Harmless Compiler Plugins

Nathaniel Nystrom
Faculty of Informatics
University of Lugano
nate.nystrom@usi.ch

Abstract

Languages such as Java and Scala allow programmers to write compiler extensions, or plugins, that extend the host programming language with new functionality to enable additional static checking and code transformations.

However, by permitting arbitrary code transformations, compiler plugins can change the host language semantics in unexpected ways. Moreover, plugins do not compose. Plugins can interfere with each other such that one plugin can undo the effects of another, or worse, cause another plugin to generate incorrect code.

In this paper, we develop a theoretical framework for harmless compiler plugins. Host language programs are annotated to limit the scope of plugins. Plugins may change the termination behavior of code outside these scopes, but they are prohibited from changing the values computed by the original computation. The framework is based on an extension of Welterweight Java and uses an information-flow type system to limit plugin effects.

1. Introduction

Today's software environment is becoming increasingly complex. Developers must write code for parallel and distributed systems, systems that are often constructed from components written in multiple languages using complex libraries and frameworks, and that operate on an alphabet soup of data formats. Ensuring reliability while providing maintainability and high performance presents a difficult challenge.

One class of tools being used more and more to address these challenges is compiler extensions, or plugins. Plugins modify or extend an existing compiler with new functionality to enable additional static checking and code transformations, including optimizations. Plugins provide a mechanism to perform static analyses and code transformations on programs written in embedded domain-specific languages.

However, these compiler extensions present a number of challenges. By permitting code transformations, plugins can change the language semantics in unexpected ways. Plugins can interfere with each other, introducing conflicting syntax or semantics. These incompatibilities can lead different developers to develop separate, incompatible language extensions: the developer community can fragment into several groups that each use a different dialect of the host language.

What is needed is a mechanism to ensure that plugins do not interfere with each other or with the host language in surprising ways. In this paper, we will develop a theoretical framework for safe, modular compiler extensions. Inspired by information-flow type systems for enforcing security policies [25, 16, 17, 24] for and by Dantas and Walker's notion of harmless advice [5] for aspect-oriented languages, we define a notion of harmlessness for compiler plugins. The key idea is to allow the user of a compiler plugin to specify what code they trust the plugin to transform, limiting the effects of a plugin to a more manageable scope. A type system ensures that values generated by the plugin do not interfere with the rest of the computation.

The rest of this paper is organized as follows. Section 2 develops the idea of harmless compiler plugins using a small extension of Java, and Section 3 presents some examples to illustrate its usefulness. Section 4 introduces a formal semantics for harmless plugins using an extension of Welterweight Java [21]. We define a soundness property that guarantees that a harmless plugin cannot change the behavior of the original program except where the programmer expressly allows it. Related work is discussed in Section 5, and Section 6 concludes with a discussion of future work.

2. Overview

Often, compiler plugins are used to implement additional semantics checking of programs, for example checking coding conventions, checking that programs follow business guidelines, checking for security errors. These restrictive plugins do not transform code; they simply accept or reject programs based on some correctness criteria. Restrictive plugins are always safe since they do not change the semantics of the underlying programming language. In this paper, however, we are concerned with a more powerful class of plugins: those that perform code transformations. These transformations are used to implement optimizations, to generate glue code, and to implement domain-specific language extensions.

We assume that plugins, like those for Scala [20], X10 [2, 19], and Thorn [1], can perform arbitrary transformations. However, this power enables plugins to change the behavior of the original computation to which the plugin is applied, except possibly by changing the termination or I/O behavior. For example, a noninterfering plugin could generate logging statements. These statements simply read values computed by the mainline computation and perform I/O. To implement logging, the plugin might allocate objects and later modify these objects; however, it cannot modify objects allocated by the original computation.

Requiring complete noninterference ensures safety and is useful, but it is still restrictive. For instance, consider a plugin used in
The implementation of software transactional memory (STM). One way to implement transactions is to record the memory operations performed by a transaction. When the transaction attempts to commit, the set of locations accessed by concurrent transactions are compared and if there is a data race, one or more of the conflicting transactions are aborted. A plugin can aid with the implementation of STM by translating the body of a transaction into code that implements memory access logging and the commit and abort operations. To implement the abort operation, any writes performed by the transaction need to be undone. This action violates the noninterference property of the plugin—the abort generated by the plugin can write to memory locations accessed by the subsequent computation. These writes are benign, however. The plugin performs exactly those writes to memory that are expected of it given that it is designed to implement STM.

To allow these writes—and more generally, to allow more expressive and powerful yet safe, compiler plugins—we introduce a language construct that allows the programmer to declare trust in a plugin, permitting it to perform some operations that can change the behavior of the underlying program. We say a plugin is harmless if it does not change the behavior of the program in an untrustworthy manner. We will formalize harmlessness in Section 4, but first describe the idea informally, illustrating with some examples.

To support harmless plugins, we introduce two language constructs that allows the programmer to specify precisely which program statements a plugin is allowed to transform, and how it can transform. Introducing a language change enables library–plugin co-design: library writers can develop code with the expectation that a given plugin will generate code to be used with the library.

The statement replace \( p \) \( s \) states that the plugin \( p \) can transform the statement \( s \) arbitrarily. No restrictions are placed on the transformations the plugin can perform in a replace statement. A plugin is free to insert statements and also to remove statements from the body of a replace statement. Moreover, plugins are free to introduce new class and method definitions (as long as existing name bindings do not change) and to invoke these new constructs from within the body of a transformed replace statement.

Figure 1. ORM example

| @table(account) @plug(ORM) |
| class account extends Tuple |
| @column(id) |
| long id; |
| @column(name) |
| String name; |
| @column(balance) |
| long balance; |
| static List<Account> load() |
| replace ("ORM") |
| /* this will be replaced */ |
| } |
| } |
| } |
| } |

Even though there is no explicit assignment from \( x \) to \( y \), the value stored in \( y \) depends on \( x \)—there is an implicit flow from \( x \) to \( y \). Since \( y \) is not declared to depend on plugin \( p \), the implicit flow from \( x \) to \( y \) must be prevented by the type system.

Although we formalize harmless plugins using a type system, they need not be implemented as one. Instead, one could have plugin users annotate classes and packages that plugins are allowed to affect. These annotations can be provided when invoking the compiler rather than as source code annotations. The primary goal is to require the programmer to explicitly name the plugins that can affect the behavior of a piece of code; the mechanism for achieving this matters less.

3. Examples

Harmless plugins are formalized in Section 4. But, to illustrate the concepts we first present some examples.

3.1 Atomic blocks

Consider a plugin for supporting software transactional memory. The programmer writes an atomic block as atomic \( s \) to run statement \( s \) within a transaction. The plugin rewrites \( s \) to log memory operations and to validate that the transaction does not conflict with another transaction.

In our language extension, one could write an atomic block replace ("atomic") {...}. Since the plugin rewrites the body of the atomic block, the type of any variable written by the body must be labeled with the atomic plugin. In addition any method called by the transaction must also be labeled with atomic.

3.2 ORM

A compiler plugin can be used to implement an object–relational (OR) mapping. Given an annotated class declaration, the plugin generates code to map between database tuples and instances of the class. For example, the class declaration in Figure 1 maps to a database table account with columns id, name, and balance. The @table annotation specifies that instances of the class map...
to a tuple of the given table. The @column annotations on fields specify which database attribute the field maps to.

The @plug annotation on line 1 specifies that a field of the class might depend on values produced by the ORM plugin—that is, that all field types \( t \) are implicitly labeled with ORM: \( \text{ORM}(t) \).

The ORM plugin also implements methods that load and save instances of Account to the account table. The load method on lines 12–16 contains a replace statement that returns an empty list. The plugin might rewrite this method as follows:

```java
static List<Account> load() {
    return null;
}
```

Since the generated load method updates the fields of the Account objects it instantiates, those fields must be labeled with the ORM plugin. The @plug annotation on the class declaration ensures that this is true.

### 3.3 Logging

In this example, we describe a plugin for injecting logging code into the computation, say to log calls to a particular method \( m \). The plugin generates code to output to a log file, but does not otherwise change the behavior of the mainline computation.

We can model this kind of plugin by wrapping the entire program (or rather, all statements in the program) in a interleave ("log") statement. This allows the plugin to locate calls to \( m \) and interleave logging code, but it prevents the plugin from performing transformations that change the mainline behavior of the code.

### 4. A calculus for harmless plugins

We formalize harmless compiler plugins in a simple calculus. The semantics are based on Weltertwerk Java (WJ) [21]. We chose WJ over Featherweight Java [11] since WJ models the heap and we wish to model the effect of assignments within a replace expression. We elide some of the semantics, especially for "standard" constructs, and refer the reader to Östlund and Wrigstad’s paper [21] for the complete semantics of Weltertwerk Java.

The grammar for the calculus is shown in Figure 2. A program \( P \) consists of a set of class declarations \( \mathcal{C} \) and a set of compiler plugins \( \mathcal{P} \). For a program \( P \), \( CT(P)(c) \) returns the class declaration for class \( c \), and \( PT(P) \) is the set of plugins for \( P \).

Plugins \( p \) are functions from programs to programs. A plugin may both introduce and remove classes, methods, and fields in its output \( p(P) \). To be harmless, the full expressive power of plugins is restricted, as described in Section 4.2.

A class declaration \( C \) has fields and methods. Fields are as in WJ. Methods are extended with a label annotation \( \ell \), which specifies the set of plugins that can invoke the method. The label also bounds the effects of the method. If a method is labeled with plugin \( p \), it is permitted to write only variables that can be influenced by \( p \). This label must be preserved when overriding the method in a subclass.

Statements consist of sequences, field accesses and updates, variable moves, allocation, calls, and type casts. For simplicity, most statements are flattened and do not have nest. The \( \epsilon \) statement is a no-op. The exceptions to this are the if and replace statements. Unlike WJ, we include if statements so that we can model explicit flows. Types consist of a class \( c \) and a label \( \ell \) specifying the plugins on which values of that type may depend.

One notable omission from the calculus is the interleave statement. Instead, interleave (\( \ell \))s can be encoded as

\[
[x]_{\ell} \cdot \text{replace}(\ell) \epsilon
\]

where \([x]_{\ell}\) is defined as follows:

\[
\begin{align*}
\text{if } (\ell) & x \cdot \text{replace}(\ell) \epsilon; \text{interleave}(\ell) \{ x \}\ell \\
\text{replace}(\ell) & x \cdot \text{replace}(\ell) \{ x \}_{\ell} \\
\text{interleave}(\ell) & x \cdot \text{replace}(\ell) \epsilon; \text{interleave}(\ell) \{ x \}_{\ell} \\
\text{replace}(\ell) & x \cdot \text{replace}(\ell) \epsilon; \text{interleave}(\ell) \{ x \}_{\ell} \\
\text{replace}(\ell) & x \cdot \text{replace}(\ell) \epsilon; \text{interleave}(\ell) \{ x \}_{\ell} \\
\end{align*}
\]

The calculus also does not model exceptions. One option is to extend the calculus to handle exceptional flows similarly to the Jif language [16]. A replace statement can be annotated with the set of exceptions it is permitted to throw. A plugin that introduces other exceptions would be rejected as non-harmless. Labels on catch statements can indicate the set of plugins that might cause the caught exception to be thrown. Other possibilities are to turn exceptions introduced by plugins into errors or to run plugin code transactionally—i.e., if a plugin throws an unexpected the replace statement is rolled back and has no effect on the rest of the program.

The formal semantics below use the auxiliary functions in Figure 4 and the grammar elements of Figure 5.

#### 4.1 Dynamic semantics

The dynamic semantics for the calculus are shown in Figure 3. These are nearly identical to the semantics in WJ [21] with the addition of rules for if and replace. Each rule transforms a heap \( H \) and call stack \( S \) into a new heap and stack. Stack frames consist of a map of local variables \( L \) and the next statement to execute. A single step can modify the local variable map and the heap. Note that the cast rule D-Cast does not use the label on the value being cast—this label is never stored. The new rule for if is straightforward. The new D-PLUG rule simply evaluates the body of the replace statement. We elide the error rules from [21].

#### 4.2 Static semantics

A plugin \( p \) is a function from programs to programs. To be harmless with respect to this calculus a plugin \( p \) must satisfy the following:

- If \( P \), then \( p(P) \). That is, if the original program \( P \) is well-formed, the transformed program is also well-formed.
- All statements in \( P \) except replace statements are preserved by \( p \). More precisely, for a statement \( s \) let unplugged(\( s \)
be the statement formed by rewriting any sub-statement “replace \((\ell) \ x'\) to \(\ell\)”. For all statements \(s\) occurring anywhere in \(P\), let \(p(s)\) be the corresponding transformed statement in \(p(P)\). Then we require that a harmless plugin \(p\) ensures unplugged \((s) = \text{unplugged}(p(s))\).

Given the type system we describe next, these conditions ensure that a plugin cannot change the computation of any values that are assigned to a variable that is not labeled with \(p\). This guarantees that if there are no variables that are labeled with \(p\), \(p\) cannot change the behavior of the program (modulo changes in termination and I/O behavior).

The static semantics for the calculi are shown in Figure 6. The semantics for well-formedness of programs, classes, and methods are similar to WJ are are elided. All judgments in the static semantics are implicitly parametrized on the program \(P\). The typing rules differ from WJ by introducing a label into the typing environment. Following other information-flow type systems, typing rules assume a label \(pc\), which can be thought of as the label associated with the program counter. The \(pc\) label represents the set of plugins on which the statement \(s\) is control-dependent. The general form of a typing judgment for a statement is thus \(pc : E = s\).

Since most of the typing rules for statements assign to a variable, we illustrate how the \(pc\) label affects the semantics by discussing one rule, S-ASSIGN. The rule first checks that the variables \(x\) and \(z\) are of the same type (as in [21]). It then checks that the \(pc\) label (a set of plugins) is a subset of the label on the type of \(x\). This ensures that if control reached this statement because of code that might have been transformed by a plugin \(p\), then \(x\)’s type indicates that it can be influenced by \(p\). The other rules handle the \(pc\) label similarly.

The result of the \(\text{mtyp}e(\cdot)\) function in Figure 4 includes a label on the function as well as on the argument and return types. The S-CALL rule additionally checks that the \(pc\) label is a subset of the label on the method being called.

The S-CAST rule preserves the label on the type. There is no way in the calculus to drop a label from a type. We plan to investigate such \(downgrading\) policies [4] in the future.

Both the S-PLUG and S-IF rules change the \(pc\) label for their sub-statements. S-PLUG adds the set of plugins in the replace statement to the \(pc\) label. S-IF adds the set of plugins on which the branch condition depends to the \(pc\) label for the arms of the if statement. This ensures that if the branch condition depends on a given plugin, that dependency will be propagated to assignments dependent on the branch.

Subtyping is the reflexive, transitive closure of the relation defined by the SUB-DIR rule. A subtype must be a subclass of its supertype. In addition, a subtype may be labeled with fewer plugin than the supertype. Types are well-formed if the class is in the class table \(CT\) and if all plugins in type’s label are in the plugin table \(PT\).

### 4.3 Noninterference

To prove that the type system and the additional conditions on harmless plugins in Section 4.2 ensure noninterference, we need to prove not only that the type system above is sound, but also that for any program \(P\) and for any statement, if there is a variable \(x\) that does not trust plugin \(p\)—that is, the label on its type does not include \(p\)—then both \(P\) and \(p(P)\) compute the same value for \(x\). More formally, we must prove the following property holds:

**Noninterference**: Let \(P\) be an program such that \(\triangleright P\), and let \(p\) be a harmless plugin. Let \(s\) be a statement such that \(pc : E[x : c] = s\) where \(p \notin \ell\) (i.e., \(s\) is a statement with a free variable \(x\) that cannot be influenced by \(p\), and that type-checks in the context of program \(P\)). Then, for all well-formed heaps \(H\) and all well-formed stacks \(S\), if

\[
H \mid S[L[x \mapsto \mathbf{null}], s] \xrightarrow{m} H_1 \mid S[L_1, s_1] \xrightarrow{m}
\]

in \(P\), then

\[
H \mid S[L[x \mapsto \mathbf{null}], s] \xrightarrow{m} H_2 \mid S[L_2, s_2] \xrightarrow{m}
\]

in \(p(P)\) and \(L_1(x) = L_2(x)\).

### 5. Related work

Our notion of harmlessness was inspired by Dantas and Walker’s harmless advice [5], an approach to ensuring safety of aspect-oriented [14, 13] programs. The primary difference between our work and theirs is that their work provides a simpler protection model: code (advice) injected into the program cannot write to any locations that affect the mainline computation; harmless advice can only write to variables created by the aspect. Moreover, aspects, unlike compiler plugins, can neither transform nor delete existing
The extended compiler is a standard port for new syntax and semantics. These frameworks differ from each other on how the language can be extended.

Frameworks such as Polyglot [18] and JastAdd [7] allow a compiler to be extended with support for new syntax and semantics. These frameworks differ from compiler plugins architecturally. The extended compiler is a standalone compiler and can often be used as a drop-in replacement for the original base compiler. Compiler frameworks are often used for making pervasive changes to the language, such as modifications of the type system, for research or pedagogical purposes. Plugins, by contrast, have (so far) been used primarily for small extensions to the base language.

Java [10] provides syntax for annotating declarations with metadata and a compiler plugin mechanism for performing additional static checking of code. The Checker framework [22] supports flow-sensitive types qualifiers in Java and has been used to implement, for example, nonnull types and immutable types. Annotations based on Checker’s type qualifiers are planned for inclusion in Java version 7 through JSR 308 [12]. These checkers, as well as many other systems, such as JavaCOP [15] and CQual [8] and semantic type qualifiers [3], perform no code transformations but rather simply check annotated programs against stronger correctness criteria than Java requires. C# [6] provides an annotation mechanism, called attributes, similar to Java 7’s.

6. Conclusions

 Harmless plugins are a class of compiler plugins that can perform safe code transformations. By declaring what statements the plugin is allowed to transform and what variables the plugin is allowed to access only to package, class, and class member declarations, not to a representation of the executable code.

Often, a more powerful approach than compiler plugins is to use an extensible compiler framework, which places no restrictions on how the language can be extended. Frameworks such as Polyglot [18] and JastAdd [7] allow a compiler to be extended with support for new syntax and semantics. These frameworks differ from compiler plugins architecturally. The extended compiler is a standalone compiler and can often be used as a drop-in replacement for

---

1 This can be worked around by using other means to access the source or bytecode for a given class.
References


