Annotations for (more) Precise Points-to Analysis

Mike Barnett Manuel Fähndrich
Francesco Logozzo
Microsoft Research
{mbarnett, maf, logozzo}@microsoft.com

Diego Garbervetsky
Departamento de Computación, FCEyN, UBA
diegog@dc.uba.ar

Abstract

We extend an existing points-to analysis for Java in two ways. First, we fully support .NET which has structs and parameter passing by reference. Second, we increase the precision for calls to *non-analyzable* methods. A method is non-analyzable when its code is not available either because it is abstract (an interface method or an abstract class method), it is virtual and the callee cannot be statically resolved, or because it is implemented in native code (as opposed to managed bytecode). For such methods, we introduce extensions that model potentially affected heap locations. We also propose an annotation language that permits a modular analysis without losing too much precision. Our annotation language allows concise specification of points-to and read/write effects. Our analysis infers points-to and read/effect information from available code and also checks code against its annotation, when the latter is provided.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Object-oriented programming, static analysis, points-to analysis, effects analysis

Keywords object-oriented, points-to analysis

1. Introduction

Object-oriented languages, as C# or Java, strongly rely on the manipulation (read/write) of dynamically allocated objects. As a consequence, static analysis tools for these languages need to compute some heap abstraction. Here, we focus our attention on a static analysis for determining the side-effects of statements and methods.

Side effect information can be used for program analysis, specification, verification and optimization. If it is known that a method m has no side-effects, then during the analysis of a caller, m can be handled in a purely functional way. Furthermore, m can be used in assertions and specifications [13, 5]. Side effect-free methods enable several optimizations such as caching the computed results and automatic parallelization.

Analysis of side-effects in mainstream OO languages is not simple as (i) different variables or fields may refer to the same memory location (aliasing); (ii) the relationship between objects can be very complex (shape); (iii) the number of objects can be unbounded (scalability); and (iv) it can be difficult or impossible to statically determine the control flow because of dynamic binding or because not all the code is not available at analysis time, e.g., when analyzing a class library or programs that use native code.

We extend an existing points-to and effect analysis presented by Salcianu et al. [22] to infer read and write effects for code targetting the .NET Common Language Runtime (CLR) [11]. The CLR is the common infrastructure for languages such as C#, Visual Basic, Managed C++, etc. Unlike Java, the CLR adds support for struct types and parameter passing by reference via managed pointers, i.e., garbage collector controlled pointers. For each method in the application we compute a summary describing a read/write effects and a points-to graph that approximates the state of the heap at the method's exit point.

The more important extension is the inclusion of additional support for *non-analyzable* calls. We can analyze programs that have calls to non-statically resolvable calls such as interface calls, virtual calls, and native calls while being less pessimistic than Salcianu's analysis. We define a concise yet expressive specification language to describe points-to and read/write effects for a method. The method annotations are used (i) as summaries, to analyze code involving calls to non-analyzable methods; (ii) to enable modular analysis, i.e., when analyzing a method n that invokes a method m, we (a) use the annotation $\mathcal{A}(m)$ in the analysis of the body of n and (b) we check m against its specification $\mathcal{A}(m)$; (iii) as documentation and contracts to impose restrictions on eventual implementations [18]. This allows our analysis to work even without computing a precise call graph.

In this work we apply our analysis primarily for checking *method purity* but it can be used for any other analysis that requires aliasing information and/or conservative read/write effect information. Purity is informally understood to mean that a method has no effect on the state. Formally, however, there are different levels of purity [6]. Our analysis computes weak purity, i.e., it infers weak purity and it checks whether a method annotated as being weakly pure lives up to its contract. A *weakly pure* method does not mutate any object that was allocated prior to the beginning of the method's execution. Because a weakly-pure method can return newly allocated objects and since object equality can be observed by clients, there may be further restrictions on weakly-pure methods in order to use them in specifications [10].

The main contributions of the paper are:

- An interprocedural read/write effect inference technique, built on the top of the points-to analysis, for the .NET memory model that relaxes the *closed world* assumption.
- A new set of annotations for representing points-to and effect information in a modular fashion. The annotations are considered valid for interprocedural analysis when the methods are called, and verified when the implementations of the methods are analyzed.
- An implementation integrated into the Spec# compiler [23] to infer and verify method purity and for checking the admissibility of specifications in the Boogie methodology [5].

1.1 The Problem

Consider the following simple, but realistic example. Figure 1 contains a method written by a programmer to copy a list of inte-

```
List<int> Copy(IEnumerable<int> src)
{
  List<int> l = new List<int>();
  foreach (int x in src)
    l.Add(x);
  return l;
}
```

Figure 1. A simple use of an iterator in C#.

```
List<int> Copy(IEnumerable<int> src)
{
  List<int> l = new List<int>();
  IEnumerator<int> iter =
    src.GetEnumerator();
  while (iter.MoveNext()){
    int x = iter.get_Current();
    l.Add(x);
  }
  return l;
}
```

Figure 2. "Desugared" version of the iterator example.

gers. In C#, the *foreach* is syntactic "sugar" which the compiler expands ("desugars") into the code shown in Figure 2. (Programmers are also able to directly write the de-sugared version.) The desugared version shows that there is one method call from the interface $IEnumerable\langle T\rangle$ and two from the interface $IEnumerator\langle T\rangle$. In addition, the constructor for the type $List\langle T\rangle$ is called, as is its Add method.

A points-to analysis produces the set of memory locations that are read and written by $Copy$. That information can then be used to determine if $Copy$ is (weakly) pure. It clearly mutates the list that it creates and returns, but that list is created after entry into the method and the original collection from which the integers are drawn is unchanged. Thus, we desire an analysis that is precise enough to recognize its purity.

Salcianu's analysis would not be able to analyze the calls to the interface methods. It would make the conservative approximation that the parameter src could escape to any location in memory and that the method has a (potential) write effect on all accessible locations, such as all static variables. This precludes $Copy$ from being pure and, perhaps more importantly, pollutes the analysis of any method that calls it because those effects then become the effects of the caller.

We have created a specification language for concisely describing the points-to graph and read/write effects of a method. The design of such a language is subject to common engineering tradeoffs: it should be precise enough to enable the recognition of common programming idioms while at the same time be concise enough for programmers to use in everyday practice.

We add annotations written in the language to method signatures. At call sites, we trust the annotation of the called method; annotations are then verified when analyzing a method implementation. Annotations are inherited: they must be respected in every subtype by overriding methods. We use the set of annotations to model non-analyzable calls with better precision than previously possible while still computing a conservative points-to graph and read and write effects of the callee. The annotations describe an approximation of the read and write effects of the method.

1.2 Paper structure

First, we review the essential ideas from Salcianu's analysis in Section 2 and present our extensions to deal with .NET memory model and non-analyzable calls. Section 3 presents our annotations and the extensions to Salcianu's analysis needed to process the points-to graphs they represent. Our preliminary experimental results appear in Section 4. Some related work is reviewed in Section 5 and our conclusions are presented in Section 6.

2. Salcianu's Analysis

Salcianu et al. [22] created an analysis for Java programs that performs an intra-procedural analysis of each method to obtain a method summary that models the result of the analysis at the end of the method's execution. We briefly review their analysis.

Their analysis relies on having a precise precomputed call graph for the entire application. Methods are traversed in a bottom up fashion, using already computed method summaries at each call site. To deal with recursion, a fixpoint computation operates over every strongly-connected component (i.e., group of mutually recursive methods). When a method invokes another method, the current state of the caller and the method summary for the callee are unified to represent the caller's state after the call.

The intra-procedural analysis is a forward analysis that computes a points-to graph (PTG) which over-approximates the heap accesses made by a method m during all its possible executions. Given a method m and a program location pc, a points-to graph P_m^{pc} is a triple $\langle I, O, L\rangle$, where I is the set of inside edges, O the set of outside edges and L the mapping from locals to nodes [1]. The nodes of the graph represent heap objects; there are basically three different types of nodes. *Inside nodes* represent objects created by m, while *parameter nodes* represent the value of an object passed as an argument to m. *Load nodes* are used as placeholders for unknown objects or addresses. A load node represents elements read from outside m.

Relations between objects are represented using two kind of edges: *inside* edges model references created inside the body of m and *outside* edges model heap references read from objects reachable from outside m, e.g., through parameters or static fields.

When the statement at the program point pc is a method call, op, the analysis uses a summary of the callee P_{callee}—a PTG representing the callee effect on the heap—and computes an inter-procedural mapping $\mu_m^{pc} :: \mathsf{Node} \mapsto \mathcal{P}(\mathsf{Node})$. It relates every node $n \in nodes(\mathsf{P}_{callee})$ in the callee to a set of existing or fresh nodes in the caller $(nodes(\mathsf{P}_m^{pc}) \cup nodes(\mathsf{P}_{op}))$ and is used to bind the callee's nodes to the caller's by relating formals with actual parameters and also to try to match callee's outside egdes (reads) with caller's inside egdes (writes).

For each program point within m, the analysis also records the locations that are written to the heap. The summary of a method represents the abstract state at the method's exit point in term of its parameters. It contains all reachable nodes from the (original) parameter nodes.

2.1 Extensions for the .NET Memory Model

We extend this analysis to support features of the .NET platform not present in Java: parameter passing by reference and struct types. Struct types have *value* semantics; they encompass both the primitive types like integers and booleans as well as user-defined record types. To accommodate both references and structs, we add

[1] The set of nodes is implicitly described by the two sets of edges and the local variables map. Salcianu's analysis also has one more element, E, the escaping node set. Instead, we represent an escaping node by connecting it to a special node that represent the global scope.

a new level of dereference using *address nodes*. In this model, every variable or field is represented by an address node. In the case of objects (or primitive types) the address node then refers to the object itself. A struct value is represented directly by its address. To access an object we first get a reference to an address node and then follow that to the value. In the case of structs we directly consider the address as the starting offset of the struct. Thus, an address node for an object has outgoing edges labeled with the "contents-of" symbol "*", while an address node for a struct value has one outgoing edge for each field of the struct: the labels are the field names.

This distinction is used in the assignment of objects and structs. For objects, we just copy the value pointed to by the address node, and for structs we also copy all the values pointed to by its fields. Figure 3 shows the representation of object and struct values and how the assignment of struct values is done. Address nodes are depicted as ovals, values as boxes.

In [4] we formally present the concrete and abstract semantics of the extended model. Basically we support the statements that operate on managed pointers. For instance the statement that loads an address `a = &b` assigns to `a` the address of `b`. If the type of `b` is a struct type `a` will contain a reference to it. Thus, `a` can be used as if it were an object. The pair of statements indirect load, `a = *b`, and indirect store, `*a = b`, allows indirect access to values and are typically used to implement parameter passing by reference. We also keep track of read effects by registering every field reference (load operation).

Figure 4 shows a simple method and three points-to graphs at different control points in the method. All of the addresses in the figure refer to objects. One node models all globally accessible objects. The graph on the left shows the points-to graph as it exists at the entry point of the method. The middle graph shows the effect of executing the body of the method: the points-to graph is shown at the exit point of the method. Finally, the right graph is the summary points-to graph for the method. It represents the method's behavior from a caller's point of view. Notice that the initial value of the parameter a has been restored since a caller would not be able to detect that it is re-assigned within the method. The summary for the method is a triple made up of a points-to graph that approximates the state of the heap, a write set $\mathcal{W}$, and a read set $\mathcal{R}$.

2.2 Extensions for Non-analyzable Methods

Salcianu's analysis computes a conservative approximation of the heap accesses and write effects made by a method. A call to a non-analyzable method causes all arguments to escape the caller and also to cause a write effect on a global location [22].

For a more precise model of non-analyzable calls, we generate summary nodes for non-analyzable methods. A load node (in particular, a parameter node) is a placeholder for unknown objects that may be resolved in the caller's context. In the case of analyzable calls, at binding time the analysis tries to match every load node with nodes in the caller. A match is produced when there is a path starting from a callee parameter that unifies with a path in the caller. That means that a read or write made on a callee's load node corresponds to a read or write in the caller. As reads and writes in the callee are represented by edges in the points-to graph, those edges must be translated to the caller.

Non-analyzable calls may have an effect on every node reachable from the parameters. That means that, unlike analyzable calls, some effects might not be translated directly to the caller points-to graph as it may not have enough context information to do the binding. For instance, a non-analyzable callee $m2$ may modify $p1.f1.f2.f3$ to point to another parameter $p2$ and a caller m that performs the method call $m2(a1, a2)$ may have points-to information only about $a1.f1$. As we don't know "a priori" the effect of $m2$ it would be unsound to consider only an effect over $a1.f1$ in the caller. We need some mechanism to update $a1$ when more information becomes available (e.g., when binding m with its caller).

2.2.1 Omega Nodes

We introduce a new kind of node, an ω node, to model the set of reachable nodes from that node. At binding time, instead of mapping a load (or parameter) node with the corresponding node in the caller, ω nodes are mapped to every node reachable from the corresponding starting node in the caller. For instance, an ω node for a parameter in the callee will be mapped to every node reachable from the corresponding caller argument.

Figure 5 shows an example of how ω nodes are mapped to caller nodes during the inter-procedural binding. Suppose that somehow we know the non-analyzable method call creates a reference from some object reachable from $p1$ to some object reachable from $p2$. Since we don't know which fields are used on the access path, we use a new edge label, $?$, that represents any field. At binding time we know that from $a1$ we can reach $IN1$ and $IN2$. Thus, we must add a reference from both nodes to the nodes reachable from $a2$.

We want to distinguish between a node being merely reachable from it being writable (e.g., an iterator may access a collection for reading but not for writing). For this purpose, we introduce a variant of ω nodes: ωC nodes. The C stands for *confined*, a concept borrowed from the Spec# ownership system [2]. These nodes have the same meaning as ω nodes for binding a callee to a caller, but they represent only nodes reachable from the caller through fields it *owns*. Ownership is specified on the class definition: a field f marked as being an *owning* field in class T means that an object o of type T owns the object pointed to by its f field, $o.f$ (if any).

To model potential read or writes we use $?$ edges to mean that the method may generate a reference using an unknown field for any object reachable from the object(s) represented by the source node to the object(s) represented by the target node. As we want a conservative approximation of the callee's effect, we only generally introduce inside edges in non-analyzable methods because they do not disappear when bound with the caller's edges. We use another wildcard edge label $\$$, that includes only a subset of the labels denoted by $?$. $\$$ denotes only non-owned fields and allows distinguishing between references to objects that can be written by a method, from references that can only be reached for reading (see Section 3 in particular the $WriteConfined$ attribute). This is the distinction that allows the use of impure methods while retaining guarantees that some objects are not written. For the worst case scenario we connect every parameter ω node of the non-analyzable method to other parameter nodes and to themselves using edges labeled as $?$ to indicate potential references created between objects reachable from the parameters. Section 3 presents our annotation language that helps eliminate some of these edges.

2.2.2 Interprocedural binding

To deal with the new nodes and edge labels, we adapt the inter-procedural mapping μ. Recall that μ is a mapping from nodes in the callee to nodes in the callee and the caller. Thus, for every ω node $n_{pc}^{L\ \omega}$ we compute the closure of $\mu(n_{pc}^{L\ \omega})$ by adding the set of reachable nodes from $\mu(n_{pc}^{L\ \omega})$ to itself.

When computing the set of reachable nodes matching an ωC node we consider only paths that pass through owned fields[2] and $?$ edges. Note that we reject paths that contain $\$$ edges.

Finally, we convert any load nodes, n_{pc}^{L}, contained in the set $\mu(n_{pc}^{L\ \omega})$ to ω nodes. This is because these nodes could be resolved when more context is available, at which point we still need to apply the effect of the non-analyzable call to those nodes. For

[2] We mean "owned fields" as defined in the Boogie methodology [2].

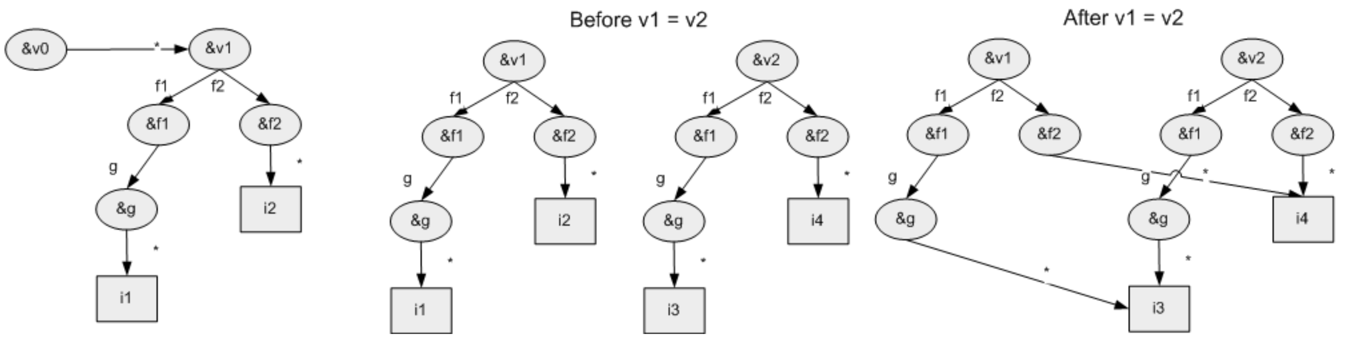

Figure 3. Modeling objects and structs. On the left v_0 is the address of v_1, which is a value of a struct type with two fields $f1$ and $f2$. (v_0 can be thought of as an object, e.g., if the struct is passed to a method that takes an object as a parameter then v_1 would be a *boxed* value.) The type of $f1$ is also a struct type with one field g which is of an object type. The type of $f2$ is an object type. The center and right figures show an assignment of two variables of struct type.

instance in Figure 5, before the binding all nodes reachable from $a1$ are inside nodes. Those nodes do not change at binding time as they were created by the caller itself and are not place holders for unknown objects. Thus, no more context is necessary to solve the binding between $a1$ and $p1$. However, $a2$ can reach the load node $L4$ meaning that more context might be necessary to resolve nodes reachable from $a2$. That is why we convert $L4$ to an ω node. Full details on the modified computation for the inter-procedural mapping μ is in [4].

We also modify the operation that models field dereference to support the $?$ and $\$$ edges. It considers those edges as "wild cards" allowing every field dereference to follow those edges.

3. Annotations

Table 1 summarizes our annotation language. The annotations provide concise information about points-to and effect information and allows us to mitigate the effect of non-analyzable calls. Annotating a method as pure is the same as marking each parameter as not being writable (unless it is an out parameter). A method annotated as being write-confined is shorthand for marking every parameter as write-confined. Obviously not all combinations of the attributes are allowed. For example, it would be contradictory to label a method as being both pure and as writing globals.

The full details for mapping the attributes into points-to and write effect information are found in [4]. Basically their impact is to a) remove $?$ edges, b) replace ω nodes by inside nodes, and c) avoid registering write effects over parameters or the global scope.

We explain the effect of the annotations using some of the methods in our running example. Figure 7 presents the full list of annotations. The $GetEnumerator$ method returns an object that is modified later on in $Copy$. Notice that the loop would never terminate unless $iter.MoveNext$ returns false at some point. So either the loop never executes or else some state somewhere must change so that a different value can be returned. If the state change involves global objects, then $Copy$ is not pure so let us assume that the change is to the object $iter$ itself. As long as that object was allocated by $GetEnumerator$, changes to it would not violate the weak purity of $Copy$. We expect $GetEnumerator$ to return a fresh object: the iterator. At the same time, it is likely that the returned iterator has a reference to the collection. We need a way to distinguish the write effects in $MoveNext$ so that we do not conclude that it modifies the collection.

Figure 6 shows the points-to graph for $GetEnumerator$. It corresponds to the following annotations.

- The return value is annotated as $Fresh$. This generates the inside node for the return value instead of an ω node.
- The receiver ($this$ variable) is annotated as $Escapes$ which means that the points-to graph must introduce edges from the nodes reachable from outside (in this case the return value) to the receiver. Note that we do not annotate it as $Capture$. This is why the edge between the return value and the collection is labeled as $\$$ which means that the receiver is reachable from outside but only for reading. A $Capture$ annotation would generate a $?$ edge. There are no edges starting from the ω node pointed by $\&this$ because of the default annotation for the receiver as $Write(false)$.
- The method is annotated as not accessing globals. This means that there is no global node (and so no write or read effects on the global state).

We believe these are reasonable constraints on the behavior of $GetEnumerator$. The points-to graph for $MoveNext$ is also shown in Figure 6. It corresponds to these annotations:

- The method is annotated as $WriteConfined$, which means that it can only mutate objects it owns. This is represented using an ωC node for the receiver. Note how this is implemented. The parameter node has two edges. The edge labeled as $?$ which leads back to the reciever means that the method can perform any write to nodes in its ownership cone. The other edge labeled as $\$$ leads to a separate ω node. That means that objects reachable using not-owned fields can be read but not modified. Thus, edges labeled as $ do not need to be considered when computing write effects for the method.

```
class List<T> {
  [GlobalAccess(false)]
  public List<T>();
  [GlobalAccess(false)]
  public void Add(T t);
  ...
}
interface IEnumerable<T>{
  [return: Fresh]
  [Escapes(true)] // receiver spec
  [GlobalAccess(false)]
  IEnumerator<T> GetEnumerator();
}
interface IEnumerator<T> {
  [WriteConfined] bool MoveNext();
  T Current { [GlobalAccess(false)] [Pure] get; }
  [WriteConfined] void Reset();
}
```

Figure 7. The methods needed for analyzing $Copy$ along with their annotations.

```
void m(A a){
  a = this;
  D d = new D();
  a.f = d;
}
```

$\mathcal{W}(\mathtt{m}) = \{\langle \mathtt{PLN(this)}, f\rangle\}$
$\mathcal{R}(\mathtt{m}) = \{\}$
$Write(m) = \{\mathtt{this.f}\}$
$Read(m) = \{\}$

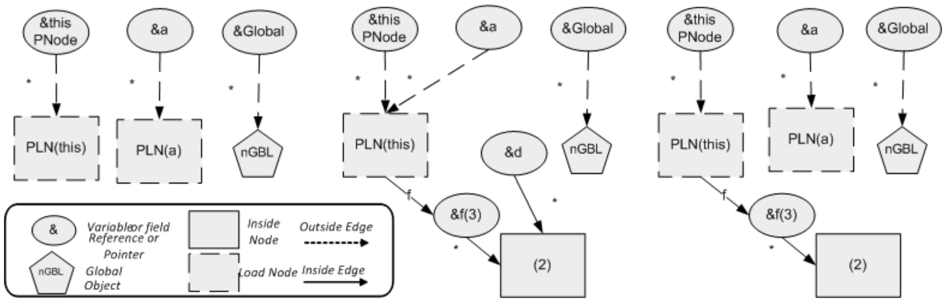

Figure 4. An example method, three points-to graphs for the beginning, end, and summary of the method and the read and write sets for the method. The latter is expressed both as the sets of nodes (`PLN` means *parameter load node*) and as the access paths.

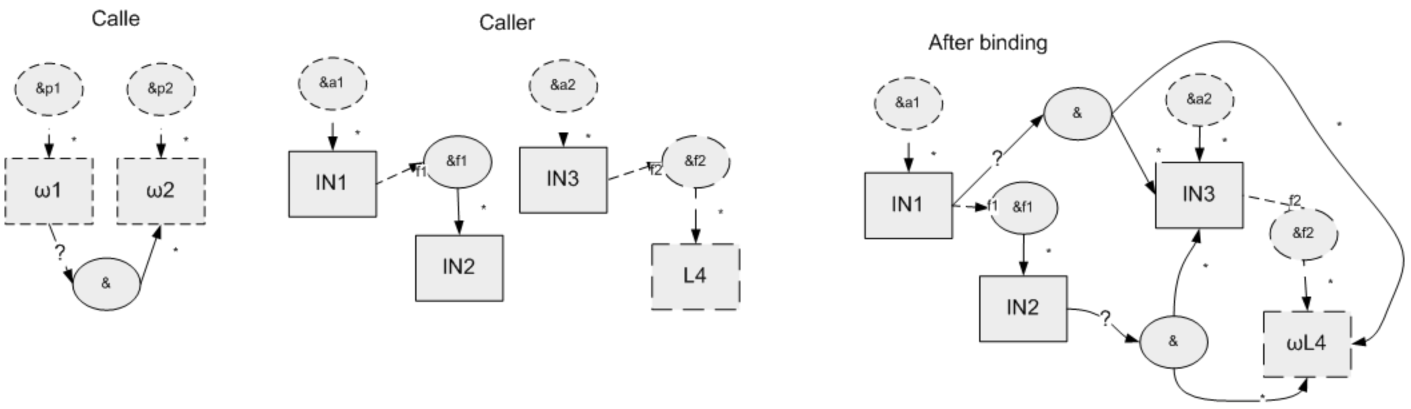

Figure 5. Effect of omega nodes in the inter-procedural mapping

Attribute Name	*Target*	*Default*	*Meaning*
Fresh	out Parameter	False	The returned value is a newly created object.
Read	Parameter	True	The content can be transitively read.
Write	Parameter	False	The content can be transitively mutated.
WriteConfined	Parameter	False	The content can transitively mutate only captured objects.
Escape(bool)	Parameter	False	Will any object reachable from the parameter be reachable from another object in addition to the caller's argument
Capture(bool)	Parameter	False	Will some caller object own the escaping-parameter's objects ?

GlobalRead(bool)
GlobalWrite(bool)
GlobalAcccess(bool)
	Method
	Method
	Method
		True
		True
		True
			Does the method read a global?
			Does the method write a global?
			Does the method read or write a global?

Pure	Method	False	The method can not mutate any object from its prestate except for out parameters
WriteConfined	Method	False	The method mutates only objects owned by the parameters (captured).

Table 1. The set of attributes used to summarize the points-to graph and the read and write sets. The attributes $Fresh$ and $Escape$ also are allowed on the "return value" of the method since we model that as an extra (out) parameter. In C#, attributes on return values are specified at the method level with an explicit target, e.g., `[return:Fresh]`.

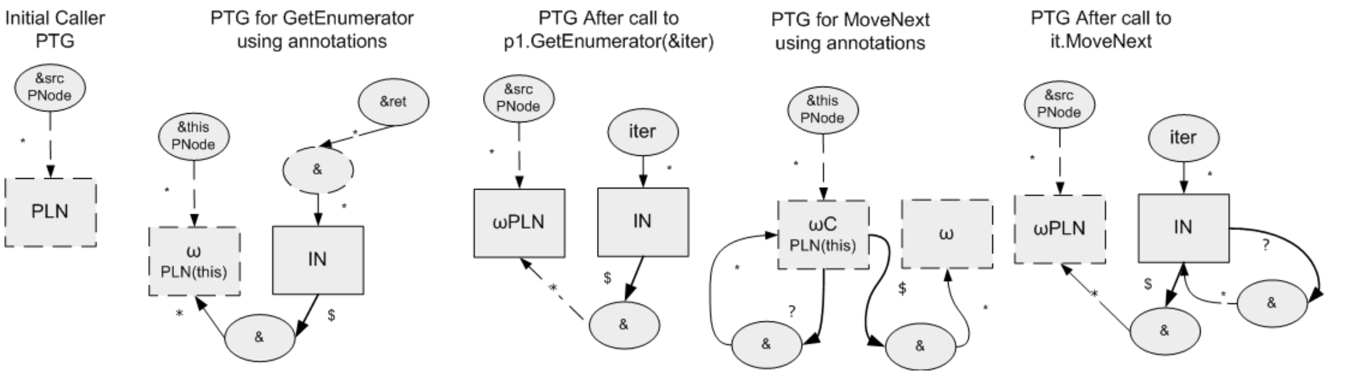
Initial Caller PTG
PTG for GetEnumerator using annotations
PTG After call to p1.GetEnumerator(&iter)
PTG for MoveNext using annotations
PTG After call to it.MoveNext
&src PNode
PLN
&this PNode
&ret
&
ω PLN(this)
IN
ωPLN
iter
ωC PLN(this)
ω
*
$
?

Figure 6. The evolution of $Copy$'s points-to graph after calling `src.GetEnumerator` and `iter.MoveNext`. We use the special field \$ to indicate that src is reachable from $iter$ but $iter$ is able to mutate objects only using fields that $iter$'s class owns. For simplicity we do not show the evolution of the newly created objects pointed to by the list l.

4. Experimental Results

Our implementation is integrated into the Spec# compiler pipeline and can also be run as a stand alone application. We analyze Boogie [3], a program verification tool for the Spec# language [2]. Boogie is itself written in Spec# and so already has some annotations. In this case we use our tool to verify methods annotated as pure. We analyzed the eight application modules using three different approaches. *Intra-procedural:* We analyze each method body independently. In the presence of method calls we use any annotations provided by the callee. *Inter-procedural (bottom up with fixpoint):* This is a whole program analysis. We compute a partial call graph and analyze methods in a bottom up fashion in order to have the callee precomputed before any calls to that method. To deal with recursive calls we perform a fixpoint computation over the strongly connected graph of mutually recursive calls. *Inter-procedural (top down with depth 3):* Again, a whole program analysis with inline simulation. For every method we analyze call chains to a maximum length of three.

Table 2 contains the results for the three kinds of analysis. We show only modules that contain purity annotations. The intra-procedural analysis is only slightly less precise than the other two analyses. Furthermore, when using annotations with intra-procedural analysis, the precision is substantially better than a full inter-procedural analysis without annotations. For this application we don't find a big difference between the two inter-procedural analyses. This is because most of the methods are not recursive.

One interesting thing is that we found that many of the methods declared pure in Boogie were not actually pure. Some are observationally pure, but others either record some logging information in static fields, or else were just incorrectly annotated as being pure.

5. Related work

Our analysis is a direct extension of the points-to and effect analysis by Salcianu et al. [22]. We add support for a more complex memory model (managed pointers and structs) and provide a different approach for dealing with non-analyzable methods. Instead of assuming that every argument escapes and the method writes the global scope, we try to bound the effect of unknown callees using annotations. Using their analysis it is difficult to decide that a method is pure when it calls a non-analyzable method (e.g., the iterator example). One alternative is to generate by hand all the information about the callee (points-to and effects) but it has to be done for every implementation of an interface or abstract class. Our annotation language simplifies that task and allows us to verify the annotations when code becomes available.

Type and effect systems have been proposed by Lucassen et al. [17] for mostly functional languages. There has been a significant amount of work in specification and checking of effect information relying on user annotations. Clarke and Drossopoulou use ownership types [9] while Leino et al. use data groups [16]. In [14], an effect system using annotations is proposed: it allows effects to be specified on a field or set of fields (regions). It also has a notion of "unshared" fields that corresponds to our ownership system. Using a purely intra-procedural analysis, they verify methods against their annotations. However, it seems that it doesn't compute points-to-information. Compared to their approach, our annotation language is less precise, but still allows enough information about escaping and captured parameters. JML [15] and Spec# [2] are specification languages that allow specification of write effects. One of the aims of our technique is to assist the Spec# compiler in the verification and inference of the read and write effects. We use the purity analysis to check whether a method can be used in specifications. Javari [24] uses a type system to specify and enforce read-only parameters and fields. To cope with caches in real applications, Javari allows the programmer to declare mutable fields; such fields can be mutated even when they belong to a read-only object. Our technique computes weak purity so mutation of prestate objects are not allowed in methods. To automatically deal with caching writes, it is necessary to infer observationally pure methods [6].

Points-to information has also been used to infer side effects [21, 19, 8, 7]. Our analysis, as well as Salcianu's analysis [22], is able to distinguish between objects allocated by the method and objects in the prestate. This enables us to compute weak purity instead of only strong purity. In more recent work, Cherem and Rugina [7] present a new inter-procedural analysis that generates method signatures that give information about effects and escaping information. It allows control of the heap depth visibility and field branching, which permits a tradeoff between precision and scalability. Our analysis also computes method summaries containing read and write effect information that are comparable with the signatures computed by their analysis but our technique is able to deal with non-analyzable library methods with a concise set of annotations that can be checked when code is available. AliasJava [1] is an annotation language and a verification engine to describe aliasing and escape information in Featherweight Java. Our work also uses annotations to deal with escape, aliasing and some ownership information but also some minimal description about read and write effects in order to compensate for information lacking at non-analyzable calls. Hua et al. [20] proposed a technique to compute points-to and effect information in the presence of dynamic loading. Instead of relying on annotations, they only compute information for elements that may not be affected by dynamic loading and warn about the others.

6. Conclusions and Future Work

We have implemented an extension to Salcianu's analysis [22] that works on the complete .NET intermediate language CIL. The extensions involve several non-trivial details that enable it to deal

Project	#Meths	DP	Using Annotations						Without Annotations					
			Intra	%	Inter 3	%	IF	%	Intra	%	Inter 3	%	IF	%
AbsInt	348	66	66	100%	66	100%	66	100%	51	77%	51	77%	51	77%
AIFramework	15063	3514	2702	77%	2725	77%	2730	78%	1631	46%	1688	48%	1688	48%
Graph	97	20	14	70%	14	70%	14	70%	10	50%	10	50%	10	50%
Core	9628	1326	1164	88%	1224	92%	1224	92%	709	53%	729	55%	729	55%
ByteCodeTrans	5564	984	781	79%	845	86%	863	88%	255	26%	297	30%	297	30%
VCGeneration	2050	187	171	91%	171	91%	171	91%	155	83%	155	83%	155	83%
Compiler Plugin	55	12	10	83%	10	83%	10	83%	8	66%	8	66%	8	66%

Table 2. Results for Boogie showing the number of methods annotated as pure that were verified as pure by our analysis. The "DP" (declared pure) column lists the number of methods in each module that were annotated as pure. The column labeled "Intra" shows the number of methods verified using the intra-procedural analysis, "Inter 3" the inter-procedural top-down analysis limited to a call-chain depth of three, and "IF" is the full bottom-up inter-procedural analysis.

with call-by-reference parameters, structs, and other features of the .NET platform. Our model provides a simple operational semantics for a useful part of CIL. Full details are presented in an accompanying technical report [4].

We have extended the previous analysis by including ω-nodes that model entire unknown sub-graphs. Together with our annotation language, this allows treatment of otherwise non-analyzable calls without losing too much precision.

The abstraction aspect of ω-nodes also holds the promise to improve the scalability of the analysis by enabling points-to graphs to be abstracted further than possible in the original analysis by Salcianu.

We believe our annotation system strikes the proper balance between precision and conciseness. The annotations are specifications that are useful not only for the analysis itself, but represent information programmers need to use an API effectively. Our technique needs to be very conservative when dealing with load nodes. We are planning to improve it by recomputing the set of egdes ($?$, $\$$, ω) when new nodes become available. We also plan to leverage type information to avoid aliasing between incompatible nodes.

Our annotation language appears to be general, but it was designed with our purity analysis in mind. It is possible to create a different set of annotations; our approach would work given a mapping from the set of annotations into points-to graphs. It is also possible to imagine the annotations being elements of the abstract domain themselves, instead of using a separate annotation laguage. Besides usability concerns for real programmers, it could make the verification of a method against its specification more difficult: our annotation language is intentionally simple enough to make the verification easy to perform.

One problematic aspect of the system is the necessity to introduce an ownership system. The concept of ownership certainly exists in real code, but the right formalization is not fully agreed upon. There are several different ownership systems in the literature and we believe the meaning of our annotations would work for any of them. For now, we have connected our annotations to the Spec# ownership system.

By relaxing the closed-world requirements so that we do not need full programs, we hope to enable the use of our system within real programming practice. In the future we hope to present results from some real-world case studies.

There are other uses for a points-to and effect analysis besides method (weak) purity. In addition to using it for checking forms of observational purity, we have adapted the analysis for studying *method re-entrancy* [12]. It is also possible to use it for inferring and checking method *modifies clauses*.

Acknowledgements

We would like to thank the anonymous reviewers for their comments.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM Press.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.

[4] Mike Barnett, Manuel Fandrich, Diego Garbervetsky, and Francesco Logozzo. A read and write effects analysis for C#. Technical Report MSR-TR-2007-xx, Microsoft Research, April 2007. Forthcoming.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[6] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, LNCS, pages 54–84. Springer, July 2004.

[7] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *CC 2007: 16th International Conference on Compiler Construction*, Braga, Portugal, March 2007.

[8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM Press.

[9] Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *ACM SIGPLAN Notices*, 37(11):292–310, November 2002.

[10] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[11] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/-ecma-335.htm, Ecma International, 2006.

[12] Manuel Fändrich, Diego Garbervetsky, and Wolfram Schulte. A reentrancy analysis for object oriented programs. In *FTfJP 2007:*

ECOOP Workshop on Formal Techiques for Java-like Programs, July 2007.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[14] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *LNCS*, pages 205–229. Springer-Verlag, New York, June 1999.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[16] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. *SIGPLAN Notices*, 37(5):246–257, May 2002.

[17] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[18] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[19] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[20] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 9–18, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[21] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 20–36, London, UK, 2001. Springer-Verlag.

[22] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2005.

[23] http://research.microsoft.com/specsharp/.

[24] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2005. ACM Press.