Type Inference Using the Cartesian Product Algorithm on a Dynamically Typed Language

Steve Hanov
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1
+1 519 888 4567
steve.hanov@mail.com

ABSTRACT

Object-oriented languages make predicting types at compile time very difficult. In order to avoid the expensive virtual method dispatch, a variety of call graph construction algorithms have been researched. Such information would also be helpful if applied to duck-typed languages--languages for which the variables are bound to types when they are assigned a value. In these languages, it is not even obvious to the programmer if calling a method will work, and any extra information that can be gleaned by the compiler would be helpful.

In this paper, we summarize the existing work on type inference. We then implement the *Cartesian Product Algorithm*, applying it to *zscript*, a simple java-like object-oriented language with duck-types. Because CPA was designed for *self*, language without variables, we make several additions and discuss the results. The efficiency of our implementation is analyzed using automatically generated programs, and possible future enhancements are discussed.

1. INTRODUCTION

Dynamically typed, (or "duck typed") languages are useful for rapid prototyping and code reuse. In languages that have duck typing, a variable's value determines what it can do, and implies that an object is interchangeable with any other object that implements the same interface, regardless of whether the objects are related by inheritance. [3]

With no types to get in the way, generic algorithms are simple to do and code is easier to write. However, because method lookup and type checking is done at runtime, typographical errors will not manifest themselves until the code is run. It would be very useful to have these errors appear when the code is compiled. Otherwise, white box testing must take the program through not only every possible execution path, but every possible program state in order to ensure correctness.

```
class BClass {
    sub bar() {
       print("Hello, world!\n");
    }
}

class AClass {
    sub main() {
        var b = new BClass;
        b.bar();
        b.foo();
// Error: b is type BClass, which has no
// foo method.
    }
}
```

Figure 1: Example Code

Without static types, one can show that it is impossible for a compiler determine whether the program is correct (all class member lookups will succeed). However, in a large class of cases, it is possible for the compiler to say with certainty that the program will fail. For example, the code in Figure 1 should result in an error during compilation. More complicated examples are constructed if b is assigned different values in multiple paths through the program, or recursion or polymorphism is used.

In this paper, we will apply the Cartian Product Algorithm for type inference to a simple scripting language, called *zscript*, invented for this project. The scripting language allows the definition of classes, but variables do not get bound to a type until the time of assignment, and they can be re-assigned to a different type later on in the program.

2. BACKGROUND

The problem of call graph construction in a dynamically typed language is similar (but not identical) to virtual method call resolution. Much research has been put into type inference in statically-typed polymorphic object-oriented languages. If the compiler can determine, for example, that a virtual method call always resolves to the same method, then it can perform more interprocedural optimizations, such as procedure inlining or elimination of the virtual method dispatch. In *zscript,* the goal is not efficiency, but providing more information for the programmer.

2.1 Prior Work

Every type inference algorithm must, in some way, keep track of the types it encounters during the analysis. In [8], Tip and Palsberg classify type inference algorithms by the number of sets that they keep. Some algorithms keep a set of types for each expression. Other algorithms, such as RTA (Rapid Type Analysis [2]) keep one set of types for the entire program, at a cost of

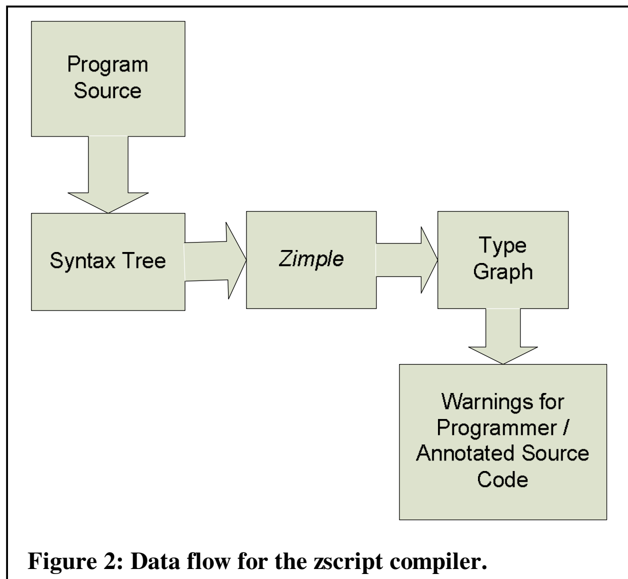

Figure 2: Data flow for the zscript compiler.

decreased accuracy. In the other extreme, *k-CFA* [7] is the most expensive, and it keeps multiple sets per expression.

The simplest algorithm is Class Hierarchy Analysis (CHA). It keeps no sets at all. When a method is called on an object, it is assumed that all possible subtypes of the statically declared type may be called. However, this method is unsuitable for a duck-typed language, where there are no statically declared types.

Another simple algorithm that would be feasible is name-based analysis. It is described in [8], but comes from multiple sources. This algorithm takes into account only the name of the method called. When method is called, it is deduced that the object must be one of the classes containing that method. This algorithm is very imprecise, because it is common for many unrelated classes to share one or more method names. However, it can be used to catch simple typos, such as "craete" instead of "create." If no known class contains the method, the program could not possibly be correct.

In the RTA algorithm [2], program analyses methods beginning at the entry point. When a method instantiates a class, the methods of that class are added to the possible methods called and they are analyzed. The algorithm is repeated until no new classes are instantiated. RTA gains much of its accuracy from having the object's static type available. Otherwise, when a method is called, no information is available other than the method name, and all methods with that name must be considered. As an example, many classes might have a create() method, so if create() is called, all of them will be added to the set of possible types. When the object's static type is known, the possible methods are restricted to subclasses of that type.

Palsberg and Schartzbach [5] specifically analyze call graphs in dynamically typed languages. They present an algorithm for creating what they call a *trace graph* on the whole program. The nodes are all of the methods in a program, and the edges are connections between a statement and the possible methods which it calls. From this graph, one can obtain a set of constraints on the types in the program (i.e. class A must have a method foo().) Plevyak and Chien [6] also present an algorithm for building call graphs, and apply it to the scheme programming language. The Cartesian Product Algorithm is based on this work.

Lattice-based methods applied to type inference are the most accurate, and seem to originate with Shiver's 1991 paper, [7]. In it, Shivers applies control flow analysis to the *scheme* programming language in order to statically gain information about the value of its variables. It is flow-sensitive, because it takes into account the order of execution of the program methods and statements. Shiver presents several variants of increasing complexity. In *0CFA*, one set of information is associated with each expression in the program. When used for type analysis, the set contains all of the possible types of the variable, and the analysis proceeds as if the variable could be any of those types. This is similar to the way CPA works. In the more advanced method, *k-CFA* recognizes that the variable's actual value can be only one element of that set, and so it analyses the program differently for each of the possible values. Each alternative analysis is called a *contour*. The increased computational cost results in greater precision.

David Grove *et al.* [3] present a generalized lattice-based model. In their model, the nodes of the lattice are call graphs. In the top node, T, all methods call all other methods. In the bottom node ⊥, no method calls any other method. The algorithm starts from T and selects nodes closer to ⊥ as it gains more information. Their paper shows that a number of algorithms for building call graphs, including both Palsberg's and Plevyak's, can be implemented using the generic lattice model, and tweaking how each statement affects the lattice and work list.

In yet another algorithm, points-to-analysis call graph construction (PTA-CG), we begin at the main method, adding it to the call graph. Upon analyzing it, we generate points-to-constraints, solve them, and then resolve further call sites. The algorithm is iterated until there are no further changes.

For this paper, however, we will implement a type inference algorithm that is based on the Cartesian Product Algorithm (CPA) [1]. It is chosen because it can be readily applied to a duck-typed language, and it is relatively simple to implement. It is also efficient because it is not iterative. It only needs to analyze methods actually called in the program, and it only needs to analyze them once for each combination of input arguments.

CPA was originally created for the *self* programming language [9]. Although *self* is object-oriented, it does not include classes or variables. CPA had to be modified to handle accessing class members. In addition, Ole Agesen's paper [1] did not provide all of the details of the implementation, including how to efficiently propagate type information throughout the nodes of the type graph.

3. ALGORITHM

In *zscript*, when the user runs the program, its source text is first transformed into a syntax tree. Then, each node of the syntax tree is traversed to build both a type tree, and an intermediate representation called *zimple*. The type tree includes classes, class methods, and class variables, and it is used during the creation of *zimple* instructions to resolve local variable names according to scoping rules.

Assign Constant	*a = 10* *a = new AClass* *a = “”*
Assignment	*a = b*
Access Update Call	*a = b.c* *a.b = c* *a = b.c(d,e,f,…)*
Control Flow	*Label:* *if (a) goto L* *goto L* *return a*
Operators	*a = !b* *a = b + c* *a = b * c* *…etc*

Figure 3: *zimple* instruction types

After each method of the program has been converted into an intermediate representation, they are processed one by one to gradually build a type graph, which will be described below. The data flow for the compiler is illustrated in Figure 2.

3.1 Intermediate Representation

To simplify analysis, the syntax tree is first converted into an intermediate representation. Each member function is packaged as a series of *zimple* instructions. The instructions are high-level and are designed to reduce the number of cases that must be handled. For example, before a constant is used, it must first be assigned to a temporary variable. As a result, the analysis needs only to consider operations on variables. Figure 3 shows the complete set of instructions that can be generated. As an example, the program statement “return a.b.foo(10).c” is converted into the these five *zimple* instructions:

```
t0 = a.b;
t1 = 10;
t2 = t0.foo(t1) ;
t3 = t2.c ;
return t3
```

3.1.1 Variable Map

For efficiency, each variable in the program is referenced by a unique positive integer, rather than by name. As the program is being converted into *zimple* code, new variables encountered in class definitions and local variables declared inside class methods are added to the variable map. Each variable record contains its name and whether it is a class member. The variable map is implemented as an array, indexed by variable ID.

3.1.2 Location Tracking

It is desirable to retain the location information of certain types of program statements, so that the source code can be annotated later with type information that is useful to the programmer. Every element in the syntax tree contains location information specifying the source file, starting and ending line number, and character positions. The location information will be transferred the certain *zimple* instructions for which it is interesting to know the types used, such as function calls. When the type graph is created later, this location information will be transferred into the corresponding type nodes. Thus, when the analysis is complete, the source code can be annotated with comments at each call site, indicating the inferred types of variables, and references to the points in the program where those variables obtained their values.

After the building of the type graph is complete, *zscript* can optionally print each line of the input to the display. Before each line is printed, the nodes of the type graph are searched for corresponding location information, and if there is a match, the information contained in the node is displayed as comments in the program source.

3.2 Type Graph

After each method has been converted into its intermediate representation, *zscript* gradually builds a type graph by each method called by the program.

The CPA is non-iterative. Only the methods that could potentially be called are processed, and (except for templates) they are only processed one time only.

In our implementation of the algorithm, we use a work list. First, the constructor of the class containing the main method, and the main method itself is added to the work list. Then, while the work list is not empty, the methods of the work list are processed. During the processing of a method, more methods may be added to the work list. The algorithm terminates when no more methods remain to be analyzed.

3.2.1 Building the Type Graph

As each *zimple* statement is analyzed, the analyzer makes incremental modifications to the type graph, depending on the nature of the instruction. Certain instructions are ignored: Because the CPA is flow-insensitive, we skip over branch instructions and instructions that evaluate mathematical operators.

The type graph consists of nodes and directed edges. Each node of the graph corresponds to a variable in the program, a function call site, a function call temple, or a member variable access or update. Node A is connected to node B if A can obtain its type from B. Nodes that correspond to variables may additionally contain a set of types. Thus, to obtain the type of any variable in the program, one can perform a depth first search started at that variable’s node. The variable’s type may be any of the types encountered during the search.

The nodes of the graph are distinguished into several classes:

- ***VariableNodes*** correspond to a local variable or a class variable in the program. Although the variables are identified by their position in the variable map (See section 3.1.1), for debugging purposes they are assigned a unique, human readable name. Local variables are prefixed by their complete function name, such as MyClass.mysub.(NUMBER, STRING).mylocal.

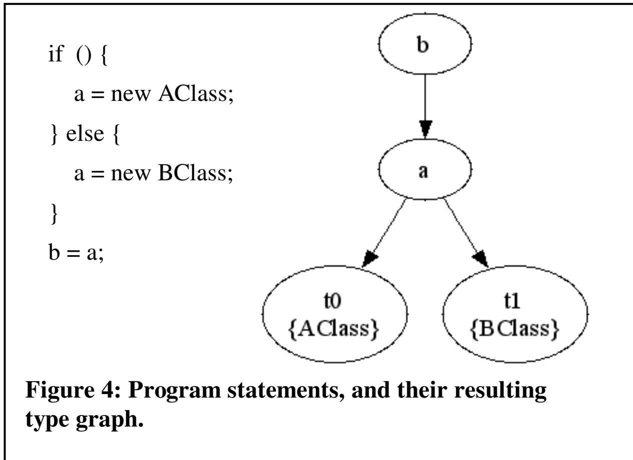

Figure 4: Program statements, and their resulting type graph.

- ***FunctionCall*** nodes correspond to a call site. They provide a way for the variable nodes to indirectly link to the result of a function call, when the exact function template is not yet known. *FunctionCall* nodes connect *VariableNodes* to *FunctionTemplate* nodes.
- ***FunctionTemplate*** nodes distinguish functions called with different arguments. Whenever a function is called with a new type of argument, a function template is created for it. A function template indirectly connects ***FunctionCall*** nodes to whatever local variable is returned from the function.
- ***AccessNodes*** and ***UpdateNodes*** are similar to FunctionCall nodes and are used to provide access to class member variables. They are further described in section 3.4.

Whenever an instruction that references a new variable is encountered, a node is created for that variable. If the variable has a type assigned to it, the variable is added to the node's type set. If one variable is assigned to another, the corresponding nodes are connected with an edge. See Figure 4 for an example.

3.3 Function Calls

3.3.1 Templates

The CPA results from the observation that the return value of function calls may depend on the input parameters. An example given in [1] is the max function. *max(float,float)* will return a float value, while *max(int,int)* will return an integer value, and *max(int, float)* may return either. CPA solves this dilemma by separately analyzing each invocation of the method. Each invocation is referred to as a *template*. (Other papers may use the more general term *contour*). Within the function template, the type of each argument is known.

Because each analysis of a function template could possibly yield different results, they all need their own copies of local variables. When a new function template is created, new copies of the nodes for all of the parameters and local variables are created, and space is created for them in the variable map (section 3.1.1).

Whenever a function call instruction is encountered, and the types of all arguments are known, the CPA computes the Cartesian product of the types of the argument, and connects destination variable to all of the resulting method templates. For example, if *a = max(b,c)*, and *b* can be an integer or a float, and *c* can be an integer, then *a* is connected to both *max(INT,INT)* and *max(INT,FLOAT)*.

3.3.2 Graph Representation

An astute reader will see that this procedure is not possible if the types of any of the variables are not known. The CPA handles this by creating two nodes. The destination variable gets its type from the *FunctionCall* node, which references the object and all of the arguments of the function call. There is a *FunctionCall* node for every call site. When the types of the object and arguments become known, then the *FunctionCall* node is connected to the appropriate *FunctionTemplate* nodes. See Figure 6 for an example.

3.4 Dot Operator

CPA does not consider access to member variables, because the

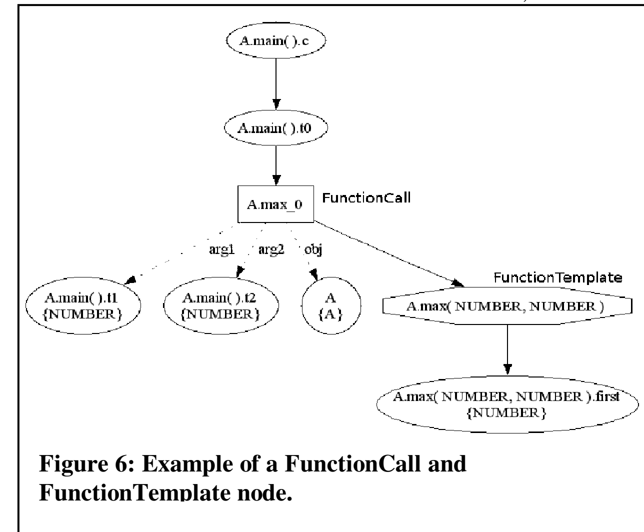

Figure 6: Example of a FunctionCall and FunctionTemplate node.

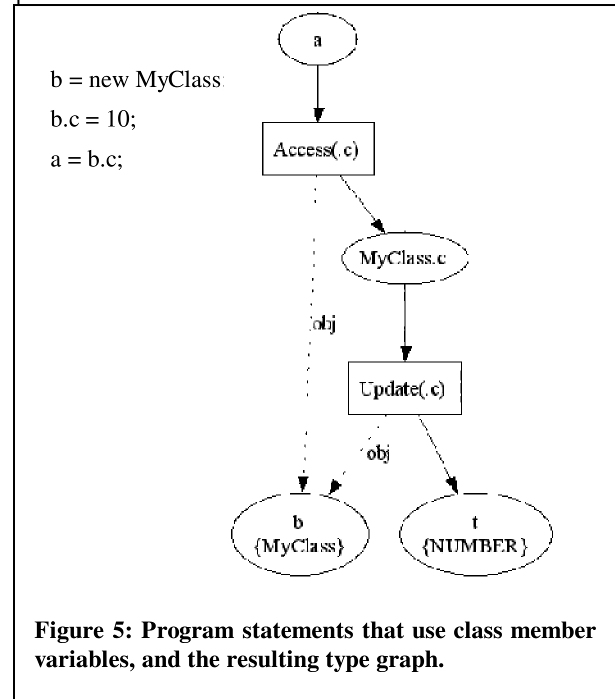

Figure 5: Program statements that use class member variables, and the resulting type graph.

self programming language does not use variables. In more conventional languages, procedures may use class member variables to communicate with each other. In order to apply CPA and give a complete the interprocedural analysis, we must first adapt it to work with member variables.

Class members may be used in two ways. They can be accessed, so that their value is written to a local variable, and they can be updated so that they take on the value of a local variable. To make these two operations work with CPA, we convert class member variable accesses and updates into function calls. We create special nodes in the type graph with references to the object variable. In addition, the nodes have stored within them the name of the member that is being accessed or updated. These Access/Update nodes may then be treated in the CPA as normal function calls. Instead of pointing to a function template, they connect to the node associated with the class member. This is illustrated in Figure 5.

4. EFFICIENCY

Zscript is a new language, so there is no existing body of benchmarks. In order to analyze the scalability our implementation, we have two choices: translate existing benchmarks into *zscript* source code, or automatically generate some benchmark programs. At the time of this writing, *Zscript* does not have the capability of running standard benchmark algorithms. Thus, we choose the second option.

4.1 Generating sample sets

In order to test the efficiency of our implementation of CPA, we designed an algorithm for automatically generating programs of a given size. The algorithm takes as input two parameters: *MaxLevels* and *MaxChildren*, and it creates as output a valid *zscript* program. The resulting program consists of two types of classes: normal classes, and leaf classes. Normal classes have MaxChildren member variables, each of which is a reference to another class. Leaf classes do not have any member variables. All classes have a single function, foo(). For normal classes, this foo() method picks a random member variable and returns the result of calling foo() on that object. For leaf classes, the foo() method returns a new instance of the class in which it was defined.

```
class Obj2 {
    sub foo() {
        return new Obj2;
    }
}
class Obj1 {
    var obj2 = new Obj2
    sub foo() {
        return obj2.foo();
    }
}
class Obj0 {
    var obj1 = new Obj1
    sub main() {
        return obj1.foo();
    }
}
```

Figure 7: Example of generated program to test the efficiency of the algorithm. In this example, *MaxChildren=1* and *MaxLevels=2*.

The resulting program forms a tree structure, consisting of $N=MaxChildren^{MaxLevels}+1$ classes. An example with *MaxChildren=1* and *MaxLevels=2* is given in Figure 7. By changing the values of *MaxChildren* and *MaxLevels*, we are able to study how CPA reacts to both a large number of classes and a long chain of polymorphic calls.

4.2 Results

We tested the algorithm by running it ten times, using a variety of values for *MaxChildren* and *MaxLevels*. *Zscript* was compiled using the Microsoft C++ Compiler version 8, with optimizations disabled. The tests were performed on a 2.1 GHz Pentium processor, with disk swapping turned off. To minimize variance, tests were performed 10 times and the most frequently occurring time was taken. Only the time spent constructing the type graph was included, after the program had already been converted into *zimple* instructions.

Tests were performed with extreme values for the values of *MaxChildren and MaxLevels.* Figure 8 illustrates the case where *MaxChildren = 1* and *MaxLevels* is increased by factors of two. In the generated program, each object has only one data member, and calls the function *foo()* on it. The resulting program therefore has a polymorphic call chain of length equal to *MaxLevels*.

The results are not that good. With a polymorphic call sequence of length 512, the time is more than 11 s, which it far too long to wait. It is worth noting, however, that such a program is highly unlikely in a non-functional programming language.

MaxLevels	*Total Classes*	*Time (ms)*
64	*65*	*47*
128	*129*	*250*
256	*257*	*1563*
512	*513*	*11543*

Figure 8: MaxChildren=1

Figure 9 shows the results with *MaxChildren=2*. Because of the tree structure of the automatically generated program, the number of classes is greatly expanded. However, the CPA only analyzes methods actually called. Since only one (random) path through the tree is actually taken by the generated program, theoretically the algorithm has to analyze only a small number of methods compared to the actual number of methods that exist. In fact, only *MaxLevels* functions need to be analyzed.

However, because the main object creates all of its children, which in turn causes all objects to be created, the the bulk of the time of the analysis is spent analyzing the constructors, and creating nodes for the member variables that will never be accessed. With 8193 classes, the analysis takes over 21 seconds.

MaxLevels	*Total Classes*	*Time (ms)*
7	*129*	*16*

8	*257*	*47*
9	*513*	*125*
10	*1025*	*469*
11	*2049*	*3875*
12	*8193*	*21812*

Figure 9: MaxChildren=2

In Figure 10, the above theory is tested. *MaxLevels* is fixed at 1, and the resulting program has a single class containing references to thousands of leaf classes. It seems that creating the nodes for member variables is taking much of the time of the analysis.

MaxChildren	*Total Classes*	*Time (ms)*
1024	*1025*	*78*
2048	*2049*	*282*
4096	*4097*	*1047*

Figure 10: MaxLevels=1

4.3 Discussion

The algorithm was not implemented with efficiency in mind, so the above results are unsurprising. The algorithm is exceptionally good at resolving long chains of polymorphic calls, but in our implementation, it spends too much time creating nodes. The implementation could be optimized use a more efficient data structure for the graph.

One thing that stands out in the algorithm is the amount of work that must be done when a new type is assigned to a variable. The graph must be searched from that variable to see if it is used as the object in any *Access* or *Update* nodes, or if the variable is used in any function call arguments. If it is, then the necessary connections must be made, and possibly more function templates will be added to the worklist. However, backtracking through the graph to find how the variable is used is probably an expensive operation that is hard to predict. It would be interesting to analyze real programs to see if the set of variables that use the type is generally small or large.

5. ACCURACY

The CPA algorithm will find, for each variable, the set of types that are assigned to it. In the simple language of *zscript*, it will never fail to find a type. So the resulting type set is always greater than or equal to the given types.

If the resulting types are used for error checking, however, the results are not accurate enough. It is not acceptable for the compiler to claim that a variable is a certain type, and fail to run the program, when in reality the variable is assigned that type for a very short time period. Consider the following code:

```
// AClass contains foo method
var a = new AClass;
a.foo();
// BClass does not contain the foo method
a = new BClass;
a.bar();
```

The CPA algorithm will claim that when foo() is called, a can be of types {AClass, BClass}. However, at that point it can be of only one class and that is AClass, in which case calling foo() is acceptable.

Much of the problem stems from the fact that CPA is a flow-insensitive analysis. That is, unlike *k-CFA*, it does not consider the order in which that statements of the program are executed. It is possible to mitigate the problem, within a method, by converting the program to SSA form prior to the analysis:

```
var a0 = new AClass;
a0.foo();
a1 = new BClass;
a1.bar();
```

The two different program points are then operating on different variables. However, that solves the problem only locally. It is unclear how to add subscripts to the member variables of classes, that are updated in multiple methods.

Another way of solving the problem is to modify the semantics of the language, and only allow a variable to be bound to one type. For example, once a certain type of class is assigned to the variable, the VM could cause an exception of any other types are ever assigned to the variable. However, such a solution is drastic and undesirable, because it makes the language less flexible.

5.1 Dynamic Class Loading

One advantage of the CPA algorithm is that it analysis each method (or method template) only once. If dynamic class loading were supported by the language, then only a minimal amount of work would be necessary upon loading a new class, provided the type graph were saved. Only the new methods loaded in would have to be analyzed, plus any new templates that result.

5.2 Flexible New Operator

In our implementation, a method cannot be analyzed unless the type of the object, and all arguments are known. If any of these have an empty type set, the method is skipped and never analyzed, because a template cannot be created.

If *zscript* were expanded, then certain features could make the CPA algorithm break down. For example, it would be very simple to allow the new operator to take a string as its argument, creating an object that is impossible to predict at compile time.

5.3 Source code Annotation

Because the CPA algorithm is a flow-insensitive algorithm it is not very suitable for error checking. For each variable, does find a complete set of types that may be assigned to that variable. However, at every program point the variable may only be one of those types, and when a class method is called using the variable of the object, it need not be in every one of those types.

Although CPA finds a superset of the values that we wanted to find, it is still good enough for source code annotation. By carefully keeping track of the source of each type, and the locations in the source code, each variable's declaration in the program text can be annotated with the types assigned to it, and the line numbers where those types are assigned. The programmer than then look for any surprises. With a duck-typed

language, any extra information that the compiler can give would be very helpful.

6. FUTURE ENHANCEMENTS

The implementation of the Cartesian product algorithm for zscript can be made more efficient in several ways. One of the more expensive steps occurs when a new type is assigned to a variable. At that moment, a depth first search is performed to see if the value of that variable is ever used in a function call, either as the object or one of the arguments. For each such call sites, the Cartesian product of the types of the arguments are recalculated, and if there are any more connections to be made, the necessary edges are added to the type graph. This could reveal more types for other variables, so the procedure is repeated recursively. Backtracking through the graph to find all users of a type is expensive, and the algorithm could potentially be improved by caching users of a node.

During implementation, we made the decision to avoid caching any values in any nodes, to avoid the problem of updating cached values. So to find the type of a variable, the complete path from the variable to all of the sources of its type must searched. It is possible that less work could be done by copying the type set from the source nodes to higher points in the graph, such as the join points.

7. CONCLUSIONS

We have presented an implementation of the Cartesian product algorithm, applied to a new dynamically typed language. It was found that the original CPA had to be modified to take into account class member variables, and the dot operator.

CPA is very good at inferring the resulting type of very long chains of polymorphic function calls. During implementation the goal was for correctness over efficiency, and the analysis revealed that the implementation is slow for enterprise-size projects. However, one must keep in mind that the target language is a scripting language and so are unlikely to have thousands of objects to analyze.

CPA is relatively easy to understand and to implement. It is efficient enough for small programming sets, and it never underreports the types of the program. Because it analyzes only methods actually called, it is well suited for object-oriented languages where the program only uses a small portion of the methods available in a class library.

CPU does a good job at inferring types, but because it is flow-insensitive, it is unsuitable for error checking programs at compile time. However, implementing CPA is an ideal building block for a comprehensive type checking system in a dynamically typed language, especially when combined with more accurate methods, such as *k-CFA*.

8. REFERENCES

[1] Agesen, O. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (August 07 - 11, 1995). W. G. Olthoff, Ed. Lecture Notes In Computer Science, vol. 952. Springer-Verlag, London, 2-26.

[2] Bacon, D. F. and Sweeney, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, California, United States, October 06 - 10, 1996). OOPSLA '96. ACM Press, New York, NY, 324-341.

[3] “Duck_typing.” Wikipedia. http://www.wikipedia.org

[4] Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. *SIGPLAN Not.* 32, 10 (Oct. 1997), 108-124.

[5] Palsberg, J. and Schwartzbach, M. I. 1991. Object-oriented type inference. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, United States, October 06 - 11, 1991). A. Paepcke, Ed. OOPSLA '91. ACM Press, New York, NY, 146-161.

[6] Plevyak, J. and Chien, A. A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications* (Portland, Oregon, United States, October 23 - 28, 1994). R. L. Wexelblat, Ed. OOPSLA '94. ACM Press, New York, NY, 324-340.

[7] Shivers, O. *Control Flow Analysis of Higher Order Languages.* PhD thesis, CMU, May 1991. CMU-CS-91-145.

[8] Tip, F. and Palsberg, J. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, United States). OOPSLA '00. ACM Press, New York, NY, 281-293.

[9] Ungar, D. and Smith, R. B. 1987. Self: The power of simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Orlando, Florida, United States, October 04 - 08, 1987). N. Meyrowitz, Ed. OOPSLA '87. ACM Press, New York, NY, 227-242.