Type Inference in Mixed Compiled / Scripting Environments

Andrew Weinrich
University of Wisconsin
weinrich at cs.wisc.edu

Ben Libit
University of Wisconsin
liblit at cs.wisc.edu

Abstract

Programs written in statically-typed languages are commonly extended with scripting engines that manipulate objects in the compiled layer. These scripting environments enhance 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. To improve the quality of analysis, idiomatic rules based on common programming patterns are used to supplement the type-inference process. 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 to implement custom behavior in game engines (8), and even in mainstream commercial applications like Adobe's recent product Lightroom (5).

The benefits of such integrated compiled / scripting environments are substantial. 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 reduces compilation costs and turnaround time for bug fixes.

However, the added flexibility of a scripting layer 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, such as using an invalid method name, that would easily be caught by a type-checking compiler. In programs where most of the scripting code is used in the user interface, such as web or GUI applications, 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.

In these programs, most of the objects used in 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.

Analysis of scripting code can also be improved by making reasonable assumptions about the coding styles and conventions that programmers follow. By taking for granted that the developer follows common conventions, idiomatic rules can be used to increase the amount of information about a program's structure that is available to the analyzer, allowing for more accurate detection of common errors.

This paper describes a generic system for performing type checking of scripting-language programs that extend a base of code written in a different, compiled, statically-typed language. It also describes how convention-based rules can aid the type-checking process. A proof-of-concept is shown in `flint`, a type-checking and static analysis tool for the language F-Script. F-Script is a scripting language derived from Smalltalk; it is unusual in that it uses the compiled, statically-typed language Objective-C (1) as an implementation platform.

2. Type Checking For Mixed Compiled / Scripting Code

One commonly suggested 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 (14), and making changes to the core language grammar is generally not feasible for a standalone tool.

An alternative is to perform type inference. Type inference algorithms are well-known and can be very effective, as demonstrated by the language ML (9). A type inference system for the mixed environments considered in this paper has the benefit of starting with a much larger set of axioms derived from the compiled code; combined with knowledge of common programming patterns, the type inference is much more tractable than in the general case.

Strictly speaking, not all compiled languages are statically typed, and not all interpreted languages lack static typing. However, in this paper, we deal exclusively with environments in which the compiled code base is statically typed, and the interpreted script code is not. The terms "compiled" and "interpreted" in this paper refer exclusively to statically typed and non-statically typed languages.

2.1 An Object-Oriented Type System

In this paper's type system, all program entities are objects; there are no primitive values such as `int`. Method calls are the only operation on objects, and the type of an object is defined solely by which methods it supports, without any explicit naming.

A type is defined as a finite function from a set of names (as defined by the grammar of the language under consideration) to ordered tuples of types. One of these name-tuple pairs is called a "method". The first type in one of the tuples is the method's return value; subsequent types belong to the method parameters. Variadic, keyword-based, or out parameters are not allowed. Instead of functions, types can also be thought of as finite sets of methods.

If we define an overriding relation for methods and a subtyping relation for types, we can define a lattice for all types in a program. This system is described formally in Table 1. In this table, types are treated as sets of methods.

2.2 Extending the Type System

The type system defined in Table 1 models very simple object-oriented languages, such as Self (13), but it is missing many common OO features like classes and inheritance. These features can be easily simulated within the type system as follows:

- **Void Types** This system does not have the concept of a "void" pseudotype like that of C-derived languages; every method must have a return value, even if there is no such semantically meaningful value. F-Script solves this problem by introducing a singleton instance of the class FSVoid, which plays a similar function as `unit` in ML: it is a placeholder that implements no methods, is never required as a parameter, and is generally useless except for solving the mapping problem. In this paper, the "void" type is referred to as κ_{FSVoid}.

Overriding relationship for methods m and n:

$$m = \langle \texttt{name\_m} \rightarrow (R_m, P_{m_1}, P_{m_2}, \ldots P_{m_j}) \rangle$$

$$n = \langle \texttt{name\_n} \rightarrow (R_n, P_{n_1}, P_{n_2}, \ldots P_{n_k}) \rangle$$

$$\begin{aligned} n \preceq m \iff & \texttt{name\_n} = \texttt{name\_m} \wedge \\ & j = k \wedge \\ & R_n \sqsubseteq R_m \wedge \\ & \forall i : 0 \leq i \leq k | P_{n_i} \sqsupseteq P_{m_i} \end{aligned}$$

Subtype relationship for types A and B (treated as sets):

$$B \sqsubseteq A \iff \forall m | m \in A : \exists n | n \in B \wedge n \preceq m$$

Type $\top \equiv$ the empty map of methods
Type $\bot \equiv$ the universal map of all methods, with $\bot$ as return type and $\top$ for all parameter types

Table 1. Type system for object-oriented programming environments.

- **Classes** Nearly all object-oriented languages are based around explicitly named classes. Our basic type system does not attach names to types; however, named classes can be simulated by the use of appropriate methods. A class can be defined as a type that contains a method that takes no arguments, returns void, and whose name is that of the class prepended with a special token, such as "$$$", that can never occur as part of a valid method name in actual code.

  Where a method requires an instance of an exact class (or subclass) as an argument, it is sufficient for the type of the parameter to contain only the special name method of that class.

  Many object-oriented languages also have "metaclasses," which are objects that implements class methods. In the Java code `MyJavaClass.doStaticStuff(x,y)`, the metaclass object would be the `MyJavaClass` "variable". In Smalltalk and F-Script, the metaclass is a true object, rather than simply a static namespace identifier.

  In this paper, the types for a class / metaclass pair, for instance that of NSFoo, are referred to as κ_{NSFoo} and $\kappa_{\text{NSFoo}'}$.

- **Field Access** Many object-oriented languages allow fields of objects to be directly accessed and assigned without method calls. Access or assignment to a field "foo" of an object can be expressed in this paper's pure method-based system as the methods $\langle \texttt{getFoo} \rightarrow (x) \rangle$ and $\langle \texttt{setFoo} \rightarrow (\kappa_{\text{FSVoid}}, x) \rangle$, where x is the type of the field.

- **Inheritance** With the representation of class types defined above, inheritance follows naturally. If ClassB is a subclass of ClassA, we define type κ_{ClassB} as a method set that contains all the methods in κ_{ClassA}, including the special class-name method. κ_{ClassB} will also contain its own class-name method and any other methods it implements. This makes it a subtype of κ_{ClassA} and provides the expected subclass semantics.

Multiple inheritance works the same way; a class with multiple parents needs only to include all methods from all its superclasses, including their name methods. In practice, multiple inheritance raises significant issues of field and method implementation conflict; our type system does not address these issues.

- **Named Interfaces** Languages like Java that do not allow multiple inheritance often provide some mechanism for creating named, abstract interfaces that classes can implement. These named interfaces can be represented in our type system in the same way as classes. A class that implements multiple interfaces will simply have to contain all the methods that those interfaces declare.
- **Overloading** Many OO languages also support some kind of overloading of method names, based on parameter count and types. In this paper's type system, overloading can be simulated by mangling the names of methods in the manner of C++. The exact mangling scheme is dependent on the target languages.
- **Constructed Types** Many statically typed OO languages support *constructed types*, such as `vector<int>` in C++. Even for languages that do not have these types (also known as *parameterized types*), such as pre-1.5 Java, it is still useful for our analysis to infer them. A constructed type can be simulated in our type system by creating a subclass of the base type (such as Array), and altering its name with the names of its constituent types (such as Array_int). It will be up to the analysis engine to ensure that these types are created and used properly.
- **Protocols** A protocol is a set of methods that represents required functionality for an object, without necessitating the object's membership in a statically-typed interface. For example, in Apple's Cocoa frameworks, protocols are used to specify the methods that delegate objects must implement, but which the type checker does not enforce. In Ruby and other scripting languages, the concept of "duck typing" is analogous to protocols, in that any object that has all necessary "duck"-like methods is considered to be a "duck".

  `flint` uses protocols to represent types when actual class membership is unknown. This is the case for most arguments to functions written in F-Script, where parameters do not have type annotations. The only logical difference between classes and protocols in `flint` is that the type for a protocol does not include any of the special marker methods used to represent class names.

2.3 Drawing Type Information from Compiled Code

Many static type inference systems for languages that lack explicit typing must operate on an entire program written in that language. For the programming environments treated in this paper, the untyped 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 is to read these headers and compile a library of all the classes and methods in the compiled layer. This library forms the set of axioms for `flint`'s type inference analysis, and contains over 700 classes and thousands of methods. Compared to the spare set of primitive types and constructions in a language like ML, `flint`'s initial base of axioms is several orders of magnitude more detailed.

3. Type Analysis

3.1 Type Variables and Constraints

`flint` uses type variables to represent types in a program. These variables represent the partial mappings from method names to type tuples. These maps may not be immediately defined; a brand new type variable is often defined only by sub- and super-type constraints against other variables.

Every syntax node in a program is assigned a type variable, which may already have been defined or may be unique to that node. As the syntax tree is analyzed, the type variable assigned to a node may be replaced by a different variable, or may be constrained to be a subtype or supertype of another variable. The constraints are always one of these three forms:

- Type variable α is a subtype or supertype of variable β
- α and β are the same
- α contains a method $\langle \texttt{method:name:} \rightarrow (\phi, \sigma_1, \sigma_2, \dots) \rangle$

The first stage of `flint`'s type analysis is to load the pre-generated information from the compiled library headers. This produces a large number of class and metaclass type variables that form the foundation for all program analysis. Constraints are added between these pre-generated variables and variables created during program analysis. After a program is completely analyzed, `flint` attempts to find a solution that satisfies all constraints.

In the simplified type system presented above, there are no explicit classes. However, in the actual Objective-C runtime system, every object is an instance of a particular class, and every class has a name. Unlike JavaScript or Ruby, there are no objects that can truly be said to have unique, unnamed types. Classes can be created programmatically, but `flint` ignores this; the justification for this decision is presented in Section 4.4.

Because classes are created declaratively, for every type variable there is a finite number of classes that satisfy the constraints placed on it. Type variables are associated with syntax nodes, so they are also finite in number. Thus, for any F-Script / Objective-C program, there are a finite number of possible solutions to the constraint system. The procedure `flint` uses to resolve constraints on type variables is described in section 5.3.

3.2 Syntax Analysis and Constraint Creation

To build the type system model for an F-Script program, `flint` performs a flow-insensitive, execution-order analysis of the nodes in the syntax tree. As nodes are analyzed, new variables are created and constraints between variables are added. Because F-Script is a simplified version of Smalltalk, there are only a few distinct kinds of syntax of nodes. A simplified grammar for F-Script is presented in Figure 1. The rules for creating and constraining type variables are as follows:

3.2.1 Literals

There are several kinds of literal objects in F-Script: strings, numbers, arrays, booleans, and the nil reference. The type of one of these nodes is the variable for its class. Blocks and arrays are also literal objects, but they are treated specially, as described below.

3.2.2 Identifiers

F-Script has lexical scoping, with variable declarations permitted only as parameters or locals of blocks. Additionally, there is an initial population of global variables for each metaclass object. Every unique identifier in the program is assigned a type variable at its point of declaration. To find the variable corresponding to an identifier, `flint` looks it up in the current scoped symbol table. If the identifier is not found, a new global identifier and variable are created. For the identifier `var`, its type is ι_{var}.

3.2.3 Assignment

For the expression `var := exp`, the constraint $\iota_{\text{var}} \sqsupseteq \tau_{\text{exp}}$ is added to the program. Unlike many languages, the type of the assignment expression itself is always κ_{FSVoid}, regardless of the type of the rvalue.

If an assignment is made to a variable that is not declared in an enclosing block, that variable is created as a top-level global identifier. Because `flint`'s analysis is flow-insensitive, if the same variable has multiple assignments, the constraints are cumulative.

program	::=	stmtlist
stmtlist	::=	exp . stmtlist
	|	ϵ
exp	::=	message
	|	assignment
	|	block
	|	array
	|	IDENTIFIER
	|	NUMBER
	|	STRING
	|	BOOLEAN
assignment	::=	IDENTIFIER := exp
array	::=	{ explist }
	|	{ }
explist	::=	exp , explist
	|	exp
block	::=	[paramlist | | locallist | stmtlist]
	|	[paramlist | stmtlist]
	|	[| locallist | stmtlist]
	|	[stmtlist]
paramlist	::=	paramdecl
	|	paramdecl paramlist
paramdecl	::=	:IDENTIFIER
locallist	::=	IDENTIFIER locallist
	|	ϵ
message	::=	exp METHOD_NAME
	|	exp keywordlist
keywordlist	::=	keywordpair
	|	keywordpair keywordlist
keywordpair	::=	METHOD_NAME: exp

Figure 1. Simplified grammar for F-Script.

3.2.4 Messages

Besides assignment, the only action in an F-Script program is method calls, also known as message sending. In F-Script's Smalltalk-derived syntax, the name of the method is broken into components, with the parameters interleaved between them. The F-Script method call

```
foo doStuff:a withThing:b
```

could be approximated as the Java method invocation

```
foo.doStuff_withThing_(a,b)
```

A method call has zero or more parameter expressions, $p_1, p_2, \dots p_n$. Most of the objects in an F-Script program are implemented in Objective-C, and their methods have concrete formal parameter types that can be compared to the types of actual parameter. However, at the time that a method call is analyzed, there may not yet be enough information to determine the type of the message receiver. We defer this comparison by creating a temporary protocol containing "semi-formal" parameter types, and then tying the actual and formal parameter types to them in separate steps.

To accomplish this, `flint` creates an anonymous protocol representing this method, and then constrains the type of the receiver to be a subtype of this protocol. Given a method name name with actual parameters $p_1, p_2, \ldots p_n$, and receiver type ρ, this protocol is defined as:

$$\pi = \{\langle \texttt{name} \rightarrow (\phi, \sigma_1, \sigma_2, \ \ldots \ \sigma_n)\rangle\}$$

`flint` will add the constraints $\rho \sqsubseteq \pi$ and $1 \leq i \leq n : \tau_{\mathrm{p_i}} \sqsubseteq \sigma_i$. The type of the node itself will be the semi-formal return type variable ϕ.

3.2.5 Statement Lists

In some F-Script contexts, such as block bodies, multiple statements can be separated by a period. A statement can be any of the above kinds of expressions. The final period is optional, unlike in C-derived syntaxes. Each statement list is analyzed individually; the type of the statement list itself is the type of the last statement.

3.2.6 Blocks

F-Script is an unusual language in that it has no special syntax for decisions, loops, or functions. The only way to create a new level of scope is by creating a literal Block, which is analogous to a lambda expression in LISP. A block has the form:

```
[ :param_1 :param_2 ... :param_n|
  |local_1 local_2 ... local_n|
      exp_1.
      exp_2.
      ...
      exp_n.
]
```

Both parameter and local lists are optional. All literal blocks are instances of the Block class, which has methods such as `value`, `value:`, `value:value:`, etc, that are used to execute the block's code with. Executing the block by calling a method with the wrong number of arguments raises an exception.

The return type of all these methods is the empty mapping $\top$; because the interpreter engine has no way of knowing what code it will execute, it is unable to say anything about the return value beyond the fact that it is an object.

Treating the type of a block literal as simply an instance of the Block class is undesirable, as it discards a considerable amount of information about the block's return type and parameter types. To rectify this, every block in an F-Script program is given a custom type like one of the following protocols:

$$\beta_{\phi} = \{\langle \texttt{value} \rightarrow (\phi)\rangle\}$$

$$\beta_{\phi,\sigma_1} = \{\langle \texttt{value:} \rightarrow (\phi, \sigma_1)\rangle\}$$

$$\beta_{\phi,\sigma_1,\sigma_2} = \{\langle \texttt{value:value:} \rightarrow (\phi, \sigma_1, \sigma_2)\rangle\}$$

τ_{x}	Generic type variable
$\iota_{\mathrm{identifierName}}$	Type associated with an identifier (program variable)
$\kappa_{\mathrm{ClassName}}$	Instance of a particular class, including subclasses
$\kappa_{\mathrm{ClassName'}}$	Class object, used for dispatching class methods
π	An anonymous protocol containing at least one method
α_{θ}	Array of objects of type θ
$\beta_{\phi,\sigma_1,\sigma_2,\ldots}$	Block that takes arguments of types $\sigma_1, \sigma_2, \ldots$ and has return type ϕ
$\Sigma_{\tau_1,\tau_2,\tau_3,\ldots}$	A "psuedo superclass" indicating that a variable must be an instance of one of several types.

Figure 2. Notation for Type Variables used in `flint`.

$$\beta_{\phi,\sigma_1,\sigma_2,\sigma_3} = \{\langle \texttt{value:value:value:} \rightarrow (\phi, \sigma_1, \sigma_3, \sigma_3)\rangle\}$$

Once the correct β form is chosen, a copy of the protocol with fresh parameter and return variables is created. Constraints are added between the type variables for the parameter identifiers and the type variables in the protocol: $\iota_{\mathrm{param}_1} = \sigma_1$, $\iota_{\mathrm{param}_2} = \sigma_2$, etc. No additional constraints are added to the local variables. The type of the block expression itself is $\beta_{\phi,\sigma_1,\sigma_2,\ldots}$.

3.2.7 Arrays

F-Script arrays are heterogeneous, so expressions of any type may be found in array literals. For an array of the form { `exp1, exp2, ... expN` }, the type of the array contents is constrained to be at least the least upper bound of the types of the expressions in the array literal. If these expressions are called exp_i, and the type of the array contents is θ, then the following constraints are added:

$$1 \leq i \leq n : \tau_{\mathrm{exp}_i} \sqsubseteq \theta$$

This type θ refers only to the objects inside the array. The type of the array itself, then, is a custom protocol that contains three methods:

$$\tau_{arr} = \alpha_{\theta} = \left\{ \begin{array}{c} \langle \texttt{NSArray} \rightarrow (\kappa_{\mathrm{FSVoid}})\rangle, \\ \langle \texttt{at:} \rightarrow (\alpha, \kappa_{\mathrm{Number}})\rangle, \\ \langle \texttt{put:at:} \rightarrow (\kappa_{\mathrm{FSVoid}}, \alpha, \kappa_{\mathrm{Number}})\rangle \end{array} \right\}$$

The first method is a special marker indicating that this object is an instance of the standard NSArray class. The other two methods are for element access and replacement, respectively. As a shorthand, this type can be called α_{θ}, indicating an array containing elements of type θ.

If the array literal is empty, its type $\alpha_{\kappa_{\mathrm{NSObject}}}$.

4. Idiomatic Type Analysis

Once all nodes have been assigned types and constraints have been added to the program type system, there is still additional work necessary to make `flint`'s analysis useful. During this second phase, `flint` identifies code structures that follow certain programming conventions, and uses them to add additional information to the program's type model.

This second pass relies on the programmer using "reasonable" program structure and standard idioms. Very advanced or very sloppy code can defeat this portion of `flint`'s analysis, but in most cases, programmers provide a wealth of information by using coding conventions. The examples below show how simple pattern recognition can greatly increase the accuracy of `flint`'s error detection.

4.1 Leveraging Coding Conventions

Many proposed type checking systems for scripting or other non-statically-typed languages, such as Smalltalk (3), JavaScript (11), and Python (12), 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 analysis is a tool for programmers that alerts them to possible problems. We 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 strictly speaking 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 (15), 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 are 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 our analysis 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 a pointer from `void*`, the programmer has extra knowledge about the program's state.

4.2 Control-Flow Graph Recognition

In F-Script, as in its parent language Smalltalk, there is no explicit language construct for conditionals. Instead, a Boolean object is sent a 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 simplicity and flexibility of this approach brings with it the inconvenient absence of a decidable control-flow graph for the program.

The Boolean class is implemented in Objective-C, so `flint` knows that it accepts the `ifTrue:ifFalse:` message from reading the header files. Unfortunately, because the compiled layer has no way of knowing the contents of the Blocks, the message has only the return type $\top$. 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 write a pattern-matching inference rule to recognize conditionals and properly compute their type. The following conjunctive list of predicates recognize an if-else structure in F-Script:

1. Is the node a message?
2. Is the message name `ifTrue:ifFalse:`?
3. Is the receiver a subtype of κ_{Boolean}?
4. Are both the arguments literal Blocks?

If all of these conditions are satisfied, the return type of the message is constrained to a member of the intersection of the return types of the two literal blocks.

For the example in Figure 3, the type computed for variable `a` is neither κ_{NSString} nor κ_{NSNumber}, but a partial supertype of the two: $\Sigma_{\kappa_{\text{NSString}},\kappa_{\text{NSNumber}}}$. This type can be though of as a pseudo-superclass that both classes inherit from. This would be the case even if if one of the branches had returned an array; in that case, the type of the if-else expression would be $\Sigma_{\kappa_{\text{NSString}},\alpha_{\theta}}$.

Both of the classes represented by $\Sigma_{\kappa_{\text{NSString}},\kappa_{\text{NSNumber}}}$ contain a method called `intValue`: for NSString, it parses an integer from the string; for NSNumber, it truncates the floating-point value to an integer. Thus, when analyzing the call to `intValue` on the last line, `flint` will be able to correctly add the constraint $\iota_{\text{b}} \sqsupseteq \kappa_{\text{NSNumber}}$.

4.3 Module Importation

F-Script does not have a special syntax or pragma for module inclusion; instead, a regular message is sent to the predefined object `sys`, which causes the interpreter to load another file of F-Script code:

```
sys import:'Dictionary'.
sys import:'switch'.
```

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

b := a intValue.
```

Figure 3. Example of inferring types based on syntax patterns.

```
sys addLibrary:'/Users/bob/FScriptLib'.
sys import:'LinkedList'.
```

The argument to `sys import:` is the name of the file, minus the `.fs` extension. There are several predefined locations where F-Script will look for library files. Additional library directories can be added by defining the `FSCRIPT_LIB` environment variable, or by using the `sys addLibrary:` method.

This "import" command is semantically more equivalent to `eval`, because the library file is executed, not just parsed, before returning to the original file's code. `flint` assumes that programmers follow the convention of library files performing no actions other than creating classes and special variables, using the procedure described below.

During `flint`'s second pass, it looks for messages sent to the `sys` identifier. If the message is `addLibrary:`, `flint` adds an entry to its own internal list of library locations, which is initially copied from the interpreter. If the message is `import:`, `flint` checks to see if the library can be found, and issues a warning if it is not, or if it is not syntactically valid F-Script. If the library exists, `flint` loads, parses, and recursively analyzes the the library code. Like the F-Script interpreter, `flint` maintains a list of libraries that have been imported, so recursive dependence will not cause files to be analyzed more than once. There is a single model of the program, and all constraints from libraries go into the same model.

For `flint`'s analysis to work properly, programmers must import libraries earlier in the source code than they are used. It is possible to write semantically correct code does not follow this convention, but that is such bad style that it is reasonable to assume that programmers will always declare all dependencies at the beginning of a file. It is also possible to defeat this analysis by using non-literal strings, which would be the case if the program itself were deciding at runtime which libraries to import. In any language, the process of dynamically linking code limits the ability of static analysis tools to determine if a running program will actually encounter errors. In this regard, `flint` is no different.

4.4 Programmatic Class Creation

Although most classes in an F-Script program are written in the compiled Objective-C layer, it is possible to write classes directly in F-Script. The language lacks any special syntax for class creation, so classes must be created programmatically, with properties and methods added to the class one at a time. During the second analysis pass, `flint` attempts to recognize the patterns of class creation, and adds constraints to the program model.

It is possible for classes to be created so that they function correctly, but in a way that defeats `flint`'s attempt to analyze them. `flint` assumes that the programmer follows the coding conventions described below. If any more "sophisticated" structure is used, the programmer is assumed to not need the help of a type checker.

4.4.1 Class Creation

Classes are created in F-Script by creating a metaclass object from a special "class factory" called `FSClass`:

```
MyNewClass := FSClass newClass:'MyNewClass'.
```

When `flint` recognizes a line like this, it creates the type variables $\kappa_{\text{MyNewClass}}$ and $\kappa_{\text{MyNewClass}'}$, and adds the constraint $\iota_{\text{MyNewClass}} = \kappa_{\text{MyNewClass}'}$.

4.4.2 Property Addition

Properties are added to classes with the command

```
MyNewClass addProperty:'bar'.
```

This line creates a new variable τ_{bar} and adds the constraints:

$$\kappa_{\text{MyNewClass}} \sqsubseteq \left\{ \begin{array}{c} \langle \texttt{bar} \rightarrow (\tau_{\text{bar}}) \rangle, \\ \langle \texttt{setBar:} \rightarrow (\kappa_{\text{FSVoid}}, \tau_{\text{bar}}) \rangle \end{array} \right\}$$

There is also a method called `addClassProperty:`, which has similar operation but affects the metaclass MyNewClass'.

4.4.3 Method Addition

Methods are installed to classes by giving the metaclass object a block to execute when an instance receives a particular message:

```
MyNewClass onMessage:#doStuff:withThing:
                  do:[ :self :stuff :thing |
    ....
].
```

There are several important points to note about this style of method declaration:

1. The token `#doStuff:withThing:` is a *selector*, a special data type that represents the name of a method.

2. As in Python and JavaScript, self must be explicitly declared; there is no implicit parameter. The use of the name self is only conventional, but it must be the first parameter.
3. Unlike methods for classes written in Objective-C, the parameters for F-Script methods will have types that are anonymous protocols determined by constraints added to their use inside the method. Instead of the single class (with descendants) that is legal for an Objective-C method, there may be many classes that could satisfy the constraints on an F-Script method parameter.
4. It is possible to provide a reference to a previously created Block as the implementation, instead of using a literal. This is useful when creating classes on the fly and using partially bound closures to implement methods. However, because flint requires information from the actual block structure (such as parameter names and count), such method addition will be ignored. It is assumed that when writing such advanced code, the programmer has no need of static analysis.

When flint encounters a node that matches this pattern, it performs the following actions:

1. The block is checked to make sure that it has the correct number of arguments: the number of colons in the selector + 1 for the receiver. If it does not, flint reports a warning to the user and performs no further action.
2. It is conventional to name the first parameter self. If this convention is ignored, flint reports a warning.
3. The type of the first block parameter, self, is constrained to be a subtype of the class to which the method is being added:
$$\sigma_1 \sqsubseteq \kappa_{\text{MyClass}}$$
4. As shown in Section 3.2.6, the return type of the block is ϕ, and the block parameters already have the types σ_i. flint creates a custom protocol containing this new method, and the the constraint:
$$\kappa_{\text{MyClass}} \sqsubseteq \{\langle \texttt{selectorName} \rightarrow (\phi, \sigma_2, \sigma_3, \ldots \sigma_n)\rangle\}$$
The method parameters start with σ_2 because of the presence of the implicit parameter.
5. If the same method is implemented in a superclass, the parameters for this method implementation are constrained to be supertypes of the superclass method's parameters:
$$\sigma_i \sqsupseteq \sigma_i^{super}$$
Likewise, the return type is constrained to be a subclass of the superclass method's return type:
$$\phi \sqsubseteq \phi^{super}$$

The command onClassMessage:do: performs a similar operation for class methods.

It is also possible to add methods to existing classes that were written in Objective-C. Such a collection of methods is called a "category", and is similar to the Ruby idea of "mixins". F-Script extends all Objective-C metaclass objects to support the onMessage:do: and onClassMessage:do: commands.

If the method is analyzed without any errors, flint adds it to its internal library of class functionality. This addition uses a separate API from that used to add type constraints; the newly added method is treated no differently than a method that was written in Objective-C.

4.5 Advanced Array Processing

The original research goal of F-Script was to demonstrate the feasibility of *object-oriented array processing* (10). To this end, F-Script incorporated the array processing functionality of APL (6). For example, if we have two arrays of equal length a and b, the following code assigns the scalar product of a and 4.5 to c, and assigns the inner product of the two arrays to d:

```
a := { 2, 3, 4, 5, 6 }.
b := { 6, 7, 8, 9, 0 }.
c := a * 4.5.
d := (a * b) \#+.
```

Here, the operation a * b performs pairwise multiplication of the corresponding elements in a and b and returns an array containing the result. Attempting to use arrays with unequal element counts raises an exception. \ is the reduction operator; in this case, the second operand is the selector for the + operator, which is internally converted into the method name operator_plus:. This allows a very concise expression of an operation that in most languages would require either a loop or an explicit collection operation. In most cases, no extra syntax beyond normal message sending is required. This functionality is provided partly by checks in the F-Script interpreter, partly by the runtime delegation functions of Objective-C, and partly by code in the Array class.

There are several variations to the array processing capabilities of F-Script; flint provides partial support for the most common cases. When flint sees a message that is sent to an object that can be positively identified as an instance of NSArray, it checks the message name against the list of methods that NSArray supports (such as count, at:, etc). If it is not one of those, the message is assumed to be sent to all the objects of the array, and the return value is an array containing the individual results. This is the same procedure that the F-Script runtime uses to implement array messaging.

In the following code, arr is a list of angles in radians, with the value of π factored out. The code first multiplies the angles by π, then performs some trigonometric operations:

```
arr  := { 0.25, 1.0, -0.27, 2.05 } * 3.14159.
cos  := arr cosine.
sin  := arr sine.
prod := cos * sin.
```

The fact that the first operand to * is an array is determined by testing the type for inclusion of the special Array marker method. In this example, we take the elements in the array to have type χ, and the solitary number in the first line to have type ψ (here, those types are both the variable κ_{Number}, but in general this cannot be assumed). From this, we know that the type of the left operand is the special array type variable α_χ.

By recognizing that the second operand to * is not an array, flint infers that the operator_star: message will be sent to every element in the array. This leads us to add the constraint:

$$\chi \sqsubseteq \{\langle \texttt{operator\_star:} \rightarrow (\gamma, \psi)\rangle\}$$

The assignment statement adds the constraint:

$$\iota_{\text{arr}} \sqsupseteq \alpha_\gamma$$

When flint attempts to resolve the types for arr in the second and third lines, it finds that the lower bound is an array of type α_γ. These unary methods add four constraints, with fresh type variables for the return values:

$$\gamma \sqsubseteq \{\langle \texttt{cosine} \rightarrow (\mu)\rangle\}$$

$$\gamma \sqsubseteq \{\langle \texttt{sin} \rightarrow (\nu)\rangle\}$$

$$\iota_{\text{cos}} \sqsupseteq \alpha_\mu$$

$$\iota_{\text{sin}} \sqsupseteq \alpha_\nu$$

Finally, when the two arrays cos and sin are multiplied together, flint notices that it is an operation between arrays, with the elements of the first array as receiver. This adds the final constraints:

$$\mu \sqsubseteq \{\langle \texttt{operator\_star:} \rightarrow (\epsilon, \nu)\rangle\}$$

$$\iota_{\text{prod}} \sqsupseteq \alpha_\epsilon$$

A complete resolution of these constraints shows all of χ, ψ, γ, μ, ν, and ϵ to be equal to κ_{Number}.

4.6 Custom Inference Rules

Although many of F-Script's idioms are well-known enough to write them directly into flint, it is desirable to allow end-users to write their own rules that can better express the behavior of their code. For example, one of the modules that comes with the F-Script standard library is the Pair class, the complete definition of which is shown in Figure 5 at the end of this paper. This class has no behavior besides having the properties first and second. It also adds the operator => to the class NSObject, which allows the arrow to be used as a pairing operator between any two objects.

This class is used often in the F-Script standard library. Ideally, Pair objects would be polymorphic, in the style of tuples in the languages ML and Haskell. By default, flint is incapable of this. Once it has analyzed the Pair.fs file, it will find no constraints on the types of the first and second properties. flint will then fix the type of κ_{Pair} as the following:

$$\kappa_{\text{Pair}} = \left\{ \begin{array}{c} \langle \texttt{first} \rightarrow (\top)\rangle, \\ \langle \texttt{second} \rightarrow (\top)\rangle \end{array} \right\}$$

This makes it impossible to perform any analysis on the use of Pair objects. To make a more useful analysis possible, flint allows for users to write custom syntax rules that match syntactic patterns. These rules are written in F-Script, and so can be added incrementally to the analyzer without recompiling the Flint bundle.

The if-else recognition predicate described in Section 4.2 is one such rule. For creating Pairs, the rule would be even simpler; it would match any method call with the name operator_equal_greater: (the internal F-Script name for the => operator). This rules relies on the coding convention that => is used exclusively for pairing objects, and does not have any other use.

An inference rule written in F-Script overrides flint's normal processing when they match, and executes a block of F-Script code that can return a custom type. In this case, if the types of the two elements in the pair are α and β, the rule would return the custom type:

$$\zeta_{\alpha,\beta} = \left\{ \begin{array}{c} \langle \texttt{first} \rightarrow (\alpha)\rangle, \\ \langle \texttt{second} \rightarrow (\beta)\rangle \end{array} \right\}$$

$$\zeta_{\alpha,\beta} \sqsubseteq \kappa_{\text{NSObject}}$$

Note that the actual Pair class does not appear in this type, nor do the methods setFirst: and setSecond: appear, even though they do technically exist. This type will then be treated by flint the same way as it would treat any other type.

5. flint Architecture

flint is implemented as a Mac OS X bundle written in Objective-C. The F-Script interpreter links with this bundle at launch. The code in flint uses several undocumented features of the F-Script engine, mostly for accessing the symbol table and parse tree. A short script, flint.fs, is used to run the analyzer on a target file. This script initializes the engine, creates an initial FlintAnalyzer object, registers idiomatic rules, and tells it to load the target program.

5.1 Compiled Code Library

Before analyzing a program, flint loads the library of compiled type information. This library is generated from Objective-C framework headers by a separate tool, and is stored in a serialized format that requires less than 2 seconds to parse and load on a modern Macbook Pro. This library will only need to be regenerated when the frameworks change.

5.2 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. Although identifiers in F-Script are lexically scoped, resolution occurs at runtime, and so the syntax tree does not contain links between identifier nodes and declarations. The F-Script interpreter also does not keep enough information on 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, fully resolved identifiers, and better location tagging.

After the syntax tree has been created, `flint` moves through the tree in execution order, applying the rules from Section 1 to create new type variables and constraints This default The code in these classes may be overridden by custom inference rules written in F-Script, as described in Section 4.6. Each syntax node of the program has a rule applied to it only once.

At various points, idiomatic rules will recognize module importation and tell the analyzer to load additional code. This may include both new F-Script code that must be analyzed, and compiled with compiled type information.

After the F-Script program has been completely analyzed, `flint` attempts to find a solution to the global system of type constraints.

5.3 Constraint Resolution

`flint` uses a constraint system based on the model of the Banshee engine (7). As described below, there is a finite set of solutions for all type variables, so `flint` is guaranteed to find a solution for a constraint system, if one exists. Some expressions in F-Script, such as arrays and closures, give rise to type variables that indicate custom behavior. In these cases, they are treated as singleton instance of custom classes. Because both closures and arrays are special syntactic forms, there are a finite number of both in a program, and the number of unique classes in F-Script's library will remain finite.

A valid solution for an F-Script program's type constraint system is one in which there is at least one class that can satisfy the constraints on every variable. The constraint system of a program can be modeled as a directed graph, with type variables as nodes and subtyping relationships as edges.

5.3.1 Identifying Type Variable Solutions

All constraints placed on a type variable reduce the number of classes that the variable may represent. This reduction may be "from above", in the case of a subtype relationship, which eliminates from contention superclasses from higher levels of the hierarchy; or it may be "from below", in the case of a supertype constraint. In both cases, the total number of candidate classes is reduced. Once all classes have been eliminated from contention, no further constraints can have any effect, and it is known that the program will encounter type errors at runtime.

It would be very inefficient to tag each type variable with the complete enumeration of classes that satisfy its constraints. In most object-oriented languages, the set of all classes (including abstract classes) can be represented as a potentially disconnected, directed acyclic graph. `flint` uses this representation to efficiently identify portions of the hierarchy that satisfy type constraints.

In this representation, a constraint to be a subtype of, for example, κ_{NSFoo} eliminates all class nodes that are not reachable from κ_{NSFoo}'s node. A superclass constraint eliminates all nodes from which κ_{NSFoo} is not reachable. A sample representation of this kind of solution restriction is shown in Figure 4, with a set of nodes defined as only the conjunction of two reachability restrictions. `flint` uses these restrictions to efficiently determine which classes satisfy the constraints on a variable.

It is possible for constraints to be added that eliminate all nodes in the class hierarchy from the solution set for a variable. This indicates a type error, as there is no class that has all of the functionality the program requires.

Pseudo-superclasses (i.e. the $\Sigma_{ClassA,ClassB,...}$ type variables) can be represented by creating brand new classes and imposing subtype constraints on them. These classes are only created in special circumstances (such as the custom rule for conditional expressions) that are based on a limited number of syntax patterns, so they do not invalidate the finitude of possible type variable solutions.

5.3.2 Constraint Data Flow Analysis

The application of constraints on type variables can be implemented as a straightforward data-flow analysis of a graph. The properties of this data-flow are shown in Table 2. All type variables start their lives as the top value, representing the set of all possible classes. As constraints are added, they move down the solution lattice. Reaching the bottom value indicates a type error.

Because the program analysis is flow-insensitive, constraints may affect type variables created at a syntactically earlier location. After all syntactic analysis is complete, `flint` recursively applies each constraint until all type variables in the program have reached fixed solutions. This global solution is the final typing of the program.

Graph Nodes	Type variables.
Graph Edges	Supertype and subtype constraints. Although a subtype and supertype constraint are semantically equivalent, it is more natural to see the information as flowing in a particular direction. For the constraints described in Sections 3 and 4, the variable on the left-hand side is the one being constrained. By default, the propagation of constraints follows this direction, rather than the direction from supertype to subtype.
Data Values	Sets of possible classes that will satisfy all constraints. These values are defined as the set of all subsets of the nodes in the program's class hierarchy.
Partial Order on Data Values	Subset.
Top Value	The set of all classes, equivalent to κ_{NSObject}. This is the default value for all type variables.
Bottom Value	The empty set of classes, which represents an invalid solution.
Transition Function	Application of constraints; because constraints always remove classes from consideration, this function is monotonically decreasing.

Table 2. Data-flow Analysis for Resolving Type Constraints

5.3.3 Optimization

In even a moderate-sized program, there will be thousands of type variables. `flint` uses several optimization techniques to reduce the amount of work it has to perform in resolving the type constraint system:

- **Online solution reduction**
  It is not always necessary to wait until the end to apply constraints. For example, if an identifier is assigned a constant, resulting in the constraint $\iota_{\text{var}} \sqsupseteq \kappa_{\text{NSFoo}}$, `flint` will immediately restrict the possible solution set of ι_{var}, reducing the number of iterations that must be performed in the final analysis.
- **Immutable type variables**
  Many of the variables in a program will represent classes from libraries, such as κ_{NSString} and $\kappa_{\text{NSNumber}'}$. There is only a single class that these variables can represent, so `flint` "hardwires" this solution to them at the beginning of its analysis. Any attempt to add additional constraints to these nodes must be incorrect, so they may be removed from consideration without affecting the rest of the program model.

  As mentioned above, it is possible to add additional methods to existing classes. However, `flint` uses a separate internal API to perform these modifications. As stated, our analysis relies on a coding convention that all class categories and libraries will be imported at a syntactically earlier point than their functionality is first used.
- **Collapsing variables**
  In the case of a literal object, such as a string constant, it is useless to have a dedicated type variable for that syntax node; the node can simply take the κ_{NSString} type. Likewise, if only a single assignment is made to a variable, the types of the left- and right-hand sides are identical, and the same variable can be used for both. `flint` aggressively collapses type variables when it can prove that they are identical.
- **Subgraph identification**
  Because the effects of constraint resolution follow a directed graph, it is possible that a change to a variable will only affect a portion of the type system. `flint` performs automatic, on-demand identification of dependent subgraphs for each type variable, and caches this information so that future iterations will not have to search the entire graph.

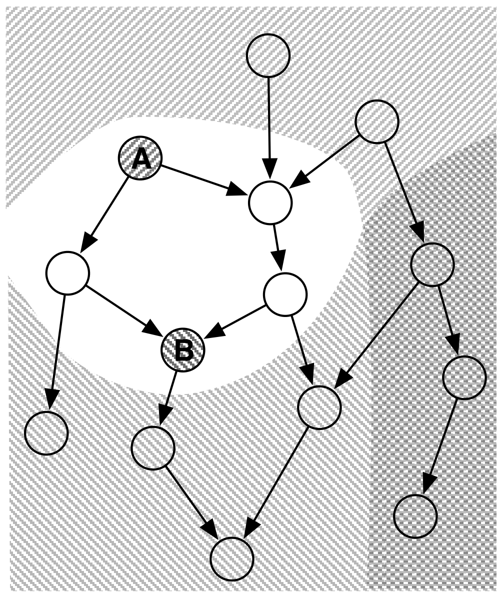

Figure 4. Sample class hierarchy. Unshaded portion shows the possible classes for type variable τ given the constraints $\tau \sqsubseteq \kappa_{\text{A}}$ and $\tau \sqsupseteq \kappa_{\text{B}}$.

5.3.4 Reporting Errors

`flint` detects and reports two kinds of errors: type errors resulting from conflicting constraints, and misuse of programming idioms that are likely to lead to run-time exceptions.

5.4 Type Errors

There are two ways that a type error may arise: calling a method that a class does not implement, and calling an existing method with an incompatible parameter type.

Method implementation errors are detected during resolution of constraints, as described above. If `flint` ever determines that the set of possible classes for a type variable is empty (the bottom value in the data-flow analysis), it reports an error and removes that type variable, along with all associated constraints, from the graph. This prevents errors from cascading, but also hides other problems that may arise when that error is fixed.

Parameter type errors are detected after the fixed point for the type variable graph has been created. As described in Sections 3 and 4, type variables for methods call receivers are constrained against protocols that contain the types of the arguments. After the set of possible classes for a type variable has been calculated, the methods that are called on it are checked against these candidate classes. If any method is called with an incompatible parameter, as defined by the inheritance hierarchy, an error is reported to the user.

5.4.1 Programming Idiom Errors

`flint` can find several common programming mistakes that, while not strictly errors, will probably cause problems at run-time. These constraints are matched by a series of heuristics during the second pass of `flint`'s analysis. These heuristics, or idiomatic rules, are in the style of the venerable C utility `lint`, hence the name of this paper's program (for "**F**-Script **lint**").

While it is possible to write a correct program that matches some of these warning patterns, in most cases they indicate the presence of serious problems. A programmer whose technique involves heavy violation of these idioms probably would not find much use in a program-analysis tool like `flint`.

Idiomatic rules are written in F-Script, using an API similar to that for the custom type inference rules. Several of the idiom violations that `flint` detects are:

- If a method is being added to a class, it is possible for the implementation block and method name to have a different number of argument slots, or for the block to be missing the implicit parameter. These errors will raise exceptions when the library is loaded, but `flint` will attempt to catch them at analysis time.
- As described above, when `flint` notices that a library file is imported, it first checks to see if that library exists before loading it. If the file does not exist, `flint` continues its analysis but reports a warning. When running the program, the absence of the library file would cause an exception.
- F-Script has several reserved identifiers, including `sys`, `args`, and the identifiers for metaclass objects. Assignment to these identifiers is syntactically legal, but raises an exception at runtime. As part of the second analysis pass, `flint` checks every assignment statement to see if the lvalue is one of these reserved identifiers. If so, it discards any constraints that had been added to the identifier and warns the user.
- κ_{FSVoid} is a placeholder that is required to map Objective-C void-valued methods to F-Script; actually using it is as inappropriate as using the `unit` value in ML. If a message node can be determined to have type κ_{FSVoid}, `flint` checks to ensure that that type is not propagated anywhere via assignment or use as an expression. If it is, `flint` reports an error.

Because idiomatic rules are written in F-Script, they can be used by the programer to extend `flint`'s analysis.

A. Appendix

Acknowledgments

Thank you to Phillipe Mougin for creating F-Script and encouraging the development of `flint`.

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] Ken Iverson, "A Programming Language", John Wiley and Sons, 1962.

[7] John Kodumal and Alex Aiken, "Banshee: A Scaleable Constraint-Based Analysis Toolkit", *Proceedings of SAS 2005*, 2005.

[8] Noel Llopis and Sean Houghton, "Backwards Is Forward: Making Better Games with Test-Driven Development", presented at the Game Developers Conference, 2006.

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

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

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

[12] Mike Salib, “Starkiller: a type inference system for Python”, presented at PyCon, 2004.

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

[14] Guido van Rossum, “Optional Static Typing: Stop the Flames!”, `http://www.artima.com/weblogs/viewpost.jsp?thread=87182`.

[15] Larry Wall, Tom Christiansen, and Jon Orwant, “Programming Perl 3rd Edition”, O’Reilly Media, 2000.

```
"Create a simple Pair class"
Pair := FSClass newClass:'Pair' properties:{'first', 'second'}.

Pair onClassMessage:#pair:with: do:[ :self :first :second | |newPair|
    newPair := self alloc init.
    newPair setFirst:first.
    newPair setSecond:second.
    newPair
].

"Add a pairing operator to NSObject, and thus to every other class"
NSObject onMessage:#operator_equal_greater: do:[ :self :second |
    Pair pair:self with:second.
].
```

Figure 5. Source code of the Pair class.