Type Inference in Mixed Compiled / Scripting Environments

Andrew Weinrich

2007-12-21

Abstract

Programs written in statically-typed languages are commonly extended with scripting engines that manipulate objects in the compiled layer. These scripting environments extend the capabilities of the program, at the cost of additional errors that would be caught by compile-time type checking. This paper describes a system for using type information from the compiled, statically-typed layer to perform type inference and checking on scripting code. A proof-of-concept of this system is shown in `flint`, a type-checking tool for the language F-Script.

1 Introduction

A recent trend in software development has been to extend programs written in a compiled language with a scripting engine that can manipulate objects from the compiled layer. The presence of JavaScript in web browsers has created an entire industry of web applications. At the OS level, many GUI environments have object-oriented scripting languages, such as AppleScript on Mac OS [2], that can tie together functionality of various programs. Scripting languages are also used in the development of commercial applications, such as Adobe's recent product Lightroom [5].

The benefits of such integrated compiled / scripting environments are obvious. The ability for end-users to customize program behavior increases the lifespan of a product and fosters the creation of an ecosystem of third-party extensions. For internal development, moving some code into a scripting layer helps speed the development process by reducing compilation costs and turnaround time for bug fixes.

However, the added flexibility of a scripting level brings with it several new costs. In particular, most scripting languages lack any sort of compile-time type checking. This leads to simple but aggravating errors that would be easily caught by a type-checking compiler, such as using an invalid method name on an object. In programs where most of the scripting code is used in the user interface, such as web applications or the aforementioned Lightroom, it is difficult to write unit tests for such code, and the presence of type-related bugs increases the time required for testing and QA.

The lack of static typing in the scripting layer is particularly unfortunate, because most of the objects used in such scripts are actually implemented in code written in the compiled layer, but all type information is thrown away when the bridge to the scripting level is crossed. A tool that could carry this type information into the flexible scripting code and detect type errors would be of great use to programmers, not as a verifier of program correctness, but as a detector of common programming mistakes.

This paper describes a generic system for performing type checking of scripting-language code that extends a base of code written in a different, compiled, statically-typed language. It demonstrates a proof-of-concept in `flint`, a type-checking and static analysis tool for the language F-Script.

2 Type Checking For Mixed Compiled / Scripting Code

One possible solution to the lack of compile-time type checking in scripting languages is to add explicit type declarations [4]. However, proposed typing systems based on explicit declarations have encountered significant resistance from developer communities, and making changes to the core language parser is generally not feasible for a standalone tool.

The obvious alternative is to perform type inference. Type inference can be very effective, as demonstrated by the language ML [6], and the algorithms for it are well-known. A type inference system for the mixed environments considered in this paper has the benefit of starting with a large base of code that already has static type annotations; combined with knowledge of common programming patterns, the type inference is much more tractable than the general case.

This section describes a generic system for performing type checking of a scripting language that extends compiled code. The terminology is drawn from Objective-C [1], but is applicable to many combinations of statically-typed and non-statically-typed languages. The procedure is concerned with finding two kinds of type errors:

1. *Is an invalid method being called on an object?* For every class, we will have complete information about what method names the class supports. If the type of a value can be determined to be a set of known classes, all invalid method errors can be detected.

2. *Are the types of parameters to methods correct?* This is a more difficult problem. For classes written in the compiled layer, the exact types of parameters will be known. For classes written in the scripting layer, only a restricted view of the parameter type called a Protocol can be determined. However, since the majority of objects in integrated compiled / scripting programs come from the compiled layer, this will not be a great problem.

2.1 Drawing Type Information from Compiled Code

Many static type inference systems for languages with dynamic checking must operate on an entire program written in that language. For the programming environments treated in this paper, the dynamically checked portion of code is typically in operational units that are far smaller than the base of compiled code. For example, the amount of JavaScript code in the world that runs on web browsers dwarfs the size of the actual browser codebases, but any particular JavaScript program is much, much smaller than the browser that supports it.

This disparity allows us to start not from scratch, but with a rich library of type information from the compiled layer. This information, which must be used by the compiler, is always written in header files or some other machine-readable format. The first step in our analysis will be to read these headers and compile a library of all the classes and methods in the compiled layer. The use of this library is described in Section 5, "`flint` Architecture".

2.2 Type Categories

The atomic types that make up a profile are called *categories* in this paper's type system. The term "category" is used because the actual number of types is unknown, due to the possible creation of new classes in the script program. Table 1 lists the compatibility rules for type categories. Required types (along the top) are known from header file information or from inference based on parameter use in methods written in the scripting layer.

- **Id** - An Id is a generic object reference that carries absolutely no information about the referent's type. It can be considered the "type" of all objects in a language that does not have static typing. Id is not an analog of the Java class Object, which does specify several methods that all objects have. If a type lattice were to be made from this system, Id would be the top element.

- **Class** - A class type represents an instance of a particular class. This may be a class that exists in the compiled layer, in which case the method information is read from header files, or it may be created programmatically in the scripting layer.

- **Metaclass** - In many scripting or higher-level languages, classes themselves are treated as first-order objects. This "class object" is called a metaclass to avoid confusion. Each metaclass category is associated with exactly one class category, and vice versa. Metaclass types function just like class types, except that they do not inherit methods from parent classes.

- **Protocol** - When trying to infer the types of arguments to methods or functions written in a language without explicit type declarations, we will only see a limited view of their capabilities. In an object-oriented language, this means that we will only be able to positively state that a parameter to a method must implement some set of methods, not that it must be an instance of an explicit class. The name for this set of methods that describes the parameter type is a Protocol. Protocols are similar to interfaces in Java, except that they do not have specific names. Any class that implements all the methods of a Protocol will be acceptable where that protocol is needed.

- **Nil** - While performing type inference and checking, it is convenient to track nil pointers at the same time. Some methods have parameters that can optionally be nil, or that may return nil under certain conditions, such as looking up a missing key in a dictionary. These methods can be manually tagged as returning a profile that also contains type Nil, which will allow the flow of null pointers to be tracked through the program.

- **Void** - In many scripting languages, every method has a return type; if no return value is reasonable (such as for property mutators), nil is typically returned. This creates a problem when using objects originally written in a C-derived language, such as Java or Objective-C. The `void` pseudo-type used for non-returning functions cannot be directly mapped to a scripting-level value. Void is a special type that solves this mapping problem, and is used as a marker to indicate that the compiled-level method did not have a return value. It is analogous to the value `unit` in ML [6].

- **Error** - When the core type category implementations detect a type mismatch, they return a profile containing an instance of the Error category. This

	Id	Class	Metaclass	Protocol
Nil	Yes, with warning	Yes, with warning	No	No
Void	No	No	No	No
Error	Yes	Yes	Yes	Yes
Id	Yes	Yes, with warning	Yes, with warning	Yes, with warning
Class	Yes	If provided class is equal to or child of required class	No	If all methods in required protocol are in provided class or parent of provided class
Metaclass	Yes	No	If provided metaclass is identical to required metaclass	If all methods of required protocol are in provided metaclass
Protocol	Yes	No	No	If all methods of required protocol are in provided protocol

Table 1: Compatibility chart for type categories. Required types are along the top, provided types are along the left. Nil, void, and error are never required types, so they do not have their own columns.

error object has a brief description of its cause, such as a type mismatch or invalid method name. When a syntax node computes its type and finds that it is an error, it adds a message to the global log and returns a profile containing an Error type. Errors propagate silently through a program after their initial creation, to avoid deluging the user in duplicate reports. In practical terms, this means that an error type responds to all methods, accepts any types of parameters, and always returns Error from every method. In this way, it has behavior similar to Id.

2.3 Type Profiles

It is possible that a variable may take on values of different types during a program, as in Listing 1, where `x` has been assigned two different kinds of views. The set of all types that have been assigned to a variable is called a *type profile*. Every value, variable, or expression in a scripting-level analysis has a proper type profile; type categories only occur as members (possibly sole members) of a set.

2.4 Limits of the Type System

The type system used in this paper is limited to languages in which the capabilities of objects are completely determined by the class, and individual instances do not have any ability to add or remove methods and properties. Scripting languages whose object model is based on the system of prototypes originally implemented in Self [11], such as JavaScript or Ruby, will not be able to express that flexibility in the above system. This paper's approach can still be useful in those languages; modifications to individual objects are usually rare, and if it is assumed that they will not occur, the type systems of those languages map easily to the above system.

2.5 Types for Object Properties

`flint`'s type system does not consider the properties of objects; they are all typed as Id. Because all property access is through methods, the use of nonexistent properties can be caught. However, no attempt is made to determine the types of these properties, or whether get / set method pairs are used consistently. The analysis of object property types would be a useful addition to a future version of `flint`.

2.6 Reporting Errors

A `flint` program representation maintains a single, global list of encountered warnings and errors. When a syntax node detects that an Error object is created, it adds a message to this list; type inference rules and action rules can also raise errors and warnings at arbitrary parts of the program. When an error or warning message is added to the list, the current syntax node must also be supplied, so that the file location can be extracted. This makes the type system simpler by removing the need for type category objects to carry around references to the nodes that originally generated them, and for potentially complicated rules on how file locations should be combined when two or more type profiles are unioned together.

At the end of this analysis, the `flint` program reports all errors and warnings to the user. Some warnings, such as

the potentially dangerous use of Id types, can be turned on or off by setting configuration flags in the `flint` program. Currently, this must be done by modifying the `flint.fs` script, as there is no command-line argument parsing.

2.7 Expanding the Type System

The type categories described above are implemented in `flint` as Objective-C classes in the analyzer core. It may occasionally be desirable for inference rules to create new type categories. Categories can be created in F-Script by subclassing the root FSTypeCategory class and overriding the following methods:

1. `respondsToMethod:(String)name` - returns a boolean indicating whether the type responds to a method with the given name.

2. `acceptsMethod:(String)methodName withTypes:(Array)types` - It is possible for an object to respond to a message, but not be able to perform the action because the provided argument types are invalid. This method is an extended version of `respondsToMethod:(String)name` that checks argument types as well as the method name.

3. `returnTypeForMethod:(String)name` - returns a type profile representing the default return type for a method name.

4. `signatureForMethod:(String)name` - returns a complete *method signature*, including return type and argument types.

5. `isCompatibleForType:(FSTypeProfile)type` - returns a boolean indicating whether this type category can be used where `type` is expected.

The root FSTypeCategory class also has predicate methods such as `isVoid`, `isId`, `isClass:(String)className`, etc, which allow inference rules to safely test the type of an expression. When creating a new type category, FSTypeCategory can be extended with an additional method like `isArray` that returns false; the new type category class can override the same method with an implementation that returns true. This sort of extension to a compiled class is due to the flexibility of the Objective-C runtime system, and would be very difficult to simulate in languages like Java or C++. Ruby's "mix-ins" architecture implements similar functionality [**?**].

```
view := nil.
(showBrowser) ifTrue:[
    view := NSBrowser alloc init.
]
ifFalse:[
    view := NSTableView alloc init.
].

view setNeedsDisplay:true.
```

Figure 1: In this example, the variable `view` will be judged to have a type profile that contains two Class types, NSBrowser and NSTableView.

3 Leveraging Coding Conventions

Many proposed type checking systems for scripting or other non-statically-typed languages, such as Smalltalk [3], JavaScript [9], and Python [**?**], attempt to make guarantees of type safety for some portion of analyzed programs. These projects are driven as much by a desire for as performance as for correctness; type guarantees are necessary if the compiler is to inline method calls or make other optimizations.

The scripting language code we consider in this paper is not a target for optimization, and our type checking is a tool for programmers that alerts them to possible problems. Our analysis can forgo guarantees of soundness, and instead concentrate on finding problems that are most likely to occur in actual code.

This is especially valuable for programs written in a language like F-Script, which has very few syntactic constraints on the organization of code. Classes in F-Script are created programmatically, so theoretically it is not decidable how many classes will be in a given program or what their capabilities are. This makes it extremely difficult to prove type correctness; the systems cited above go to great lengths to be certain they have not missed any changes to the type environment.

However, in practice, most classes in scripting languages are written in a fairly standard way. For example, in Perl [12], each class is defined in one file that has the same name as the class. F-Script has a similar convention. If we assume that the programmer will be following this convention, our type inference algorithm can be simplified significantly.

In short, we take for granted that the programmer is being "reasonable" and is writing "reasonable" code that hews to common coding standards, even if the language does not strictly enforce them. These assumptions will cause `flint` to miss some errors in very sophisticated or flexible code, or to report errors where none actually exist. These cases we leave to the judgement of the programmer, just

as the C compiler assumes that, when casting to `void*`, the programmer knows what is going on.

4 Rule-Based Program Analysis

`flint`'s analysis is based on "rules": heuristics that perform type inference and model program state. All rules in `flint` are based on standard programming practices, and work by recognizing syntactic and type patterns. Rule-based analysis and inference allows `flint` to concentrate on finding the most common errors, without using an overly complicated algorithm that is completely sound.

`flint` uses rules for two purposes: custom type inference, and changes to the model of the program being analyzed. Both are triggered by recognition of syntactic patterns, based on coding conventions. Both kinds of rules are applied directly to syntax nodes, with the current program state available as a global variable. Model rules are applied once to every syntax node in the program; type rules will be executed only on demand, when a syntax node needs to compute its type.

A `flint` rule is composed of two pieces: a *matching condition* and an *action*. The matching condition is an array of anonymous functions, each of which takes the current syntax node as an argument. Each function must return true for the rule to match; the evaluator will short-circuit on the first function that returns false. With this behavior, the matching condition could be interpreted as a series of predicates in conjunctive normal form.

The CNF model has two benefits. First, short-circuiting allows unnecessary parts of the condition to be skipped, speeding evaluation time. This is especially important for model rules, all of which will be applied to every node in the program. For improved efficiency, `flint` attaches both kinds of rules to specific kinds of nodes; a rule that is only relevant to message nodes will never be attached to identifiers.

The second benefit is that short-circuiting acts as a guard against conditions that may raise exceptions. Consider the following rule that evaluates node n:

> (n is a message) AND (the receiver of n is a String) AND (the message name is `toUpperCase`)

If n is not a message node, the second condition will throw an exception, because other kinds of nodes do not have a "receiver" property. In a language like F-Script that does not have static typing, there is no way to protect against these errors besides using guard conditions.

4.1 Type Inference Rules

Every syntax node has a default behavior for computing its type. Literals return the literal object type, identifiers look up the currently computed type for the variable, Blocks return the type of their last expression, and so on. These behaviors are overridden when a type inference rules matches that node. `flint` will match type inference rules against nodes only when it is strictly necessary; that is, only when some other piece of the program, such a model rule, has requested the type of the node.

The matching condition for type inference rules can be dependent on other type information, such as the types of a node's children. Since the program representation is a tree, and since each node will only determine its type once, the type inference process is guaranteed to terminate, and the final type model for the program will always be decidable.

4.2 Program Model Rules

Program model rules build up `flint`'s model of the program state, including the symbol table, types of variables, names and capabilities of classes, and lists of errors and warnings. Unlike type rules, every model rule is matched against every relevant node; i.e., if `flint` has 6 rules that apply to method calls, every one of those rules' match conditions will be applied to every single method call in the entire program. Also unlike type rules, more than one model rule can match a node, and a single node can thus make multiple changes to the program state.

There are four types of syntax nodes that `flint` considers (other node types, such as Program and File, are used only for organization):

1. Identifier - Identifiers are the names of either variables, which are lexically scoped to either a closure or the entire program; or built-ins, such as `sys`, `true`, `false`, and the names of classes.

2. Assignment - Assignment to a variable, in the format `var := expression`. One rule for assignments computes the type of the rvalue and sets it as the type for the identifier on the left-hand side. Another rule checks to see if the assignment is in a more deeply nested scope than that at which the variable was declared; this indicates that the assignment may be inside a conditional or loop, and is not guaranteed to be executed. In this case, the rvalue type is added to the type profile for the identifier, rather than replacing it. The latter rule will raise a warning that the `flint.fs` script will report to the user.

3. Literal - Literal objects in F-Script include arrays,

strings, booleans, numbers, and selectors (analogous to atoms or symbols in other languages).

4. Message - A "message", in Smalltalk terminology, is the invocation of a method on an object. A message node has a receiver (which may be an identifier or another expression), a name (also called the "selector" of the message), and a variable number of argument expressions.

5 `flint` Architecture

The `flint` tool is composed of two pieces: a compiled module that hooks into the F-Script interpreter to build a syntax tree, and a program written in F-Script, `flint.fs`, that loads custom type and action rules and drives the analyzer.

5.1 Syntax Analyzer

`flint` uses the F-Script interpreter framework to parse script files, but it does not directly use the generated syntax tree. The F-Script internal representation of a program is designed for immediate evaluation, not static analysis, and has a number of nodes that are either not supported by `flint` (such as advanced messaging patterns) or that can be trivially reduced to other kinds of nodes. The F-Script interpreter also does not keep accurate enough file locations to be useful for error reporting. For these reasons, the `flint` core builds a separate, but similar, parse tree based on that used by the F-Script interpreter, with a simplified syntax and better location tagging.

The other half of the `flint` core is the type inference system and classes for the built-in type categories. These classes implement the default type inference rules, which are overridden by custom rules in the `flint.fs` script. Each syntax node computes its type at most once. When a node's type is requested with the `type` method, it checks to see if it has cached a type profile from the last time `type` was called. If not, it runs through all the currently registered custom rules for its kind of node, and executes the first that matches. If no rule matches, it executes its `defaultType` method.

The `flint` analyzer is currently implemented as a customized version of the F-Script framework. To be publicly released, the `flint` code will have to be fully extracted, as requiring users to replace their F-Script installation is not reasonable. Luckily, there are only two firm points at which `flint` directly ties into the F-Script code: in the implementations for closures and symbol tables.

Otherwise, the `flint` only uses the existing methods of the F-Script interpreter classes. These methods are unpublished and partly undocumented, but can be accessed from outside code if the interface is known. There are workarounds for the few alterations mentioned above; although not officially sanctioned, they will permit `flint` to be extracted into a completely separate framework with minimal effort.

5.2 `flint.fs` Script

The user-interface of the tool is `flint.fs`, an F-Script program that is invoked by passing it the name of an F-Script file. `flint.fs` will load the flint framework, have it parse the file, and then perform the program analysis.

While the analyzer core is responsible for the application of type inference rules, `flint.fs` drives the application of program model rules. The tool walks through the syntax tree returned by the analyzer core in depth-first order, trying every relevant rule against each node. Any rule that matches will have its action executed. A global state of the program model, in the form of symbol tables and class information that is maintained by the analyzer core, is altered by these actions as more information about the program becomes available.

As the program is analyzed, certain rules will trigger actions that load and parse more code, such as `sys import:` statements. These rules recursively invoke the tree walker on the new code. The top-level FSProgram syntactic object keeps track of which module files have been loaded, preventing circular references or multiple inclusion.

When the program has been completely processed, `flint.fs` prints out any errors and warnings that were raised, along with files and line numbers / positions on which they occurred.

All of the custom type inference and program model rules are contained within `flint.fs`. To add additional rules, the tool's code must be modified; these modifications only require adding additional rules to the end of arrays in appropriate places.

6 Type Inference Example

Figure 2 lists a snippet of F-Script code that will demonstrate `flint`'s type-inference process. In F-Script, as in its parent language Smalltalk, there is no explicit language construct for conditionals. Instead, a Boolean object is sent a special message, `ifTrue:ifFalse:`, with two closures as arguments. It picks the correct one, evaluates it, and returns the value returned by that closure. In this

sense, it is closer to the boolean conditional of simple lambda calculus than the special form of LISP.

The Boolean class is implemented in Objective-C, so we can read the headers to find that it accepts the `ifTrue:ifFalse:` message. Unfortunately, because the compiled layer has no way of knowing the contents of the Blocks, the message has only the return type Id. This is undesirable, because it obliterates any possibility of recognizing type errors involving the return value of the if/else or anything that uses it downstream.

However, we can use our pattern-matching inference rules to recognize conditionals and properly compute their type. The following conjunctive list of predicates will recognize an if-else structure in F-Script:

1. Is the node a message?
2. Does the type profile of the receiver contain the Class Boolean?
3. Is the message name `ifTrue:ifFalse:`?
4. Are both the arguments literal Blocks?

If all of these conditions are satisfied, the action taken will be to return a type profile that is the union of the two type profiles of the Block arguments. This will involve asking the Blocks to compute their types, which may involve recursive computation of types inside the Block, if there are other nested conditionals. Since nodes always cache type information and only compute it once, the cost of this process is at most linear in the number of nodes under the `ifTrue:ifFalse:` message node.

For the example in Figure 2, the type computed for variable `a` will be a profile containing the Class type categories NSString and NSNumber. When attempting to discover the type for the assignment to `b`, `flint` will look in the header-derived information for the two classes in `a`'s profile. Both implement an `intValue` method: for NSString, it parses an integer from the string; for NSNumber, it truncates the floating-point value to an integer. Both return a type profile containing the lone category Class NSNumber. The union of these two identical profiles will result in the type profile for `b` containing a single Class NSNumber category.

If `(x < y)` is not a valid operation (i.e., if `x` is a String and `y` is a Number), then the type of that receiver will be Error, and the error will be silently propagated to `a`, `b`, and other downstream variables. Similar rules can be used to recognize loops, class creation, and method addition, none of which have language-defined syntactic forms.

```
a := (x < y) ifTrue:[
    out println:'blah'.
    '6'
]
ifFalse:[
    5.5
].

b := a intValue.
```

Figure 2: Example of inferring types based on syntax patterns.

7 Conclusions and Further Work

`flint` serves as a proof-of-concept for performing type checking in mixed programing environments. The system it implements is promising, but more work is needed before it can be used on production code.

7.1 Public Release

To release `flint` to the F-Script community, three tasks must be completed, in increasing order of time required:

1. Extract `flint` code from the modified F-Script core and repackage it as a separate module.
2. Clean up the `flint.fs` script and improve the user interface.
3. Port `flint` and associated tools to Mac OS X 10.5; this includes building 64-bit versions for both x86 and PowerPC.

The last step will take the longest, probably one or two weeks; the tools `flint` uses are heavily involved in the Objective-C runtime, which was completely rewritten by Apple for Mac OS X 10.5. Once the supporting tools are ported, building `flint` for release will be relatively easy.

7.2 Additional Testing

So far, `flint` has only been tested on small cases constructed specifically for development. It has not been tested on a large body of production code. There are several commercial and academic programs that use F-Script (listed on `www.fscript.org`), which could provide more information on `flint`'s real-world utility. The F-Script developer community is small, but enthusiastic; it is likely that the public release of `flint` would generate enough feedback to determine whether the approach in

this paper is useful.

7.3 Typing for Collections

One weakness of `flint` is that it does not have any support for collections. The compiled layer does not provide any help, as Objective-C arrays are heterogeneous, like unparameterized Java collection classes. It would be very useful for developers to have better collection typing, so that arrays could keep track of the types of objects they contain.

Beyond catching errors on simple insertion/access, support for F-Script's special array handling is also desirable. F-Script extends APL's array processing semantics to collections of objects; this extension is called "object-oriented array programing" [7]. For example, the following is legal F-Script code. It will produce a new array that contains the values in `arr1`, incremented by 10:

```
arr1 := { 5, 6, 7, 8 }.
arr2 := arr1 + 10.
```

`flint`'s current type inference system would flag `arr2` as containing an error, because the `+` operator is not supported by the NSArray class. Array handling could be implemented with inference rules that create custom types whenever arrays are encountered, either by literal notation or using allocation methods of NSArray and NSMutableArray. A custom type category, written in F-Script, could handle the special style of method application, providing more information than is strictly available in the Objective-C headers.

7.4 Object Properties

As previously mentioned, `flint` does not attempt to determine the types of object properties for classes written in F-Script. This leads to a great loss of precision when analyzing code that implements methods for these classes. Being able to infer the types of properties and detect the consistency of their use would be very valuable to programmers. This would be a considerable effort, and designing an algorithm to accurately infer these types would probably take more time than any other future step in this section.

References

[1] Apple Inc, "The Objective-C Programming Language", 2006.

[2] William R. Cook, "The Development of AppleScript", *Proceedings of the Third Conference on History of Programming Languages*, 2007.

[3] Craig Chambers and David Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs", *SIGPLAN 1990 Conference Proceedings*, 1990.

[4] Gilad Bracha and David Griswold, "Strongtalk: Typechecking Smalltalk in a Production Environment", *Proceedings of the ACM SIG-PLAN conference on OOPSLA*, 1993.

[5] Mark Hamburg, "Its All Glue: Building a desktop application with Lua", presented at the Lua Workshop, 2005.

[6] Robin Milner, Mads Tofte, and Robert Harper, "The Definition of Standard ML", MIT Press, 1990.

[7] Phillipe Mougin and Stephane Ducasse, "OOPAL: Integrating Array Programming in Object-Oriented Programming", *OOPSLA 2003 Conference Proceedings*, 2003.

[8] Nathanael Schaerli, Stephane Ducasse, and Oscar Nierstrasz, "Classes = Traits + States + Glue", *Proceedings of The Inheritance Workshop at ECOOP*, 2002.

[9] Peter Thiemann, "Towards a Type System for Analyzing JavaScript Programs"; *European Symposium On Programming*, 2005.

[10] Mike Salib, "Starkiller: a type inference system for Python", presented at PyCon, 2004.

[11] David Ungar and Randall B. Smith, "Self: The Power of Simplicity", *OOPSLA 1987 Conference Proceedings*, 1987.

[12] Larry Wall, Tom Christiansen, and Jon Orwant, "Programming Perl 3rd Edition", O'Reilly Media, 2000.