Using Static Analysis For IDE's for Dynamic Languages

Julian Dolby
IBM Thomas J.Watson Research Center
`dolby@us.ibm.com`

October 14, 2005

Abstract

Modern IDE's for languages such as Java exploit the static type system of the language to provide shortcuts and hints to aid programmers with common programming tasks. An example of this help is auto-completion of field and method names when a programmer types a "." operator. In dynamic languages, such as scripting languages like JavaScript[1] and programming languages like Scheme[3], there is no such static type system that can be exploited in this manner. However, a deeper analysis of the source program may be able to provide equivalent information that allows similar levels of functionality in an IDE for dynamic languages. To enable this, we discuss various uses to which analysis data could be put, and we argue for a layered approach that exploits the same analysis infrastructure across multiple languages

Modern IDE's provide a range of shortcuts and wizards to ease, and to some extent automate, many tedious aspects of programming. While some of these shortcuts are straightforward insertion of boilerplate text— such as a "new class" operation for Java that simply inserts some syntax—there are many that are context-sensitive in some manner—such as auto-completion of field and method names. In statically-types languages such as Java and C++, much of this context information can be derived from declared type information in a relatively straightforward manner. For instance, the explicit type information for the left-hand side expression of a "." or "->" operator can be used to generate a fairly good list of possible completions for constants on the right hand side. Similarly, IDEs often indicate visually when some method declaration is overriding a parent declaration. In a class-based language, this information can be derived by examining the declarations of methods and inheritance in each class. Another common feature is to provide a list of possible called functions at a particular call site. Once again, the type of the receiver object can often be used to provide a good list. Overall, with myriad helpers such as these, modern IDEs provide a range of popular shortcuts that are, at least conceptually, simple to implement.

The situation is rather different for dynamic and scripting languages: there is, in general, little or no static type information to exploit, making context-dependent shortcuts a much more challenging proposition. Consider JavaScript as an example; it is dynamically-typed and has a prototype-based object model which makes inheritance potentially dynamic too. If one were to attempt auto-completion of the ".' operator in JavaScript, it is unclear how a plausible list of completions could be derived. In general, properties could be set and deleted anywhere in the script and one would need precise alias analysis to determine which properties might have been assigned to the left-hand side expression of the ".". Similarly, deciding whether one property definition overrides a parent one would require alias analysis to work out which objects may inherit from which others in the dynamic prototype-based model that JavaScript has.

One solution implicit in the foregoing paragraph is to use some kind of alias analysis on the program being edited. How practical this may be depends to some extent on the language, and what kinds of assumptions one is willing to make about the completeness of the program being edited. On the other hand, a system that employs inter-procedural alias analysis does not need to make more assumptions about completeness than the type-system-based shortcuts of current IDE's already do. Those already assume a closed world to some extent, since it can hardly suggest things that do not exist. However, even if assumptions sufficient for non-trivial inter-procedural analysis are reasonable, there is still the issue that a static analysis infrastructure is a major undertaking. This difficulty is multiplied by the fact there are so many different popular scripting languages; it hardly seems practical to write inter-procedural analysis for all of them.

Our solution comes from the observation that the diverse collection of popular dynamic and scripting languages mostly share a common core and fundamental approach. This approach has been called languages for managed runtimes and sometimes called type safe languages in the sense of not allowing low-level misuse of memory. They also tend to share a range of common structures, such as common types of loops. Features such as a class-based object model, proper lexical scoping and first-class functions are by no means universal, but they tend to share some fundamental basics when they do appear.

Thus, we believe that we can build an analysis framework that comprises a large shared core that understands how to analyze many of these basic structures, and a collections of relatively thin layers that adapt from the various scripting languages to the shared core. On top of this analysis layer, IDE's for the various languages can exploit sophisticated analysis components such as whole-program, context-sensitive alias analysis to provide the same kinds of context-dependent help as do current IDE's for more static languages.

So far, we have built the beginnings of this approach, with an Abstract Syntax Tree based intermediate layer feeding into the DOMO[2] analysis infrastructure developed at Watson. So far, we handle JavaScript from two front-end parsers, the Rhino parser from Mozilla and an internal IBM parser. We also have support for a substantial and increasing portion of the Java language. Further planned work includes PHP and possibly other scripting languages popular in open-source web community.

In this document, we will first motivate some of the ways in which program analysis can be used to enhance IDE's for dynamic and scripting languages in Section 1. We will then outline our approach for supporting program analysis across our targeted range of programming languages in Section 2. In Section 3, we illustrate our approach by briefly describing some issues in analyzing JavaScript.

1 Uses of Program Analysis

In this brief paper, we will restrict our attention to what we see as two of the most pressing concerns for enabling IDE's for dynamic and scripting languages: the notion of types and the notion of a call graph.

1.1 Approximating Types

The most obvious use of program analysis in IDE's for dynamic languages is to substitute for the static type information exploited by IDE's for Java and C++. There are really two distinct aspects of this use. The first is to approximate the possible types of variables in the program being edited, which substitutes for the required type declarations for such variables in Java or C++. The other use is to approximate the notion of "type" itself for languages such as JavaScript, which have no static notion at all of classes or types.

Approximating Type Declarations The simpler notion is to analyze to determine what types of objects a particular value could hold at runtime. Traditional pointer analysis essentially computes this sort of information, if one uses a pointer analysis that distinguishes heap objects at least on a per-class basis. That means that the pointer analysis approximates all possible heap-allocated objects using one abstract object to represent all objects of a given concrete type. A concrete type is one that may actually be allocated by the program. The results of this pointer analysis for a particular variable will be the set of concrete types that variable could possibly hold at runtime. This information could either be used directly, or could be used to compute a "most general type", i.e. the least superclass of all the concrete types that pointer analysis finds.

Consider the example PHP program in Figure 1. This is a trivial example, but it serves to illustrate the analysis of types. It would require some analysis to infer that the type of `$arg_1` is always a `User`; this information would typically be declared by the programmer in Java, but would need to be inferred here.

```
<?php
function printIt($arg_1) {
  echo $art_1;
}

class User {
   var $test;
   var $test2;
}

$user1 = new User();
$user2 = new User();
$user1->test = "user1's test";
$user1->test2 = "user1's test2";
$user2->test = "user2's test";
$user2->test2 = "user2's test2";

printIt($user1)
printIt($user2)
?>
```

Figure 1: Simple PHP Example

Approximating Types Themselves Some dynamic languages do not have any declarative notion of types at all; such languages are no longer only weird niche players, but include popular Web scripting languages such as JavaScript. In this situation, analysis can be used to synthesize a notion of "type" by statically differentiating possible runtime objects into groups that must have similar behavior. In JavaScript, this would mean associating "types" with sets of possible objects that have the same set of properties, especially the same set of properties that hold function values. This approach can extend to "sub-typing' using a behavioral idiom in which a "sub-type" has all the function properties of a "super-type" and some additional ones. This notion actually conforms to the "inheritance" mechanisms that JavaScript supports.

```
function User(test, test2) {
  this.test = test;
  this.test2 = test2;
}

var user1 = new User("x", "y");
var user2 = new User("xx", "yy");

user2.test3 = "whatever";
```

Figure 2: Simple JavaScript Example For Types

Consider the contrived JavaScript code in Figure 2. While JavaScript does not have class declarations, it allows functions to be used as constructors, which can often have a similar result. In this case, it might be helpful to say that `user1` is of the type `User`, even though that is not really a type in JavaScript. And consider `user2`; it might be plausible to say this is a subtype of `User`, since it has more properties. And since these objects could be given additional properties elsewhere, it would require some form of pointer analysis to decide what reasonable types might be in general.

This notion of approximating types also fits directly with the traditional notions of pointer analysis. In this case, the pointer analysis would need some fairly fine-grained approximation of possible runtime objects, for example using one abstract object per allocation site in the program. Then, one could look at the possible property values of the abstract objects, and group them into "types" based on the set of properties they might contain.

This approach actually has a long history, based on program analysis work done on prior prototype-based languages, most notably the Self system.

1.2 Approximating Call Targets

IDE's for Java reply on two related facts to provide an often quite reasonable approximation of the possible callee methods at a given call site; the first of these is that Java lacks any notion of first-class functions, and the second is that the receiver of any method call has a statically-declared type. Thus, the set of possible callee methods is simply the set of methods with appropriate names and signatures declared in the receiver type and its subtypes. For dynamic languages the situation is more complex: they typically do have first-class functions of various kinds and do not have static type declarations. The type issue can be addressed as we discussed above in Section 1.1, but that does not help much with first-class functions. However, alias analysis can again provide an approximation of the needed information.

In a language with first-class functions, functions are objects like any others and can be tracked with pointer analysis just as other objects can. Thus, the set of possible callee methods at a call site for a first-class function is simply the set of functions to which that variable can refer. Furthermore, the number of function allocation sites (usually function declarations and function expressions) is typically limited, since each such site is a distinct piece of code. Hence, aggressive pointer analysis techniques can be employed to distinguish the various functions created in the program and to track their flow to call sites.

```
var whatever;

var foo = function(x) { return x + 5; };
var bar = function(x) { return x - 5; };
var dead = function(x) { return x * 5; };

some_other_function();

if (...) {
  whatever = foo;
  whatever(6);
} else
  whatever = bar;

whatever(7);
```

Figure 3: Simple JavaScript Example For Call Targets

Consider the snippet of JavaScript code in Figure 3. If an IDE is providing a list of callees at the call sites of `whatever`, one might expect the result to be the function assigned to `foo` for the first site and that plus the function assigned to `bar` at the second site. However, providing this list and being sure it is right is rather complex. First of all, pointer analysis needs to discover that `foo` and `bar` get assigned to `whatever`, and that only `foo` reaches the first call site. Furthermore, analysis must ensure that `some_other_function` cannot modify the `whatever` variable using lexical scoping. All in all, this requires some form of inter-procedural alias analysis once again.

2 Analysis Infrastructure

Given that we believe that program analysis—particularly traditional pointer analysis—is a good way to enable IDE's of dynamic languages with the kinds of features found in IDE's for static ones, the immediate question becomes how to provide such analysis. One unavoidable feature of the world of dynamic languages is the diversity of such languages. Even restricting our attention to the world of Web scripting, there are still several popular languages: JavaScript, PHP, Python[4], Perl. Expanding our view slightly, we could consider UNIX tools such as Awk and embedded macro languages in office suites. It is not feasible to build independent program analysis infrastructures for all such languages, and yet these languages all have their individual quirks that make it impossible to simply use a single unmodified infrastructure for them all.

Our approach starts with a core program analysis infrastructure that understands basic notions of objects, method calls, control flow and the like; on top of this basic infrastructure, for each language of interest, we build an adaptation layer that has two purposes. The first purpose is to generate our analysis internal forms given the source code of the given language. The second purpose is implement semantic quirks of the given language in terms of our analysis internal forms.

In this section, we present our approach by describing the adaptation layer in Section 2.1 and then how the core solver is designed for multiple languages in Section 2.2.

2.1 Analysis Adaptation Layer

The core idea behind our adaptation layer is the notion of an Abstract Syntax Tree (AST); we define a stylized AST that encompasses the kinds of statements support by our analysis IR, such as function calls, object creation and the like. Given that AST form, our IR generation has two steps: the first is a front-end-dependent step that generates our AST form from program source, and the second is a language-influenced step that handles translation of the AST into IR. The first step takes whatever data structures the front end provides, such as a parse tree, and generates our AST from them. The second step converts the AST into IR. The second step is language influenced in that there is a common core of translation that understands things like generating Control Flow Graphs (CFGs) from the AST and converting to Static Single Assignment (SSA) form, but the common core calls out to a language-dependent portion to handle details like the exact semantics of a field access.

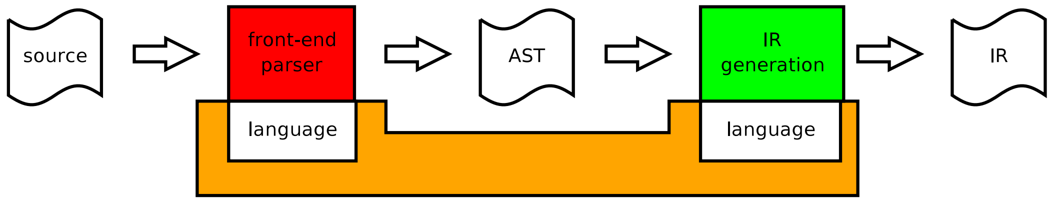

Figure 4: Overall IR Generation Mechanism

This process is illustrated in Figure 4. The red box represents a pre-existing front end for the language to be analyzed; the purpose of this box is to provide some kind of parse tree and any needed semantics such as name resolution. We have used the Mozilla Rhino JavaScript engine and also an internal IBM JavaScript engine for this purpose when analyzing JavaScript. The green box represents the language-independent portion of IR generation, such as forming CFGs from well-understood control constructs. The orange portion is what needs to be written to analyze a new language with our mechanism: the generation of our stylized ASTs from the parse trees on the left, and the handling of language specific details of IR generation on the right hand side. This setup minimizes the work to generate IR for a new language in our system: front end parsers that perform necessary type checking and the like can be reused, requiring only a layer transforming whatever parse tree it generates into our stylized AST format. Similarly, the AST translator can be reused, and what needs to be implemented is certain hooks for generating IR details for the language.

2.2 Core Analysis Engine

Our core analysis engine comprises the usual collection of tools for inter-procedural program analysis: we have an SSA-based IR, and, on top of that, a range of algorithms for call graph construction, pointer analysis and more specialized solvers. We will not present the implementation details and optimizations embodied in our engine here, since that is not the focus of this paper. Instead, we will focus on how the system is suited for extensibility and reuse across multiple languages. The basic structure of the system is as a set of analyses that operate over a common IR structure; we will discuss how the IR structure is extensible and then talk about how extended forms of IR can be integrated into some of the more common algorithms.

2.2.1 IR

The core IR represents values in SSA form and contains instructions for the usual operations, such as conditionals for control flow, binary and unary primitive operations, ϕ nodes for SSA form, a primitive allocation operation and so on. However, operations such as object field access and function call—which share a general flavor across languages but can differ significantly in details—are broken into an abstract class that captures the general operation and language-specific subclasses that are implemented as new languages are added.

2.2.2 Analysis Components

The core analysis components such as call graph construction and pointer analysis are adapted to work with the extensible IR. The common pattern is for the analysis component to be implemented using a factory-constructed visitor across the IR of a method, and to provide a visitor implementation that covers the core of the IR. This is illustrated in Figure 5. For new languages, a derived visitor class that also handles the any new IR forms for that language must be created. This seems unavoidable in that someone has to figure out what the any new language semantics should be handled in existing analyses, but it at least minimizes the effort to handle a new language by allowing reuse of the handling of the existing IR.

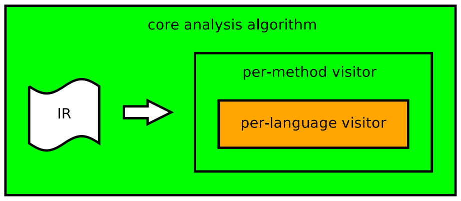

Figure 5: Common IR-Based Analysis Structure

3 Analyzing JavaScript

We illustrate our approach by describing how the system currently works for JavaScript. While we will use JavaScript as our example, we currently have significant portions of Java analyzed for source code in the same infrastructure, and we believe the same design can be used for other languages. We have plans to analyze PHP and possibly other Web scripting languages in the same manner.

JavaScript is a good representative of the many dynamic languages that are popular in the Web server community; it has a fairly simple, reference model core but on top of that is a range of features that complicate analysis. We shall not discuss exhaustively how we analyze this language, but shall focus on a representative feature—function calls—that illustrate how we can adapt this language to our simpler analysis core.

3.1 Function Calls

Function calls in JavaScript are a good illustration of how our mechanism works for analyzing idiosyncratic constructs in a language. Function calls in JavaScript come in a variety of flavors, but there two features that exercise our analysis mechanism: one is how it handles argument passing, and the second is a distinction that is perhaps unique to JavaScript: its notion of method calls versus function calls.

```
function foo(x) {
  if (x < 7)
    return arguments[1] + 5;
  else
    return this.bar;
};

var obj = {
 bar: 8,
 fooMethod: foo
};

var x = foo(3, 6);
var y = obj.fooMethod(10, 7);
```

Figure 6: Simple JavaScript Example of Issues With Calls

First, we discuss method calls versus function calls. Functions are first-class objects in JavaScript, and there is no such thing as a method per se. However, functions can be assigned as properties of objects, so there is a type of call that looks a lot like a method call in Java or C++. For such calls, the JavaScript specification defines that the containing object be used as the receiver object, and passed as the `this` argument to the function. In other calls, the `this` parameter is simply null (or rather `Undefined`). Thus, the same function instance can be called as a method in one case and a function at some other site, and the notion of the `this` object can vary for call to call for the same function. This is illustrated in Figure 6. The call `foo(3, 6)` is an ordinary function call, in which there is no `this` object, whereas the `obj.fooMethod(10,7)` call has `obj` as the `this` object. But they both call the same actual function.

In order to analyze a model such as this, we need to adapt it to the underlying analysis framework that has a more traditional notion of function and method calls. Our approach is to give every function an extra formal parameter that represents `this`, and to generate function calls that supply the appropriate extra argument. Since the rules for determining what object to use as `this` are idiosyncratic, we make the JavaScript adaptation layer hide them by handing the core analysis call statements with the appropriate `this` argument computed. That is, we generate ASTs that have any required `this` parameters as explicit arguments in call expressions. This shows up during the IR construction phases (see Figure 7) in two places: the first is that `this` parameters are added during AST generation, and the second is that special call IR instructions are generated for calls seen in the AST.

The other distinctive JavaScript feature of calls is how it handles argument passing. Functions in JavaScript do have named arguments in their declarations, but call sites can pass fewer or more arguments than are declared. And all arguments get stored in an array called `arguments` that is scoped in the function. Once again, this is illustrated in Figure 6, with the use of `arguments[1]`, which references the second argument passed at each call. Thus, we needed to create a customized call IR instruction and customized logic in the call graph builders to process them. This customized instruction and logic is added

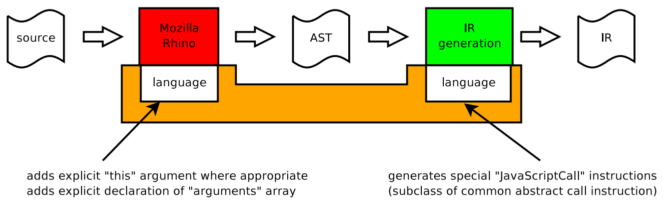
source
Mozilla Rhino
language
AST
IR generation
language
IR
adds explicit "this" argument where appropriate
adds explicit declaration of "arguments" array
generates special "JavaScriptCall" instructions
(subclass of common abstract call instruction)

Figure 7: IR Generation for Calls in JavaScript

to our analysis engine using subclasses of various analysis implementation classes. This logic shows up in the IR generation step (see figure 7) with the generation of a customized call IR instruction, and during call graph construction (see Figure 8) with special handling of calls to update the extra `arguments` object.

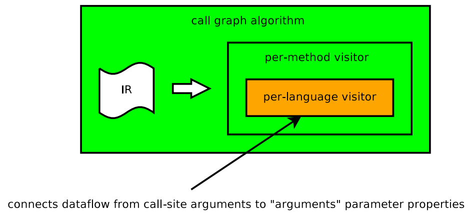
call graph algorithm
IR
per-method visitor
per-language visitor
connects dataflow from call-site arguments to "arguments" parameter properties

Figure 8: Call Graph Construction for JavaScript

4 Summary

We have argued that IDE's for dynamic languages can benefit from static analysis to provide functionality equivalent to many of the features provided by IDE's for Java. We have also described our approach to providing an implementation of a static analysis engine that can handle the range of dynamic and scripting languages out there.

References

[1] Ecma. Ecmascript language specification.

[2] S. Fink, J. Dolby, , and L. Colby. Semi-automatic J2EE transaction configuration. Technical Report RC23326, IBM T.J. Watson Research Center, March 2004.

[3] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[4] Guido van Rossum. Python reference manual. Report CS-R9525, April 1995.