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 underlying programming language or to interfere with one another. To prevent this, we adapt the notion of *noninterference* from information-flow type systems. A completely noninterfering plugin can transform code and can generate code; however, it cannot 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

FTfJP'11, July 26, 2011, Lancaster, UK.

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.

The second statement, **`interleave`** (p) s, allows the plugin p to interleave new statements between individual statements of s, but not to transform s arbitrarily. We show in Section 4 how **`interleave`** statements can be rewritten into **`replace`** statements.

The plugin is prohibited from performing transformations on statements outside the body of **`replace`** or **`interleave`** statements. However, because arbitrary transformations can occur, the *effects* of a plugin may escape the body of these statements. For example, a plugin can add (or remove) an assignment to a field declared in the original program. Thus, harmless plugins also provide a mechanism to permit the programmer to specify memory locations that the plugin is allowed to affect. Plugins can insert and remove writes to these locations within a **`replace`** statement.

To specify these locations, we use an information-flow type and effect system [25, 16]. Variables are labeled with the sets of plugins that can write to those variables. We write c^ℓ for the type with class c and label ℓ; ℓ is simply a set of plugins p. The type system ensures that any value that depends on code produced by a plugin p cannot be stored in a variable unless that variable is labeled with p.

Thus for a plugin p, if a variable x is not labeled with p, that variable cannot be influenced by p: it cannot be assigned inside a **`replace`** (p) statement; and moreover, any value produced within a **`replace`** (p) statement or any value that transitively depends on that value, cannot be assigned into x, even if the assignment is done outside the **`replace`** body.

The type system must handle not only explicit flows (such as through an assignment), but also *implicit flows*. For example, consider the statements:

$$\mathbf{boolean}^p\ \mathtt{x};$$
$$\mathbf{boolean}\ \mathtt{y};$$
$$\mathbf{replace}\ (p)\ \mathtt{x} = \mathtt{true};$$
$$\mathbf{if}\ (\mathtt{x})\ \mathtt{y} = \mathtt{true};\ \mathbf{else}\ \mathtt{y} = \mathtt{false};$$

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.

```
@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 */
    }
  }
  ...
}
```

Figure 1. ORM example

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`: $t^{\texttt{ORM}}$.

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:

```
static List<Account> load() {
  List<Account> l = new ArrayList<Account>();
  ResultSet rs = execute("SELECT * FROM account");
  while (rs.next()) {
    Account a = new Account(); l.add(a);
    a.id = rs.getLong("id");
    a.name = rs.getString("name");
    a.balance = rs.getLong("balance");
  }
  return l;
}
```

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 Welterweight 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 Welterweight Java.

The grammar for the calculus is shown in Figure 2. A program P consists of a set of class declarations $\overline{C}$ and a set of compiler plugins $\overline{p}$. 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 ℓ, 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 ε 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 implicit flows. Types consist of a class c and a label ℓ specifying the plugins on which values of that type may depend.

$$
\begin{array}{rcll}
P & ::= & (\overline{C}, \overline{p}) & \textit{program} \\
C & ::= & \textbf{class}\ c\ \textbf{extends}\ d\ \{\ \overline{F}\ \overline{M}\ \} & \textit{class} \\
F & ::= & t\ f & \textit{field} \\
M & ::= & t\ m_\ell(\bar{t}\ \bar{x})\ \{\ \overline{t'}\ \overline{x'}\ s\ \} & \textit{method} \\
s & ::= & \varepsilon \mid s_1\,;\,s_2 \mid x = y.f \mid x = z & \textit{statement} \\
 & \mid & y.f = z \mid x = \textbf{new}\ t\,() & \\
 & \mid & x = y.m(\bar{z}) \mid x = (c)\ z & \\
 & \mid & \textbf{if}\ (x == \texttt{null})\ s_1\ \textbf{else}\ s_2 & \\
 & \mid & \textbf{replace}\ (\ell)\ s & \\
t & ::= & c^\ell & \textit{type} \\
\ell & ::= & \overline{p} & \textit{label}
\end{array}
$$

Figure 2. Grammar

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

$$[\![s]\!]_\ell;\ \textbf{replace}\ (\ell)\ \varepsilon$$

where $[\![s]\!]_\ell$ is defined as follows:

$$
\begin{aligned}
[\![\textbf{interleave}\ (\ell')\ s]\!]_\ell &= \textbf{replace}\ (\ell)\ \varepsilon;\ \textbf{interleave}\ (\ell')\ [\![s]\!]_\ell \\
[\![\textbf{replace}\ (\ell')\ s]\!]_\ell &= \textbf{replace}\ (\ell)\ \varepsilon;\ \textbf{replace}\ (\ell')\ [\![s]\!]_\ell \\
[\![s_1;\ s_2]\!]_\ell &= [\![s_1]\!]_\ell;\ [\![s_2]\!]_\ell \\
[\![\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2]\!]_\ell &= \textbf{replace}\ (\ell)\ \varepsilon;\ \textbf{if}\ (e)\ [\![s_1]\!]_\ell\ \textbf{else}\ [\![s_2]\!]_\ell \\
[\![s]\!]_\ell &= \textbf{replace}\ (\ell)\ \varepsilon;\ s \qquad \text{otherwise}
\end{aligned}
$$

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—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 $\vdash P$, then $\vdash p(P)$. 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 $\mathsf{unplugged}(s)$

$$\frac{L(z) = v}{H \mid S\langle L, x = z;\ s\rangle^m \longrightarrow H \mid S\langle L[x \mapsto v], s\rangle^m} \quad (\textsc{D-Assign})$$

$$\frac{L(y) = \imath \quad H(\imath.f) = v}{H \mid S\langle L, x = y.f;\ s\rangle^m \longrightarrow H \mid S\langle L[x \mapsto v], s\rangle^m} \quad (\textsc{D-Select})$$

$$\frac{L(y) = \imath \quad L(z) = v}{H \mid S\langle L, y.f = z;\ s\rangle^m \longrightarrow H(i.f := v) \mid S\langle L, s\rangle^m} \quad (\textsc{D-Update})$$

$$\frac{\begin{array}{c} F = [f \mapsto \texttt{null} \mid f \in \mathsf{dom}(\mathsf{fields}(t))] \\ H' = H[\imath \mapsto (c, F)] \quad \imath \text{ is fresh} \end{array}}{H \mid S\langle L, x = \textbf{new}\ c^{\ell}();\ s\rangle^m \longrightarrow H' \mid S\langle L[x \mapsto \imath], s\rangle^m} \quad (\textsc{D-New})$$

$$\frac{\begin{array}{c} L(y) = \imath \quad H(\imath) = (c, \dots) \\ \mathsf{mbody}(c.m) = (\overline{x'}, \overline{x''}, s') \quad L(\bar{z}) \\ L' = [\texttt{this} \mapsto \imath][\overline{x'} \mapsto \bar{v}][\overline{x''} \mapsto \texttt{null}] \end{array}}{H \mid S\langle L, x = y.m(\bar{z});\ s\rangle^m \longrightarrow H \mid S\langle L, x = y.m(\bar{z});\ s\rangle^m, \langle L', s'\rangle^{c.m}} \quad (\textsc{D-Call})$$

$$\frac{(L(z) = \imath \quad H(\imath) = (d, \dots) \quad \vdash d <: c) \quad \vee \quad L(z) = \texttt{null}}{H \mid S\langle L, x = (c)\ z;\ s\rangle^m \longrightarrow H \mid S\langle L[x \mapsto L(z)], s\rangle^m} \quad (\textsc{D-Cast})$$

$$\frac{i = 1 \text{ if } L(x) = \texttt{null}, \text{else } 2}{H \mid S\langle L, \textbf{if}\ (x == \texttt{null})\ s_1\ \textbf{else}\ s_2;\ s\rangle^m \longrightarrow H \mid S\langle L, s_i;\ s\rangle^m} \quad (\textsc{D-If})$$

$$H \mid S\langle L, (\textbf{replace}\ (\ell)\ s);\ s'\rangle^m \longrightarrow H \mid S\langle L, s;\ s'\rangle^m \quad (\textsc{D-Plug})$$

Figure 3. Operational semantics

$\mathsf{fields}(c) = \mathit{fst}(CT(P)(c))$	Lookup all fields for class c
$\mathsf{ftype}(c.f) = \mathsf{fields}(c)(f)$	Lookup type of field f in class c
$\mathsf{methods}(c) = \mathit{snd}(CT(P)(c))$	Lookup all methods for class c
$\mathsf{mtype}(c.m) = \bar{t} \xrightarrow{\ell} t$	Lookup signature of method m in class c
$\mathsf{mbody}(c.m) = (\bar{x}, \overline{x'}\texttt{ret}, s)$	Lookup variables and body of method m in class c
$\mathsf{label}(c^{\ell}) = \ell$	Get the label of a type

Figure 4. Lookup functions

be the statement formed by rewriting any sub-statement "**replace** (ℓ) s'" to ε. 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 ensure $\mathsf{unplugged}(s) = \mathsf{unplugged}(p(s))$.

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 no variables 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 calculus 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 $pc;\ E \vdash s$.

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 (just 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 $\mathsf{mtype}(\cdot)$ 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 $\vdash P$, and let p be a harmless plugin. Let s be a statement such that $P;\ pc;\ E[x : c^{\ell}] \vdash s$ 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*

$$H \mid S\langle L[x \mapsto \texttt{null}], s\rangle^m \to^* H_1 \mid S\langle L_1, s_1\rangle^m$$

in P, then

$$H \mid S\langle L[x \mapsto \texttt{null}], s\rangle^m \to^* H_2 \mid S\langle L_2, s_2\rangle^m$$

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

$$\frac{E \vdash x : t \quad E \vdash z : t \quad pc \subseteq \mathsf{label}(t)}{pc;\ E \vdash x = z} \tag{S-Assign}$$

$$\frac{E \vdash x : t \quad pc \subseteq \mathsf{label}(t)}{pc;\ E \vdash x = \mathbf{new}\ t()} \tag{S-New}$$

$$\frac{E \vdash y : t \quad E \vdash x : t' \quad t' = \mathsf{ftype}(t.f) \quad pc \subseteq \mathsf{label}(t')}{pc;\ E \vdash x = y.f} \tag{S-Select}$$

$$\frac{E \vdash y : t \quad E \vdash z : t' \quad t' = \mathsf{ftype}(t.f) \quad pc \subseteq \mathsf{label}(t')}{pc;\ E \vdash y.f = z} \tag{S-Update}$$

$$\frac{\begin{array}{c} E \vdash y : t \quad \mathsf{mtype}(t.m) = \bar{t} \xrightarrow{\ell} t' \\ E \vdash \bar{z} : \bar{t} \quad E \vdash x : t' \quad pc \subseteq \ell \quad pc \subseteq \mathsf{label}(t') \end{array}}{pc;\ E \vdash x = y.m(\bar{z})} \tag{S-Call}$$

$$\frac{E \vdash z : d^{\ell} \quad E \vdash x : c^{\ell} \quad pc \subseteq \ell}{pc;\ E \vdash x = (c)\ z} \tag{S-Cast}$$

$$\frac{pc \cup \ell;\ E \vdash s}{pc;\ E \vdash \mathtt{replace}\ (\ell)\ s} \tag{S-Plug}$$

$$\frac{E \vdash x : t \quad pc \cup \mathsf{label}(t);\ E \vdash s_1 \quad pc \cup \mathsf{label}(t);\ E \vdash s_2}{pc;\ E \vdash \mathtt{if}\ (x == \mathtt{null})\ s_1\ \mathtt{else}\ s_2} \tag{S-If}$$

$$\frac{P(c) = \mathtt{class}\ c\ \mathtt{extends}\ d\ \{\ \overline{F}\ \overline{M}\ \} \quad \ell \subseteq \ell'}{\vdash c^{\ell} <: d^{\ell'}} \tag{Sub-Dir}$$

$$\frac{c \in \mathrm{dom}(CT(P)) \quad \ell \in PT(P)}{\vdash c^{\ell}} \tag{Type}$$

Figure 6. Static semantics

$S ::= \varepsilon \mid S\langle L, s\rangle^{m}$	Stack
$L ::= [] \mid L[y \mapsto v]$	Stack frame
$v ::= \iota \mid \mathtt{null}$	Value
$H ::= [] \mid H[\iota \mapsto (c, F)]$	Heap
$F ::= [] \mid F[f \mapsto v]$	Fields
$E ::= [] \mid E[y : t] \mid E[f : t]$	Local type environment

Figure 5. Syntax of type environments, stacks, heaps

code. Other approaches to ensuring aspects are safe and composable include CompAr [23], which uses a constraint system to ensure advice can be composed, and StrongAspectJ [9], which enforces safety through a type system.

Our calculus is similar to security-typed languages [25, 16, 24] and the type system is similar to type systems that enforce integrity policies. Work in this area suggests some extensions to the type system proposed here, including supporting label polymorphism [16] and downgrading policies [4] that would relax strict noninterference.

Several languages support compiler plugins. Scala [20], X10 [2, 19], and Thorn [1] all permit plugins to perform arbitrary transformations. Writing plugins requires intimate knowledge of the compiler implementation and few, if any, safety guarantees are provided.

Java, also supports compiler plugins. Java version 6 supports a form of compiler plugin called an *annotation processor*, however these are not capable of transforming code. Indeed, they have access only to package, class, and class member declarations, not to a representation of the executable code.[1]

Often, a more powerful approach than compiler plugins it 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 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 modify, a programmer can limit the effects of a compiler plugin to an expected set of classes and methods.

Introducing a language construct enables a library–plugin codesign methodology. Library writers develop their code with the expectation that a given plugin will generate code or transform code for that library.

Soundness and noninterference properties of the calculus need to be proved. Incorporating features from security-typed languages such as downgrading [4] and label polymorphism [16] can help improve the expressiveness of the language.

We also look forward to implementing this framework in the Scala compiler and to evaluating its usefulness by developing several compiler plugins, exploring different ways to express how the scope of a plugin should be limited.

---

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

References

[1] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn–robust, concurrent, extensible scripting on the JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, October 2009.

[2] Philippe Charles, Christian Grothoff, Christopher Donawa, Kemal Ebcioğlu, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, October 2005.

[3] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–95, 2005.

[4] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, pages 198–209, 2004.

[5] Daniel S. Dantas and David Walker. Harmless advice. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 383–396, January 2006.

[6] ECMA. Standard ECMA-334: C♯ language specification (4th edition). `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, June 2006.

[7] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 1–18, New York, NY, USA, 2007. ACM.

[8] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 29th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–12. ACM Press, June 2002.

[9] Bruno De Fraine, Mario Südholt, and Vivane Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *Proceedings of the Seventh International Conference on Aspect-Oriented Software Development*, pages 60–71, March 2008.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[12] JSR 308: Annotations on Java types. `http://jcp.org/en/jsr/detail?id=308`.

[13] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[15] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems*, 32(2):1–37, January 2010.

[16] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[17] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release, at http://www.cs.cornell.edu/jif, July 2001–.

[18] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *12th International Conference on Compiler Construction (CC 2003)*, number 2622 in Lecture Notes in Computer Science, pages 128–152, Warsaw, Poland, April 2003. Springer-Verlag.

[19] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.

[20] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala (2nd ed.)*. Artima, 2010.

[21] Johan Östlund and Tobias Wrigstad. Welterweight Java. In *Proceedings of the 48th international conference on objects, models, components, patterns (TOOLS'10)*, number 6141 in Lecture Notes in Computer Science, pages 97–116, 2010.

[22] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 201–212, July 2008.

[23] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. CompAr: Ensuring safe around advice composition. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, number 3535 in Lecture Notes in Computer Science, pages 163–178, 2005.

[24] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[25] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.