactly compatible with the base language is not always a trivial undertaking. The lack of a (complete) source level semantic analyzer results in error messages by the base compiler about code fragments generated by the preprocessor, rather than the source code written by the programmer. Separate compilation is only possible if the compilation units of the extended language align well with those of the base language. This fails in the case of new modularity abstractions. In other words, considering the base compiler as a black box condemns the preprocessor implementer to reimplement the front-end of the base compiler.

***Reusing an Existing Compiler*** To avoid reimplementation efforts, language extensions are often implemented by extension of the front-end of a compiler. This level of integration ensures that existing compiler components, such as a parser and semantic analysis, can be reused in the extended compiler. By generating code in the front-end language, it can then be further compiled using the base compiler. Traditional monolithic compilers are typically not designed for extensibility, and adding a new feature may require extensive refactoring of the implementation. Since such refactorings are not incorporated upstream, this effort needs to be repeated with each release of the compiler. Extensible compilers, such as Polyglot [28], ableJ [42], and the JastAdd extensible Java Compiler [13], are designed for extensibility with the principle that the implementation effort should be proportional to the size of the language extension. This *front-end extension pattern* is illustrated in Figure 1. However, even these systems do rely on white box extension, by exposing their internal structure.

Extending a compiler purely by transformation to the base language is sometimes inadequate. Compilers typically consist of multiple stages, parsing and gathering semantic information in the early stages (the *front-end*), and generating and optimizing code in later stages (the *back-end*) [1]. This strict separation of stages is imposed to ensure straightforward modularization and reuse of compiler components. As such, some compilers for different languages share a common intermediate language and associated back-end. Only the final stage or stages of a compiler back-end actually output the target-machine code.

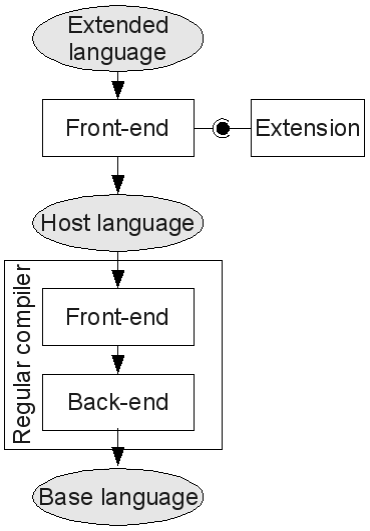


**Figure 1.** The front-end extension pattern, applied by many conventional extensible compilers.

Integration into the *back-end* makes it possible to also manipulate compiled code. This can be necessary to output specific instruction sequences or clauses not otherwise generated by the base compiler. For example, for Java, the front-end does not expose a 'goto' operation, unbalanced synchronization primitives, or means of including debugging information. The back-end of a compiler also forms an essential participant in composition of source and compiled code. Consider language extensions aimed at modularity, such as *traits* [12] and *aspects* [20]. To support separate compilation, such extensions require integration into a compiler's back-end. Separate compilation enables distribution of modules in compiled form, or weaving of code into compiled classes. Using a classic, multi-staged compiler, implementing such extensions is a demanding task that requires in-depth understanding of the compiler. Extensions that span across multiple compilation stages get tangled throughout the different components of a compiler, and create a large dependency on its implementation.

***Mixing Source and Bytecode*** Summarizing, the decision to extend a compiler using a simple, front-end based approach, or a more deeply integrated approach, comes down to a choice between black box or white box reuse of a base compiler, each with their own drawbacks. Rather than dismissing preprocessors, we want to embrace their simplicity and modularity and propose a compiler architecture that deals with their flaws.

In this paper, we propose an open compiler architecture based on *mixing source and bytecode* in order to enable *compilation by normalization*. That is, the base language

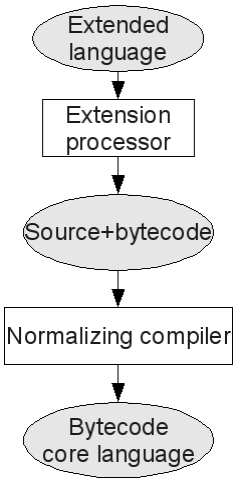


**Figure 2.** Modular extension of a normalizing compiler.

is a combination of the high-level source language (Java) and the low-level bytecode core language. This means that it is possible to use bytecode primitives, such as the `goto` instruction, directly from Java, as in the statement

```
if (condition) `goto label;
```

where the backtick (`) operator is used to distinguish between the syntax of the two constituent languages. Similarly, source code expressions and statements can be embedded in bytecode, nested to arbitrary depth. The compiler for this mixed base language *normalizes* an input program in the combined language to a program in the bytecode core language. Thus, the language produced as output of the compiler is a subset of the input language.

This architecture combines the light weight construction of extensions using a preprocessor, with the access to the compiler back-end of a compiler extension. Without knowing the internals of the compiler, a preprocessor can generate code in the core language where needed, as well as the extended input language where possible (see Figure 2). In other words, an extension preprocessor extends the base compiler by adding transformations from the extended language into a mix of low-level and high-level code, as is convenient for the definition of the particular extension. The idea of transformation-based compilers is not new. In particular, Peyton Jones and Santos [31] applied the idea in the implementation of the Glasgow Haskell Compiler GHC in which compiler optimizations are formulated as transformations on core language constructs. The front-end of the compiler consists of type analysis and simple desugarings. However, in GHC the core language is not a subset of the compiler's input language. As a result it is not possible to feed the output of the compiler back into the compiler. In our approach the *complete compiler* can be used as a normalization tool, which allows the construction of pipelines of preprocessors, and the implementation of separate compilation for new abstraction mechanisms. By keeping track of origin information when generating code, error messages produced later in the chain can refer to the original source input.

To evaluate the notion of compilation by normalization, we have created a prototype implementation of a Java compiler based on this principle. This prototype, the *Dryad Compiler*[1], unifies the Java language with the underlying bytecode instruction language. Extending a regular Java typechecker, we provide full typechecking for this combined language. Using the collected type information, we introduce *overloaded instructions* to the bytecode language that can be normalized to regular instructions, and facilitate code generation.

***Outline*** We proceed as follows. In Section 2 we discuss the design of a number of language extensions, evaluating how they benefit from compilation by normalization when implemented as an extension of the Dryad Compiler. We describe the syntax and semantics of the mixed Java/bytecode language and the architecture of its compiler in Section 3. In Section 4 we discuss how normalization rules incrementally transform the Java/bytecode language and its extensions to the core language. In Section 5 we offer a discussion of the architecture of the Dryad compiler, and compilation by normalization in general. We present related and future work in Section 6, and finally conclude in Section 7.

## 2. Extending the Dryad Compiler

In this section, we discuss a number of compiler extensions for the Dryad Compiler. We first discuss extension at the *class level*, with partial and open classes in Section 2.1 and traits in Section 2.2. In Section 2.3 we discuss how the principle of compilation by normalization can be applied at the *statement level* for iterators, and in Section 2.4 we show how it can benefit the implementation of *expression-level* extensions.

### 2.1 Extension with Partial and Open Classes

In Java, classes are defined in a single source file. *Partial classes* enable the distribution of class members over multiple source files. Partial classes can be used for separation of concerns, for example dividing GUI-related and event-handling code, or as a way of modifying existing classes included in a third-party application or library. Another application of partial classes is merging generated code fragments, such as code generated from partial models [44] or from modular transformations generating code for specific aspects [16]. Partial classes can be implemented relatively easily as an extension of an existing compiler, by merging them to regular classes.

*Open classes* extend the notion of partial classes by allowing the extension of compiled classes, rather than just classes in source code form. One implementation of open

[1] http://strategoxt.org/Stratego/TheDryadCompiler

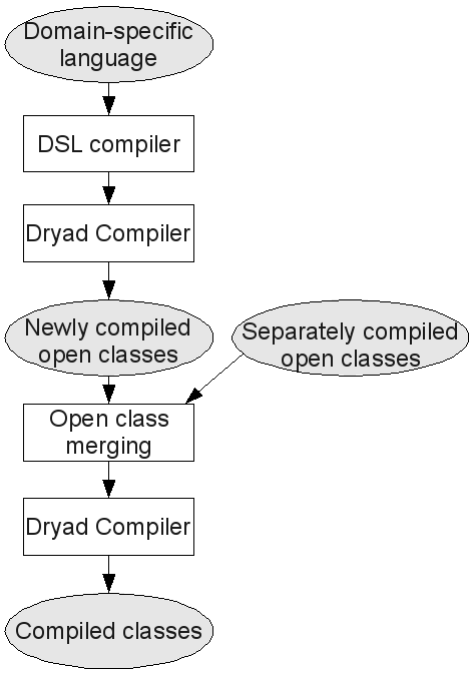


**Figure 3.** Using open classes for incremental compilation.

classes for Java is provided by MultiJava [8, 9]. MultiJava is derived from the Kopi Java Compiler, a fairly small open-source implementation written in Java itself. This makes it a relatively accessible candidate for such an extension. MultiJava uses different stages that add to the existing stages of the Kopi compiler (i.e., parsing, importing definitions, different typechecking stages, grouping of multi-methods, bytecode generation). This required a significant part of the development effort to be spent in understanding the base compiler. Clifton notes that this was in fact the case, attributing this to a lack of documentation on Kopi [8].

Existing implementations of partial and open classes only allow the definition of classes that augment others in source form, merging them into either source or compiled classes. A third scenario, merging together multiple classes in compiled form, is not supported. This scenario can be applied to use open classes for supporting separate compilation in compilers that target the Java platform, a technique for example applied in [44] and [16]. Consider Figure 3, where open classes facilitate merging of newly and previously compiled class fragments. In this architecture, a compiler for a domain-specific language (DSL), for example producing GUI-related code, is implemented as an extension of the Dryad Compiler. The code it produces is processed by another extension, which merges the open classes for final compilation.

The mixed source/bytecode language allows us to think of source and bytecode classes as instances of a single language; there is no fundamental difference in the merge process required for them. Figure 4 shows the architecture of

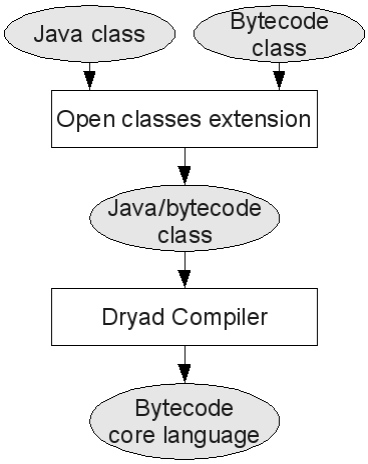


**Figure 4.** Extending the Dryad Compiler to support open classes.

```
class Calculator {
  // From Calculator_Operations.java
  public void add() {
    operation = new Add(getDisplay());
    ⋮
  }

  // From Calculator_Gui.class
  `private setDisplay(int number : void) [
    iload number;
    ⋮
  ]
  ⋮
}
```

**Figure 5.** Open classes merged to a single Java/bytecode definition.

the open classes extension. It merges together members of class fragments, either in source or bytecode form. The resulting Java/bytecode class (see Figure 5) is passed to the Dryad Compiler, which provides the support for compilation of these composed fragments. This design allows a straightforward implementation of the extension, no longer requiring implementation-level knowledge of the open compiler it builds on. Compilation of the merged classes is handled by the base compiler; the extension only provides the composition mechanism. Using the technique of source tracing (on which we elaborate in Section 3.4), it maintains location information relative to the original source files for compile-time errors and debugging information.

### 2.2 Extension with Traits

Traits are primitive units of reusable code that define a set of methods that can be imported into regular classes. Using

```
class Shape with TDrawing {
  Vertices getVertices() { ... }
  ⋮
}

trait TDrawing {
  void draw() { ... }

  require Vertices getVertices();
}
```

**Figure 6.** Example of a class `Shape` importing a trait `TDrawing`.

a set of operators, traits can be composed and manipulated. For instance, the `with` operator composes two or more traits, and is also used to extend a class with the methods defined in a trait (see Figure 6). Traits were originally introduced by Schärli et al. in the context of Smalltalk [12]. They have since been ported to statically typed languages, such as Java, C#, and Scala. To support traits in a statically typed context, they must explicitly specify their *required methods*, i.e. all methods that are referenced but not provided by a trait.

To the best of our knowledge, only Scala – which supports the feature natively rather than in the form of an extension – supports separate compilation of traits [29]. This allows for statically verified, binary distribution of traits in libraries, but requires a significantly different compilation model than the source-to-source transformation commonly applied by implementations of traits.

To enable separate compilation to binary class files in our traits extension, we designed a compilation scheme translating traits to regular, abstract classes. This way, required methods can be mapped to abstract methods. Although we have no intention for the Java Virtual Machine (JVM) to load these classes at run-time – traits are merely compile-time entities – this mapping enables us to use the base compiler's verification and compilation facilities for abstract classes.

After traits are compiled, the resulting class files can be composed according to the composition operators. For the `with` operator this means that a trait's methods are added to a client class. Similarly, the `minus` operator removes methods from a trait. The `rename` operator renames a method declaration and all occurrences of it in the core language invocation constructs. Unlike in Java, the names used in these constructs are fully qualified and unambiguous, making this a straightforward operation. The composition operations are followed by a basic consistency check that confirms that all required methods are defined and that there are no duplicate definitions. More extensive typechecking is performed by the base compiler. Consider Figure 7, which illustrates the architecture of this extension. By leveraging the Java/bytecode language for inclusion of compiled code, the extension remains loosely coupled from the base compiler. Our implementation

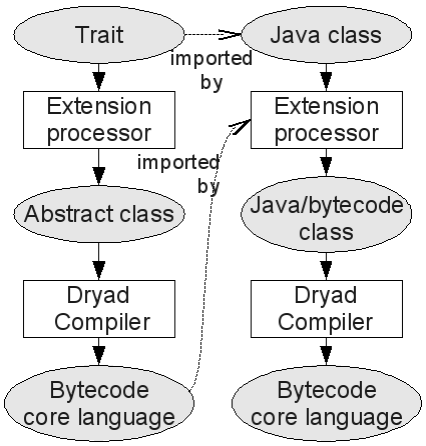


**Figure 7.** Separate compilation of a trait (left) and a class importing it (right).

```
for (String s : getShortWords("foo bar")) {
  System.out.println(s);
}
```

```
Iterator<String> it =
        getShortWords("foo bar").iterator();
while (it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

**Figure 8.** The enhanced for loop (top) and its desugared form (bottom).

of traits currently spans 104 lines of code[2], making it a relatively lightweight implementation. Still, it includes support for separate compilation, consistency checking, and source tracing. As it does not require implementation-level knowledge of the base compiler, but only of its input language, the focus shifts to these issues rather than on direct adaptation of the existing compilation process.

### 2.3 Extension with Iterator Generators

Java 5.0 introduced the *enhanced for loop*, a language feature that allows programmers to iterate over collections. The Java compiler treats this as a form of syntactic sugar, and translates it to a regular `while` loop that uses the `java.lang.Iterable` and `java.util.Iterator` interfaces (see Figure 8). As such, the enhanced for loop can be used to conveniently iterate over any type that implements the `Iterable` interface.

Implementing the iterator interfaces is somewhat involved and follows a common pattern with considerable

[2] Not included is the syntax definition. Implemented as a stand-alone program, this figure does include 47 lines of I/O code, imports, and comments, all written in the Stratego program transformation language [6].

boilerplate code. A complementary language extension that deals with this problem is that of *iterator generators*, a feature supported by other languages such as Python and Sather [27]. Like the enhanced for loop, this feature abstracts away from the underlying interfaces. Consider Figure 9, which uses it to define an iterator method that splits a string at every space character. It loops over the results of a call to `String.split()`, and uses the `yield` statement to return all substrings with less than four characters as elements of the resulting iterator. The iterator method operates as a coroutine: control is "yielded" back to the client of the iterator at every `yield` statement. When the client requests the next element, the iterator method is resumed after the last `yield` statement.

In earlier work, we implemented the `yield` statement as a source-to-source transformation, abstracting over regular Java control flow statements [19]. The `yield` statement introduces a form of *unstructured control flow*: the method can be entered at any arbitrary point it is used. To express this in a legal Java program, the desired control-flow graph must be transformed to make use of Java's *structured* control flow statements, a non-trivial problem, also faced when decompiling bytecode [26]. We accommodated for this by using a `switch` statement and `case` labels at every point in the control flow graph. Since the Java `switch` statement can only be used in a structured fashion (i.e., it disallows `case` labels inside blocks of code nested in it), all existing control flow statements in the method must be rewritten to become part of the `switch`. This turned out to require significant effort, essentially re-implementing a considerable part of the existing Java language.

The infomancers-collections library [11] aims to avoid the complications of source-to-source transformation. It effectively hides the language extension from the Java compiler, by using a dummy `yield()` method that can be invoked from anonymous classes that implement the iterator interfaces. A regular Java compiler can then be used to compile the code, unaware of the special semantics of these invocations. The resulting bytecode is then altered by the library, replacing the dummy invocations with an actual implementation, modifying the (unstructured) control flow of the method. This is a rather intricate process that requires the use of a bytecode manipulation library to generate a new class. Since the Java compiler is oblivious to the special semantics of the dummy invocations, it is unable to perform proper control flow analysis during the initial compilation, which may lead to unexpected results. In particular, compilers may optimize or otherwise generate code that violates the stack-neutrality property of statements (see Section 3.3), which can result in invalid programs after inserting new jump instructions.

In our approach, we treat the language extension as a form of syntactic sugar that can be projected to the base language (i.e., Java/bytecode), just like the enhanced for

```
public Iterable<String> getShortWords(String t) {
  String[] parts = t.split(" ");
  for (int i = 0; i < parts.length; i++) {
    if (parts[i].length() < 4) {
      yield parts[i];
    }
  }
}
```

**Figure 9.** Iterator definition with the `yield` statement

```
class ShortWords implements Iterator<String> {
  int _state = 0;
  String _value;
  boolean _valueReady;

  String[] _parts;
  int _i;

  private void prepareNext() {
    if (_valueReady || _state == 2) return;
    if (_state == 1) `goto afterYield;

    _parts = t.split(" ");
    for (_i = 0; _i < _parts.length; _i++) {
      if (_parts[_i].length() < 4) {
        _state = 1;
        _valueReady = true;
        _value = _parts[_i];
        return; // yield value
afterYield:
      }
    }
    _state = 2;
  }

  public String next() {
    prepareNext();
    if (!_valueReady)
      throw new NoSuchElementException();
    _valueReady = false;
    return _value;
  }

  public boolean hasNext() {
    if (!_valueReady) prepareNext();
    return _valueReady;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

**Figure 10.** Iterator definition, generated from Figure 9.

loop. For this, the `yield` statement is rewritten to a `return` statement to exit the iterator method, in conjunction with a small, bytecode-based jump table at the beginning of the method for re-entry. Based on a finite state machine model of the method, a field `_state` is maintained to indicate the next method entry point. Consider Figure 10, which shows the class that will be generated local to the original `getShortWords()` method. It highlights all lines of code that directly correspond to the original method. The `prepareNext()` method is central to this implementation, and includes an adapted version of the original method body. Its control flow is adapted by the addition of unstructured control flow primitives, in the form of bytecode instructions. In this example, we implement this using a jump table and `goto` instructions, embedded in Java using the backtick (`) operator. Alternatively, a `tableswitch` instruction could be used to form an unstructured switch statement.

It is not possible to statically determine if the iterator will return a value or not for a given state, and we cannot make assumptions on the order of invocation of the `Iterator.next()` and `Iterator.hasNext()` interface methods. Therefore, a call to `prepareNext()` is used in both these methods, caching the value as necessary. To ensure persistence of local variables during the lifetime of the iterator, all local variables must be replaced or shadowed by fields.

The amount of code required to define a custom iterator by implementing the iterator interfaces (Figure 10) illustrates the increase in productivity that can be gained by abstracting over this using iterator generators (Figure 9). The projection to Java/bytecode that realizes this abstraction is relatively straightforward, as it maintains the structure of the original method, unlike in our earlier source-to-source approach. On the other hand, this approach also avoids the complexity of the low-level, purely bytecode-oriented approach, eliminating the need for special libraries and using the convenience and familiarity of the Java language for most part of the implementation.

### 2.4 Assimilating Expression-Level Extensions

Embedded domain-specific languages add to the expressivity of general-purpose languages, combining domain-specific constructs with the general-purpose expressivity of the host language. Examples include embedded database queries, or integrated regular expressions. We have explored DSL embedding with Java as a base language and described MetaBorg [7], a general approach for DSL embeddings. In that context, we have coined the term *assimilation* for the transformation that melds the embedding with its host code. Assimilation preserves the semantics of language extension constructs, while making it possible to compile them using the base language compiler.

Small, statement- or expression-level language extensions are especially well-suited for assimilation. They can often be assimilated locally to an implementation in the host language (often API-calls), without disturbing the surrounding code. However, for specific kinds of language extensions this is not possible. One class of such extensions is that of language extensions that take the form of *expressions* but require assimilation to *statements* in the host language (e.g., to use control flow constructs or declare local variables, which is not possible in expressions). In-place assimilation to an expression does not suffice in these cases, because Java and similar languages do not allow nesting of statements in expressions. One technique to overcome this problem is using an intermediate syntax in the form of *expression blocks* [7]. These enable the use of statements in expressions, facilitating in-line expression assimilation.

```
System.out.println(e1 ?? e2);
```

(a) The `??` operator, with operands of type *T*.

```
System.out.println(
  {| T lifted = e1; // evaluate e1 only once
     if (lifted == null) lifted = e2;
   | lifted
   |});
```

(b) In-line assimilation to an expression block.

```
T lifted = e1;
if (lifted == null) lifted = e2;
System.out.println(lifted);
```

(c) Lifted to pure Java.

**Figure 11.** Assimilation using statement-lifting.

Expression blocks take the form

$$\texttt{\{|}\ \textit{statements}\ \texttt{|}\ \textit{expression}\ \texttt{|\}}$$

where the statements are executed before the evaluation of the expression, and the value of the expression block is the value of the embedded expression. A separate, generally applicable normalization step in the form of *statement-lifting* can be used to *lift* the statement component of the expression to the statement level [7]. Consider for example the C# coalescing operator:

```
e1 ?? e2
```

This operator returns the value of expression *e1* if it is non-null, or otherwise the value of *e2*. It could be used in conjunction with Java's nullable (or "boxed") types and its support for auto-unboxing, providing a default value when converting to a non-nullable (i.e., primitive) type.

Consider Figure 11, which shows how the coalescing operator can be assimilated to regular Java statements. In Figure 11(a), the operator is used in a method call. Using an expression block, it is straightforward to translate it in-line to regular Java statements, as seen in Figure 11(b). Finally, in Figure 11(c), the expression block is lifted to the statement level.

```
Iterator<Integer> it = ...;

for (int i=i0; it.hasNext(); i = it.next() ?? 0) {
   ...
}
```

(a) The coalescing operator in a loop.

```
Iterator<Integer> it = ...;

Integer lifted = next();
if (lifted == null) lifted = 0;
for (int i = i0; it.hasNext(); i = lifted) {
   ...
}
```

(b) Incorrect program after statement-lifting.

**Figure 12.** Statement-lifting in a `for` loop.

Unfortunately, proper statement-lifting is not trivial to implement: it requires *syntactic knowledge* of every language construct that an expression may appear in, including various statements, other expressions, and possibly other language extensions. Furthermore, it requires *semantic knowledge* of the language, as it is not always sufficient to simply move statements to the syntactic statement level. For instance, the simple lifting pattern from [7] cannot be applied if the expression is the operand of short-circuiting operators (e.g., `||` in Java) or the conditional of a `for` loop (see Figure 12). In these cases, simply lifting the translated statements to the statement level changes the semantics of the program.

In the bytecode core language, there is no statement level or expression level. This distinction only exists in Java source code, and is simply a consequence of the Java syntax. Thus, we can overcome this limitation by assimilating the operator directly to the bytecode core language, using instructions in place of the original operator (see Figure 13).

Given that the core language is enriched with constructs of the more convenient Java language, we can also apply the complete, mixed language to synergistic effect and assimilate the operator in a more elegant fashion. Consider Figure 14, which assimilates the coalescing operator to Java statements, embedded in a bytecode block. In addition to these statements, we use a `push` pseudo-instruction to place the `lifted` variable on the stack (we elaborate on the role of the stack and the `push` instruction in Section 3.1). This value forms the result of the expression, and is used as the argument of the call to `System.out.println`. Like expression blocks, the bytecode fragment can contain any number of statements and a single resulting expression, making normalization of the expression block extension to a bytecode block trivial. As such, this pattern effectively does away with the complications associated with statement-lifting, and thereby simplifies the implementation of expression-level language extensions using statements.

```
System.out.println(`[
  push `e1;
  astore lifted;
  ifnull else;
    push `e2;
    goto endif;
  else:
    aload lifted;
  end:
]);
```

**Figure 13.** Assimilation of the `??` operator to bytecode.

```
System.out.println(`[
  `T lifted = e1;
  `if (lifted == null) lifted = e2;
  push `lifted;
]);
```

**Figure 14.** Assimilation to bytecode-embedded Java.

## 3. Realization of the Base Compiler

The Dryad Compiler operates by normalization of the mixed Java/bytecode language to its core (bytecode) language. We implemented a parser for the language as an extension of the regular Java language, using the modular syntax definition formalism SDF [4]. The language is normalized through normalization rules expressed in the Stratego [6] program transformation language. In the remainder of this section, we give an overview of the design of the language and its compiler.

### 3.1 Language Design

Key to compilation by normalization is the notion of a core language that is – often incrementally – extended with new abstractions to form a rich, high-level language. For the Dryad compiler, we build upon the existing bytecode language, an assembly-like core language, and mix it with the standard Java language [14]. The two syntax forms are integrated using the *backtick operator* `` ` ``, which toggles the syntax form of a fragment of code. Figure 15 gives an overview of the syntax of the language. In this figure we use italics to refer to other symbols in the syntax, and an overline to indicate lists of symbols (e.g., $\overline{T\,x}$ indicates a list of type/identifier pairs). For brevity, we left out some of the more advanced Java language constructs. For mixing of class, method, and field declarations, the `` ` `` notation is optional, and was also left out from this overview.

The bytecode assembly language we use shares similarities with existing languages such as Jasmin [25] and its derivatives. It provides a somewhat abstracted representation of the underlying binary class format. For instance, it allows the use of symbolic names rather than relative and absolute *offsets* for locals and jump locations. Like Jasmin, our rep-

**General**

| | | |
|---|---|---|
| $p$ | ::= $\overline{cd}$ | Program (start symbol) |
| $T$ | ::= $C$ \| void \| int \| int \| long \| double \| float \| boolean \| char \| byte | Types |

**Java**

| | | |
|---|---|---|
| $cd$ | ::= class $C$ extends $C$ { $\overline{fd}$ $\overline{md}$ $\overline{cd}$ } | Class declaration |
| $md$ | ::= $C$ ($\overline{T\ x}$) { super ($\overline{e}$); $\overline{s}$ } \| $T\ m$ ($\overline{T\ x}$) { $\overline{s}$ } | Method/constructor declaration |
| $fd$ | ::= $T\ f$; | Field declaration |
| $s$ | ::= { $\overline{s}$ } | Statement block |
| | \| $e$; | Expression statement |
| | \| $C\ x = e$; | Local variable declaration |
| | \| if ($e$) $s$ else $s$ \| while ($e$) $s$ \| for ($e$; $e$; $e$) $s$ \| return; \| return $e$; | Control flow |
| | \| throw $e$; | Throw exception |
| | \| synchronized ($e$) { $\overline{s}$ } | Synchronization |
| | \| try { $\overline{s}$ } catch($C\ x$) { $\overline{s}$ } | Exception handling |
| | \| $l$: | Label |
| $e$ | ::= $x = e$ \| $x$ | Local variables |
| | \| ($T$) e | Cast |
| | \| $e + e$ \| $e$ - $e$ | Basic operators |
| | \| $e.m$ ($\overline{e}$) | Method invocation |
| | \| new $C$ ($\overline{e}$) | Object creation |

**Bytecode**

| | | |
|---|---|---|
| $cd$ | ::= classfile $C$ extends $C$ fields $\overline{f}$ methods $\overline{md}$ | Class declaration |
| $md$ | ::= $C$ ($\overline{T\ x}$ : $T$) [ $\overline{I}$ ] \| <init> ($\overline{T\ x}$ : $C$) [ $\overline{I}$ ] | Method/constructor declaration |
| $fd$ | ::= $f$ : $T$ | Field declaration |
| $I$ | ::= catch ($\overline{l}$ : $C$) [ $\overline{I}$ ] | Exception handling |
| | \| *(see Figure 16)* | Bytecode instruction |

**Mixing**

| | | |
|---|---|---|
| $s$ | ::= `$I$ \| `[$\overline{I}$] | Bytecode statement |
| $e$ | ::= `$I$ \| `[$\overline{I}$] | Bytecode expression |
| $I$ | ::= `$s$ \| `[$\overline{s}$] \| push `$e$ | Embedded Java |

**Tracing**

| | | |
|---|---|---|
| $s$ | ::= trace ($o$) [ $\overline{s}$ ] | Statement trace |
| $e$ | ::= trace ($o$) [ $e$ ] | Expression trace |
| $I$ | ::= trace ($o$) [ $\overline{I}$ ] | Instruction trace |
| $md$ | ::= trace ($o$) [ $\overline{md}$ ] | Method trace |
| $o$ | ::= $\overline{s}$ @ $loc$ \| $e$ @ $loc$ \| $\overline{I}$ @ $loc$ \| $\overline{md}$ @ $loc$ | Trace originating code |
| $loc$ | ::= $path$:$i$:$i$ | Location specification |

**Figure 15.** Syntax for the mixed Java/bytecode language.

| **Arithmetic** | **Stack** | **Arrays** | **Control flow** | **Comparison** | **Conversions/truncations** |
|---|---|---|---|---|---|
| add | ldc $c$ | aload | ifeq $l$ | lt | x2i |
| div | ldc2_w $c$ | astore | ifne $l$ | gt | x2l |
| mul | new $C$ | arraylength | goto $l$ | eq | x2d |
| neg | pop | newarray $T$ | $l$: | le | x2f |
| rem | pop2 | multianewarray $T\ n$ | athrow | ge | i2b |
| sub | dup | | return | ge | i2s |
| shl | dup_x1 | **Fields** | xreturn | | i2c |
| shr | dup_x2 | getstatic $C.f$ : $T$ | tableswitch $n$ to $n$: $\overline{l}$ default: $l$ | **Miscelleneous** | checkcast $C$ |
| ushr | dup2 | putstatic $C.f$ : $T$ | lookupswitch $\overline{n : l}$ default: $l$ | instanceof $C$ | |
| xor | dup2_x1 | getfield $C.f$ : $T$ | | monitorenter | **Invocations** |
| and | dup2_x2 | putfield $C.f$ : $T$ | | monitorexit | invokevirtual $C.m(\overline{T} : T)$ |
| or | swap | | | nop | invokestatic $C.m(\overline{T} : T)$ |
| inc | | | | breakpoint | invokeinterface $C.m(\overline{T} : T)$ |
| dec | | | | | invokespecial $C.m(\overline{T} : T)$ |

**Figure 16.** The (reduced) bytecode instruction set.

resentation allows constants to be used in-line, eliminating the need to maintain a separate constant pool with all constants and types used in a class. Still, our representation remains close to the underlying binary bytecode instructions. Most notably, it preserves the *stack machine*-based nature of the instruction set. More abstract representations are for example provided by Soot [39], but this does not match our design goal of exposing the complete core language functionality. Instead, we embrace the operand stack to provide low-level operations, and use it for interoperability between the two languages. Figure 15 shows the basic elements of the bytecode language, while Figure 16 gives an overview of the instruction set. Note that this figure shows a reduced bytecode instruction set, using overloaded instructions, on which we elaborate in Section 4.2. For compatibility, we also support the complete, standard set of instructions, which can be mapped to the reduced set.

***Interoperability between Java and Bytecode*** Java and bytecode have many of the same basic units of code, e.g. classes, methods, and fields. In our mixed language, the Java and bytecode representations of these units can be combined arbitrarily. How to combine the two languages at a finer level of granularity, i.e. inside method bodies, is less obvious. In Java, a method body is a tree-like structure of statements that may contain leafs of expression trees. Bytecode methods consist of flat, scopeless lists of instructions. Perhaps the most elementary form of mixing at this level is to allow bytecode fragments in place of statements, forming *bytecode statements*, and vice versa. This – among other things – allows statement-level separate compilation and basic insertion of source code into compiled methods. At the statement level, state is maintained through local variables and fields. These can be shared between Java and bytecode statements, as the language uses a common, symbolic representation for both fields and local variables.

Expressions can be used as the operands of statements or other expressions, passing a value to the enclosing construct. *Bytecode expressions* are fragments of bytecode that can be mixed into expressions. They are conceptually similar to bytecode statements and share the same syntax for embedding. However, as expressions, they must produce a resulting value. At the bytecode level, such values are exchanged using the *operand stack*. For example, the *load constant* instruction (`ldc`) pushes a value onto the stack, and the `add` instruction consumes two values and pushes the addition onto the stack. Such instructions can be used to form a legal bytecode expression:

```
int i = `[ ldc 1; ldc 2; add ];
```

Vice versa, the `push` pseudo-instruction places the value of a Java expression onto the stack:

```
push `"Java" + "expression";
```

```
void locals() {
  { // Java blocks introduce a new  scope
    `[ ldc 1; store var ];
    System.out.println(var);
  }

  // 'var' is out of scope and can be redefined
  int var = 2;
}
```

**Figure 17.** Local variable scoping.

### 3.2 Name Management and Hygiene

Local variables shared between Java and bytecode follow the standard Java scoping rules. Bytecode has no explicit notion of declaration of variables, only of assignment (using the `store` instruction). In the mixed language, the first assignment of a local variable is treated as its declaration, and determines its scope (see Figure 17). In regular bytecode there exists no notion of local variable scoping; all scopes are lost in the normalization process. To ensure proper hygiene during normalization, this means that all local variable identifiers – both intermediate and user-defined – need to be substituted by a name that is unique in the scope of the entire method. For the example in Figure 17, two unique variables can be identified in separate scopes. After normalization, these variables get a different name.

### 3.3 Typechecking and Verification

Typechecking is an essential part of the compilation of a statically typed language. The Java Language Specification specifies exactly how to perform name analysis and typechecking of the complete Java language [14]. The analyses provide type information required for the compilation (e.g., for overloading resolution) and give feedback to the programmer in case of errors. We employ a Stratego-based typechecker for Java, which is implemented as a tree traversal that adds semantic information to the abstract syntax tree.

The typechecker for the mixture of Java source and bytecode is a modular extension of a typechecker for Java source code. The source code typechecker is designed to handle language extensions by accepting a function parameter that is invoked for extensions of the Java source language. The extension can inspect or modify the environment of the typechecker, recursively apply the typechecker, or completely take over the traversal of the abstract syntax tree. For the mixed Java/bytecode language, the extended language constructs are the mixing constructs of Figure 15, where bytecode is embedded in Java. If any of these constructs are encountered, the extension of the typechecker takes over the traversal by switching to the bytecode verifier. The current typechecker environment is passed to the verifier, ensuring all variables and other identifiers are shared with the surrounding code. The verifier in turn returns the resulting operand stack of the bytecode fragment.

For bytecode expressions, these must consist of a single type, which is passed back to the Java typechecker. Vice versa, the bytecode verifier invokes the source code typechecker for any embedded Java constructs, using it to resolve the types of embedded Java expressions and variables declared in Java statements.

***Stack-Consistency of Mixing Constructs*** The bytecode verifier ensures correct stack behavior and type safety of a program. In the mixture of Java and bytecode, we impose specific rules on the stack behavior to ensure safe interoperability and natural composability between Java and bytecode fragments. These are verified by the bytecode verifier.

One restriction we impose on bytecode expressions is that they must leave a single value on the stack, just like Java expressions[3]. Leaving no value on the stack is considered an error. Likewise, leaving more than one value on the stack is considered illegal as this can lead to a stack overflow when it is not properly cleaned up.

Unlike Java expressions, statements do not leave a value on the stack; they compile to a sequence of *stack-neutral* bytecode instructions. That is, they may use the stack for intermediate values during evaluation, but must restore it to its original height afterwards. This ensures that all Java statements can safely be placed in a loop construct, without the risk of a stack overflow or underflow. Even more so, the JVM actually *requires* that methods have a fixed maximum stack height, disallowing loops with a variable stack height [22]. For compound statements (e.g., `for`, `while`), stack-neutrality extends to the contained statements: the input stack of all nested statements must be the same as the stack outside the containing statement. This restriction goes hand in hand with the JVM restriction that at any point in a method, the types of the values on the stack must statically be known, and must be the same for all incoming edges of the control flow graph. Any other jumps to a location are considered illegal. The restriction can always be satisfied on the statement level based on the property of stack-neutrality. A jump from one statement to another is therefore always legal. To preserve this property in the mixed language, we place the same restriction of stack-neutrality on bytecode statements: only bytecode sequences that restore the stack height are legal bytecode statements. This ensures that fragments of Java and bytecode can be composed naturally without risk of stack inconsistencies, and ensures support for arbitrary control flow between Java and bytecode statements.

***Verifier Implementation*** The JVM specification [22] includes a description of how a bytecode verifier should operate. It describes the process as a fix-point iteration, where a method's instructions are iterated over a number of times until all possible execution paths have been analyzed. Because of restrictions on the form of a method and its instructions, this is a straightforward process. One restriction is that for all instructions the effect on the stack can be statically determined. For instance, for a method invocation instruction, this means that it must specify the arguments it takes from the stack and what the return type is. This also means that the verification can be done in an intraprocedural setting, using a static, global environment. This allows it to be tightly integrated into the Java source code typechecker, as it can be used to verify individual fragments at a time.

We implement our analysis using a monotone framework [18, 1]. This representation allows a generic formulation of such analyses using a specific set of operators and constants for each framework instance. Bytecode verification is a forward data-flow analysis, and assumes an empty stack at the beginning of a method. The operators that determine the types on the stack through fix-point iteration are defined as follows:

- *The transfer function* determines the resulting stack of an instruction, given the input stack. For instance, for an `ldc` instruction, a single value is loaded onto the stack.
- *The join operation* merges the stack at branch targets (i.e., labels and exception handlers), unifying the types of the stack states if they are compatible (e.g., `String` and `Integer` unify to the `Object` type).

### 3.4 Source Tracing

During compilation, code often undergoes multiple normalization steps. If there are any errors in any of these steps, they should reflect the originating source code, and not the intermediate code. To maintain this information, we introduce *source tracing* information that indicates the source of a fragment of code, in the form of a path and location of the originating file and an abstract syntax tree reference. Source tracing is explicitly available as a language feature, using the `trace` keyword (see Figure 15). This ensures maximal, source-level interoperability with other tools that may exist in the compilation chain. Consider Figure 18, which shows a class file compiled using the traits extension. In addition to language-level support, we provide an API to transparently include this information in the result of normalization rules in Stratego. Using this facility, extensions of the compiler can fall back on the error reporting and checking mechanisms already provided by the base compiler, and may catch these errors or to improve the user experience.

Source tracing information is also used to generate debugging information in the produced Java class file. This takes the form of a table that maps instruction offsets to original source file locations, and enables stepping through the code using a debugger, as well as accurate position information in run-time exceptions.

### 3.5 Data-Flow Analysis on the Core Language

Leveraging the bytecode verifier and the source tracing facility, we implemented analyses such as unreachable code

[3] Actually, `void` method invocations are an exception to this, but these cannot be nested into other expressions.

```
classfile Shape
methods
  trace (void draw() {...} @ TDrawing.java:2:2) [
    public draw(void) [ ... ]
  ]
  ⋮
  trace (... getVertices() {...} @ Shape.java:9:2)
  [
    public getVertices(void) [ ... ]
  ]
```

**Figure 18.** Compiled methods with source tracing.

```
normalize-finally:
  ⟦ synchronized (e) { bstm* } ⟧ →
  ⟦ Object locked = e;
    try {
      `[ push `locked; monitorenter ];
      bstm*
    } finally {
      `[ push `locked; monitorexit ];
    }
  ⟧
```

**Figure 19.** Normalization of the `synchronized` statement.

analysis and checking for definite assignment at the bytecode level. By performing these analyses on the core language, we can formulate them as a monotone framework instance, or make use of the information already provided by the verifier. Furthermore, dealing with a normalized form of the language, these analyses have fewer constructs to deal with, reducing their complexity.

Reachability of statements can be determined by use of the regular bytecode verifier: it returns a list of stack states, leaving the stack states of all unreachable instructions uninitialized. Using source tracing information, the Java statements that generated such code (which is illegal in Java) can be reported as unreachable. For testing definite assignment, we formulated another monotone framework instance that maintains a list of all assigned variables, matching against the `load` and `store` instructions. Through iteration it determines whether or not variables are assigned before use, and if variables marked `final` are not assigned more than once. While the core language has no direct notion of `final` variables, this information can be retrieved using the method's source tracing information.

In addition to verification, we applied these analysis techniques for optimizations on the core language, including dead code elimination and peephole optimizations. Similarly, other optimizations such as constant propagation can be implemented at this level.

## 4. Normalization Rules for Code Generation

Using normalization rules, high-level language constructs can be rewritten to lower-level equivalents, e.g. from constructs of language extensions to the Java/bytecode language, and then to the core bytecode language. We express these rules as Stratego rewrite rules, which take the form

$L$`:` $p_1$ → $p_2$ `where` $s$

where $L$ is the name of the rule, $p_1$ is the left-hand side pattern to be matched against, and $p_2$ is the result of the rewrite rule. The `where` clause $s$ may specify additional conditions for the application of the rule, or may declare variables to be used in the result. Using the technique of *concrete syntax embedding*, Stratego rewrite rules may use the concrete syntax of the language being transformed as patterns [43, 6]. These concrete syntax fragments are parsed at compile-time and converted to equivalent match or build operations using the abstract syntax tree. These patterns are typically enclosed in "semantic braces":

```
normalize-if:
  ⟦ if (e) s ⟧ →
  ⟦ if (e) s else ; ⟧
```

this rule normalizes the basic, single-armed `if` statement to the more general two-armed `if` statement, with the empty statement in the `else` clause. This normalization ensures that other rules only have to deal with the latter form.

### 4.1 Mixed Language Normalization

Rather than directly normalizing from high-level language constructs to bytecode, this is often done through a series of small normalization steps. Often, these are rules that produce a mixture of Java and bytecode, which is further normalized in later steps. Iterative rule application and leveraging the primitives made available in the mixed Java/bytecode language make it possible to normalize the complete language using relatively small steps that focus on a single aspect.

Consider Figure 19, which demonstrates a normalization rule for the standard Java `synchronized` statement. It is rewritten to a mix of more low-level Java statements and the `monitorenter` and `monitorexit` unbalanced synchronization instructions. The resulting fragments can in turn be normalized to core language constructs themselves.

We apply normalization rules in both the core compiler as well as in the language extensions. For example, in Section 2.2 we discussed the extension of Java with traits, mapping trait classes to Java abstract classes for separate compilation. Consider Figure 20, which illustrates this mapping by means of a normalization rule. This rule makes use of a `where` clause, and depends on two helper functions: `trait-name`, which determines and registers a new name for the resulting abstract class, and `trait-methods`, which determines the set of methods to be included. Given the mapping of traits to regular Java classes, the core normalization rules can proceed to normalize the result further down to bytecode.

```
normalize-trait:
  ⟦ trait x trait* { method1* } ⟧ →
  ⟦ abstract class y { method2* } ⟧
  where
    y        := <trait-name> x
    method2* := <trait-methods> (method1*,
                                 trait*)
```

**Figure 20.** Normalization of traits to abstract classes.

### 4.2 Pseudo-Instruction Normalization

The JVM supports over two hundred instructions, ranging from very low-level operations, such as manipulation of the stack, to high-level operations, such as support for synchronization and arrays. Many of these instructions are specialized for specific types, such as the `iadd` and `fadd` instructions that respectively add two integers or two floats. Rather than requiring the code generator to select the proper specialization of `<T>add`, we introduce overloaded pseudo-instructions to defer this to the final step of the compilation. This simplifies the implementations of specific language extensions, as they do not have to reimplement this specialization process.

The pseudo-instructions form a reduced set of only 67 essential instructions (see Figure 16). These are normalized to regular bytecode instructions, based on the type information provided by the verifier. For this, the verifier is extended with a transformation operator, which uses the type information provided by the transfer function of the verifier to replace all overloaded instructions with type-specific, standard bytecode instructions.

Consider Figure 21, which illustrates how very different bytecode instructions must be generated for identical Java expressions if they operate on different types. Instructions such as `iload` and `iadd` have a type prefix, and the `dup` or `dup2` instruction depend on the number of words that a value occupies (i.e., two for `long` values). In the reduced set, identical instructions can be used for many such patterns, thus simplifying normalization rules from Java (or another language) to the instruction set, and reducing their total number.

| Java | Reduced instruction set | Regular bytecode |
|---|---|---|
| `intVar++` | `load intVar` | `iload_1` |
| | `dup` | `dup` |
| | `inc` | `iconst_1` |
| | | `iadd` |
| | `store intVar` | `istore_1` |
| `longVar++` | `load longVar` | `lload_1` |
| | `dup` | `dup2` |
| | `inc` | `lconst_1` |
| | | `ladd` |
| | `store longVar` | `lstore_1` |

**Figure 21.** Instructions generated for identical Java expressions of type `int` and `long`.

## 5. Discussion

***Compilation by Normalization in Practice*** From an external view, the Dryad Compiler has no discernible stages, and simply normalizes the input code to resulting code that can be normalized again. At each normalization step, the transformed code forms a valid Java/bytecode program. This design enables extensions – as well as built-in features – to make use of a wider range of language constructs, and prevents scattering their implementation across different compilation stages. Furthermore, it allows for different (separate) compilation scenarios than possible with conventional open compilers.

Still, the internal architecture of the Dryad Compiler does employ separate, discernible components. For instance, it employs a (global) semantic analysis phase, based on the Dryad typechecker component. As such, it does not conform to what may be an idealized image of how a normalizing compiler should work: by pure, iterative application of declaratively defined normalization rules. As it is, the Dryad Compiler uses strategies to maintain a degree of control over the application order of normalization rules. To simplify their implementation, the rules are formulated without special logic for preserving the semantic information collected in the analysis. This means that in some cases, the analyzer must be reapplied. While this may not be ideal, this architecture does not hinder the applications presented here: for the extensions, the Java/bytecode language acts as the interface of the compiler. The internal implementation of the compiler, how this language is normalized, and how this design may (and likely *will*) change in the future, is of no concern to the developer of an extension.

***Core Language-Based Analysis*** We perform a number of data-flow analyses and optimizations on programs after they have been normalized to the core language. By doing these at this level they can be applied independently of the source language. The analyses have fewer constructs to deal with than for source code, reducing their complexity. Still, reducing a program to the bytecode form can also mean a loss of precision, due to the missing high-level structure and the reduced size of the code window used by transfer functions. Logozzo and Fähndrich [23] offer a discussion comparing the relative completeness of either approach.

***Composition of Language Extensions*** Modular definition of language extensions aids in the composability with other extensions, possibly developed by different parties. Ideally, these can be imported into a language without requiring any implementation level knowledge of the extensions. At the syntax level, this can be achieved using a modular syntax definition formalism such as SDF [4]. On the semantic level, the primary determinant of the compositionality of extensions is the type of analysis and transformations that

are performed (global or local). For global transformations, an ordering of application must be determined, which requires implementation-level knowledge. Thus, composable language extensions should use primarily local transformations; composition of global transformations is an orthogonal issue. Using small normalization steps, facilitated by expression blocks, pseudo-instructions, and the increased expressivity of the general Java/bytecode language, many extensions can be expressed using local transformations.

## 6. Related and Future Work

***Compiler Extension*** Extensible compilers, such as the JastAdd Extensible Java compiler [13], ableJ [42], Polyglot [28], and OpenJava [36] provide a foundation for compilation of language extensions. Polyglot, ableJ, and OpenJava follow the front-end extension approach; they offer an extensible semantic analysis stage and aid in projection to the base language. JastAdd on the other hand provides its own, modular back-end. Our approach does not preclude the use of these tools, but can add to them by offering at once the Java language for projection and direct use of the underlying bytecode primitives or inclusion of compiled class fragments.

*Forwarding*, as supported by ableJ, avoids the need to implement full semantic analysis for a language extension [42, 41]. Instead, forwarding allows the metaprogrammer to define the projection of a language construct to the base language, and by default applies semantic analysis over the forwarded program fragment. Our work is related to forwarding, as we similarly define a mapping from a source language to a base language, using normalization rules. Semantic analysis can be performed on the projected code, and any errors can be reported in reference to the user code using source tracing. Unlike in ableJ, we introduce core language constructs into the source language, to increase its expressivity and to facilitate normalization to this core language form.

Macro-based systems, such as JSE [3] and the Jakarta Tool Suite [33] implement direct projection to a base language, forgoing semantic analysis. This can lead to confusing error messages in references to locations and constructs in the generated code. We support semantic analysis on the source language using an extensible typechecker and verifier, and provide source tracing facilities to ensure any errors on code resulting from transformations can be traced back to user code.

Source tracing is a technique related to origin tracking [40], which maintains the origins (i.e., source terms and positional information) of terms during rewriting. We extend this notion by defining language constructs to maintain and share this information at the source level, to help interoperability between tools and simplify the internal representation.

***Language Composition*** A similarity can be drawn between the mixed Java/bytecode language presented here and existing languages such as C++ and Ada that allow inline assembly code languages. These too can be used for optimizations or obfuscation and provide access to low-level operations. Assembly code, however, is a representation of instructions for a specific CPU architecture, and therefore is much more low-level and less safe than bytecode. This makes it more difficult to use it for composition of source and compiled code.

The Java/bytecode language also shows similarities with the Jeannie language [17], which integrates Java with C code, compiling to code that makes use of the Java Native Interface (JNI) [21], the JVM's standard foreign function interface. The C language can be used for accessing platform functionality or (legacy) libraries, but does not actually form the core language of the Java platform. Similar to the Dryad Compiler, Jeannie performs semantic analysis on the combined language, and introduces a number of bridging constructs for conversions and interoperability between the two constituent languages.

***Different Platforms*** We have applied the *compilation by normalization* architecture for the Java platform. Java provides a safe, mature, and ever-evolving platform for developing applications, and has been frequently used to host other languages and language extensions [15, 24, 32]. A similar architecture can be realized for other platforms based on a bytecode language. For instance, .NET may be an interesting case given it was designed from the ground up to host multiple languages, and includes support for *unsafe code*, which allows direct manipulation of object pointers. Other platforms provide a more high-level intermediate language, such as the Glasgow Haskell Compiler CORE language [31]. CORE is not a strict subset of the Haskell language and cannot be directly compiled [38], but this and similar languages make good candidates for use as a core language that is grown to a more extensive, (not necessarily existing) high-level language, by introducing new abstractions, while preserving the core functionality.

***Other Language Extensions*** We presented a number of compiler extensions, demonstrating how the Dryad Compiler can facilitate their implementation. Future work could include other extensions, such as adding full support for aspect-oriented programming (AOP) [20], building upon the implementation of open classes (i.e., intertype declarations in AOP) and composition of code at the statement and expression level. Full aspect weaving can be performed by composition of compiled classes with aspect source code, or vice versa. A related design is used for the AspectBench Compiler (abc) [2], which applies aspect weaving on the Jimple language, a three-address (stackless) representation of bytecode. The abc compiler uses Soot [39] to (de)compile Java and bytecode to Jimple. Thereby they avoid some of the complexities associated with bytecode-level weaving. Using

the Java/bytecode language instead would enable the direct insertion of regular Java code into class files for both advice and any dynamic checks or other wrapping code that accompanies it. By providing typechecking and verification for the combined language, as well as the guarantee of stack-neutrality of inserted statements, we provide the same level of safety as is possible with weaving using Jimple. Similar techniques can be used in the implementation of annotation processing tools (such as Java APT [35]), which typically operate purely based on source code.

There is a growing interest in running dynamic programming languages on the JVM. These compilers, such as Jython [30] for the Python language, typically compile to Java source code. Using the Dryad Compiler as a basis, such compilers can make use of specific bytecode features and optimizations, such as the newly considered dynamic invocation instruction.[4] Similarly, other Java code generators could be retrofitted to generate selected bytecode features. The F2J Fortran compiler, for instance, used to generate Java source code, before it was re-engineered to directly generate bytecode instead, to support `goto` instructions and specific optimizations [32]. JCilk also generates Java code, to assimilate an extension with fork-join primitives and exception handling for multithreaded computations [10]. Similar to the problems with implementing the `yield` statement as a source-to-source transformation (Section 2.3), this significantly complicates the control flow semantics. However, rather than directly generating bytecode for the complete JCilk language, the current implementation depends on a modified version of the GNU Compiler for Java as its backend. It would be straightforward to use the Dryad Compiler instead, making use of the mixed language and introducing support for source tracing.

Java's source model of single inheritance with interfaces does not always match that of a given language that is intended to target the JVM. For instance, a restriction of Java interfaces is that only one method with a given signature may exist. This renders interfaces incompatible if they define methods with the same signature but with a different return type. For generated code, such as interfaces generated from annotations, this can be a restricting or incalculable factor. At the bytecode level, this restriction does not exist. Additionally the `synthetic` modifier, used for instance on bytecode methods of inner classes, can be used to mark methods inaccessible from user code (JVM Spec. [14], §4.7.6). It can be used to hide multiple methods with the same signature, and can enable friend class semantics for generated code.

## 7. Conclusions

To increase programmer productivity, language extensions, both domain-specific and general-purpose, have been and continue to be developed. These may generate source code or bytecode; either approach has its advantages. Mixing source code and bytecode, a new language can be formed that has a synergistic effect, resulting in a language that at once provides the low-level expressivity of bytecode and the convenience and familiarity of Java. The combined language allows rapid development of language extensions through normalization steps that can remain loosely coupled from the base compiler. Using intermediate forms such as expression blocks and pseudo-instructions, these steps remain relatively small and maintainable.

Mixing source and bytecode opens the doors for new, more fine-grained forms of separate compilation, providing a foundation for composition of source and bytecode classes up to instruction- and expression-level precision. By use of source tracing, these composed fragments of code can be traced back to their original source files, enabling accurate location information for error messages and debugging.

***Acknowledgements*** This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

[4] JSR 292: http://www.jcp.org/en/jsr/detail?id=292.

## References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Aspect-oriented software development (AOSD'05)*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, volume 36 of *ACM SIGPLAN Notices*, pages 31–42, New York, NY, USA, 2001. ACM.

[4] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[5] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12, New York, NY, USA, October 2007. ACM.

[6] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[7] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented*

*Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[8] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Nov. 2001.

[9] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):517–575, 2006.

[10] J. Danaher, I. Angelina Lee, and C. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, 2006.

[11] A. B. Dov. infomancers-collections. http://code.google.com/p/infomancers-collections/.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[13] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA'07)*, pages 1–18, New York, NY, USA, 2007. ACM.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, Boston, Mass., third edition, 2005.

[15] J. C. Hardwick and J. Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, August 1996.

[16] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. A case study in transformation modularity. In J. Gray, A. Pierantonio, and A. Vallecillo, editors, *Proceedings of the International Conference on Model Transformation (ICMT 2008)*, volume 5063 of *Lecture Notes in Computer Science*, pages 183–198. Springer, June 2008.

[17] M. Hirzel and R. Grimm. Jeannie: granting Java Native Interface developers their wishes. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA'07)*, pages 19–38. ACM, 2007.

[18] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[19] L. C. L. Kats. java-csharp: C#-inspired language extensions for Java. http://strategoxt.org/Stratego/JavaCSharp/.

[20] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[21] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1999.

[23] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In L. Hendren, editor, *Compiler Construction (CC'08)*, volume 4959 of *Lecture Notes in Computer Science*, pages 192–212. Springer, 2008.

[24] J. Melton and A. Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan Kaufmann, 2000.

[25] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[26] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. *Workshop on Decompilation Techniques, appeared in Proceedings of the Working Conference on Reverse Engineering (WCRE'01)*, pages 368–374, 2001.

[27] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(1):1–15, 1996.

[28] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An Extensible Compiler Framework for Java. *Compiler Construction (CC'03)*, 2622:138–152, Apr. 2003.

[29] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[30] S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly Media, Inc., 2002.

[31] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[32] K. Seymour and J. Dongarra. Automatic translation of Fortran to JVM bytecode. In *Joint ACM Java Grande — ISCOPE 2001 Conference, Stanford University, California, June 2–4, 2001*, New York, NY 10036, USA, 2001. ACM.

[33] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[34] G. Steele. Growing a language. *Higher-Order and Symbolic Compututation*, 12(3):221–236, October 1999.

[35] Sun Microsystems. The annotation processing tool (apt). http://java.sun.com/j2se/1.5.0/docs/guide/apt.

[36] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *First OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99)*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer, Nov. 1999.

[37] The XDoclet team. XDoclet: attribute-oriented programming. http://xdoclet.sourceforge.net/.

[38] A. Tolmach. An external representation for the GHC core language. http://haskell.org/ghc/docs/papers/core.ps.gz, September 2001.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM, 1999.

[40] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[41] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes on Computer Science*, pages 128–142, London, UK, 2002. Springer-Verlag.

[42] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In E. Ernst, editor, *European Conference on Object Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes on Computer Science*, pages 575–599. Springer Verslag, July 2007.

[43] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[44] J. Warmer and A. Kleppe. Building a flexible software factory using partial domain specific models. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM 2006)*, volume TR-37 of *Computer Science and Information System Reports*, pages 15–22, Finland, October 2006. University of Jyväskylä.