Biblioteca Digital
F C E N - U B A

BIBLIOTECA CENTRAL LUIS F LELOIR
BIBLIOTECA CENTRAL LELOIR
FACULTAD DE CIENCIAS EXACTAS Y NATURALES UBA

Tesis Doctoral

Síntesis de especificaciones paramétricas de utilización de la memoria dinámica

Garbervetsky, Diego

2007

Este documento forma parte de la colección de tesis doctorales y de maestría de la Biblioteca Central Dr. Luis Federico Leloir, disponible en digital.bl.fcen.uba.ar. Su utilización debe ser acompañada por la cita bibliográfica con reconocimiento de la fuente.

This document is part of the doctoral theses collection of the Central Library Dr. Luis Federico Leloir, available in digital.bl.fcen.uba.ar. It should be used accompanied by the corresponding citation acknowledging the source.

Cita tipo APA:
Garbervetsky, Diego. (2007). Síntesis de especificaciones paramétricas de utilización de la memoria dinámica. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.

Cita tipo Chicago:
Garbervetsky, Diego. "Síntesis de especificaciones paramétricas de utilización de la memoria dinámica". Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 2007.

EXACTAS UBA
Facultad de Ciencias Exactas y Naturales

UBA
Universidad de Buenos Aires

Dirección: Biblioteca Central Dr. Luis F. Leloir, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Intendente Güiraldes 2160 - C1428EGA - Tel. (++54 +11) 4789-9293

Contacto: digital@bl.fcen.uba.ar

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Síntesis de especificaciones paramétricas de utilización de la memoria dinámica

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

Diego Garbervetsky

Director de tesis: Dr. Víctor Braberman
Director asistente: Dr. Sergio Yovine

Buenos Aires, 2007

Síntesis de especificaciones paramétricas de utilización de la memoria dinámica

Resumen: En los últimos años se ha visto un gran interés en las comunidades de sistemas de tiempo real y embebidos en el uso de de lenguajes orientados a objetos tipo Java. Los motivos de este interés se deben en parte a que este tipo de tecnologías facilitan la encapsulación de abstracciones y la comunicación mediante interfaces bien definidas. Otro aspecto importante es la gran comunidad de desarrolladores y la cantidad de bibliotecas y herramientas de desarrollo disponible.

Sin embargo, para poder adoptar lenguajes de este tipo en ambientes embebidos y de tiempo real hay que solucionar al menos dos grandes problemas: la impredictibilidad temporal dada por las interrupciones relacionadas con la colección de objetos (garbage collector) y poder analizar requerimientos de memoria de las aplicaciones.

Ha habido un número importante de trabajos donde se intenta atacar el problema de impredictibilidad temporal de los administradores de memoria automáticos desde distintos enfoques tales como garbage collectors con ciertas garantías temporales o directamente utilizando modelos alternativos de administración de memoria. Sin embargo, no ha habido muchos avances con respecto al estudio cuantitativo de requisitos de memoria.

En esta tesis abordamos el problema de predecir automáticamente certificados de utilización y requisitos de memoria. Para ellos presentamos primero una técnica que permite obtener expresiones paramétricas de las solicitudes de memoria dinámica sin considerar ningún mecanismo de colección de objetos. Luego proponemos un esquema alternativo de administración de memoria junto con una técnica que permite la transformación de código Java convencional en otro con la misma funcionalidad pero adaptado para la nueva política de manejo de la memoria. Bajo este nuevo esquema, proponemos una técnica que permite determinar de manera paramétrica la cantidad de memoria necesaria para correr el programa o parte de él.

Todas estas técnicas fueron implementadas en un prototipo que nos permitió analizar automaticamente un conjunto interesante de aplicaciones siendo los resultados iniciales bastante promisorios.

palabras clave: Administración de memoria dinámica, consumo de memoria, sistemas embebidos, análisis estático, análisis de escape.

Parametric specifications of dynamic memory utilization

Abstract: Current trends in the embedded and real-time software industry are leading towards the use of object-oriented programming languages such as Java. From the software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces.

However, in order to be able to successfully adopt languages with object oriented features like Java in embedded and real-time systems, is necessary to solve at least two problems: eliminate execution unpredictability due to garbage collection and automatically analyze memory requirements.

There has been some work trying to deal with the first problem but the problem of computing memory requirements is still challenging. In this thesis we present our approach to tackle both problems by presenting solutions towards more predictable memory management and predicting memory requirements. The effort is mainly focused in the latter problem as we found it hard, less explored, strongly relevant for all kinds of embedded systems and its applicability and usefulness is beyond real-time applications.

This thesis presents a series of techniques to automatically compute dynamic memory utilization certificates. We start by computing a technique that produces parametric specifications of memory allocations without consider any memory reclaiming mechanism. Then, we approximate object lifetime using escape analysis and synthesize a scoped-based memory organization where objects are organized in regions that can be collected as a whole. We propose a technique to automatically translate conventional Java code into code that safely adopts this memory management mechanism. Under this new setting we infer parametric specifications of the size of each memory regions. Finally, we predict the minimum amount of dynamic memory required to run a method (or program) in the context of scoped memory management by computing parametric specifications of the size of memory regions and by modeling the potential configurations of the regions in run time.

We develop a prototype tool that implemented the complete chain of techniques and allow us to experimentally evaluate the efficiency and accuracy of the method on several Java benchmarks. The results are very encouraging.

keywords: dynamic memory management, memory consumption, embedded systems, static analysis, escape analysis.

Agradecimientos

A toda mi familia, especialmente a mis padres y hermanas por todo el apoyo y afecto que me dieron durante todos estos años.

A mis amigos y autoridades de la Escuela Técnica ORT por los buenos tiempos, por haber comenzado a definir mi interés por la computación y en especial por la investigación científica. Un agradecimiento especial a la profesora Clara Freud ya que influyo fuertemente en que finalmente me decida a estudiar en la FCEyN.

A la Universidad de Buenos Aires, primero por brindarme la posibilidad de obtener una educación de calidad tanto desde lo académico como desde lo humano y luego por permitir desarrollar mi vocación docente y científica. La UBA es una universidad pública, gratuita, gobernada y administrada por estudiantes, graduados y docentes, donde a pesar de las dificultades económicas muchos ponen lo mejor de ellos mismos para el bien de la comunidad universitaria y de la población en general. Espero poder colaborar a convertirla en un lugar aún mejor.

A Víctor y Sergio mis directores por ayudarme, guiarme y aguantarme durante todos estos años. A Víctor un especial agradecimiento por saber ser mi amigo en momentos difíciles y por incentivarme a investigar cuando aún no estaba decidido. A Sergio por haberme invitado a realizar las pasantías en Verimag donde surgieron las principales ideas de esta tesis.

A mis estudiantes "estrella" Diego Piemonte, Andrés Ferrari, Federico Fernández y Guido de Caso por colaborar en partes importantes de mi trabajo y a todos los presentes y pasados integrantes de Dependex/Laphis por los gratos momentos en la mega oficina.

A los amigos que conoci en la facu: Nico K, Dani, Sergio M, Chapa , Esteban, Public, Mariela, Laura G, Ariel C, Ariel D, Diego FS, Sergi D, Techas, Greg, Pablo M, Charly LP, Charly LH, Juan Pablo, Flavia, Guille, y los que me olvide en este momento...

A mis amigos de afuera de la facu: Patu (Groc), Diego S, Juanjo, Fede, Luigi, Diego Q, Ruben, Eli, todos sus novias\os esposas\os, y todos los que me olvido en este momento...

Al CONICET, la fundación YPF Estenssoro y Microsoft Research por su ayuda económica durante parte del doctorado así como a la Agencia Nacional de Promoción Científíca y Tecnológica, Microsoft Research e IBM por financiar algunos de nuestros proyectos.

Finalmente, una dedicación especial a MAGDA por todo su amor, dulzura y paciencia infinita.

Contents

CHAPTER 1

Introduction

1.1. Motivation

Current trends in the embedded and real-time software industry are leading towards the use of object-oriented programming languages such as Java. From the software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces.

Because programmer-controlled memory management inhibits modularity, object oriented languages like Java, provide built-in garbage collection (GC) [JL96], that is, the automatic reclamation of heap-allocated storage after its last use by a program. Dynamic memory management is a serious challenge for real-time embedded systems based on Java technology. Unlike the standard Java paradigm, garbage collection is rarely used in such real-time environments, since execution times and memory occupancy become difficult to predict and thus significantly complicates the implementation of real-time scheduling policies. Many different garbage collection algorithms have been developed and they achieve very good performance (e.g. [BCGV05, BCG04, Hen98, HIB+02, RF02, Sie00]), but they all have a very high worst-case complexity. As the GC can stop the application program at any time, and for an unpredictable amount of time, it seems impossible to use it in a real-time context. Still, several projects like Metronome [BCR03], and JamaicaVM [Sie99] address the problem of building a real-time GC. In addition to an optimized design, the key idea of these algorithms is to use a statistical model of the application program behavior: the GC is then scheduled according to application-dependent parameters, such as the allocation rate, and more importantly, the garbage generation rate.

An interesting approach is to change the memory organization model, and to group objects in regions [TT97, GA01]. The idea behind region-based memory management is to group objects of similar lifetimes: within a region, one cannot deallocate any individual object, but must wait until the region can be destroyed as a whole. There are several variants of this memory model: the regions may either have a fixed size, or be allowed to expand when they become full; inter-region pointers may either be allowed or not; etc. The common point is to trade object deallocation, which is accurate but time-unpredictable, for region destruction, which presents a better temporal behavior, at the expense of some space overhead. Several approaches [GA01, SHM+06] propose to add region constructs to an existing language, but the

resulting programming model is still very difficult to use, because the programmer must decide in which region to place each object, and when to create and destroy regions. Regions are advocated by the Real-time Specification for Java (RTSJ) [GB00]. It proposes several extensions to the syntax and semantics of Java that aim at making the execution more predictable. To get rid of the garbage collector for time-critical tasks, the RTSJ offers lexically scoped memory regions called ScopedMemory areas. This environment is appealing, as it guarantees constant-time memory operations, but it is very restrictive for the programmer: the size of the regions is fixed, and must be decided at programming time. Moreover, RTSJ includes assignment rules that forbid an object in a short-lived region to be referenced by an older object. However, programming for the RTSJ is thus very difficult [PFHV04], as it makes it impossible to reuse any old code (even the Standard Library has to be fully rewritten), and it forces the programmer to adopt new coding habits and to reason in a new paradigm quite different from Java.

Still, in order to develop efficient region-based memory managers, as stated in the RTSJ, it is necessary to give (to the memory manager) upper-bounds of the amount of memory to be allocated in each region. However, automatically evaluating quantitative memory requirements becomes inherently hard. Indeed, finding a finite upper-bound on memory consumption is undecidable [Ghe02]. This is a major drawback since embedded systems have (in most cases) stringent memory constr*mua*or are critical applications that cannot run out of memory.

In summary, in order to be able to successfully adopt languages with object oriented features like Java, is necessary to solve at least two problems:

1. Eliminate execution unpredictability due to garbage collection

2. Automatically analyze memory requirements

There has been some work trying to deal with the first problem but the problem of computing memory requirement is still challenging.

In this thesis we present our approach to tackle both problems by presenting solutions towards more predictable memory management and predicting memory requirements. The effort is mainly focused in the latter problem as we found it hard, less explored, strongly relevant for all kinds of embedded systems and its applicability and usefulness is beyond real-time applications.

1.2. About this work

We try to tackle the problem of the GC by adopting a scoped-memory managed discipline and by proposing a series of techniques that allow programmers to automatically produce scope-memory managed code from conventional Java code. Given a standard Java program we divide the dynamic memory space in regions that are associated with its computing units (i.e. methods, threads). Region inference is done by escape analysis [Bla03, SYG05, SR01], for which we propose two different techniques [SYG05, BFGL07a] (see chapters 4 and 5) and using a tool that allow manual editing of memory regions [FGB+05] (see chapter 6).

The main focus of this thesis is in trying to solve the problem of *predicting memory requirements*. Our aim is to have a technique that allows us to reason about memory consumption in order to know *a priori* the amount of memory required to safely run a program (or part of it). We also deal with the problem of automatically computing the size of memory regions. We develop a series of techniques for

computing parametric upper-bounds of the amount of dynamic memory utilization in Java-like imperative object-oriented programs (see chapter 2).

Our first technique quantifies the explicit dynamic allocations made by a method. Given a method m with parameters $p_1, \ldots, p_k$ we exhibit an algorithm that computes a non-linear expression over $p_1, \ldots, p_k$ which over-approximates the amount of memory *requested* during the execution of m. By requested we mean the amount of memory that is solicited to the system (i.e. a virtual machine or operating system) through "new" statements, without considering any kind of collection mechanism.

This technique is insensitive to any memory management mechanism. Nevertheless, it serves as a basis for solving the problem of computing region sizes. Combining this algorithm with static pointer and escape analyses, we are able to compute memory region sizes to be used in scope-based memory management. Given a method m with parameters $p_1, \ldots, p_k$, we develop two algorithms that compute non-linear expressions over $p_1, \ldots, p_k$ which over-approximate, respectively, the amount of memory that *escapes from* and is *captured by* m. The prediction of the amount of memory captured is directly related with the size of memory region as objects captured by the method are not live after the scope of that method. Thus it can be safely allocated in its associated region. On the other side, the objects that escape the method have to be captured by some method in the outer scope, so following scoping rules, they have to be allocated in another region. As a consequence, the prediction of the amount of memory escaping serves as a measure of the residual memory that will remain occupied after the execution of the method.

Finally, we propose a new technique to over-approximate the amount of memory *required* to run a method (or a program). Given a method m with parameters $p_1, \ldots, p_k$ we obtain a polynomial upper-bound of the amount of memory necessary to *safely* execute the method and all methods it calls, without running out of memory. This polynomial can be seen as a *pre-condition* stating that the method requires that much free memory to be available before executing, and also as a *certificate* engaging the method is not going to use more memory than the specified.

1.3. Overview

The long term goal of our work is to have a tool that is able to start from conventional Java code and automatically produce equivalent Java code that runs under a more predictable memory management together with certificates of memory requirements to guarantee proper execution.

In this work we use a simple but useful scoped-based memory manager in which objects are allocated in regions that are associated with methods. Consequently, a region is created at method's entry and is destroyed at its end. When an object is created it has to be allocated in one region but when a region is collected all the objects within that region are also collected.

Under this setting our prototype tool is capable of predicting the size of the memory regions and the amount of memory required to run an application without crashing because running out of memory. This initial prototype is able to analyze single-threaded Java programs provided they do not feature recursion.

A view of the most important functional components that appear in our solution is shown in Fig. 1.1. Every component in the diagram is related with a technique we developed or adapted during this work. We can divide the components in two main categories: *Region Inference* related components and techniques and *Memory specification* related components and techniques.
Region inference related components:

- Region Inference
  - Escape Analysis: to automatically approximate object lifetime (see section 1.5.3).
  - Memory Region Inference: to produce memory regions from escape analysis information (see section 1.5.4).
- Region Management
  - Region-based API: to interact with a region-based memory manager (see section 1.5.2).
  - Region-based Code Generator: to translate conventional code to region-based code using the computed region information (see section 1.5.4).

Memory specification related components:

- Dynamic Memory utilization analyzer: to obtain parametric specifications of the amount of memory requested by a method (see section 1.4).
- Region Size inference: to obtain parametric information about the size of a memory region (see section 1.5.5).
- Memory requirements inference: to obtain parametric certificates of the amount of the dynamic memory required to safely run a method (see section 1.6).
- Local Invariant generation: to generate invariants required by the memory prediction techniques (see section 1.4.2).

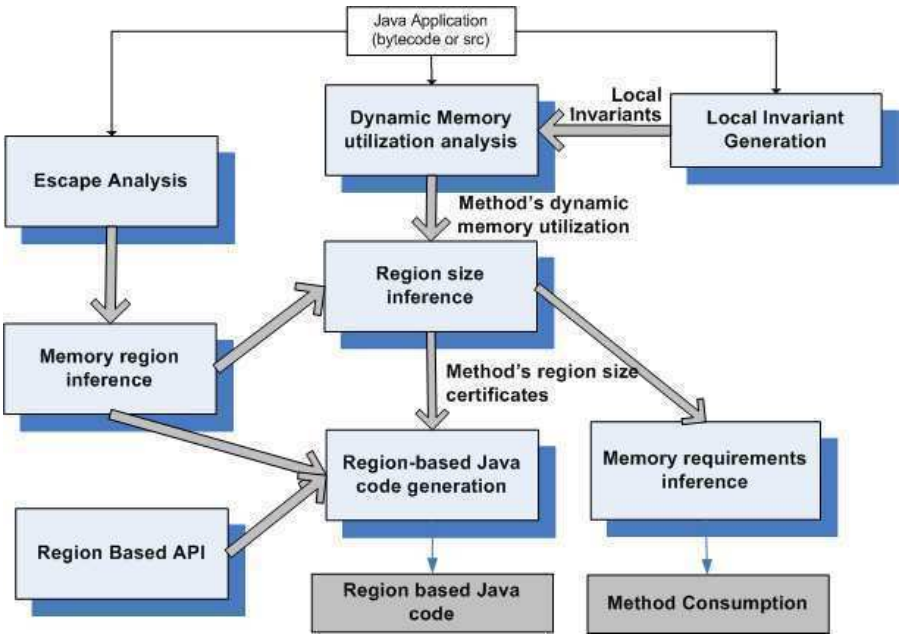


Figure 1.1: Main functional components of our solution.

The most conceptually challenging problems are the *Dynamic memory utilization analysis* and the *Memory requirements inference* and represent the core of this thesis work. The former approximates total allocations made by the application without

considering any kind of collection mechanism. The latter computes memory requirements taking into account that there might be some collection mechanism. As we will see later, both techniques require program invariants. That is why an important part of the work is involved in solving the problem of producing useful invariants.

For region inference we develop two escape analysis techniques and implement also a tool to visualize and refine the inferred region information. Using this region information we are able to produce Java code that uses a *Region-based API* that bypasses the standard Java memory manager. An interesting aspect of the API is that is uses the register/subscriber paradigm that eases the task of object allocation (see section 1.5.2).

Hereinafter, we overview the main contributions of this thesis. More technical details are provided later in the respective chapters.

1.4. Dynamic memory utilization analysis

As we have mentioned, one of the main contributions of this thesis is a technique to obtain parametric upper bounds of dynamic memory utilization. By dynamic memory utilization we mean an expression that approximates the amount of dynamic memory *requested* to the system (or virtual machine) during the execution of the application (or selected method) in terms of its parameters.

To get a flavor of the approach, consider for instance the following program[1]:

```
void m1(int k) {
1:  for(int i=1;i<=k;i++)  {
2:     A a = new A();
3:     m2(i);
  }
}
```

```
void m2(int n) {
4:  for(int j=1;j<=n;j++) {
5:    B b = new B();
  }
}
```

For $m2$, our technique computes the expression:

$$size(\texttt{B}) \cdot n$$

which is the amount of memory requested if the program starts at $m2$. For $m1$, the computed expression is:

$$size(\texttt{A}) \cdot k + size(\texttt{B}) \cdot \frac{1}{2}(k^2 + k)$$

because starting at $m1$, the program will invoke $m2$ k times and, at each invocation $i \in [1, k]$, $m2(i)$ will request i instances of `B`, resulting in a total amount of:

$$\sum_{i=1}^{k} i = \frac{1}{2}(k^2 + k)$$

instances of `B`, which have to be added to the k instances of `A` directly allocated by $m1$.

Our general technique to infer dynamic memory requests relies on the following idea: The amount of memory *requested* is closely related to the number of visits to `new` statements. Using a combinatorial approach, this can be related to the number of possible valuations of variables that it might feature at its control location.

[1] We assume that calls to constructor are analyzed like any other call. In this example, the constructor has no code to analyze.

Furthermore, this can be related to the number of integer solutions of a predicate constraining variable valuations at its control location (i.e. an invariant). For linear invariants, the number of integer solutions is equivalent to the number of integer points which can be expressed as an Ehrhart polynomial [Cla96].

$$\phi \equiv \{1 \leq i \leq k, n = i, 1 \leq j \leq n\}$$

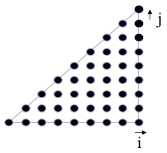


Figure 1.2: Invariant representing the iteration space at the statement `new B()`

In Fig. 1.2 we show an invariant which is used to model the potential valuations of variables for a program state at the control location 5 (creation of an object of type `B`) and called from the control location 2. On the right we show its geometrical representation showing the number of integer points inside the triangle represented by variables i and j.

Assuming that k is a fixed value (i.e. a parameter) the number of integer points for this invariant is expressed by the polynomial:

$$\sum_{i=1}^{k} i = \frac{1}{2}(k^2 + k)$$

Observe that in the invariant also appears the variables of method $m1$ since we want to count allocations made by runs starting at $m1$ and the invariant must represent the global state when the parameters of method $m1$ and method $m2$ binded in some way.

Our approach is basically the following:

1. Identify every allocation site (`new` statement) reachable from the method under analysis (MUA).

2. Generate linear invariants describing possible variables valuations at each allocation site.

3. Count the number solutions for the invariant in terms of MUA parameters (# of visits to the allocation site)

4. Adapt those expressions to take into account the size of object allocated (their types).

5. Sum up the resulting expressions for each allocation site.

A detailed view of the components involved in the tool that implements this approach is shown in Fig. 1.3.

- Allocation site identification is done by the *Creation Site finder*.

- Invariants are generated by combining local invariants obtained from the *Local Invariant Generator* and the *Control State Invariant generator*.

- The *Symbolic Polyhedral Calculator* provides techniques and tools to produce polynomials that represent parametric expressions of the number of solutions of the given invariants.

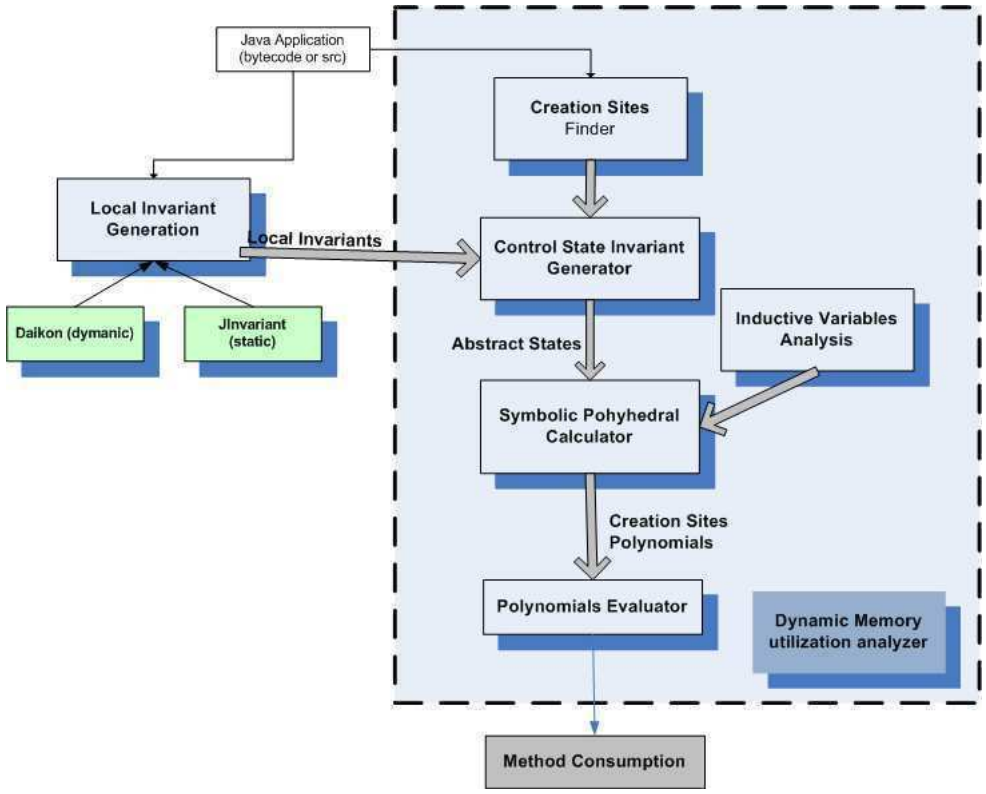


Figure 1.3: View of the components of the dynamic memory requests inference engine

- *Polynomials evaluator* allows us to manipulate and evaluate the resulting polynomials.

Our work is inspired in techniques appearing in the field of parallelizing and optimizing compilers where the use of linear constrains to model iteration spaces were traditionally applied to work on performance analysis, cache analysis, data locality, and worst case execution time analysis [Fah98, LMC02, Cla97, Lis03]. As far as we know, the combination of such techniques to obtain specifications that predict dynamic memory utilization is novel and most of the existing work (see related work in section 1.9) is focused on functional languages using type inference mechanism [HP99, USL03, HJ03], or abstract interpretation based approaches [USL03]. The use of linear invariants allows us to produce non-linear easy-to-evaluate expressions keeping the acceptable computational cost and tool support of linear programming (against other approaches that rely on Presburger arithmetic or polynomial algebra).

In Fig. 1.4 we present an slightly more complex example that we use to introduce the different aspects of the technique.

1.4.1. Identifying allocation sites

In order to obtain more precise bounds we distinguish program locations not only by a "method-local" control location but also by the different control stack configurations that lead to that location. Specifically, we identify allocation sites by the call chain starting from the MUA and finishing in a `new` statement.

We call these chains *Creation Sites* and are a particular case of *Control States* which are basically a sequence of program locations that models the *control* part of call stack configurations. The *data* counterpart of a control state is the *Control State*

```
void m0(int mc) {
1:  m1(mc);
2:  B[] m2Arr=m2(2 * mc);
}

void  m1(int k) {
3: for (int i = 1; i <= k; i++) {
4:   A a = new A();
5:   B[] dummyArr= m2(i);
  }
}

B[] m2(int n) {
6: B[] arrB = new B[n];
7: for (int j = 1; j <= n; j++) {
8:   arrB[j-1] = new B();
9:   C c = new C();
10:  c.value = arrB[j-1];
  }
11: return arrB;
}
```

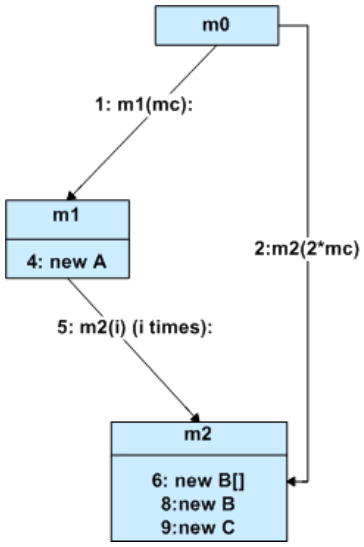


Figure 1.4: An example program with its detailed call graph

Invariant which are used to model sets of states for a given of control state.

As creation sites represent a traversal trough several methods, we use global invariants to model the potential set of valid states of control state (i.e. the data part of the call stack).

For instance: $m0.2.\mathbf{m2.6}$ is a creation site that represent the program location $m2.6$ with control stack $m0.2$. $m0.1.m1.5.\mathbf{m2.6}$ is a creation site that represent the program location $m2.6$ with control stack $m0.1.m1.5$

Example 1.1. In this example the creation sites reachable from $m0$, $m1$ and $m2$ are:

$$\begin{aligned} CS_{m0} &= \{\text{m0.1.m1.4, m0.1.m1.5.m2.6, m0.1.m1.5.m2.8, m0.1.m1.5.m2.9,} \\ &\quad \text{m0.2.m2.6, m0.2.m2.8, , m0.2.m2.9}\} \\ CS_{m1} &= \{\text{m1.4, m1.5.m2.6, m1.5.m2.8, m1.5.m2.9 }\} \\ CS_{m2} &= \{\text{m2.6, m2.8, m2.9 }\} \end{aligned}$$

□

To accurately compute call chains and to get all allocation sites reachable from the application under analysis we rely on computing a precise call graph. Call graphs are obtained with `Soot` [VRHS+99].

Please note that to compute control states we are making two strong assumptions

1. There is no recursion and all allocation sites in the application can be reached by static analysis.

2. The amount of "hidden" memory allocated by native methods or by the virtual machine itself cannot be quantified with this technique.

For those cases that violate these assumptions, we will assume that a memory utilization specification is given.

1.4.2. Computing invariants

Our technique relies on having invariants that constraint the possible variable assignments of a specific program point. Control state invariant are fundamental for

our approach as they not only are used to model the potential variables valuation at a control state, they also are used to bind the parameters of the MUA with the different variables in the global state.

Local Invariants

Local invariants can be either provided by programmer assertions "à la" JML [LLP+00], or computed using general analysis techniques [CH78, CC02] or Java-oriented ones[NE01, FL01, ECGN99, CL05].

Local invariants can be computed using static analysis, e.g., [PG06, CH78, IS97], or dynamic analysis, e.g. [EPG+07]. In our work, we have explored both alternatives.

Dynamic invariant generation We have first used `Daikon` for dynamic detection of "likely" invariants by executing the program over a set of test cases. Even if the properties generated by `Daikon` have a high probability of being true in all runs, that is, being invariants, they might not be. In our experiments, we have manually verified all properties to be invariants (see section 2.5). Our tool "guide" `Daikon` in his search for program invariants [Gar05] (see section A.1). Basically we generate new method variables for expressions we presumed may have impact in the number of times allocation sites are visited (e.g, integer class fields, size of collections, length of arrays and strings, etc) and we produce a dummy method before every point of interest (call site and allocation sites) whose arguments are the variables we detected as relevant for that point of interest. Using that procedure the precondition of the generated method contains a invariant for the instrumented program point which predicates only about the specified variables.

Static invariant generation More recently, we have implemented and extended Hallwachs and Cousot´s seminal work [CH78] based on abstract interpretation to support method calls (interprocedural analysis), and to conservatively model the heap (points-to information) and some characteristics of Java language such as inclusion polymorphism. We developed a tool as we could not get a freely available static analysis tool which this capacity. We call this tool `JInvariant` [PG06]. However, we found several scalability and precision issues that complicate the use of this approach. Operating with linear invariants is costly, and a dataflow analysis requires lots of operation on this data structure. Another issue is the loss of precision in the presence of loops. This is due to the widening operation that makes the resulting invariants inappropriate for the posterior counting phase. We found that in practice the `Daikon` based dynamic invariant generation was able to compute more precise invariants than the static counterpart. Thus, we decided not to include details about this tool in this document. `JInvariant` is still a work in progress and we plan to improve it in the future.

Control State Invariants

None of the techniques for computing invariants deal with our concept of control state invariant since they only compute local invariants. Thus, the tool builds a control state invariant by computing the conjunction of the local invariants that hold in the control locations along the path. That task is performed by the *Control State Invariant Generator*.

Example 1.2. Consider the following local invariants for the example in Fig. 1.4.

$$\mathcal{I}^{m0}_{m0.1} = \{k = mc\}$$
$$\mathcal{I}^{m1}_{m1.5} = \{1 \leq i \leq k, n = i\}$$
$$\mathcal{I}^{m2}_{m2.8} = \{1 \leq j \leq n\}$$

Then, the invariant for the control states are:

$$\begin{aligned}
\mathcal{I}^{m0}_{m0.1.m1.5} &= \{k = mc, 1 \leq i \leq k, n = i\} \\
\mathcal{I}^{m1}_{m1.5.m2.8} &= \{1 \leq i \leq k, n = i, 1 \leq j \leq n\} \\
\mathcal{I}^{m0}_{m0.1.m1.5.m2.8} &= \{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}
\end{aligned}$$

□

1.4.3. Counting the number of visits

Recall that the number of visits at a control state (in particular a creation site) is related to the number of solutions of an invariant that describes (typically over-approximates) all valuations of variables for that point (the iteration space). The *Symbolic polyhedral calculator* represents a tool that can manipulate linear invariants. It consist of the algorithms used to count the number of solutions of a given invariant [Cla96]. To count the number of solutions of a predicate we need to select which variables are fixed (parameters) and which are free.

Example 1.3. Consider the following invariant for the creation site:

$$\mathcal{I}^{m0}_{m0.1.m1.5.m2.8} = \{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$$

Let mc be the parameter since it is the MUA parameter. Then the number of solutions (visits) for $\mathcal{I}$ in terms of mc is

$$\begin{aligned}
\mathcal{C}(\mathcal{I}^{m0}_{m0.1.m1.5.m2.8}, mc) &= \#\{(k, i, j, n) \mid k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\} \\
&= \frac{1}{2}(mc^2 + mc)
\end{aligned}$$

□

In the case of linear invariants we can use a technique due to Ehrhart [Ehr77]. Roughly speaking, it generates a polynomial whose variables are the parameters of the invariants. We compute Ehrhart's polynomials by the aid of `Polylib` [Pol] and a technique based on Barvinok's work [VSB+04].

Note that for this example the invariant mentions and constrains all variables visible in the control state. Producing such invariants can be difficult to do automatically or may require careful annotations if provided manually. Nevertheless, in general invariants do not need to predicate about every variable in a global state. We explain later (see section 1.4.5) that it is enough to constrain the set of inductive variables which are the variables that have a real impact in the number of visits that the analyzed control location may have.

1.4.4. Computing memory consumption expressions

To get the expressions that bound the amount of memory requested by a method we first compute a function called $\mathcal{S}(mua, cs)$ (see chapter 2) that given a creation site cs yields an expression in terms of mua parameters that bounds the amount of memory requested by cs. To compute that function we simply need to multiply the number of visits of a creation site by the size corresponding to the type of the allocated object. For arrays the computation is a little bit trickier (see section 2.3).

Once we compute the size of all creation sites we can compute the total amount of memory requested by a method. We simply need to sum the size expression for all creation sites reachable from the method.

Example 1.4. For instance, the total amount of memory requested by $m0$ is the following:

$$\begin{aligned}
\texttt{memalloc}(m0)(mc) &= \sum_{cs \in CS_{m0}} \mathcal{S}(m0, cs) \\
&= \mathcal{S}(m0, m0.1.m1.4)(mc) \\
&\quad + \Big(\mathcal{S}(m0, m0.1.m1.5.m2.6)(mc) \\
&\quad + \mathcal{S}(m0, m0.1.m1.5.m2.8)(mc) \\
&\quad + \mathcal{S}(m0, m0.1.m1.5.m2.9)(mc) \Big) \\
&\quad + \Big(\mathcal{S}(m0, m0.2.m2.6)(mc) + \mathcal{S}(m0, m0.2.m2.8)(mc) \\
&\quad + \mathcal{S}(m0, m0.2.m2.9)(mc) \Big) \\
&= (\texttt{size}(B[]) + \texttt{size}(B) + \texttt{size}(C))(\frac{1}{2}mc^2 + \frac{5}{2}mc) \\
&\quad + \texttt{size(A)}mc
\end{aligned}$$

□

Note that the precision of our analysis depends on the accuracy of both invariant and call graph generation techniques (specially in the presence of dynamic binding). The technique gets a counting expression for every allocation site assuming that allocation sites that cannot appear in the same iteration (e.g, `else` and `then` branches of if statements) are constrained by the corresponding invariant. Weak invariants and infeasible calls make our technique over-approximate too much. In section 2.6.3 we show some ideas in order to try to mitigate this problem. In particular it is fundamental to discover what we call set of inductive variables.

1.4.5. Computing set of inductive variables

As we mentioned, we do not need to constrain the valuations of each variable in a global state. A key concept for our characterization of iteration spaces is the set of *inductive variables* for a control location. That is, a subset of program variables which cannot repeat the very same value assignment in two different visits of the given control state (except in the case where the program loops forever).

An invariant that only involves parameters and a set of inductive variables is called an *inductive invariant*. As we associate the number of visits to statements with the number of solutions, relying on inductive invariants guarantees soundness. An invariant that does not constrain the values of an inductive variable would lead

to an over approximation of the upper bounds. However, selecting a smaller set of inductive variables might lead to invalid bounds.

To compute inductive variables we developed a conservative dataflow analysis that combines a live variables analysis augmented with field sensitivity with a loop inductive analysis [NNH99]. This problem has been studied for programs that make use of iteration patterns composed of `for` and `while` loops with simple conditions.

Example 1.5. The inductive set of variables for the creation site $m0.1.m1.5.m2.8$ is $\{mc, k, i, j, n\}$. □

Handling more complex iteration patterns and types beyond integers is a challenging issue related to finding variant functions for the iteration. In 2.6.2 we brifely discuss our general strategy and we show how the tool currently deals with an iteration pattern pervading Java applications as it is the case of looping over collections.

Indeed, while not dealing with recursive programs is an underlying limitation of the approach, handling complex data-structures (such as collections) is not precluded, but it is a challenge for building good linear invariants.

1.4.6. Some Experiments

The initial set of experiments were carried out on a significant subset of programs from JOlden [CM01] and JGrande [DHPW01] benchmarks. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications – the target application classes we had in mind when we devised the technique. That is why we could not analyze some of the programs since they were highly recursive and our technique at the moment cannot handle recursion.

The tool was able to synthesize very accurate and non-trivial estimators for the number of object instances created (and memory allocated) in terms of program parameters for several examples that do not feature recursion. In contrast to [CNQR05], all these results were achieved using the **original code** as input for the method and reducing human intervention to a minimum (i.e., creation of test cases for `Daikon`, strengthening some of the automatically detected invariants and reducing some of automatically detected inductive sets of variables). Remaining obstacles that prevent fully automatic analysis of some examples are complex data-structures which must be considered part of any set of inductive variables and thus, an integer interpretation of them should be provided by the user to build a useful linear invariant.

In order to make the result more readable, the tool computes the number of object instances created when running the selected method, rather than the actual memory allocated by the execution of the method[2]. Also, we set aside analyzing the standard Java library in order to keep examples manageable.

Table 1.1 shows the computed polynomials and the comparison between real executions and estimations obtained by evaluating the polynomials with the corresponding values of parameters. The last column shows the relative error ((#Obs - Estimation)/Estimation).

These experiments shows that the technique is indeed efficient and very accurate, actually yielding exact figures in most benchmarks. For the (*)`health` example, it is impossible to find a non-trivial linear invariant. It actually turns out that memory consumption happens to be exponential.

More details about the benchmarks can be found in chapter 2.

[2] By memory allocated we consider the amount of memory occupied by the objects, not the actual memory reserved by a particular memory manager for internal accounting purposes. In this setting, we assume for simplicity that the function `size(T)`=1 for all type `T`

Example:Class.Method	Static Analysis		Precision Analysis			
	$\#CS_m$	memAlloc	Param.	#Objs	Estim.	Err%
mst:MST.main(nv)	13	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+4nv+5$	10 20 100 1000	240 940 22700 2252000	245 985 22905 2254005	2,00 5,00 1,00 0,09
mst:MST.computeMST(g, nv)	1	$nv-1$	10 20 100 1000	9 19 99 999	9 19 99 999	0,00 0,00 0,00 0,00
mst:Graph.Graph(nv)	6	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+3nv$	10 20 100 1000	230 920 22600 2251000	230 960 22800 2253000	0,00 4,17 0,88 0,09
mst:Graph.addEgdes(nv)	2	$2nv^2$	10 20 100 1000	180 760 19800 1998000	200 800 20000 2000000	10,00 5,00 1,00 0,10
Em3d.main(nN, nD)	28	$6nD \cdot nN + 4nN + 14$	(10, 5) (20, 6) (100, 7) (1000, 8)	350 810 4610 52010	354 814 4614 52014	1,13 0,49 0,09 0,01
(*)health: (recursive) Village.createVillage(l, lab, b, s)	8	$11(4^l-1)/3$	2 4 6 8	55 935 15015 240295	∞ ∞ ∞ ∞	∞ ∞ ∞ ∞

Table 1.1: Experimental results

1.5. Scoped Memory Inference and Management

Scoped-memory management is based on the idea of allocating objects in *regions* associated with the lifetime of a computation unit, i.e., its scope. A computational unit can be a method, a thread, etc. When a computational unit finishes its execution, its objects are automatically collected.

For instance, the Real-Time Specification for Java (RTSJ) [GB00] proposes a new memory hierarchy which incorporates this kind of memory management. In particular it proposes several kinds of memory models: *Heap memory* (garbage collected), *Immortal memory* and *Scoped memory*. Neither Immortal nor Scoped memory use garbage collection. Objects allocated in Immortal memory are never collected and live throughout program lifetime. This approach imposes restrictions on the way objects can reference each other in order to avoid the occurrence of dangling references. An object $o1$ belonging to region r references an object $o2$ only if one of the following conditions holds: $o2$ belongs to r; $o2$ belongs to a region that is always active when r is active; $o2$ is in the Heap; $o2$ is in Immortal (or static) memory. An object $o1$ cannot point to an object $o2$ in region r if: $o1$ is in the heap; $o1$ is in immortal memory; r is not active at some point during $o1$'s lifetime.

From \ To	Heap	Immortal	Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	if active

Table 1.2: Scoped-memory reference rules.

At runtime, region activity is related to the execution of computational units (e.g., methods or threads). In a single-threaded program, if each region is associated with one method, then there is a region stack where the number and ordering of active regions corresponds exactly to the appearances of each method in the call stack. In a multi-threaded program, where regions are associated with threads and methods, there is a region tree whose branches are related to each execution thread.

We adopted this kind of memory management mechanism because it is useful to overcome the problem of predictability of garbage collector (the same motivation

that made the real-time and embedded community adopt similar approaches like the RTSJ) but also because it imposes an order in the allocation and the deallocation of objects we will leverage to predict memory requirements. In particular, we assume that, at method invocation, a new region is created which will contain all objects captured by this method. When it finishes, the region is collected with all its objects. We will call this kind of region associated with a method an m-region. An implementation of scoped memory following this approach is described in chapter 3.

As mentioned, the main contributions of this thesis are related with automatic prediction of quantitative memory requirements. We also present some interesting results for the automatic generation of scoped-memory based Java code.

1.5.1. Inferring method regions

Programming using region-based allocation is very difficult [PFHV04], as it makes it impossible to reuse any old code (even the Standard Library has to be fully rewritten), and it forces the programmer to adopt new coding habits and to reason in a new paradigm quite different from Java.

In Fig. 1.5 we show a graph representing part of the heap space. Each box in the graph represents the potential objects created at the indicated program location. The first graph shows how objects point-to each other. The second graph represents the same heap but organized in m-regions. Regions are associated with method lifetimes. Thus, the lifetime of region $M0$ is longer than the lifetime of region $M1$ which is longer than $M2$. In this particular case, the organization does not respect the scoping rules since long-lived objects represented by the creation site $m0.1.m1.5.m2.6$ in region $M1$ may point-to short lived objects represented by creation site $m0.1.m1.5.m2.8$ in region $M2$. This may lead to a dangling reference because region $M2$ is freed before region $M1$. The third graph shows an organization which respects the scoping rules and can run safely. Notice that to solve the problem we enlarge the lifetime of objects represented by the creation site $m0.1.m1.5.m2.8$ by moving it to an outer region.

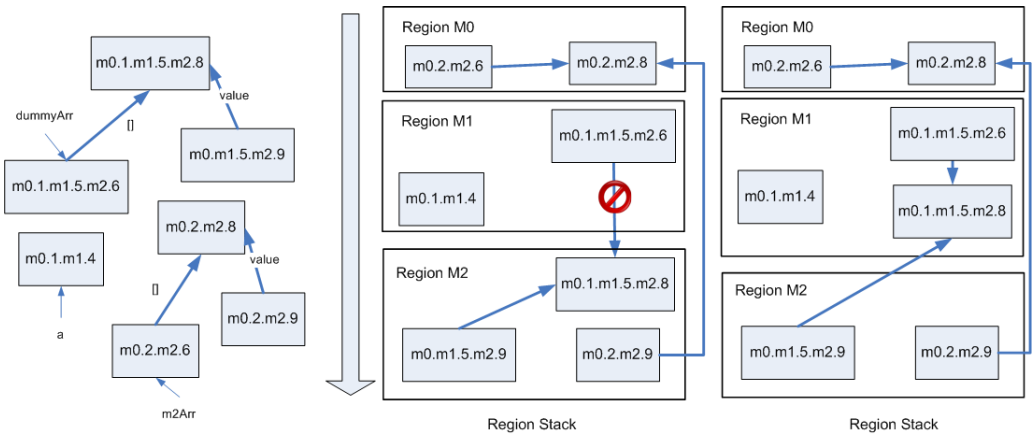


Figure 1.5: First: a graph showing the points-to relation between objects. Each box represents the potential set of objects created at that location. Second: An invalid region assignment because long lived objects may refer to short lived objects. Third: A valid region assignment.

In order to take advantage of the region-based memory model without having to suffer from the above mentioned difficulties that arise from the manual operation of regions, we propose to automatically infer memory region from program code based on escape analysis [BGY04] (see chapter 3).

Intuitively, an object *escapes* a method m when its lifetime is longer than m's lifetime. It cannot be safely collected when this unit finishes its execution. An object is *captured* by the method m when it can be safely collected at the end of the execution of m.

It is possible to synthesize a memory organization that associates a memory region (called m-region) with each method m in such a way that scoped-memory restrictions (like RTSJ) are fulfilled by construction. It can be done by allocating in each m-region the object that are captured by its associated method.

Example 1.6. Using escape analysis we can infer the creation sites that escape and are captured by $m0$, $m1$, and $m2$ in the example presented in Fig. 1.4. The objects referred to by the allocation site $m2.9$ do not escape the scope of $m2$ provided they are referenced from outside $m2$ by any parameter or return value.

The objects referred-to by the allocation site $m2.6$ are pointed-to by `arrB` which is returned by $m2$ escaping its scope. The same happens with the objects referred-to by $m2.8$ which are pointed-to by `arrB[i]` which is referenced-to by `arrB`.

The object referred-to by $m1.4$ is allocated in method $m1$ and is not referenced from outside. Since `dummyArr` refers to the objects returned by $m2$ this variable has references to objects referred-to by $m2.6$ and $m2.8$. Since `dummyArr` is a local variable not referenced by a variable or field reachable from outside, those objects do not escape the scope of method $m1$. In fact, all objects reachable from $m1$ are captured by this method.

The same procedure is applied to $m0$. The resulting escape and capture information is the following:

$$\begin{aligned} \mathit{escape}(m0) &= \{\} \\ \mathit{capture}(m0) &= \{m0.2.m2.6, m0.2.m2.8\} \\ \mathit{escape}(m1) &= \{\} \\ \mathit{capture}(m1) &= \{m1.4, m1.5.m2.6, m1.5.m2.8\} \\ \mathit{escape}(m2) &= \{m2.6, m2.8\} \\ \mathit{capture}(m2) &= \{m2.9\} \end{aligned}$$

Using this information we can safely infer the following regions:

$$\begin{aligned} \mathit{region}(m0) &= \{m0.2.m2.6, m0.2.m2.8\} \\ \mathit{region}(m1) &= \{m1.4, m1.5.m2.6, m1.5.m2.8\} \\ \mathit{region}(m2) &= \{m2.9\} \end{aligned}$$

□

1.5.2. An API for a Region-Based memory manager

In order to perform scoped-memory management at program level, we propose an API where memory scopes are bound to methods (m-regions).

The API is shown in Table 1.3. This API has constructs to create and destroy m-regions and proposes a registration mechanism to inform the memory manager that an m-region wants to allocate a set of objects in its region. At object creation,

the object identifies itself by presenting its id. The memory manager checks whether that id is registered by one m-region. Then, the manager allocates the object in the last one that register the object or, by default, in the active region. More details can be found in chapter 3.

`enter(r)`	push r into the region stack
`enter(r, CS)`	push r into the region stack and indicate that creation sites identified by $l \in CS$ have to be allocated in r
`exit()`	collect the objects in top region
`current()`	return the top region
`determineAllocationSite(CS)`	indicate that creation sites identified by $l \in CS$ have to be allocated in the current region
`newInstance(l,c)`	create an object of class c identified by l
`newAInstance(l,c,n)`	same but for arrays of dimension n

Table 1.3: Scoped-memory API.

1.5.3. Escape Analysis

The idea, as already said, is to apply pointer and escape analysis techniques (e.g., [SR01, Bla03, SYG05, BFGL07a]) to the conventional program to synthesize scopes [SYG05].

In this work we extend two existing points-to and escape analysis techniques: a lightweight but less precise analysis and a more precise but expensive technique.

A simple escape and fast escape analysis for region inference

In 4 we extend an algorithm for escape analysis inspired by Gay and Steensgaard work [GS00]. One of the original objectives of G&S's analysis was to determine which objects can be allocated in the stack. Our goal is to determine in which regions to allocate objects.

The original analysis computes two boolean properties for each local variable u of reference type. The property $escaped(u)$ is true if the variable holds references that may escape due to assignment statements or a throw statement. The property $returned(u)$ is true if the variable holds references that escape by being returned from the method in which u is defined. Other properties are introduced to identify variables that contain freshly allocated objects and methods returning freshly allocated objects. $vfresh(u)$ returns a Java reference type if u is assigned to a freshly allocated object and $mfresh(m)$ is a boolean indicating whether a method m returns a fresh object. Once these properties are computed the analyzes determines (for each method) which object can be allocated on the stack by checking if fresh object are referenced only by variables that do not escape.

We perform basically two extensions. One is focused on improving the precision when computing $escape(u)$ by performing an interprocedural analysis and by computing a local points-to graph to keep track of some points-to information (for instance to be able to determine to which object an expression like $u.f$ may refer to).

The second extension consist in computing for each variable u where objects pointed to by u live. We call this property $side(u)$ where $side(u) =$ `INSIDE` means that objects pointed to by u are captured by m and can be allocated in m's region

or if they are created by callees, m can ask for them to be allocated in its region. On the contrary $side(u) = \texttt{OUTSIDE}$, means that objects pointed to by u live longer than m. If they are created by m, they must be allocated outside its stack frame.

The analysis is a tradeoff between performance and precision. It is more precise that the Steensgaard's escape analysis because it actually computes more properties but trying to keep its simplicity and locality. The main sources of imprecision are because the analysis is flow and context insensitive, and because of the decision of keeping only local information in the points-to graphs. Nevertheless, the analysis is precise enough to determine the same regions we showed in example 1.6 for our example of Fig. 1.4 (see Table 1.4) but in general it tends to be conservative in the sense that most objects go to regions that have a larger lifetime.

loc	var	escape	def	points-to graph	side
$m0.2$	m2Arr	BOTTOM	RETVAL	$[m2.6 \rightarrow m2.8]$	INSIDE
$m1.4$	a	BOTTOM	NEW	$[m1.4]$	INSIDE
$m1.5$	dummyArr	BOTTOM	RETVAL	$[m2.6 \rightarrow m2.8]$	INSIDE
$m2.6$	arrB	RETURNED	NEW	$[m2.6 \rightarrow m2.8]$	OUTSIDE
$m2.8$	tmp	FIELD	NEW	$[m2.8]$	OUTSIDE
$m2.9$	c	BOTTOM	NEW	$[m2.9]$	INSIDE

Table 1.4: Output of our escape analysis for the example given in Fig. 1.1

Program	Lines	Allocation sites	INSIDE		G&S's analysis
			variables	sites	*stackable* variables
bh	1128	41	34	21	23
bisort	340	10	7	7	7
em3d	462	26	13	11	11
health	562	28	18	13	10
mst	473	16	8	8	7
perimeter	745	13	7	7	7
power	765	21	9	9	5
treeadd	195	11	6	6	6
tsp	545	12	7	7	7
voronoi	1000	35	34	20	31

Table 1.5: Analysis results

Table 1.5 presents the results of our algorithm on the Jolden benchmarks [CM01] comparing with the original G&S's analysis [GS00]. The first two columns are the size of the program in lines, and the number of allocation sites. The last three columns give the number of `INSIDE` variables and allocation sites, as computed by our algorithm, and the number of *stackable* variables, as computed by our implementation of G&S's analysis [GS00]. Information about the time spent by the analysis can be found in Table 4.1.

Our analysis is more precise than [GS00] as it subsumes all its rules. That is, all *stackable* variables in the sense of [GS00] are `INSIDE` variables, but the converse is not true.

In our experiments, we did not use any inlining of analyzed code. As noted in [GS00], both analyses will benefit from method inlining.

It is worth to mention that analysis is currently implemented in our tool (see Fig. 1.1). More details about the technique can be found in chapter 4.

A more precise but expensive points-to, escape and purity analysis

In chapter 5 we extended a well known points-to and escape analysis by Salcianu and Rinard [SR05]. This is a flow sensitive, interprocedural analysis that computes for each method a summary points-to graph and information about write effects. This analysis is more precise than our previous analysis but it is more expensive

because the points-to information is transferred from callees to callers leading to a more costly interprocedural analysis and larger points-to graphs.

The original analysis to be precise requires a detailed call graph and to be able to analyze all methods reachable by the application. The technique is not very precise in dealing with *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).

We extend the analysis to increase the precision for calls to non-analyzable methods[3] (see chapter 5). For such methods, we introduce extensions that model potentially affected heap locations. We also propose an annotation language that increases the precision of a modular analysis.

The annotation language allows concise specification of points-to and read/write effects. They are applied at the interface level. For instance we can declare that objects returned by the method are fresh, the objects pointed-to by a parameter may escape in some way, or some restrictions on the effects as objects pointed by a parameters are readonly or can be written by some particular objects.

At analysis time, when a non-analyzable method is called, the analysis trusts the provided annotations. Later, when the code of the non-analyzable code is available, it is analyzed to verify if it complies with its annotations.

Our initial experiments show that adding a small amount of annotations in the most commonly used libraries actually increases the precision of the analysis The experiments (see section 5.4) were focused in evaluation the of precision of the technique in inferring purity of methods, but this precision relies on the ability of the analysis in computing accurate points-to and escape information. For the example we annotate some typical library classes, like collections, iterators, and most frequently accessed methods like `equal`, `hashCode`, etc. Using annotations we were able to infer (and to prove were an annotation was given) the purity of more methods and allows us, in some cases, to improve the speed of the analysis by performing only the intraprocedural analysis relying on annotations in case of methods calls.

Although, in this work we focus on computing method purity the analysis can be used to compute m-regions.

1.5.4. Tool support for region editing and program transformation

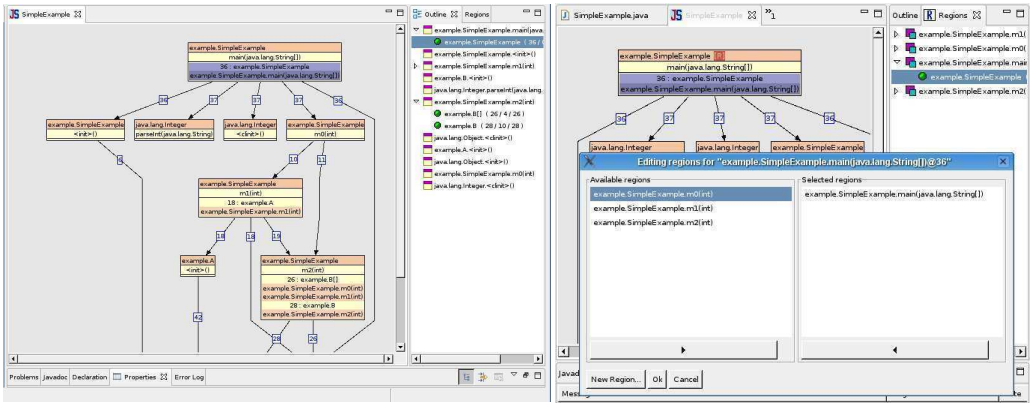


Figure 1.6: On the left: the callgraph browser window. On the right: the Region Manager.

[3]This work has been done as part of an internship at Microsoft Research with the original goal of having a points-to and effects analysis to check for method purity.

Determining precise object lifetime is undecidable. This forces static analysis to be conservative, which in practice may lead to overly large regions. Therefore, we advocate a semi-automatic approach by making the programmer participate in the analysis and the transformation process. For this, we have developed `JScoper` [FGB+05] (see chapter 6), an Eclipse plug-in providing visualization, navigation and editing of the results generated at different stages of the process (call graph generation, escape analysis, region synthesis, program instrumentation, etc.).

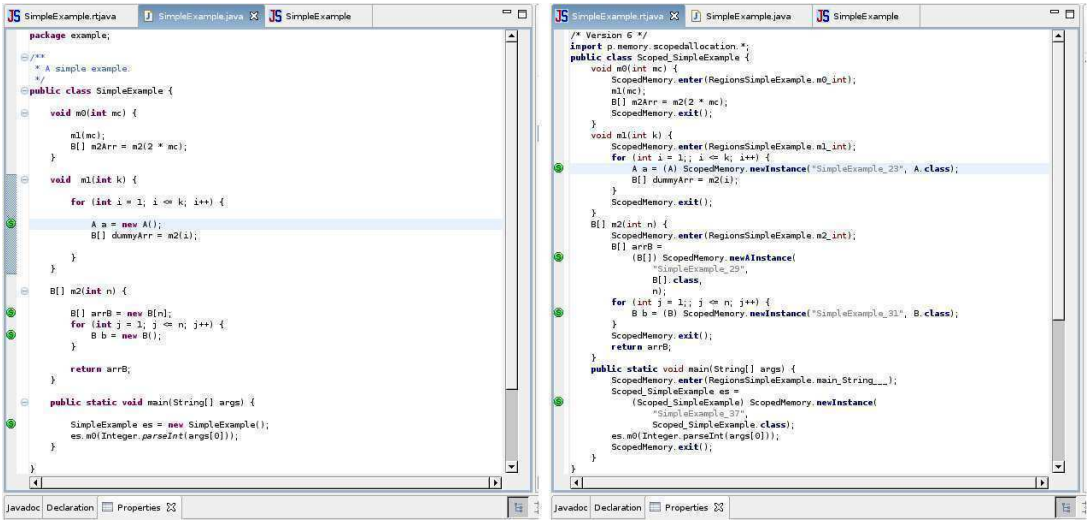


Figure 1.7: A side by side view of the two code editors. Left: the standard Java Editor. Right: the Scoped-Memory Java Editor.

`JScoper` provides a graphical interface to display call graph information enriched with information about allocation sites (see left picture on Fig. 1.6). This can be used for program understanding but more importantly to manipulate memory regions. Regions can be created or edited by moving allocation sites from the call graph to a region or by moving them between from one region to another (see right picture on Fig. 1.6). Of course manual region editing may be unsound as an invalid assignment of an allocation sites to a short-live region may produce dangling reference. Nevertheless, we assume that the programmer knows what he/she is doing or simply wants to experiment with the effect of moving some object from one region to another in terms of memory consumption, performance, etc.

Once regions are defined, the tool runs the code generator that basically instrument by inserting the corresponding calls to an API for region-based memory management (see chapter 3). The tool provides syntax highlighting for the generated code and special links that allow the user to move back and forth to the original code (see Fig. 1.7) and the application call graph.

Fig. 1.8 shows the region-based code that is automatically generated for the example in Fig. 1.4. The generated code uses the API presented in 1.5.2 to allocate objects in regions. More details about the tool can be found in chapter 6.

1.5.5. Computing region sizes

Recall that, we have fixed the memory model to a region-based one where regions are associated with the lifetime of methods and we have proposed a technique to infer those memory m-regions relying on escape analysis. In this setting, m-regions are defined by the set creation sites that are captured by a m. Thus, the size of the m-region is directly associated with the size and the number of objects it captures.

```
class CSs {
   public static final String m1_4 = "m1.4";
   public static final String m2_6 = "m2.6";
   public static final String m2_8 = "m2.8";
   public static final String m2_9 = "m2.9";
}

class Regions {
   public static final Region rm0 =
      new Region("m0",new String[] {
          CSs.m2_6, CSs.m2_8} );
   public static final Region rm1 =
      new Region("m1",new String[]
          {CSs.m1_4, CSs.m2_6, CSs.m2_8} );
   public static final Region rm2 =
      new Region("m2",new String[]
          {CSs.m2_9} );
}
public class TestIntroRegions {
   void m0(int mc) {
      ScopedMemory.enter(Regions.rm0);
      m1(mc);
      B[] m2Arr = m2(2 * mc);
      ScopedMemory.exit();
   }
   void m1(int k) {
      ScopedMemory.enter(Regions.rm1);
      for (int i = 1; i <= k; i++) {
         A a =(A)ScopedMemory.
               newInstance(CSs.m1_4, A.class);
         B[] dummyArr = m2(i);
      }
      ScopedMemory.exit();
   }
   B[] m2(int n) {
      ScopedMemory.enter(Regions.rm2);
      B[] arrB = (B[])ScopedMemory.
               newAInstance(CSs.m2_6, B.class,n);
      for (int j = 1; j <= n; j++) {
         arrB[j - 1] = (B)ScopedMemory.
                  newInstance(CSs.m2_8, B.class);
         C c = (C)ScopedMemory.
             newInstance(CSs.m2_9, C.class);
         c.value = arrB[j - 1];
      }
      ScopedMemory.exit();
      return arrB;
   }
}
```

2

Figure 1.8: Instrumented version of the example of Fig. 1.4

Since our memory utilization analysis uses creation sites as input to its algorithm, we can reuse the same technique to obtain parametric upper-bounds of region sizes by simply applying the technique to the set of creation sites captured by a method.

Notice that the prediction is parametric in terms of method parameters. That means, the size of the region will not be a fixed value and it will vary according to the calling context, matching the real memory region size. In a similar way, we can compute the amount of memory that escapes the method and has to be collected by methods that precede it in the call stack.

Example 1.7. For the example presented in Fig. 1.4 and using the synthesized capture information (see example 1.6), the size of regions for $m0$, $m1$, and $m2$ can be approximated as:

$$\begin{aligned}
\texttt{memCaptured}(m0)(mc) &= \mathcal{S}(m0, m0.2.m2.6)(mc) + \mathcal{S}(m0, m0.2.m2.8)(mc) \\
&= (\texttt{size}(B[]) + \texttt{size}(B)).(2mc) \\
\texttt{memCaptured}(m1)(k) &= \mathcal{S}(m1, m1.4)(k) + \mathcal{S}(m1, m1.5.m2.6)(k) \\
&\quad + \mathcal{S}(m1, m1.5.m2.8)(k) \\
&= \texttt{size(A)}k + (\texttt{size}(B[]) + \texttt{size}(B)).(\frac{1}{2}k^2 + \frac{1}{2}k) \\
\texttt{memCaptured}(m2)(n) &= \mathcal{S}(m2, m2.9)(n) = \texttt{size}(C).n
\end{aligned}$$

□

Some Experiments

Table 1.6 shows the polynomials that over-approximate the amount of memory captured by methods of the MST and Em3d examples from the JOlden benchmark. We show only methods that capture some creation sites. For the others, the estimation yields 0 as they do not allocate objects or they escape their scopes.

m	$\#\mathbf{CS_m}$	`memCaptured(m)`
mst		
MST.main(nv)	13	`size(mst.Graph)` + (`size(Integer)` + `size(mst.HashEntry)`) $\cdot nv^2$ + $[1/4, 0, 0, 0]_{nv}$ $\cdot$ `size(mst.Hashtable)` $\cdot nv^2$ + (`size(mst.Vertex)` + `size(mst.Vertex[])`) $\cdot nv + 5 \cdot$ `size(StringBuffer)`
MST.parseCmdLine()	2	`size(java.lang.RuntimeException)`+`size(Integer)`
MST.computeMST(g, nv)	1	`size(mst.BlueReturn)` $\cdot (nv - 1)$
em3d		
Em3d.main(nN,nD)	26	`size(em3d.BiGraph)` + $nN \cdot$ (2 $\cdot$ `size(em3d.Node)` + 4 $\cdot$ `size(em3d.Node[])` $\cdot nD$ + 2 $\cdot$ `size(double[])` $\cdot nD$) + 8 $\cdot$ `size(em3d.Node1Enumerate)` + 4 $\cdot$ `size(java.lang.StringBuffer)` + `size(java.util.Random)`
Em3d.parseCmdLine()	6	3 $\cdot$ `size(Integer)` + 3 $\cdot$ `size(java.lang.Error)`
BiGraph.create(nN,nD)	2	`size(em3d.Node[])` $\cdot nN$

Table 1.6: Capturing estimation for MST and Em3d examples.

1.6. Predicting dynamic-memory requirements

Now, we address the problem of computing memory requirements. We propose a technique to over-approximate the amount of memory *required* to run a method. Given a method we obtain a polynomial upper-bound of the amount of memory necessary to *safely* execute the method and all methods it calls, without running out of memory. This polynomial can be seen as a *pre-condition* stating that the method requires that much free memory to be available before executing, and also as a *certificate* ensuring the method is not going to use more memory than the specified amount.

As a first approach we can be tempted to try to use directly the `memalloc` estimator presented in section 1.4. However, using this technique we would obtain overly conservative upper bounds because it does not consider any kind of memory reclaiming mechanism.

Our strategy is to leverage on our knowledge about how to infer memory regions, how to compute their sizes and the fact that we know where (and when) regions are created and destroyed. Specifically, assuming our particular scoped-memory based memory model we know that objects captured by a method m are collected after it finishes its execution and escaping objects have to be collected by some other method in the call stack. Thus, an algorithm for computing dynamic-memory requirements should take into account region activations and deactivations that may occur during method execution.

To compute the amount of memory necessary to safely run a method we need to consider every potential region-stack configuration starting from the MUA and consider the largest size m-regions can get. We want this estimation to be expressed in terms of the formal parameters of MUA but the amount of memory required may depend also on the requirements of the callees which are expressed in terms of their own parameters. Therefore, there is a need of some sort of binding between MUA parameters and callees parameters. We do that binding leveraging on program invariants as we have done when computing consumption for creation sites. We call

those invariants *binding invariants* since they are control state invariants for control states finishing in the entry of a method.

Given a method *mua* we know how to compute the size of its *mua*-region (see section 1.5.5). But this is not enough: to compute the amount of memory required to run a method we need to include also the sizes of all m-regions of every method that may be called during the execution of *mua*. There are two important facts to take into account:

1. There are some region stack configurations that cannot happen at the same time.

2. Although a method can be potentially invoked several times, there will be at most one active m-region instance for m whose size may change depending of the values assigned to its parameters each time it is invoked.

To illustrate the first fact, consider the example of Fig. 1.4. In this example, method $m0$ calls to $m1$ which calls $m2$ several times. As we associated an m-region to each method there will be at most 3 active regions (i.e. $m2$ running and $m0$ and $m1$ in the stack). Later, when method $m1$ returns the control to $m0$, the only active region is an $m0$-region. Afterwards, when $m0$ calls $m2$ a new $m2$-region is activated. As it is shown in Fig. 1.9 there are some regions that cannot be active when other regions are. In this example, the regions corresponding to the call chains $m0 \xrightarrow{1} m1 \xrightarrow{5} m2$ and $m0 \xrightarrow{2} m2$ share only the $m0$-region. Since, both region stacks cannot live together, it suffices to consider the amount of memory required by the configuration that requires more space.

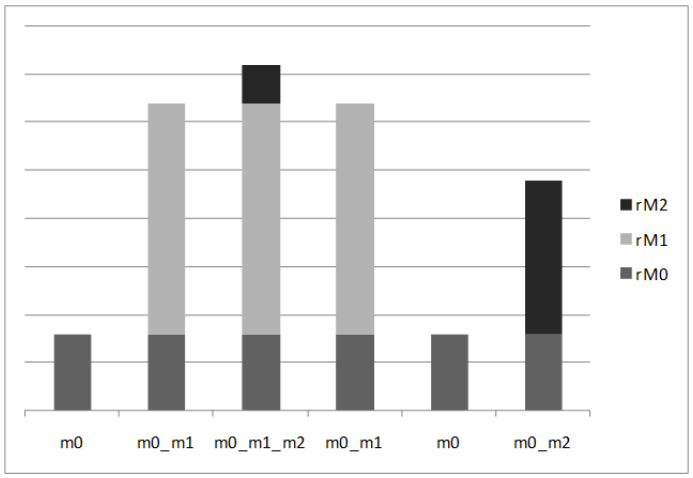


Figure 1.9: Potential regions stack configurations

Now, consider again the call chain $m0 \xrightarrow{1} m1 \xrightarrow{5} m2$. The method $m2$ will be called k times ($k = mc$) with n assigned to i ranging from 1 to k. At each invocation a new $m2$-region is created which is collected when $m2$ returns the control to $m1$. That means that it will be at most only one active $m2$-region and its size varies according to the value of n. Thus, when analyzing memory requirements it suffices to take the maximum size that a $m2$-region may reach considering the calling context given by the call chain $m0 \xrightarrow{1} m1 \xrightarrow{5} m2$.

We call $\texttt{rSize}^{\pi.m}_{mua}$ the function that yields an expression in terms of the MUA parameters of the size of the largest m-region created by any call to m with control stack π in a program starting at method *mua*. π represents the calling context used to restrict the maximization.

Suppose we can compute `rSize` for each method in each call chain. Then, to compute the amount of memory required to run a method *mua*, we basically need to consider the size of its own region and add the amount of memory required to run every method it calls. Since every call (a branch in the call graph) lead to an independent region stack, we can select the branch the would require the maximum amount of memory. This procedure is applied recursively by traversing the application call graph.
In general, this function can be defined as follows:

$$\begin{aligned}\texttt{memRq}^{\pi.m}_{mua}(p_{mua}) \quad = \quad & \texttt{rSize}^{\pi.m}_{mua}(p_{mua}) + \\ & \max\{\texttt{memRq}^{\pi.m.l.m_i}_{mua}(p_{mua}) \mid (m, l, m_i) \in edges(CG_{mua} \downarrow \pi.m)\}\end{aligned}$$

where $CG_{mua} \downarrow \pi.m$ is a projection over the path $\pi.m$ of the call graph of the program starting at method *mua* and *edges* is the set of its edges.

Note that this recursive definition lead to an *evaluation tree* (see Fig. 1.10) where leaves are related with `rSize` problems and nodes with *max* or *sum* operations. Since our objective is to evaluate this formula in run-time (i.e. when method parameters are instantiated) we would like to make the evaluation as fast as possible. That is why is it important to simplify the underlying evaluation tree as much as possible.

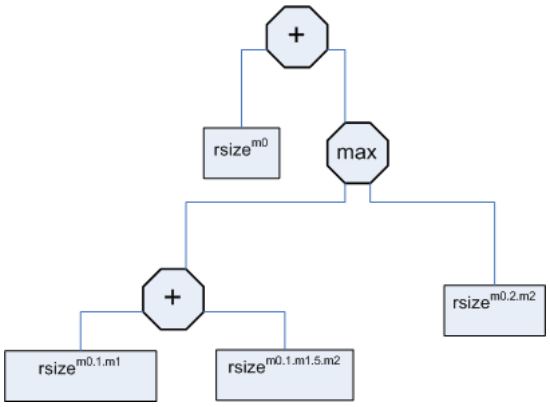


Figure 1.10: Evaluation tree for $\texttt{memRq}^{m0}_{m0}$

In order to properly define `memRq` we must rule out recursive calls. In other words, the underlying evaluation tree has to be finite[4].

Finally, in order to safely predict the amount of memory required by the MUA, we need to consider the objects that were allocated during its execution but cannot be collected when it finished. Since the escape property is absorbent is enough to consider only objects escaping the MUA (see section 7.3). Thus, we define the function that approximates memory requirements as follows:

$$\texttt{memRq}_{mua}(p_{mua}) = \texttt{memEscapes}(mua)(p_{mua}) + \texttt{memRq}^{mua}_{mua}(p_{mua})$$

Example 1.8. Assume we call $\texttt{rSize}^{m0\ldots m'}_{m0}$ to the size of the maximum m'-region in terms of $m0$ for a control stack given by a path starting from $m0$ and finish in a method m'. We can compute the size of memory required to run $m0$ as follows:

[4] Mutually recursive methods have to be removed by program transformation or provide one requirement specifications for every strongly connected component in the call graph (i.e. treat every set of mutually recursive methods as one method)

$$\begin{aligned}\texttt{memRq}_{m0}(mc) &= \texttt{memEscapes}(m0)(mc) + \texttt{rSize}_{m0}^{m0}(mc) \\ &\quad + max\{\texttt{rSize}_{m0}^{m0.1.m1}(mc) + \texttt{rSize}_{m0}^{m0.1.m1.5.m2}(mc), \\ &\qquad \texttt{rSize}_{m0}^{m0.2.m2}(mc)\}\end{aligned}$$

□

1.6.1. Maximizing region memory sizes

As mentioned, we need to model the fact that the size of every m-region may vary according to its calling context. Thus, for every method m' reachable from the MUA, we need to get an expression that represents the maximum size an m'-region may reach restricted by a call chain π starting from the MUA (`rSize`). As we did for the technique presented in 1.5.5, we use invariants to bind the method m' parameters with the MUA parameters (e.g. $m0$ in the example) and to constrain the valuation of variables according to the calling context.

Let $mua...m'$ a path starting from mua finishing in a method m'. We can model the size maximum region as follows:

$$\begin{aligned}\texttt{rSize}_{mua}^{mua...m'}(P_{mua}) &= \text{Maximize } \texttt{memCaptured}(m')(P_m) \\ &\quad \text{subject to } I_{mua}^{mua...m'}(P_{mua}, P_m, W)\end{aligned}$$

$\texttt{memCaptured}(m')$ (the size of an m'-region) is a polynomial in terms of m' parameters and the invariant $I_{mua}^{mua...m'}$ binds m' parameters with the MUA parameters[5].

This formula characterizes a non-linear maximization problem whose solution is an expression in terms of MUA parameters. Since our goal is to avoid expensive run-time computations we need to perform off-line reduction as much as possible at compile time. Off-line calculation also means that the problem must be stated parametrically.

To solve this parametric maximization problem we resort to an approach based in a technique presented by Clauss [CT04] which proposes the use of the Bernstein expansion [Ber52, Ber54] for handling parameterized multivariate polynomial considered over a parametric polyhedron. Roughly speaking, given a polynomial and a restriction given by a parametric polyhedron the technique provides a set of polynomials candidates which bound the original polynomial in the domain given by the parametric polyhedron. The most interesting aspect of the technique is that obtained polynomials are in terms of the parameters of the restriction.

In our case, the polynomial is given by the `memCaptured` estimator (see section 1.5.5) and the restriction is given by a binding invariant. The maximization problem can be therefore solved by picking the maximum Bernstein coefficient.

Example 1.9. For our example, the expression for `rSize` for each possible region

[5] Actually the invariant binds the parameters of all the sequence of methods that appears in the call chain.W are local variables appearing in the other methods in the call chain.

is:

$$\begin{aligned}
\texttt{rSize}_{m0}^{m0.2.m2}(mc) &= \text{Maximize } \big(\texttt{size}(C).n\big) \text{ subject to } \{n = 2mc\} \\
&= (\texttt{size}(C))2mc \\
\texttt{rSize}_{m0}^{m0.1.m1.5.m2}(mc) &= \text{Maximize } \big(\texttt{size}(C).n\big) \text{ subject to } \{k = mc, 1 \leq i \leq k, n = i\} \\
&= (\texttt{size}(C))mc \\
\texttt{rSize}_{m0}^{m0.1.m1}(mc) &= (\texttt{size}(B[]) + \texttt{size}(B)).(\frac{1}{2}mc^2 + \frac{1}{2}mc) + \texttt{size(A)}mc \\
\texttt{rSize}_{m0}^{m0}(mc) &= (\texttt{size}(B[]) + \texttt{size}(B))2mc
\end{aligned}$$

□

Once we are able to obtain a bound of every region in terms of the MUA, the problem of computing the memory requirement is reduced to traverse all paths starting from the MUA and applying basically *max* and *sum* operations.

Example 1.10. Let's assume (for simplicity) that the size of all types is 1. In this case, the amount of memory required to safely run $m0$ is:

$$\begin{aligned}
\texttt{memRq}_{m0}(mc) &= \texttt{memEscapes}(m0)(mc) + \texttt{rSize}_{m0}^{m0}(mc) \\
&\quad + max(\texttt{rSize}_{m0}^{m0.1.m1.5)}(mc) \\
&\qquad + \texttt{rSize}_{m0}^{m0.1.m1.5.m2}(mc), \texttt{rSize}_{m0}^{m0.2.m2}(mc)) \\
&= 0 + 2(2mc) + max(2(\frac{1}{2}mc^2 + \frac{1}{2}mc) + 1mc + 1mc, 2mc) \\
&= 4mc + max\{mc^2 + 3mc, 2mc\} \\
&= mc^2 + 7mc
\end{aligned}$$

□

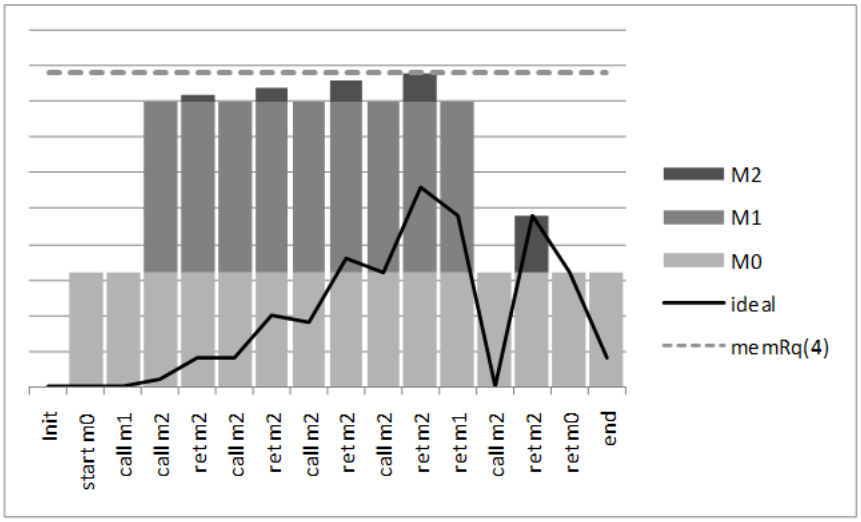


Figure 1.11: Computed memory requirements against actual memory consumption

In Fig. 1.11 we show a run of the example of Fig. 1.4 together with the evolution of the size of regions for $m0$, $m1$ and $m2$ and the computed approximation of the

amount of memory required to safely run $m0$, called $memRq$ in the figure. The prediction is accurate for our region-based memory management since the expression exactly matches the actual value that an execution using m-regions will require to run.

In this case there is an overhead in the memory usage that comes from the use of m-regions instead of using a more aggressive collection mechanism (represented by *ideal* in the figure). The overhead comes from the fact that in the $m0$-region we reserve the amount of memory necessary to allocate the objects escaping from $m2$. However, that space is only needed when $m0$ calls $m2$ near the end of its execution. This overhead is produced because the granularity of the regions is at the method level.

Notice that this approach obtains safe bounds even for other memory reclaiming mechanisms (see chapter 7). The intermediate region inference generation adds an additional level of over-approximation.

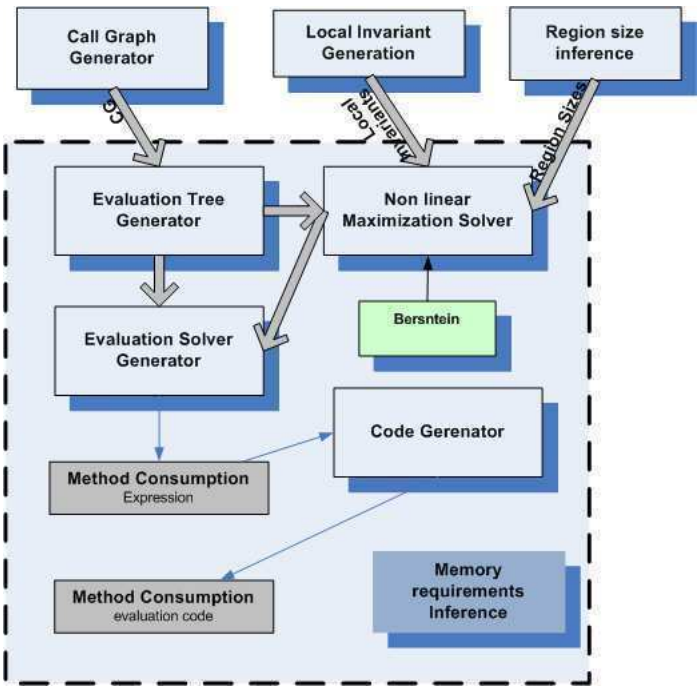


Figure 1.12: Components of our approach for predicting memory requirements

In Fig. 1.12 we show the main components of a tool that computes the memory requirements. As we have mentioned, we generate an evaluation tree by traversing all methods reachable from the MUA by following the application call graph. To compute $\texttt{rSize}^{\pi}_{mua}$ we need to provide invariants which are obtained using the same ideas we show in 1.4.2. The evaluation tree can be simplified off-line in order to try to reduce the size of the memory requirements expressions. Finally, this expression can be translated to code for evaluation in runtime.

1.6.2. Some Experiments

The initial set of experiments were carried out on a subset of programs from JOlden [CM01] benchmarks. Again we only select programs that are not recursive or the set of recursive methods is treatable by eliminating the recursion or by proposing consumption specifications.

In order to make the result more readable, we show the number of object instances created when running the selected method, rather than the actual memory

allocated by the execution of the method. Table 1.7 shows the computed peak expressions, and the comparison between real executions and estimations obtained by evaluating the polynomials. The last column shows the relative error ((#Objs - Estimation)/Estimation).

Example	`memRq`	Param.	#Objs	Estimation	Err%
`MST`(nv)	$1 + \frac{9}{4}nv^2 + 3nv + 5 + \max\{nv - 1, 2\}$	10	253	270	6%
		20	943	985	4%
		100	22703	22905	1%
		1000	2252003	2254005	0%
`Em3d`(nN, nD)	$6nN.nD + 2nN + 14 + \max\{6, 2nN\}$	(10,5)	344	354	3%
		(20,6)	804	814	1%
		(100,7)	4604	4614	0%
		(1000,8)	52004	52014	0%
`BiSort(n)`	$6 + n$	10	13	16	19%
		20	21	26	19%
		200	133	206	35%
		64	69	70	1%
		128	133	134	1%
`Power()`	32656	-	32420	32656	1%

Table 1.7: Experimental evaluation of memory requirements prediction

These experiments show that the technique produced quite accurate results, actually yielding almost exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of allocations associated with exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the bisort example, the reason of the over-approximation is that the actual number of instances is always bounded by $2^i - 1$, with $i = \lfloor log_2 n \rfloor$. Indeed, the estimation was exact for arguments power of 2.

1.7. Summary of Contributions

As we have mentioned, the most important contributions are a set of techniques in the realm of predicting memory requirements. Nevertheless, we also make some contributions in escape analysis, region inference and scope-based region management. Summarizing:

- A technique to compute parametric expressions of dynamic memory allocations (see chapter 2).
- An application of the technique to compute region sizes in a scoped memory management setting (see chapter 2).
- A region-based memory manager that allows programmers to allocate objects in regions using a simple API and a technique to automatically produce region-based code by the aid of escape analysis (see chapter 3).
- Two techniques to compute points-to and escape information which can be used to infer memory regions. One is an extension of an efficient but not very precise escape analysis ([GS00]) and the other is an extension of a precise points-to and escape analysis ([SR01]) which incorporates points-to and effects annotations to try to keep precision even in the case of call to native or virtual methods (see chapters 4 and 5).
- A tool that is able to edit memory regions and to translate conventional Java code to Java code that runs in an scoped memory management setting (see chapter 6).

- A technique that is able to predict parametric certificates of dynamic memory requirements to ensure that applications will have enough space to run (see chapter 7).

- A proof of concept tool able to compute the memory utilization certificates. The tool implements and integrate solutions for invariant discovery, escape analysis, polyhedral manipulation and non-linear optimization techniques and include other analysis techniques such as program instrumentation, dataflow analysis, inductive variable set inference, points-to analysis, call graph computation, etc. (see appendix A).

1.8. Some limitations and weaknesses of our approach

This work was one of the first ones to predict dynamic memory requirements in imperative languages[6] (see related work in section 1.9) since when we started our research, there were only very few works on the topic, mainly focused on first-order functional languages [HP99, USL03, HJ03]. Our main application domain is embedded and real-time systems where applications tend to be implemented using imperative languages and avoiding recursive method calls. We believe that the ability to compute polynomial approximation of memory consumption from program invariants is an interesting contribution to cope with the problem of quantifying memory requirements. However, our approach suffers from some intrinsic limitations and weaknesses that we would like to address in the near future (see section 8).

1.8.1. Limitations

Restrictions on the input

Our techniques cannot analyze any kind of program. Here we briefly describe some features that our techniques cannot handle.

Recursion: Our approach does not support recursive method calls. This is in principle acceptable for our application domain where programs tend to avoid recursion. However, we found this limitation an important obstacle to apply our approach to a broader spectrum of applications.

Implicit allocations: We only account for allocations made by the methods we can actually analyze. Allocations made by native methods or internal allocations made by the runtime system (virtual machine) are not considered.

Data Structures: Our analysis is better suited to deal with programs that operate with arrays and integer variables but we also showed that it can handle some iteration patterns like based on collections and iterators. However, memory consumption is not always directly related with method parameters but to some values stored in complex data structures (e.g a database, a graph, etc.) which are not always possible to model using linear invariants.

[6] Some object oriented features such as polymorphic calls are also supported if they can be solved at compile time computing a detailed call graph.

Restrictions on the output

Polynomials: We believe that one of the most important features of our technique is the generation of polynomials which are easy-to-evaluate non-linear parametric expressions. However, many programs require an exponential amount of dynamic memory which cannot be bounded by polynomials. In general, exponential memory consumption is produced by recursive programs which anyway are not supported by our approach.

1.8.2. Weaknesses

Here, we discuss some weaknesses of the approach we have followed.

Theoretical

Linear invariants: It is impossible to capture all the potential variable valuations of program locations using linear invariants. Non-linear expressions have to be ignored or approximated. That means the output of the technique will be inevitably approximated.

Complexity: Our approach relies on two techniques that manipulate polyhedra and polynomials. The first technique is Ehrhart [Ehr77] used to count the number of solutions of invariants. It first implementation [Cla96] was exponential in the number of variables but nowadays a polynomial solution exists [VSB+04]. The second technique is Bernstein [Fer06, CFGV06] which computational complexity has not been determined yet, although we found it quite efficient in practice. However, both techniques are theoretically computationally expensive in terms of the number of variables. Moreover, the number of times they are called is related with the number of paths in the application call graph. For instance, we call Ehrhart for every creation site where the number of creation sites is determined by the number of paths that lead to allocation sites. In a similar fashion, we call Bernstein for every call chain that leads to a region. Theoretically, the number of paths in a graph can be exponential. Nevertheless, we found that in practice, the number of paths in call graphs is usually not large.

Practical

Multiple sources of imprecision: Our techniques rely on obtaining program invariants to count visits to creation sites and to constrain method calls (binding invariants). We assume they can be automatically inferred or manually attached by programmers to allocation and call sites using appropriate annotations[7]. We implemented tools (see section 1.4.2) to automatically compute them. However, automatically generated invariants can be too imprecise and would require manual intervention in order to improve their quality. This can be a burdensome task for real-world applications. Something similar occurs with the discovery of sets of inductive variables which has an impact on the precision of the counting mechanism. In the case of the computation of peak-memory requirements, another source of imprecision comes from escape analysis which is used to infer memory regions.

Moreover, our prototype tool suite (see appendix A) integrates several tools like invariant generation tools, static analyzers to compute call graphs and object lifetime

[7] In our prototype tool we use our own annotation language, but other languages like [LLP+00, BLS05] can be used as well

information, linear programming tools, polyhedra calculation, polynomial maximization, etc. Every analysis may introduce some sort of approximation impacting the final precision of our analysis and in the overall cost of our techniques.

Scalability: We need further experimentation to assess how the analysis performs in real-world applications. For Jolden and Java Grande benchmarks it took between 5 to 30 seconds to obtain memory consumption expressions. We observed that an important part of that time was spent in inferring the invariants. As mentioned, assuming invariants are already given, the cost of the analysis is directly related with the number of paths in the application call graph and the cost of Ehrhart and Bernstein implementations whose complexity is a function of the number of variables. The latter can be considerably reduced in practice by simplifying invariants using inductive variables.

The problem of the number of creation sites remains since it is related to the number of paths leading to allocations sites which is exponential in the number of methods in the worst case. Although it was not an issue in the (small) set of benchmarks we have analyzed, it may cause problems in real-world applications. To cope with this, the idea would be to resort to a more modular approach (see 8.2).

1.9. Related Work

There has been a lot of work in escape analysis and region inference techniques. Some discussion about them can be found in chapters 3, 4, 5 and 6. In this section we focus on the most relevant related work regarding dynamic memory consumption analysis. Additional discussion about related work can be also found in the corresponding chapters.

Most of the related work is focused in ensuring that programs do not violate resource policies which are enforced by using an enriched type system [HJ03, CNQR05, HP99] or by using a program logic [AM05, BHMS04, CEI+07, BPS05]. We found just a few works focused in the inference of dynamic memory consumption [Ghe02, HJ03, CJPS05, USL03, AAG+07]. Most of these approaches are based in type inference [HJ03, CNQR05, HP99], by program transformation [USL] or by abstract interpretation [AAG+07].

To our knowledge, the use of program invariants to automatically synthesize method-centric parametric non-linear over-approximations of memory consumption is novel. We also believe that modeling the memory requirement as a non-linear problem and its symbolic solution using Bernstein basis is also novel. Our approach combines techniques appearing in the field of parallelizing and optimizing compilers. They are traditionally applied to works on performance analysis, cache analysis, data locality, and worst case execution time analysis [Fah98, LMC02, Cla97, Lis03]. The use of linear invariants allows us to produce non-linear expressions but keeping the manipulability of linear constrains together with their tool support and acceptable computational cost of linear programming (against other approach like Presburger or polynomial algebra).

In Table 1.8 we present a chronological overview of the works that we believe are the most relevant in dynamic memory consumption analysis. We include also our contributions to situate them in this timeline. For every work, we highlight the year of publication, the target language paradigm (functional, imperative, etc.), the main purpose of the analysis (inference, checking, verification, etc), the kind of expressions the analysis can handle, the kind of memory reclaiming mechanism that it supports and the benchmarks used to text to test the approach. It is noticeable that most

of the techniques have not been tested using well known benchmarks, specially in the case of inference techniques. In particular the most complicated benchmark used was indeed Jolden which was been applied by us and by Chin et al. [SYG05] but in their case only for verification purposes (not for inference).

Work	Year	Target Language	Purpose of Analysis	Type of Expressions	GC	Benchmarks
Hughs & Pareto [HP99]	99	Functional (ML)	Checking (Type Based)	Presburger	Regions	No
Gheorghioiu [Ghe02]	02	Imperative (Java Like)	Inference (Abs.Int.)	Non-linear	No	No
Hofman &Jost [HJ03]	03	Functional (First Order)	Inference (Type Based, Linear Prog)	Linear	Explicit	No (some examples)
Unnikrishnan et al. [USL03]	03	Functional (First Order)	Inference (Transformation)	Non-Linear (recursive function)	Reference Counting	Ad hoc (list manipulation)
Garbervetsky et al. [BGY04, BGY05]	04, 05	Imperative (Java like)	Inference (Invariants)	Non-linear	No	Jolden, Java Grande
Chander et al. [CEI+07]	05, 07	Imperative	Checking (static and dynamic, SMT)	Linear	No	No
Cachera et al. [CJPS05]	05	Imperative (Java like)	Inference (Abs.Int)	$<$ Linear	No	Ad hoc
Barthe et al. [BPS05]	05	Imperative (Java like)	Checking (SMT)	Non-linear	No	No
Chin et al. [CNQR05]	05	Imperative (MemJ)	Checking (Type Based)	Presburger	Explicit	Jolden, RegJava (translated)
Garbervetsky et al. [BGY06, BFGY07]	06, 07	Imperative (Java like)	Inference (Bernstein)	Non-linear	Regions	Jolden
Albert et al. [AAG+07]	07	Imperative	Inference (Abs.Int.)	Non-Linear (recurrence equations)	No	No

Table 1.8: Dynamic memory consumption's chronology

First, we overview the most relevant work focused in checking memory consumption by type checking or by using theorem provers and then we compare more thoroughly works that infer dynamic memory consumption.

1.9.1. Type Based Checking

In general, the idea behind type based approaches is to enforce memory consumption properties by typing rules meaning that well typed programs do not consume more memory than specified.

Hughes and Pareto [HP99] proposed a variant of ML extended with region constructs [TT97] together with a type system based on the notion of sized types [HPS96] (Presburger constrains), such that well typed programs are proven to execute within the given memory bounds given as linear constrains. Although, their work is meant for first-order functional languages, they also rely on regions to control objects deallocation.

The method proposed by Chin et al. [CKQ+05, CNQR05] relies on a type system and type annotations, similar to [HP99]. It does not actually infer memory bounds, but statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear, and to include aliasing and object deallocation information. One interesting feature is that they specify individual object deallocation which allows them to check precise bounds. We do not support that feature since, in our case that will require

the ability of inferring lower bounds, a feature that we do not support up to the moment. Nevertheless, we do support region deallocation and we can infer non-linear specifications of method consumption whereas they are limited only to linear ones.

1.9.2. Cheking using program logics

Beringer et al. [BHMS04] proposes a program logic for verifying heap consumption of a low-level imperative program which is based on a general purpose program logic for resource verification proposed by Aspinall et al. [ABH+04] designed for a proof-carrying code scenario. It basically allows the same reasoning as in their previous work that uses a linear type system [HJ03] (see later in this section) but in an imperative setting. The work presented in the paper is more focused in the formal presentation of the technique and no benchmarking is available to assess the impact of this technique in practice.

Chander et al. [CEI+07] explored combinations of static and dynamic methods by proposing a language extended with idioms for reserving and consuming resources. "Reserve" statements are checked dynamically and "consume" statements are checked statically assuming reserve statements as valid assertions. The approach requires programmers to annotate the programs with loop invariants, pre and post conditions for methods and reserve statements. The verification is performed using SMT[8] SAT solvers. The technique is presented using one interesting example (a simplified version of tar) but no benchmarking is available.

Barthe et al. [BPS05] described a technique for proving memory consumption of programs using a program logic for bytecode which is a variant of JML [LLP+00]. Roughly speaking, they introduce a special variable to the specification language that denotes the amount of memory utilized and extend the semantics of allocation statements (i.e. `new` statements) to update this variable accordingly. The ability of proving consumption predicates is constrained only by the power of the verification tool that is behind. One drawback of the technique is it lack of support of some garbage collection mechanism.

1.9.3. Memory consumption inference

The technique of Gheorghioiu [Ghe02] manipulates symbolic memory-consumption expressions on unknowns that are not necessarily parameters, but added by the analysis to represent, for instance, the number of loop iterations. The analysis basically computes a memory-consumption expression for each method by traversing its code and assigning a cost to each instructions. For method calls the analysis instantiates the pre-computed method consumption of the callee. For loops and recursive calls the analysis introduces new unknowns to model the number of times they are performed. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper-bound. Although it is difficult to analyze this work due to the lack of examples, it seems not to be suitable for programs with dynamically created arrays whose size depends on loop variables. In such cases, it yields an over approximation similar to multiplying the maximum array size by the number of loop iterations. In contrast, our approach produces more accurate estimates for dynamic memory creation inside loops. No benchmarking is available to assess the impact of this technique in practice and object deallocation is not considered.

Hofmann and Jost [HJ03] proposed a solution to obtain linear bounds on the heap space usage of first-order functional programs. Closer to our spirit, their work stat-

[8] Satisfiability Modulo Theories

ically infers, by typing derivation and linear programming, easy-to-evaluate expressions that depend on function parameters to predict memory consumption. However, their work differs from ours in many aspects. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching where memory recovery can be supported within each function, but not across functions in general. The obtained bound are *linear* bounds expression involving the sizes of various part of data. In particular, the size of the freelist required to evaluate the function is an expression on the input, while the freelist left is an expression on the result. On the other hand, our approach is meant for imperative languages with high level memory management and relies on pointer and escape analysis to infer objects lifetimes and it does not require explicit declaration of deallocations. The obtained bounds are polynomials in terms of methods parameters instead of linear expressions.

The technique proposed by Unnikrishnan et al. [USL03] computes memory requirements considering garbage collection. It consists in a program transformation approach that, given a function, constructs a new function that symbolically mimics the memory allocations of the former. The function encodes a reference counting collection mechanism. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does. In any case the cost of evaluation can be expensive and difficult to predict beforehand. Our approach computes easy-to-evaluate expressions allocations are modeled as solutions to program invariants meaning that, even in case of non-terminating programs, our analysis always finishes (returns infinite).

Cachera et al. [CJPS05] proposed a constraint-based memory analysis for a Java like bytecode. For a given program their loop-detecting algorithm can detect methods and instructions that execute an unbounded number of times, thus can be used to check whether the memory usage is bounded or not. The analysis trade precision for efficiency since it is meant to run in small embedded system such as smartcards.

More recently Alter et al. [AAG+07] propose a technique for parametric cost analysis for sequential Java code. The code is translated to a recursive representation with a flattened stack. Then, they infer *size relations* which are similar to our linear invariants. Using the size relation, and the recursive program representation they compute *cost relations* which are set of recurrent equation in term of input parameters. Applied to memory consumption the bounds that this technique is able to infer are not limited to polynomials. However, solving recurrence equations is not a trivial task and is not always possible to obtain closed form solutions for a set of recurrence equations. They outline some proposals to approximate solutions. Object deallocation is not considered.

All those works are also related with seminal works on automatic asymptotic complexity analysis such as the work of Rosendahl [Ros89] which proposes an approach based on abstract interpretation or the work of Le Métayer [Mét88] based on computing a cost function by program transformation.

We find a similar approach to us in [GBD98, ZM99]. Both use the counting technique to address the problem of finding memory size bounds in array computations for DSP and multimedia processing. However, their aim is to optimize the number of memory cells used in (potentially parallel) array intensive applications which typically involve several nested loops. Our approach was meant to deal with *dynamic* memory allocation (dynamically created object instances and *dynamic* array creation) and extended to support memory deallocation.

1.10. Thesis Structure

In chapter 2 we present the techniques related with *Dynamic Memory Utilization analysis* and for *Region size inference*. They were published in the "Journal of Object Oriented Technologies (JOT06)" [BGY06] and is based in the work presented in "Formal Techniques for Java like Languages (FTFJP05)" [BGY05] and a technical report [BGY04].

Chapters 3,4, 5 and 6 are focused in the necessary analyses and transformations from conventionally garbage-collected Java code to a new region-based Java code. In chapter 3 we present our region-based memory model, and a technique to automatically infer scoped-regions using escape analysis. We also present a technique for automated transformation of standard Java code to region-based code. This work was presented in the "International Workshop of Runtime Verification (RV04)" In chapter 4 we propose an extension of an existing escape analysis technique that make it for suitable for region inference. It was published in the "First Workshop of Abstract Interpretation for Object Oriented Languages (AIOOL05)". In chapter 5 we extend a points-to and effect analysis to support a small annotation language that enables specifications about points-to, escape, effect and ownership. It was published in "International Workshop of Aliasing, Confinement and Ownership (IWACO07)". Finally in chapter 6 we present our tool that integrates region edition and visualization with program transformation. It was presented in " eclipse Technology eXchange Workshop at OOPSLA (eTX 2005)".

In chapter 7 we present the technique developed for *Memory requirement inference*. This work is currently published as a technical report [BFGY07] and has been sent for publication. It is based on preliminaries ideas published in a technical report [BGY04] and some work we have done during a Master Thesis by Federico Fernandez [Fer06] that I co-advised where we implemented the Bernstein basis to partially solve the non-linear maximization problem of computing `rSize`.

In chapter 8 we present our conclusions and future work. Finally, in appendixes A and B we discuss some implementation details of our prototype tool and also present some details about how we deal with the problem of *Local invariant generation*.

CHAPTER 2

Computing Parametric specifications of dynamic memory utilization

In this chapter we present a static analysis for computing a parametric upper-bound of the amount of memory dynamically allocated by (Java-like) imperative object-oriented programs. We propose a general procedure for synthesizing non-linear formulas which conservatively estimate the quantity of memory explicitly allocated by a method as a function of its parameters. We have implemented the procedure and evaluated it on several benchmarks. Experimental results produced exact estimations for most test cases, and quite precise approximations for many of the others. We also apply our technique to compute usage in the context of scoped memory and discuss some open issues[1].

2.1. Introduction

The embedded and real-time software industry is leading towards the use of object-oriented programming languages such as Java. This trend brings in new research challenges.

A particular mechanism which is quite problematic in real-time embedded contexts is automatic dynamic memory management. One problem is that execution and response times are extremely difficult to predict in presence of a garbage collector. There has been significant research work to come up with a solution to this issue, either by building garbage collectors with real-time performance, e.g. [BCG04, Hen98, HIB+02, RF02, Sie00], or by using a scope-based programming paradigm, e.g. [GB00, CR04, GNYZ05, GA01]. Another problem is that evaluating quantitative memory requirements becomes inherently hard. Indeed, finding a finite upper- bound on memory consumption is undecidable [Ghe02]. This is a major drawback since embedded systems have (in most cases) stringent memory constraints or are critical applications that cannot run out of memory.

In this work we propose a novel technique for computing a parametric upper-bound of the amount of memory dynamically allocated by Java-like imperative object-oriented programs. As the major contribution, we present a technique to quantify the explicit dynamic allocations of a method. Given a method m with parameters $p_1, \ldots, p_k$ we exhibit an algorithm that computes a non-linear expression

[1] This chapter is based on the results published at the "Journal of Object Technology" (JOT) [BGY06]. A preliminary version was frist published in Formal Techniques for Java like Programs"(FTFJP'05) [BGY05].

over $p_1, \ldots, p_k$ which over-approximates the amount of memory allocated during the execution of m.

Roughly speaking, our technique works as follows. For every allocation statement, we find an invariant that relates program variables in such a way that the amount of consumed memory is a function of the number of integer solutions of the invariant. This number is given in a parametric form as a polynomial where unknowns are method parameters. Our technique does not require annotating the program in any form and produces parametric non-linear upper-bounds on memory usage. The polynomials are to be evaluated on program (or method) inputs to obtain the actual bound.

To get a flavor of the approach, consider for instance the following program:

```
void m1(int k) {
  for(int i=1;i<=k;i++)  {
     A a = new A();
     m2(i);
   }
}
```

```
void m2(int n) {
  for(int j=1;j<=n;j++) {
     B b = new B();
  }
}
```

For $m2$, our technique computes the expression $size(\texttt{B}) \cdot n$ which is the amount of allocated memory if the program starts at $m2$ [2]. For $m1$, the computed expression is $size(\texttt{A}) \cdot k + size(\texttt{B}) \cdot \frac{1}{2}(k^2 + k)$ because starting at $m1$, the program will invoke $m2$ k times and, at each invocation $i \in [1, k]$, $m2(i)$ will allocate i instances of `B`, resulting in a total amount of $\sum_{i=1}^{k} i = \frac{1}{2}(k^2 + k)$ instances of `B`, which have to be added to the k instances of `A` directly allocated by $m1$.

Combining this algorithm with static pointer and escape analyses, we are able to compute memory region sizes to be used in scope-based memory management. Given a method m with parameters $p_1, \ldots, p_k$, we develop two algorithms that compute non-linear expressions over $p_1, \ldots, p_k$ which over-approximate, respectively, the amount of memory that *escapes from* and is *captured by* m.

These techniques can be used to predict explicit memory requirements, both during compilation and at runtime. Applications are manyfold, from improvements in memory management to the generation of parametric memory-allocation certificates. These specifications would enable application loaders and schedulers (e.g., [KNY03]) to make decisions based on available memory resources and the memory-consumption estimates.

It should be noted that our analysis only copes with allocations explicitely made by a program through `new` statements in its code. The amount of "hidden" memory allocated by native methods or by the virtual machine itself cannot be quantified with this technique. This is a very important issue that deserves further research.

2.1.1. Related Work

The problem of dynamic memory estimation has been studied for functional languages in [HJ03, HP99, USL03]. The work in [HJ03] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching). The computed expressions are linear constraints on the sizes of various parts of data. In [HP99] a variant of ML is proposed together with a type system based on the

[2] For simplicity, we assume here the constructor `B()` does not allocate memory. This issue will be handled later when we present the technique in detail.

notion of sized types [HPS96], such that well typed programs are proven to execute within the given memory bounds. The technique proposed in [USL03] consists in, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed in [CKQ+05, CNQR05, Ghe02]. The technique of [Ghe02] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper-bound. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, two points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables. The method proposed in [CKQ+05, CNQR05] relies on a type system and type annotations, similar to [HP99]. It does not actually synthesize memory bounds, but statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear.

Our approach combines techniques used for performance analysis [Fah98], cache analysis [Cla97], data locality [LMC02], worst case execution time analysis [Lis03], and memory optimization [GBD98, ZM99]. To our knowledge, their use to automatically synthesize method-centric parametric non-linear over-approximations of memory consumption is novel.

Outline

In Section 2.2 we introduce useful definitions, notations, and some already developed techniques we rely on. In Section 2.3, we explain our general method for calculating memory consumption. In Section 2.4 we show our method for region-size estimation in scope-based memory management. In section 2.5 we show the results of applying our technique to some well known benchmarks. Section 2.6 discusses some extensions and future work. Section 2.7 presents some conclusions.

2.2. Preliminaries

2.2.1. Counting the number of solutions of a constraint

Let $\mathcal{I}$ be an arithmetic constraint over a set of integer variables $V = W \uplus P$ where P represents a set of distinguished variables (called parameters) and W is the remaining set of variables. We write $\mathbf{v}$, $\mathbf{p}$ and $\mathbf{w}$ to denote assignments of values to variables. $\mathcal{I}(\mathbf{v})$ is the result of evaluating $\mathcal{I}$ in $\mathbf{v}$.

$\mathcal{C}(\mathcal{I}, P)$ denotes the symbolic expression over P which provides the *number of integer solutions* of $\mathcal{I}$ for the set of variables W, assuming P has fixed values. More precisely:

$$\mathcal{C}(\mathcal{I}, P) \quad = \quad \lambda \mathbf{p}.\ \#\{\ \mathbf{w} \in \mathbb{Z}^{|W|} \mid \mathcal{I}(\mathbf{w}, \mathbf{p})\ \}$$

There are several techniques which can be used to obtain these symbolic expressions, e.g., [Cla96, Fah98, Pug94, VSB+04]. Here, we will briefly present the one described in [Cla96, VSB+04] which applies to linear constraints.

A *linear parametric set* S_P is defined as $S_P = \{\ \mathbf{w} \in \mathbb{Q}^{|W|}\ \mid\ A\mathbf{w} \geq B\mathbf{p} + \mathbf{c}\ \}$ where A and B are integer matrices, and $\mathbf{c}$ is an integer vector. S_P is called a *parametric polytope* whenever the number of points in S_P is finite for each $\mathbf{p}$.

A $|P|$-*periodic* number is a function $U : \mathbb{Z}^{|P|} \to \mathbb{Z}$ for which there exists $\mathbf{r} \in \mathbb{N}^{|P|}$ such that $U(\mathbf{p}) = U(\mathbf{p}')$ whenever $p_i \equiv p'_i \mod r_i$, for $1 \leq i \leq |P|$. The least common multiple of all r_i is called the *period* of U.

A *quasi-polynomial* in $|P|$ variables is a $|P|$-dimensional polynomial in variables over $|P|$-periodic numbers. That is, the coefficients of a quasi-polynomial depend periodically on the variables.

Ehrhart [Ehr77] showed that $\mathcal{C}(S_P, P)$ for a parametric polytope S_P, can be represented as a *quasi-polynomial*, provided S_P can be represented as a *convex* combination of its parametric *vertices*, where each vertex is an affine combination of the parameters with *rational* coefficients. This result can be extended to unions of parametric polytopes defined as $\{\ \mathbf{w} \in \mathbb{Q}^{|W|}\ \mid\ A\mathbf{w} \geq B\mathbf{p} + \mathbf{c}\ ,\ M\mathbf{w} \mod \mathbf{d} \geq \mathbf{e}\ \}$, where M is an integer matrix, and $\mathbf{d}, \mathbf{e}$ are integer vectors.

Example Consider, for instance, the linear parametric set $S^1 = \{\mathbf{w} \mid \mathcal{I}^1(\mathbf{w}, \mathbf{p})\}$, where $\mathcal{I}^1$ is defined as follows:

$$\mathcal{I}^1 \quad = \quad \{k = mc,\ 1 \leq i \leq k,\ 1 \leq j \leq i,\ n = i\}$$

where $W = \{k,\ i,\ j,\ n\}$, and $P = \{mc\}$. The corresponding Ehrhart polynomial is:

$$\mathcal{C}(S^1, mc) \quad = \quad \frac{1}{2}mc^2 + \frac{1}{2}mc$$

For the linear parametric set $S^2 = \{\mathbf{w} \mid \mathcal{I}^2(\mathbf{w}, \mathbf{p})\}$, with

$$\mathcal{I}^2 \quad = \quad \{k = mc,\ 1 \leq i \leq k,\ 1 \leq j \leq i,\ n = i,\ j \mod 3 = 0\}$$

the Ehrhart polynomial is:

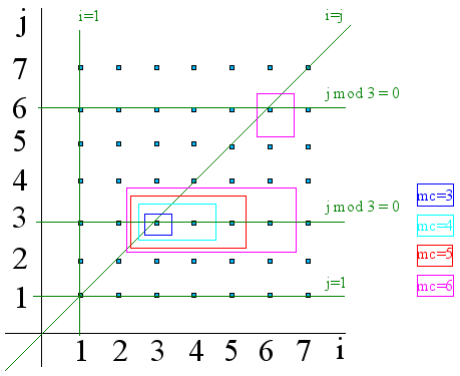
$$\mathcal{C}(S^2, mc) \quad = \quad \frac{1}{6}mc^2 - \frac{1}{6}mc + \left[0, 0, -\frac{1}{3}\right]_{mc}$$

where the period is 3 and the last coefficient of the polynomial depends periodically on mc as follows:

$$\left[0, 0, -\frac{1}{3}\right]_{mc} = \begin{cases} -\frac{1}{3} & \text{when } mc \mod 3 = 2 \\ 0 & \text{otherwise} \end{cases}$$

The following illustration depicts the result of evaluating $\mathcal{C}(S^2, mc)$ in the interval $[1, 6]$.

mc	$\mathcal{C}(S^2, mc)$
1	0
2	0
3	1
4	2
5	3
6	5



When $mc \in \{1,2\}$, the are no solutions, therefore $\mathcal{C}(S^2, mc) = 0$. For $mc = 3$, there is only one solution given by $k = mc = j = i = n = 3$ (blue box), and $\mathcal{C}(S^2, mc) = 1$. For $mc = 4$, there are two solutions ($\mathcal{C}(S^2, mc) = 2$), given by $k = mc = 4$, $j = 3$, and $i = n \in \{3,4\}$ (cyan box). For $mc = 5$, the number of solutions is three ($\mathcal{C}(S^2, mc) = 3$): $k = mc = 5$, $j = 3$ and $i = n \in \{3,4,5\}$ (red box). For $mc = 6$, the solution space is non-convex and contains $\mathcal{C}(S^2, mc) = 5$ points (magenta box). □

Several algorithms have been proposed for computing Ehrhart polynomials. The first one is discussed in [Cla96]. This algorithm is not complete and has exponential-time complexity, even when the number of variables in the inequalities is fixed. This happens because the periods are only bounded by the values of the coefficients in the linear inequalities of the input. A more efficient algorithm proven to have polynomial-time complexity for fixed dimensions has been developed in [VSB+04]. Still, the output polynomials can be relatively large in some *degenerate* cases. Recently, a fast algorithm for computing Ehrhart polynomials that over-approximate $\mathcal{C}(S_P, P)$ has been proposed in [Mei04]. All these algorithms are implemented in the Polyhedral Library `PolyLib` [Pol] used in this article. Computing Ehrhart polynomials is quite involved as it resorts to very technical results in discrete mathematics which are out of the scope of this work. The interested reader is referred to [Cla96, Mei04, VSB+04] for a detailed explanation.

2.2.2. Notation for Programs

We define a program as a set $\{m_0, m_1, \ldots\}$ of methods. A method has a list P_m of parameters ($\mathbf{p_m}$ will denote the method arguments when m is called by another method m') and a sequence of statements.

Programs are sequential and non-recursive. We assume that there is no variable name clashing including formal parameters, local and global variable names. For the sake of the presentation, we assume that method parameters are of integer type. This restriction is, however, not essential as later discussed in Section 2.6.

Example In Figure 2.1 we present the program we will use throughout this work to illustrate our approach. The program creates two arrays: a (bi-dimensional) and e, whose cells can contain an Integer (`new Integer`) or an array of Integers (`newA Integer`) depending on an expression evaluated over a loop variable. □

Each statement in a program is identified with a *control location* $\ell = (m, n) \in Label =_{def} Method \times \mathbb{N}$ (a method and a position inside the method) which uniquely characterizes the statement via the *stm* mapping ($stm : Label \to Statement$). We write $\mathsf{mth}(\ell)$ to denote m.

The *call graph* $G \subseteq Method \times Label \times Method$ of a program is such that $(m, \ell, m') \in G$ whenever $\ell = (m, n)$ and $stm(\ell)$ is a method call to m'. A (finite) *path* π in G is a sequence $m_1.\ell_1.\ldots.m_k.\ell_k.m_{k+1}$, $k \geq 1$, such that $(m_i, \ell_i, m_{i+1}) \in G$. $|\pi| = k$ is the *length* of π. For $j \in [1, |\pi|]$, we define $\pi_{\ldots j}$ to be the sub-sequence $m_1.\ell_1.\ldots.m_j.\ell_j$ of π, and we write $\mathsf{ploc}(\pi, j)$ to denote the control location ℓ_j. For $j \in [1, |\pi| + 1]$, $\mathsf{pmth}(\pi, j)$ denotes the method m_j.

Example The call graph of our example is $\{(m0, 2, m1), (m0, 3, m2), (m1, 5, m2)\}$ (see Fig. 2.2). $m0.2.m1.5.m2$ is a path. For simplicity, in the examples we will only use the position of the control location rather than the label. □

```
    void m0(int mc) {
1:    Ref0 h = new Ref0();
2:    Object[] a = m1(mc);
3:    Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k) {
1:    int i;
2:    Ref0 l = new Ref0();
3:    Object[] b = newA Object[k];
4:    for(i=1;i<=k;i++)  {
5:        b[i-1] = m2(i,l);
      }
6:    Object[] c = newA Integer[9];
7:    return b;
    }
class Ref0 {
   public Object ref;
}
```

```
Object[] m2(int n, Ref0 s) {
1:    int j;
2:    Object c,d,e;
3:    Object[] f = newA Object[n]

4:    for(j=1;j<=n;j++) {
5:      if(j % 3 == 0) {
6:        c = newA Integer[j*2+1];
        }
        else {
7:        c = new Integer(0);
        }
8:      d = newA Integer[4];
9:      f[j-1] = c;
      }
10:   e = newA Integer[1];
11:   s.ref = e;
12:   return f;
    }
```

Figure 2.1: Motivating example

2.2.3. Representing a program state

For the sake of simplicity, we would not formally define program semantics. Such a formalization is given in, for instance, [Sal]. Informally, a state σ of a program in run-time is given by the values of the variables, the heap, the control location and the call stack. A program run is a sequence $\sigma_1 \ldots$ of states. Notice that, the absence of recursion and name clashing implies that mapping variable names to values is enough to model program data (i.e., no environment or data stacks are required).

A static analysis for safely estimating memory consumption requires defining an abstraction that conservatively describes program states and runs in a suitable way. In our case, this abstraction only needs to keep enough information about the program state to be able to *count* the number of times object creation statements are executed in a program run. For simplicity, we assume that counting only depends on non heap-allocated[3], integer-valued variables. Therefore, it is important to notice that the heap in a program state can be abstracted away. This is due to the fact that the points-to relationship between objects in the heap *is not relevant* for computing the amount of explicitly allocated memory, which is, indeed, equal to the size of the portion of the heap *directly* created by `new` statements in the program code.

For the purpose of the analysis, the program control state can be characterized by the control location and the call stack. A *control state* ζ is the sequence $\pi.\ell$, where $\ell \in \mathit{Label}$ is a location and π is a path to method $\mathsf{mth}(\ell)$ in the call graph G.

Example $m0.2.m1.5.m2.3$ is a control state. □

Let $\zeta = \pi.\ell$ be a control state and $\sigma_1 \ldots \sigma_t$ be a finite run such that the location of state σ_t is ℓ, and the call stack of σ_t is π. Then, there exists a set of indexes $\{i_1, \ldots, i_{|\pi|}\}$, such that the control state ζ_j of σ_{i_j} is $\pi_{\ldots j}$, $j \in [1, |\,\pi\,|]$. That is, $\mathsf{pmth}(\pi, j)$ is the method on the top of the stack in state σ_{i_j}, $\mathsf{ploc}(\pi, j)$ is the control location corresponding to the method call to $\mathsf{pmth}(\pi, j+1)$, and $\mathsf{pmth}(\pi, |\pi| + 1)$ is $\mathsf{mth}(\ell)$. We say that run $\sigma_1 \ldots \sigma_t$ *reaches* the control state ζ.

[3] We will discuss about relaxing this assumption in Section 2.6.

Example Let ζ be the control state $m0.2.m1.5.m2.3$. Consider the run $\sigma_1 \dots \sigma_{10}$ defined as: $(m0.1, \theta_1)$ $(m0.2, \theta_2)$ $(m1.1, \theta_3)$ $(m1.2, \theta_4)$ $(m1.3, \theta_5)$ $(m1.4, \theta_6)$ $(m1.5, \theta_7)$ $(m2.1, \theta_8)$ $(m2.2, \theta_9)$ $(m2.3, \theta_{10})$, where θ_i, $1 \leq i \leq 10$, record the valuations of program variables, the heap, and the call stack. We have that the $\sigma_1 \dots \sigma_{10}$ reaches ζ, the call stack of σ_{10} is the path $\pi = m0.2.m1.5.m2$, and the set of indexes $\{2, 7\}$ is such that $\zeta_1 = \pi_{\dots 1} = m0.2$ is the control state of $\sigma_{i_1} = \sigma_2$, and $\zeta_2 = \pi_{\dots 2} = m0.2.m1.5$ is the control state of $\sigma_{i_2} = \sigma_7$. These indexes correspond to the times in the run where a method yet in the call stack of state σ_{10} (i.e., $m1$ at 2 and $m2$ at 7), or equivalently, in π, has been pushed (i.e., called). □

An *invariant* for a control state ζ is an assertion over program variables (local, global and method parameters) that holds whenever such a control state is reached in any run.

Given a method m and a control state $\zeta = \pi.\ell$ such that $\mathsf{pmth}(\pi, 1) = m$, that is, π is a path in the call graph G that starts in m, $\mathcal{I}^m_\zeta$ denotes an invariant predicate for ζ. We call the pair $(\zeta, \mathcal{I}^m_\zeta)$ an *abstract state* as it is a *conservative* approximation of the possible program states at location ℓ and stack π in any run starting at method m. That is, for every run $\sigma_1 \dots \sigma_t$ starting at $(m, 1)$, that reaches ζ, $\mathcal{I}^m_\zeta(\sigma_t)$ holds.

Example Let $\zeta = m0.2.m1.5.m2.8$. The constraint $\mathcal{I}^{m_0}_\zeta$ defined by set of linear inequalities $\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ is an invariant for ζ. □

Whenever $(m, \ell, m') \in G$ (i.e., $stm(\ell)$ is a method call), we assume the invariant $\mathcal{I}^m_\zeta$, for any $\zeta = \pi.\ell$, constrains not only the values of variables local to the caller m, but also equates actual parameters (local variables of the caller m) with formal parameters (local variables of the callee m'). This assumption simplifies the presentation.

Let m, m' be two methods such that $(m, \ell, m') \in G$, $\zeta = m_1 \dots m.\ell$ and $\zeta' = m' \dots m_s.\ell_s$ be two control states, and $\mathcal{I}^m_\zeta$ and $\mathcal{I}^{m'}_{\zeta'}$ be two invariants. We have that $\zeta.\zeta'$ is a control state and $\mathcal{I}^m_{\zeta.\zeta'}$ defined as $\mathcal{I}^m_\zeta \wedge \mathcal{I}^{m'}_{\zeta'}$ is an invariant for $\zeta.\zeta'$. In words, the invariant of a control state obtained by concatenating two control states is the conjunction of the respective invariants.

Example Let $\zeta = m0.2$ and $\zeta' = m1.5.m2.8$. We have that

$$\mathcal{I}^{m0}_{m0.2} = \{k = mc\}, \quad \mathcal{I}^{m1}_{m1.5} = \{1 \leq i \leq k, n = i\}, \text{ and } \quad \mathcal{I}^{m2}_{m2.8} = \{1 \leq j \leq n\}$$

are invariants, which gives that

$$\begin{aligned} \mathcal{I}^{m0}_{m0.2.m1.5} &= \{k = mc, 1 \leq i \leq k, n = i\} \\ \mathcal{I}^{m1}_{m1.5.m2.8} &= \{1 \leq i \leq k, n = i, 1 \leq j \leq n\} \\ \mathcal{I}^{m0}_{m0.2.m1.5.m2.8} &= \{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\} \end{aligned}$$

are also invariants. □

Given a control state $\zeta = m_1.\ell_1 \dots m_k.\ell_k$, the property above provides means for computing the invariant $\mathcal{I}^{m_1}_\zeta$ as the conjunction $\bigwedge_{i=1}^k \mathcal{I}^{m_i}_{m_i.\ell_i}$. Each $\mathcal{I}^{m_i}_{m_i.\ell_i}$ is called a *local* invariant.

Example Table 2.1 shows invariants that define iteration spaces and corresponding Ehrhart polynomials for some control states starting at method $m0$. □

ζ	$\mathcal{I}_\zeta^{\mathbf{m0}}$	$\mathcal{C}(\mathcal{I}_\zeta^{\mathbf{m0}}, \mathbf{P_{m0}})$
$m0.2.\mathbf{m1.2}$	$\{k = mc\}$	1
$m0.2.m1.5.\mathbf{m2.3}$	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.2.m1.5.\mathbf{m2.6}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + [0, 0, -\frac{1}{3}]_{mc}$
$m0.2.m1.5.\mathbf{m2.7}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 > 0\}$	$\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}$
$m0.2.m1.5.\mathbf{m2.8}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$	$\frac{1}{2}mc^2 + \frac{1}{2}mc$
$m0.2.m1.5.\mathbf{m2.10}$	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.3.\mathbf{m2.3}$	$\{n = 2mc\}$	1
$m0.3.\mathbf{m2.6}$	$\{n = 2mc, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{2}{3}mc + [0, -\frac{2}{3}, -\frac{1}{3}]_{mc}$
$m0.3.\mathbf{m2.7}$	$\{n = 2mc, 1 \leq j \leq n, j \ mod \ 3 > 0\}$	$\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}$
$m0.3.\mathbf{m2.8}$	$\{n = 2mc, 1 \leq j \leq n\}$	$2mc$
$m0.3.\mathbf{m2.10}$	$\{n = 2mc\}$	1

Table 2.1: Some invariants and Ehrhart polynomials for $m0$

2.2.4. Counting the number of visits of a control state

Let $(\zeta, \mathcal{I}_\zeta^m)$ be an abstract state such that the invariant $\mathcal{I}_\zeta^m$ defines a *polyhedral iteration space* [Cla96], that is, a polytope that characterizes all possible values of loop-control variables and parameters involved in a program interation that passes through ζ.

Example Let ζ be the control state $m0.2.m1.5.m2.8$. The invariant $\mathcal{I}_\zeta^{m_0}$ defined by set of linear inequalities $\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ defines a polyhedral iteration space for ζ. □

Therefore, given an invariant $\mathcal{I}_\zeta^m$ that defines a polyhedral iteration space, it follows that counting the number of integer solutions of $\mathcal{I}_\zeta^m$ yields an expression that over-approximates *the number of times* a concrete state, whose abstraction is $(\zeta, \mathcal{I}_\zeta^m)$, is reached in a run starting at m.

Example Let $\zeta = m0.2.m1.5.m2.8$. We have that ζ is reached at most $\frac{1}{2}mc^2 + \frac{1}{2}mc$ times in a run starting at m_0 for any value of parameter mc. □

2.3. Synthesizing memory consumption

In this section we present our technique for synthesizing non-linear formulas (actually, quasi-polynomials) to conservatively over-estimate memory consumption in terms of method parameters. First, we show how to adapt the counting technique discussed in Section 2.2.4 to cope with memory allocations. Second, we show how to compute the total amount of memory allocated by a method.

2.3.1. Memory allocated by a creation site

We now focus on statements that create new objects (i.e., allocate memory): `new` and `newA` statements. We assume that those statements only create object instances

and constructors are called separately and handled as any other method call. We call *creation site*, and denote cs, a control state associated to such operations: $cs \in CS = \{\ \pi.\ell \in Label^{+} \mid stm(\ell) \in \{\texttt{new T}, \texttt{newA T}[\cdot]\dots[\cdot]\}\ \}$.

To compute the amount of memory allocated by a creation site cs we define the function $\mathcal{S}$ (see below). Given an invariant $\mathcal{I}^m_{cs}$ for cs and method m with parameters P_m, $\mathcal{S}$ computes the parametric number of visits to cs and multiplies the resulting expression for the size of the allocated object. This parametric expression over-estimates the memory allocate by cs whenever cs is a `new` statement. Nevertheless, when cs is an array allocation (i.e., `newA T`$[e_1]\dots[e_n]$), this technique needs to be slightly adapted considering the fact that an array is a collection of elements of the same type. In fact, the `newA T`$[e_1]\dots[e_n]$ statement creates the same number of instances (and, therefore, allocates the same amount of memory) as n nested loops of the form:

```
for( h_1 = 1; h_1 ≤ e_1; h_1++ )
  ...
    for( h_n = 1; h_n ≤ e_n; h_n++ )
      newA T[1]
```

whose iteration space can be described by the invariant $\bigcup_{i=1..n}\{1 \le h_i \le e_i\}$.

Thus, we define the function $\mathcal{S}$ as follows:

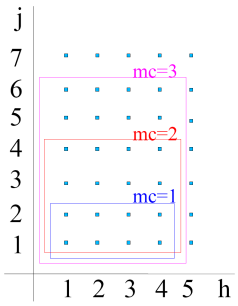
```
S(I^m_cs, P_m, cs) // returns an Expression over P_m
ℓ = last(cs); // (cs = π.ℓ)
if stm(ℓ)= new T
   res:=size(T) · C(I^m_cs, P_m);
else if stm(ℓ)= newA T[e_1]...[e_n]
   Inv_array := I^m_cs ∪ ⋃_{i=1..n}{1 ≤ h_i ≤ e_i}
   res:=size(T[]) · C(Inv_array, P_m);
end if;
return res;
```

where $size(\texttt{T})$ is a symbolic expression that denotes the size of an object of type `T`, and $size(\texttt{T[]})$ is a symbolic expression that denotes the size of a *cell* of an array of type `T` [4]. $\mathcal{C}$ is the symbolic expression that counts the number of integer solutions for an invariant as defined in Section 2.2.1.

As linear invariants are conservative, $\mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)$ *over-approximates*, in general, the amount of memory allocated by cs in any run starting at m. That is, for any run $\sigma_1 \dots \sigma_t$ that starts at m and reaches cs, the amount of memory in the heap of σ_t occupied by objects *allocated by* creation site cs is bounded by the result of evaluating $\mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)$ in the values of parameters P_m in σ_1.

Example Consider the creation site $m0.3.m2.8$, which corresponds to statement `d = newA Integer[4]` in line 8 of method $m2$ when called from $m0$ at line 3.

$$
\begin{aligned}
&\mathcal{S}(\mathcal{I}^{m0}_{m0.3.m2.8}, mc, m0.3.m2.8) = \\
&\quad = \ size(\texttt{Integer[]}) \cdot \mathcal{C}(\mathcal{I}^{m0}_{m0.3.m2.8} \cup \{1 \le h \le 4\}, mc) \\
&\quad = \ size(\texttt{Integer[]}) \cdot \mathcal{C}(\{n = 2mc, 1 \le j \le n, 1 \le h \le 4\}, mc) \\
&\quad = \ size(\texttt{Integer[]}) \cdot \mathcal{C}(\{1 \le j \le 2mc, 1 \le h \le 4\}, mc) \\
&\quad = \ size(\texttt{Integer[]}) \cdot 8mc
\end{aligned}
$$



[4] $size(\texttt{T[]})$ will be the same for all `Object` subclasses and will differ for arrays of basic types.

The figure on the right depicts the sets of points in the invariant for several values of parameter mc. □

Example Table 2.2 shows the polynomials that over-approximate the amount of memory allocated for (some selected) creation sites reachable from method $m0$. □

cs	$\mathcal{S}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{cs}}, \mathbf{P_{m0}}, \mathbf{cs})$
$m0.2.m1.2$	$size(\texttt{Ref0})$
$m0.2.m1.6$	$size(\texttt{Integer[]}) \cdot 9$
$m0.2.m1.5.m2.3$	$size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{1}{2}mc\right)$
$m0.2.m1.5.m2.6$	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}]_{mc} \cdot mc + [0, -\frac{4}{9}, -\frac{11}{9}]_{mc}\right)$
$m0.2.m1.5.m2.7$	$size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}\right)$
$m0.2.m1.5.m2.8$	$size(\texttt{Integer[]}) \cdot (2mc^2 + 2mc)$
$m0.3.m2.3$	$size(\texttt{Object[]}) \cdot 2mc$
$m0.3.m2.6$	$size(\texttt{Integer[]}) \cdot \left(\frac{4}{3}mc^2 + [2, -\frac{2}{3}, \frac{2}{3}]_{mc} \cdot mc + [0, -\frac{2}{3}, -\frac{2}{3}]_{mc}\right)$
$m0.3.m2.7$	$size(\texttt{Integer}) \cdot \left(\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}\right)$
$m0.3.m2.8$	$size(\texttt{Integer[]}) \cdot 8mc$

Table 2.2: Polynomials of memory allocation.

2.3.2. Memory allocated by a method

Having shown how to compute the amount of memory allocated by a single creation site, we determine how much memory is allocated by a run starting at method m. Basically, our technique identifies the creation sites reachable from method m, gets the corresponding invariants, computes the amount of memory allocated by each one and finally yields the sum of them.

Let $CS_m \subseteq CS$ denote the set of creation sites reachable from method m that is, the set of creation sites $cs = \pi.\ell \in CS$, where π is a path starting at m.

Example The creation sites of the example in Fig. 2.1 are:

$CS_{m0} = \{$ $m0.1$, $m0.2.m1.2$, $m0.2.m1.3$, $m0.2.m1.6$, $m0.2.m1.5.m2.3$, $m0.2.m1.5.m2.6$, $m0.2.m1.5.m2.7$, $m0.2.m1.5.m2.8$, $m0.2.m1.5.m2.10$, $m0.3.m2.3$, $m0.3.m2.6$, $m0.3.m2.7$, $m0.3.m2.8$, $m0.3.m2.10$ $\}$

$CS_{m1} = \{$ $m1.2$, $m1.3$, $m1.6$, $m1.5.m2.3$, $m1.5.m2.6$, $m1.5.m2.7$, $m1.5.m2.8$, $m1.5.m2.10$ $\}$

$CS_{m2} = \{$ $m2.3$, $m2.6$, $m2.7$, $m2.8$, $m2.10$ $\}$

Fig. 2.2 shows the call graph augmented with creation sites. This graph is automatically constructed with the tool described in [FGB+05]. □

Observe that, since we are not dealing with recursive programs, the number of paths in the call graph and thus the number of control states is finite. Now, the problem of computing a parametric upper-bound of the amount of memory allocated by a method m can be reduced to: for each $cs \in CS_m$, obtain an invariant, compute the function $\mathcal{S}$ and sum up the results.

The function `computeAlloc` computes an expression (in terms of method parameters) that over-approximates the amount of memory allocated by a selected set of creations sites:

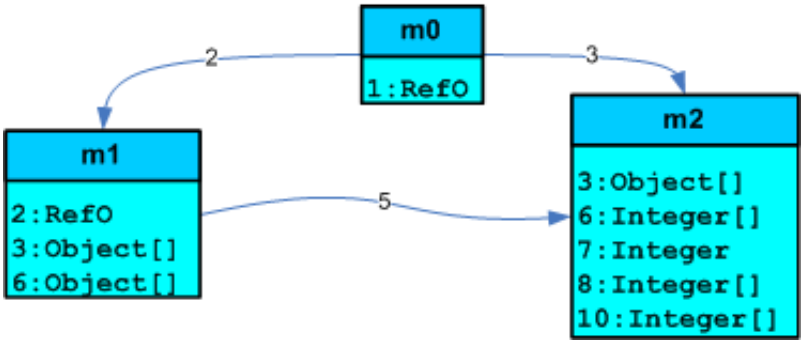


Figure 2.2: Call Graph and Creation Sites

$$\texttt{computeAlloc}(m,\ CS) = \sum_{cs\ \in\ CS} \mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)\ \text{, where } CS \subseteq CS_m$$

Given a method m, the symbolic estimator of the memory dynamically allocated by m is defined as follows:

$$\texttt{memAlloc}(m) = \texttt{computeAlloc}(m,\ CS_m)$$

That is, for any run $\sigma_1 \ldots$ that starts at m, the amount of memory, in the heap of any state in the run, occupied by objects *allocated by* a creation site in CS_m reached by the run, is bounded by the result of evaluating `memAlloc(`m`)` in the values of parameters P_m in σ_1.

Notice that the over-estimation may arise because invariants are conservative, but also as a consequence of summing up *all* creation sites reachable in the call graph, which may not all be executed by a given run.

Example Table 2.3 shows the expressions computed for $m0$, $m1$ and $m2$. □

memAlloc($m0$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{23}{6}mc^2 + ([\frac{29}{2}, \frac{71}{6}, \frac{25}{2}]_{mc}) \cdot mc + [11, \frac{83}{9}, \frac{79}{9}]_{mc}\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}m^2 + 2mc + [0, \frac{2}{3}, \frac{2}{3}]_{mc}\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + 2 \cdot size(\texttt{RefO})$
memAlloc($m1$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{5}{2}k^2 + [\frac{23}{6}, \frac{23}{6}, \frac{19}{6}]_k \cdot k + [9, \frac{77}{9}, \frac{70}{9}]_k\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + [0, 0, \frac{1}{3}]_k\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{3}{2}k\right) + size(\texttt{RefO})$
memAlloc($m2$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{3}n^2 + [\frac{16}{3}, \frac{14}{3}, 4]_n \cdot n + [2, 1, \frac{2}{3}]_n\right) + size(\texttt{Integer}) \cdot \left(\frac{2}{3}n + [0, \frac{1}{3}, \frac{2}{3}]_n\right) + size(\texttt{Object[]}) \cdot n$

Table 2.3: Memory allocated by methods $m0$, $m1$, and $m2$

The complexity of the method depends on the number of configurations of the call stack from the analyzed method to each creation site. Though this number is in the worst case exponential in the number of methods, in many cases, the topology of the call graph leads to few paths and thus the presented technique is still feasible. This actually happens for the benchmarks analyzed in Section 2.5. Further discussion on this topic can be found in Section 2.6.

Notice that, using the technique we are able to evaluate the consumption of a program starting at *any* method m. For instance, in case of a batch program it

would be reasonable to compute the consumption from the actual main method of the program since the consumption usually depends on command line arguments or contextual objects like the size of a referenced file. Nevertheless, the ability to compute consumption for any given method is useful to get different context-independent consumption specifications at a finer level of granularity. Besides, in cases where the application model is reactive event-driven, the consumption should be measured from a dispatched method according to the parameter values conveyed in the event.

2.4. Applications to scoped-memory

Scoped-memory management is based on the idea of grouping sets of objects into regions associated with the lifetime of a computation unit. Thus, objects are collected together when their corresponding computation unit finishes its execution. In order to infer scope information we use pointer and escape analysis (e.g., [Bla99, SR01]). In particular, we assume that, at method invocation, a new region is created which will contain all objects *captured* by this method. When it finishes, the region is collected with all its objects. An implementation of scoped memory following this approach can be found in [GNYZ05].

An object *escapes* a method when its lifetime is longer than the method's lifetime, and it cannot be safely collected when this unit finishes its execution. Let $escape : Method \to \mathbb{P}(CreationSite)$ be a function that given a method m returns (an over-approximation of) the set of creation sites $escape(m) \subseteq CS_m$ that escape m.

An object is *captured* by the method m when it can be safely collected at the end of the execution of m. Let $capture : Method \to \mathbb{P}(CreationSite)$ be a function that given a method m returns (an under-approximation of) the creation sites $capture(m) \in CS_m$ that are captured by m.

These functions can be computed using any escape analysis technique.

Example For instance, for our example in Figure 2.1 we have:

$$
\begin{aligned}
escape(m0) &= \{\} \\
escape(m1) &= \{m1.3, m1.5.m2.3, m1.5.m2.6, m1.5.m2.7\} \\
escape(m2) &= \{m2.3, m2.6, m2.7, m2.10\} \\
capture(m0) &= \{m0.1, m0.2.m1.3, m0.2.m1.5.m2.3, m0.2.m1.5.m2.6, m0.2.m1.5.m2.7, \\
&\quad m0.2.m1.5.m2.10, m0.3.m2.3, m0.3.m2.6, m0.3.m2.7, m0.3.m2.10\} \\
capture(m1) &= \{m1.5.m2.10, m1.2, m1.6\} \\
capture(m2) &= \{m2.8\} \quad \square
\end{aligned}
$$

2.4.1. Memory that escapes a method

In order to symbolically characterize the amount of memory that *escapes* a method, we use the algorithm developed in Section 2.3, but we restrict the search to creation sites that escape the method:

```
memEscapes(m) = computeAlloc(m, escape(m))
```

This information can be used to know how much memory the method leaves allocated in the active regions (the caller region or their parent regions in the call stack) after its own region is deallocated, or to measure the amount of memory that cannot be collected by a garbage collector after the method terminates.

Example In Table 2.4 we show the memory-consumption expressions for the creation sites escaping $m1$. Observe that expressions are defined only on the method parameters. □

$\text{memEscapes}(m1) = size(\texttt{Object[]}) \cdot k$	$m1.3$
$+size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{1}{2}k\right)$	$m1.5.m2.3$
$+size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{1}{2}k^2 + [\frac{5}{6}, \frac{5}{6}, \frac{1}{6}]_k \cdot k + [0, -\frac{4}{9}, -\frac{11}{9}]_k\right)$	$m1.5.m2.6$
$+size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + [0, 0, \frac{1}{3}]_k\right)$	$m1.5.m2.7$

Table 2.4: Amount of memory escaping from $m1$.

2.4.2. Memory captured by a method

To compute the expression over-estimating the amount of allocated memory that is *captured* by a method, we use the algorithm developed in Section 2.3, but we restrict the search to creation sites that are captured by the method:

$$\texttt{memCaptured}(m) = \texttt{computeAlloc}(m, capture(m))$$

Example Table 2.5 shows the expression that over-approximates the amount of memory captured by each method for our example. □

$\text{memCaptured}(m0) = size(\texttt{Ref0})$	$m0.1$
$+size(\texttt{Object[]}) \cdot mc$	$m0.2.m1.3$
$+size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{1}{2}mc\right) +$	$m0.2.m1.5.m2.3$
$+size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}]_{mc} \cdot mc + [0, -\frac{4}{9}, -\frac{11}{9}]_{mc}\right)$	$m0.2.m1.5.m2.6$
$+size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}\right)$	$m0.2.m1.5.m2.7$
$+size(\texttt{Integer[]}) \cdot mc$	$m0.2.m1.5.m2.10$
$+size(\texttt{Object[]}) \cdot 2mc$	$m0.3.m2.3$
$+size(\texttt{Integer[]}) \cdot \left(\frac{4}{3}mc^2 + [2, -\frac{2}{3}, \frac{2}{3}]_{mc} \cdot mc + [0, -\frac{2}{3}, -\frac{2}{3}]_{mc}\right)$	$m0.3.m2.6$
$+size(\texttt{Integer}) \cdot \left(\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}\right)$	$m0.3.m2.7$
$+size(\texttt{Integer[]})$	$m0.3.m2.10$
$= size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 + ([\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]_{mc}) \cdot mc + [2, \frac{2}{9}, -\frac{2}{9}]_{mc}\right) +$ $size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + 2mc + [0, \frac{2}{3}, \frac{2}{3}]_{mc}\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) +$ $size(\texttt{Ref0})$	Total
$\text{memCaptured}(m1) = size(\texttt{Ref0})$	$m1.2$
$+size(\texttt{Integer[]}) \cdot 9$	$m1.6$
$+size(\texttt{Integer[]}) \cdot k$	$m1.5.m2.10$
$\text{memCaptured}(m2) = size(\texttt{Integer[]}) \cdot 4n$	$m2.8$

Table 2.5: Memory captured by methods $m0$, $m1$ and $m2$

Assuming the resulting expression is a symbolic estimator of the size of the memory region associated to the method's scope, this information can be used to specify the size of the memory region to be allocated at run-time, as required by the RTSJ [GB00]. Moreover, it can be used to improve memory management algorithms.

2.5. Method Validation

We have developed a proof-of-concept tool-suite to perform the initial experiments aiming at validating our approach for Java applications. This section identi-

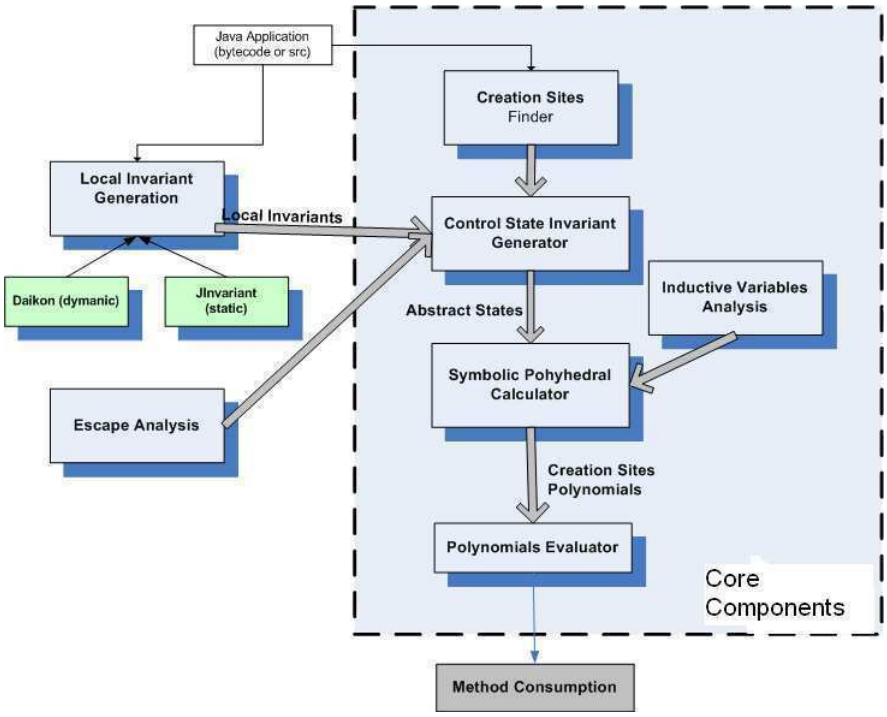


Figure 2.3: Proof-of-concept tool-suite

fies the key conceptual components of the technique, their associated challenges and briefly describes the implemented solution that was suitable to treat some well-known benchmarks.

2.5.1. Tool

The proof-of-concept architecture is shown in Fig.2.3. The tool can effectively analyze single-threaded Java programs provided they do not feature recursion or complex data structures.

Call graphs are obtained with `Soot` [VRHS+99]. Invariants can be either provided by programmer assertions "à la" JML [LLP+00], or computed using general analysis techniques [CH78, CC02] or Java-oriented ones[NE01, FL01, ECGN99, CL05]. `PolyLib` [Pol] is used to compute Ehrhart polynomials. In the experiments, local invariants were generated using `Daikon`[ECGN99]. It should be noted here that `Daikon` is a tool for dynamic dectection of "likely" invariants by executing the program over a set of test cases. Even if the properties generated by `Daikon` have a high probability of being true in all runs, that is, being invariants, they might not be. In our experiments, we have manually verified all properties to be invariants.

None of the techniques for computing invariants deal with our concept of control state invariant since they only compute local invariants. Thus, the tool builds a control state invariant by computing the conjunction of the local invariants that hold in the control locations along the path as explained in Section 2.2.

Note that the precision of our analysis depends on the accuracy of both the invariant generation and call graph generation techniques (specially in the presence of dynamic binding). Weak invariants and unfeasible calls make our technique to over-approximate too much. In section 2.6 we comment this issue in more detail.

In order to increase the precision of computed upper-bounds, it is preferable to obtain invariants that only capture what is required to be known about the relevant

iteration spaces [Cla96]. A key concept for our characterization of iteration spaces is the set of *inductive variables* for a control location, that is, a subset of program variables which cannot repeat the very same value assignment in two different visits of the given control state (except in the case where the program halts). An invariant that only involves parameters and an inductive variables is called an *inductive invariant*.

To compute inductive variables we developed a conservative dataflow analysis that combines a live variables analysis augmented with field sensitivity with a loop inductive analysis [NNH99]. This problem has been studied for programs that make use of iteration patterns composed of `for` and `while` loops with simple conditions. Handling more complex iteration patterns and types beyond integers is a challenging issue related to finding variant functions for the iteration. In section 2.6.2 we brifely discuss our general strategy and we show how the tool currently deals with an iteration pattern pervading Java applications as it is the case of looping over collections. Indeed, while not dealing with recursive programs is an underlying limitation of the approach, handling complex data-structures (such as collections) is not precluded, but is a challenge for building good linear invariants.

2.5.2. Experiments

The initial set of experiments were carried out on a significant subset of programs from JOlden [CM01] and JGrande [DHPW01] benchmarks. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications – the target application classes we had in mind when we devised the technique.

Indeed, our method faced serious obstacles when dealing with these examples. First, in most examples some of the memory-consuming methods reside into recursive structures. Second, inductive variables include not only integer-typed variables but also object fields and complex data-structures.

Despite these issues, the tool was able to synthesize very accurate and non-trivial estimators for the number of object instances created (and memory allocated) in terms of program parameters for two examples that do not feature recursion (mst and em3d examples). In all test cases, execution times were less than 30 sec. in a Pentium 4 3Ghz PC for the core components (Fig. 2.3): (1) find creation sites, and compute (2) control-state invariants, (3) inductive variables, and (4) Ehrhart polynomials. Moreover, the tool was also able to analyze most non-recursive (and tail-recursive) application methods for the rest of the examples.

All these results were achieved using the **original code** as input for the method and reducing human intervention to a minimum (i.e., creation of test cases for `Daikon`, strengthening some of automatically detected invariants and reducing some of automatically detected inductive sets). Remaining obstacles that prevent fully automatic analysis of some examples are complex data-structures which must be considered part of any set of inductive variables and thus, an integer interpretation of them should be provided by the user to build a useful linear invariant.

These experimental results focused on the allocation estimation (Section 2.3). The application of our technique to the scoped memory management (Section 2.4) needs further work.

In order to make the result more readable, the tool computes the number of object instances created when running the selected method, rather than the actual memory allocated by the execution of the method[5]. Also, we set aside analyzing the

[5]For simplicity we assume that the function `size(T)`=1 for all type `T`

standard Java library in order to keep examples manageable.

Table 2.6 shows the computed polynomials, the analysis time (of core components), and the comparison between real executions and estimations obtained by evaluating the polynomials with the corresponding values of parameters. The last column shows the relative error ((#Obs - Estimation)/Estimation).

Example:Class.Method	Static Analysis			Precision Analysis			
	$\#CS_m$	memAlloc	Time	Param.	#Objs	Estim.	Err%
mst:MST.main(nv)	13	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+4nv+5$	26.04s	10	240	245	2,00
				20	940	985	5,00
				100	22700	22905	1,00
				1000	2252000	2254005	0,09
mst:MST.computeMST(g, nv)	1	$nv-1$		10	9	9	0,00
				20	19	19	0,00
				100	99	99	0,00
				1000	999	999	0,00
mst:Graph.Graph(nv)	6	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+3nv$		10	230	230	0,00
				20	920	960	4,17
				100	22600	22800	0,88
				1000	2251000	2253000	0,09
mst:Graph.addEgdes(nv)	2	$2nv^2$		10	180	200	10,00
				20	760	800	5,00
				100	19800	20000	1,00
				1000	1998000	2000000	0,10
Em3d.main(nN, nD)	28	$6nD \cdot nN + 4nN + 14$	30.57s	(10, 5)	350	354	1,13
				(20, 6)	810	814	0,49
				(100, 7)	4610	4614	0,09
				(1000, 8)	52010	52014	0,01
Bigraph.create(nN, nD)	22	$6nD \cdot nN + 4nN + 8$		(10, 5)	348	348	0,00
				(20, 6)	808	808	0,00
				(100, 7)	4608	4608	0,00
				(1000, 8)	52008	52008	0,00
Node.makeFromNodes	2	$2 \cdot this.fromCount$		10	20	20	0,00
				20	40	40	0,00
				100	200	200	0,00
				1000	2000	2000	0,00
Tree.createTestData(nb)	23	$17nb + 26$	7.22s	10	196	196	0,00
				20	366	366	0,00
				100	1726	1726	0,00
				1000	17026	17026	0,00
Value.createTree(size,sd)	1	$size-1$	2.74s	10	7	9	22,22
				20	15	19	21,1
				200	127	199	36,2
				64	63	63	0,0
				128	127	127	0,0
				256	255	255	0,0
power:Root.<init>	14	32622	5.82s	-	32412	32622	0,64
(*)health: (recursive) Village.createVillage(l, lab, b, s)	8	$11(4^l-1)/3$		2	55	∞	∞
				4	935	∞	∞
				6	15015	∞	∞
				8	240295	∞	∞
FFT.test(n)	10	$4n + 8$	5.02s	8	38	40	5,00
				32	134	136	1,47
				256	1030	1032	0,19
				1024	4102	4104	0,05
JGFHeapSortBench.JGFinitialise	2	1000001	4.63s	-	1000001	1000001	0,00
JGFCryptBench.JGFinitialise	7	9000113	5.76s	-	9000113	9000113	0,00
JGFSeriesBench.JGFinitialise	1	20000	5.16s	-	20000	20000	0,00

Table 2.6: Experimental results

These experiments showed that the technique was indeed efficient and very accurate, actually yielding exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of creation sites associated with exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the `bisort` example, the reason of the over-approximation is that the actual number of instances is always bounded by $2^i - 1$ being $i = \lceil \log_2 size \rceil$. Indeed, the estimation was exact for arguments power of 2. For the (*)`health` example, it was impossible to find a non-trivial linear invariant. It actually turns out that memory consumption happens to be exponential[6] (the given result was calculated by hand). For `fft`, the argument n was required to be a power of 2 for not throwing an exception.

Table 2.7 shows the polynomials that over-approximate the amount of memory captured by methods of the MST and Em3d examples from JOlden. We show only

[6]Some JOlden programs not considered here also lead to exponential memory usage

methods that capture some creation sites. For the others, the estimation yields 0 as they do not allocate objects or they escape their scope.

m	$\#\mathbf{CS_m}$	memCaptured(m)
mst		
MST.main(nv)	13	size(mst.Graph) + (size(Integer) + size(mst.HashEntry)) $\cdot nv^2$ + $[1/4, 0, 0, 0]_{nv}$ · size(mst.Hashtable) $\cdot nv^2$ + (size(mst.Vertex) + size(mst.Vertex[])) $\cdot nv + 5 \cdot$ size(StringBuffer)
MST.parseCmdLine()	2	size(java.lang.RuntimeException)+size(Integer)
MST.computeMST(g, nv)	1	size(mst.BlueReturn) $\cdot (nv - 1)$
em3d		
Em3d.main(nN,nD)	26	size(em3d.BiGraph) + nN · (2 · size(em3d.Node) + 4 · size(em3d.Node[]) · nD + 2 · size(double[]) · nD) + 8 · size(em3d.NodeIEnumerate) + 4 · size(java.lang.StringBuffer) + size(java.util.Random)
Em3d.parseCmdLine()	6	3 · size(Integer) + 3 · size(java.lang.Error)
BiGraph.create(nN,nD)	2	size(em3d.Node[]) · nN

Table 2.7: Capturing estimation for MST and Em3d examples.

Additional experiments and details about the the tool can be found in [BGY04].

2.6. Discussion and Future Work

2.6.1. Dealing with recursion

As stated, currently we are not dealing with general recursion. This is probably the most challenging theoretical obstacle for our method since some basic concepts are rooted in the assumption of finite call chains. However, not supporting recursion does not constitute a major drawback in many cases since our focus are embedded applications where recursion is a "rara avis". Nevertheless, we are looking for ways of relaxing this limitation like counting the number of possible stack configurations when recursion is eliminated.

2.6.2. Beyond classical iteration spaces

State of complex data-structures may impact the number of times a control state is visited (e.g., iterating a collection). The basic idea to handle this problem is, firstly, to abstract away data-structures into "integer views" (e.g., size of a collection, array length, integer class-attributes, counters standing for iteration progress, largest integer member of the collection, the size of the largest collection inside the collection, the number of objects satisfying a given property, etc.). Then, inductive invariants may be built using those integer-typed variables that capture the relevant state of the data structure (e.g., current index position) and integer-typed expressions over the data-structure that may serve as complexity parameters (e..g, size of array). The tool provides basic functionality to apply this pre-processing for structures such as collections and arrays.

As an example, we illustrate here how to handle collections. Consider an iteration of the form:

```
Iterator it1= collection1.iterator();
while (it1.hasNext() && condition) {
   a = (Type)it1.next();
   ...
}
```

To analyze this kind of pattern the following pre-processing is to be done:

1. As the counting method deals with integer-valued inductive variables, each iterator `it` should be associated to a "virtual" counter *it*. This counter is initialized when the iterator is created and incremented when the corresponding `it.next()` is called. Consequently, loop invariants involving iterators will include a constraint of the form $\{0 \leq it < \texttt{collection.size()}\}$.
2. The parameter to be used when computing the invariant is its `size`.

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  ArrayDim() {
1:  list= new Vector();
2:  len = 0;    }
  void add(Object o) {
1:  Object[] block;
2:  if (len % BSIZE == 0)
3:    block = newBlock(BSIZE);
    else
4:    block=(Object [])
            list.lastElement();
5:  block[len % BSIZE] = o;
6:  len++;
}
  Object[] newBlock(int how) {
1:  Object[] block=new Object[how];
2:  list.add(block);
3:  return block;
    }
  void addAll(Collection c) {
1:  for(Iterator it=c.iterator();
        it.hasNext();)
    {
2:    add(it.next());
    }
  }
}
```

Figure 2.4: Collection Example

Figure 2.4 shows a (very simple) implementation of a dynamic array using a list of fixed sized nodes. The memory allocated by the method `addAll` depends on the size of the collection passed as a parameter. The actual allocation takes place in the method `newBlock` where a new block of memory is allocated only when the previous block is full. Our method yields the following invariant for the control state *addAll*.2.*add*.3:

$$\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1} = \{\text{BSIZE} = 5,\ 0 \leq it < \text{c.size}(),\ \text{len} = it,\ \text{len} \mod \text{BSIZE} = 0, \text{how} = \text{BSIZE}\}$$

and the corresponding allocation expression in terms of the collection size[7]:

$$\mathcal{S}(\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1}, \{c\}) = \text{c.size}() + [0, 4, 3, 2, 1]_{\text{c.size()}}$$

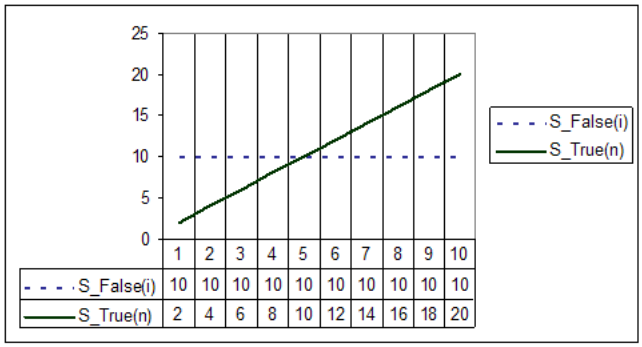
2.6.3. Improving method precision

When programs feature `if` statements with non-linear conditions or polymorphic invocations, it is usually the case of having control states that, by the control structure, are mutually exclusive but their invariants have non-empty intersection. This implies that some statement occurrences are counted more than once by the current technique.

Consider the following example:

```
0: void test(int n,Object a[]) {
1:    for(int i=1;i<=n;i++) {
2:       if(t(i))
3:           a[i] = new Integer[2*i];
4:       else
5:           a[i] = new Integer[10];
      }
   }
```

[7] The function $\mathcal{S}$ will add the constraint $\{\ 1 \leq h_1 \leq how\}$ since the involved creation site is a `newA` statement.



	1	2	3	4	5	6	7	8	9	10
S_False(i)	10	10	10	10	10	10	10	10	10	10
S_True(n)	2	4	6	8	10	12	14	16	18	20

Figure 2.5: Evolution of size functions for the "test" example

If `t(i)` is abstracted away, the invariants at $test.3$ and $test.5$ will be identical:
$\mathcal{I}^{test}_{test.3} = \mathcal{I}^{test}_{test.5} = \{1 <= i <= n\}$
and their corresponding size expressions[8]:
$\mathcal{S}(\mathcal{I}^{test}_{test.3}, n) = n^2 + n$, $\mathcal{S}(\mathcal{I}^{test}_{test.5}, n) = 10n$.

The `computeAlloc` function will sum up these expressions and yield the expression $n^2 + 11n$. This result, although safe, would be too conservative. For instance, for $n = 6$, the estimated memory utilization for `test` will be 102. Nevertheless, analyzing the program, it is easy to see that the maximum amount consumed is 62. This corresponds to choosing creation site $test.5$ when `i` is between 1 and 5 and taking creation site $test.3$ when `i` is greater than 5 (see figure 2.5). In [BGY04] we show some advances in the direction of improving precision.

2.6.4. Hybrid technique

Approaches like [CKQ+05, CNQR05] seem suitable for the verification of Presburger expressions accounting for memory consumption annotations for class methods. We believe that it is possible to devise a technique integrating our analysis together with those mentioned type-checking based ones. The approach would be as follows. While methods for data container classes (like the ones provided by standard libraries) are annotated and verified by type-checking techniques, loop intensive applications built on top of those verified libraries may be analyzed using our approach. The idea is to resort to verified annotations in the same spirit as we handle array creation. That is, it would be not necessary to reach the underlying creation sites of the library. Instead, invariants at the method invocation sites may be built by introducing an integer variable with the Presburger expression as upper-bound. Benefits are twofold: first, work done by our technique would be reduced since we would had to deal with significatively smaller call graphs, and second, our ability to synthesize non-linear consumption expressions would entail an increase of expressive power and usability of type-checking based techniques.

2.7. Conclusions

We have developed a technique to synthesize non-linear symbolic estimators of dynamic memory utilization. We first presented an algorithm for computing the estimator for a single method. We then specialized it for scope-based memory management. Our approach resorts to techniques for finding invariants and counting integer solutions of linear constraints. We believe that the combination of such techniques, and in particular, their application to obtain specifications that predict dynamic

[8] To simplify the explanation, we intentionally omit the size(Integer) factor.

memory utilization is interesting and novel. Besides, it is suitable for accurately analyzing memory utilization in the context of loop-intensive programs. Memory estimators can be used both at compile- and run-time, for example, to set up the appropriate parameters required by the RTSJ scoped-memory API, to over estimate heap usage, to improve memory management and to accurately determine whether a new program can be safely dynamically loaded and scheduled without disturbing other programs behavior.

We have developed a prototype tool that allowed us to experimentally evaluate the efficiency and accuracy of the method on several Java benchmarks. The results were very encouraging. We are currently improving the tool in order to thoroughly test the complete approach (in particular integration with escape analysis) and make the approximations tighter.

Other aspect to explore is the optimization of our method. Slicing techniques and techniques to find inductive variables could help in reducing the number of variables and statements considered when building the invariants. On the other hand, techniques like [Ghe02] can be used to eliminate from our analysis creation sites that can be statically pre-allocated.

CHAPTER 3

A region-based memory manager

We present a method to analyze, monitor and control dynamic memory allocation in Java. It first consists in performing pointer and escape analysis to detect memory scopes. This information is used to automatically instrument Java programs in such a way memory is allocated and freed by a region-based memory manager. Our source code instrumentation fully exploits the result of scope analysis by dynamically mapping allocation places to the region stack at runtime via a registering mechanism. Moreover, it allows executing the same transformed program with different implementations of scoped-memory managers and perform different run-time analysis without changing the transformed code. In particular, we consider a class of managers that handle variable-size regions composed of fixed-size memory blocks for which we provide analytical models for the intra- and inter-region fragmentation. These models can be used to observe and control fragmentation at run-time with negligible overhead. We describe a prototype tool that implements our approach[1].

3.1. Introduction

Current trends in the embedded and real-time software industry are leading towards the use of object-oriented programming languages such as Java. From the software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces. Because programmer-controlled memory management inhibits modularity, object-oriented languages, like Java, provide built-in garbage collection [JL96] (GC), that is, the automatic reclamation of heap-allocated storage after its last use by a program. However, automatic memory management is not used in real-time embedded systems. The main reason for this is that the temporal behavior of software with dynamic memory reclaiming is extremely difficult to predict.

Several GC algorithms have been proposed for real-time embedded applications. For instance, [Hen98] proposes to use an incremental copying algorithm [Bro84] during the execution of low-priority tasks. To insure that high-priority tasks will not run out of memory, enough storage space must be pre-allocated. Besides, the sharing of garbage collection time among low-priority tasks is not evident. [Sie00] adapts the

[1] This chapter is based on the results published at the "International Workshop on Runtime Veirfication" (RV'04) [GNYZ05].

incremental mark-and-sweep algorithm for a JVM that allocates objects as a collection of small memory blocks. The inconvenience of this algorithm is that the number of increments required per allocated block depends on the size of the whole reachable memory. [RF02] adapt the classical reference-counting algorithm [Bro85]. Its response time depends on the total number of reachable objects when it has to collect a non-referenced cycle. [HIB+02] propose a picoJava-II hardware implementation of an adaptation of the incremental treadmill algorithm [Bak92]. This approach is not portable and it does not ensure predictable execution times.

To overcome the drawbacks of current GC algorithms, the RTSJ [GB00] proposes a memory management API based on the concept of "scoped memory". The idea is to allocate objects in regions [GA01, TT97] which are associated with the lifetime of a computation unit (method or thread). Regions are freed when the corresponding unit finishes its execution. However, determining objects' scope is difficult. Therefore, programming using the RTSJ API is error-prone.

To avoid using the RTSJ API directly, [DC02] proposes to automatically instrument a Java program and to replace (whenever possible) Java new statements by calls to the RTSJ scoped-memory API. Doing so requires analyzing the program to determine the lifetime of dynamically allocated objects. Their approach is based on a weighted graph of references, where nodes are allocation points, arcs represent the points-to relation, and weights correspond to depths in the call chain. Roughly speaking, weights are associated with scopes, and dynamic programming is used to minimize weights, that is, to bind any allocation point to the smallest depth of an allocation point of an object that transitively points to some object created at the former.

To build the graph, [DC02] uses a profiler. Thus, there is no assurance that the graph over-approximates the possible references to an object in all possible runs. In consequence, scoped-memory rules are not necessarily respected which forces corresponding run-time checks to be performed by the API implementation, with the implied running time overhead. Besides, the instrumentation is such that each creation site is statically assigned to a fixed region. This technique may make objects live significantly longer than needed.

Here, we propose a method that attempts to tackle these two issues. The first step is to apply pointer and escape analysis techniques [Bla99, CGS+99, SR01] to the program to synthesize scopes. Using pointer and escape analysis it is possible to conservatively determine if an object "*escapes*" or is "*captured by*" a method. Intuitively, an object escapes a method when its lifetime is longer than the method's lifetime, so it can not be collected when the method finishes its execution. An object is captured by the method when it can be safely collected at the end of its execution.

Based on the information above we synthesize a memory organization that associates a memory region with each method in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled by construction. Thus, run-time checks can be safely eliminated to enhance performance. To instrument the program, we define an API that avoids the RTSJ overhead of creating a runnable object each time a new memory scope is created. Our instrumentation fully exploits the result of the scope analysis by dynamically mapping creation sites to the region stack at runtime via a registering mechanism. This allows to control at run-time where the object is actually allocated according to given performance criteria (e.g., minimizing memory fragmentation), without changing the source-level instrumentation.

We also address the issue of monitoring and evaluating run-time performance of the scoped-memory manager. In this work, we focus on region-based memory man-

agers that handle variable-size regions composed of fixed-size memory blocks. For this class of managers, we provide an analytical model of the intra- and inter-region fragmentation for several allocation algorithms (e.g., first-fit and best-fit). These models can be used to observe and control fragmentation at run-time with negligible overhead. Run-time analysis also allows tuning the parameters to accommodate to the needs of the program.

We finally describe a prototype tool that implements our approach.

3.2. Preliminaries

Following [SR01], we define a program to be a set $\{m_0, m_1, \ldots\}$ of *Methods*. A method m has a list P_m of parameters. Each statement is identified with a $Label =_{def} Method \times \mathbb{N}$ which uniquely characterizes its location.

A *Call Graph* of a method m is a directed graph $CG_m =< N, E >$ where $N = Methods$ represents the program methods and $E = (Methods \times Label \times Methods)$ represents the call relation. $(c, l, m) \in E$ means that the method c, at location l, calls method m. We assume that we can determine at compile time, for each call, exactly which method will be invoked, not being able to have more than one possible invocable method. Supporting inheritance and late binding is outside the scope of this work.

Since currently we do not deal with recursive programs, a finite *Call Tree* $CT_m =< N, E >$ can be obtained by unfolding the call graph. This unfolding is done by cloning the nodes that have more than one parent. $N = Methods_{CT} = Label^+ \times Method$ represents the path from the root node and $E = (Methods_{CT} \times Label \times Methods_{CT})$

Let $\alpha \in Label^+$. Let $\alpha = \alpha'.i$, $i \in \mathbb{N}$, we define $trim(\alpha) = \alpha'$. Let $l \in Label$ such that $\alpha = \alpha_1.l.\alpha_2$, and l does not appear in α_i, $i = 1, 2$. We define $pref(\alpha, l) = \alpha_1.l$, and $suff(\alpha, l) = l.\alpha_2$. We define $last(\alpha.l) = l$ and $first(l.\alpha) = l$. The projection $mth()$ of $Label^+$ onto $Method$ is recursively defined as $mth(m.i) = m$ and $mth(\alpha.m.i) = mth(\alpha).m$. These operations are naturally extended to nodes of the call tree. We define $paths(CT_m)$ to be the set of paths of CT_m, and $pred_m(\rho)$ to be the subtree of CT_m composed of all paths of the form $\rho'.mth(first(\rho))$ such that $\rho'.\rho \in paths(CT_m)$.

A *control flow graph* (CFG) is a directed graph $G =< N, E, entry, exit >$ where N is the set of nodes and E is the set of edges. *entry* and *exit* are specials nodes indicating unique start and ending points. Given a method m, G_m is the CFG of m which includes transitively the CFG of every method that m calls. Each node $n \in N$ corresponds to one statement and has a label $l \in Label^+$. Notice that, since a called method is macro-expanded in the control flow graph each time it is invoked, labels are composed by the corresponding path in CT_m and its relative location.

By convention, $m0$ is the *main* method. Thus, G_{m_0} is the control flow graph of the program, and CT_{m_0} its call tree.

We call *Creation Site* every place (defined by its $Label^+$) of the program where an object is created (i.e. there is a *new* or a *newA* statement). For simplicity we assume that new statements only create object instances. Constructors are assumed to be called separately. Calls to constructors are handled as any other method call. CS_m denotes the set of creation sites reachable from the entry point of the method m control flow graph.

We call *Call Site* every place (defined by its $Label^+$) of the program where there is method call. $Calls_m$ denotes the set of method calls in G_m.

```
    void m0(int mc) {
1:      Ref0 h = new Ref0();
2:      Object[] a = m1(mc);
3:      Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k)    {
1:      int i;
2:      Ref0 l = new Ref0();
3:      Object[] b = newA Object[k];
4:      for(i=1;i<=k;i++)  {
5:          b[i-1] = m2(i,l);
        }
6:      Object[] c = newA Integer[9];
7:      return b;
    }

    Object[] m2(int n, Ref0 s)  {
1:      int j;
2:      Object c,d;
3:      Object[] f = newA Object[n]

4:      for(j=1;j<=n;j++) {
5:          if(j % 3 == 0) {
6:              c = newA Integer[j*2+1];
            }
            else {
7:              c = new Integer;
            }
8:          d = new Integer[4];
9:          s.ref = d;
10:         f[j-1] = c;
        }
11:     return f;
    }

class Ref0 {
    public Object ref;
}
```

Figure 3.1: Motivating example

Example

In Figure 3.1 we present one motivating example. The Call Graph and Call Tree for method $m0$ are depicted in Figure 3.2.

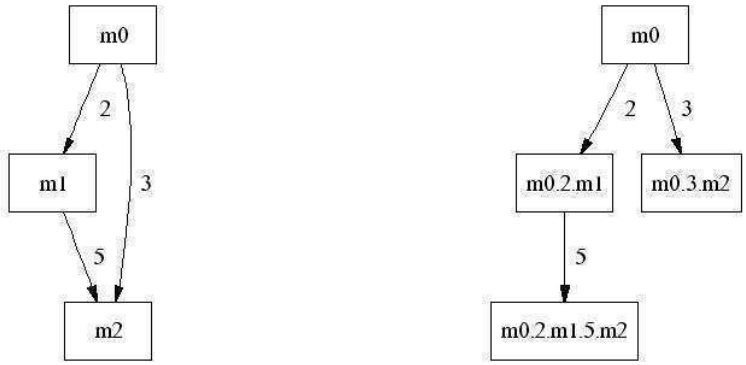


Figure 3.2: Call Graph and Call Tree for method $m0$ of the proposed example

The creation sites for each method of our example are:
$CS_{m0} = \{\ m0.1,\ m0.2.m1.2,\ m0.2.m1.3,\ m0.2.m1.5.m2.3,\ m0.2.m1.5.m2.6,$
$m0.2.m1.5.m2.7,\ m0.2.m1.5.m2.8,\ m0.2.m1.6,\ m0.3.m2.3,$
$m0.3.m2.3,\ m0.3.m2.6,\ m0.3.m2.7,\ m0.3.m2.8\ \}$
$CS_{m1} = \{\ m1.3,\ m1.5.m2.3,\ m1.5.m2.6,\ m1.5.m2.7,\ m1.5.m2.8,\ m1.6\ \}$
$CS_{m2} = \{\ m2.3,\ m2.6,\ m2.7,\ m2.8\ \}$
The call sites for each method of our example are:
$Calls_{m0} = \{\ m0.2, m0.3\ \}$
$Calls_{m1} = \{\ m1.5\ \}$
$Calls_{m2} = \{\ \}$ □

3.3. Scoped memory management

In the Real-Time Specification for Java (RTSJ) [GB00] scoped-memory management is based on the idea of allocating objects in regions which are associated with

the lifetime of a runnable object. This approach imposes restrictions on the way objects can reference each other in order to avoid the occurrence of dangling references. An object $o1$, belonging to a region r, can point to other object $o2$ only if one of the following conditions holds: $o2$ belongs to r; $o2$ belongs to a region that is active when r is active; $o2$ is in the heap; $o2$ is in the inmortal (or static) memory. An object $o1$ can not point to an object $o2$ in region r if: $o1$ is in the heap; $o1$ is in inmortal memory; r is not active sometime during $o1$'s lifetime.

At runtime, region activity is related to the execution of computational units (e.g., methods or threads). In an single-threaded program, where each region is associated with one method, there is a region stack, where the number and ordering of active regions corresponds exactly to the appearances of each method in the call stack. In a multi-threaded program, where regions are associated with threads and methods, there is a region tree which branches are related to each execution thread. In this work, we assume that threads do not share regions, that is, threads only interact through the immortal memory [GB00].

Programming with scoped-memory management is difficult and error-prone. One solution is to statically check whether a program satisfies the restrictions above. This approach is followed in [GA01], where a type system is proposed. Here we propose to automatically infer scopes by static analysis and automatically instrument the program with the appropriate region-based allocations in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled by construction.

3.3.1. Inferring scopes

In order to infer scope information we use pointer and escape analysis [Bla99, CGS+99, SR01]. This is a static analysis technique that discovers the relationship between objects themselves and between objects and methods. It has been used in several applications such as synchronization removal, elimination of runtime checks, stack and scoped allocation, etc.

Here, we are interested in conservatively determining if an object "*escapes*" or is "*captured by*" a method. An object escapes a method when its lifetime is longer than the lifetime of the method. Let $escape : Method \rightarrow \mathbb{P}(CreationSite)$ be the function that returns the creation sites that escape a method. An object is captured by the method when it can be safely collected at the end of the method's execution. Let $capture : Method \rightarrow \mathbb{P}(CreationSite)$ be the function that returns the creation sites that are captured by a method.

For the sake of simplicity, we do not explain here how these two functions are computed. The interested reader is referred to [Bla99, CGS+99, SR01]. Instead, we use our example to illustrate the technique.

Example

The creation sites that escape and are captured by are the following:

$\text{escape}(m0) = \{\ \}$
$\text{escape}(m1) = \{\ m1.3,\ m1.5.m2.3,\ m1.5.m2.6,\ m1.5.m2.7\ \}$
$\text{escape}(m2) = \{\ m2.3,\ m2.6,\ m2.7,\ m2.8\ \}$

$\text{capture}(m0) = \{\ m0.2.m1.3,\ m0.2.m1.5.m2.3,\ m0.2.m1.5.m2.6,\ m0.2.m1.5.m2.7,\ m0.3.m2.3,\ m0.3.m2.6,\ m0.3.m2.7,\ m0.3.m2.8\ \}$
$\text{capture}(m1) = \{\ m1.5.m2.8,\ m1.6\ \}$
$\text{capture}(m2) = \{\ \}$

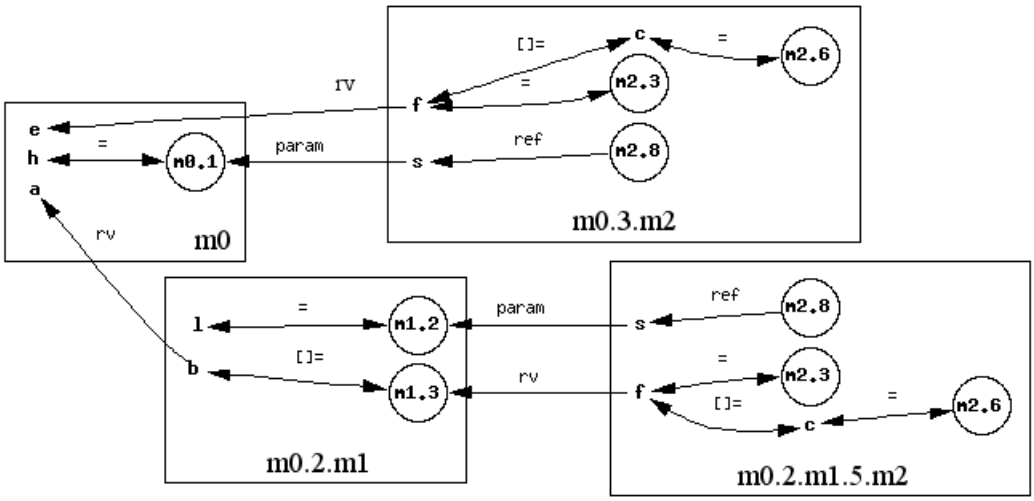


Figure 3.3: Escape analysis for creation sites $m0.1$, $m1.2$, $m1.3$, $m2.3$, $m2.6$ and $m2.8$

Let us consider a few cases. For instance, $m1.3$ escapes from $m1$. This is because $m1.3$ is the creation site of the object assigned to b (represented in Fig. 3.3 as the bi-directional arc from node b to node $m1.3$), which is returned by (and therefore escapes from) method $m1$ (depicted as the arc from b to a labeled rv [2]). Creation site $m2.3$ escapes from $m2$. This is because the memory allocated in line 6 of $m2$ is first referenced by c and then by an entry of f (line 11), which is returned by $m2$. Since the returned object is assigned to an entry of b when $m1$ calls $m2$ in line 5, and b is returned by $m1$, we have that $m1.5.m2.6$ escapes. Besides, $m0.2.m1.5.m2.6$ is captured by $m0$. Also, $m2.8$ escapes from $m2$ because the memory allocated is referenced by s which is passed to $m2$ as a parameter, but, in this case, the creation site is captured by $m1$ and $m0$ depending on the corresponding call chain. □

Let m be a method and $l \in Calls_m$, we define:

$$register(l) \quad = \quad \{last(cs) \mid cs \in capture(mth(l)) \wedge first(cs) = l\}$$

Example

The creation sites registered to call sites in the example are the following:
$register(m0.2) = \{\ m1.3, m2.3, m2.6, m2.7\ \}$
$register(m0.3) = \{\ m2.3, m2.6, m2.7, m2.8\ \}$
$register(m1.5) = \{\ m2.8\ \}$ □

3.3.2. Synthesizing memory regions

Based on the information above we can synthesize a memory organization that associates a memory region r_m with each method m in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled.

The properties of escape analysis ensure that the lifetime of objects allocated by creation sites captured by a method m does not exceed the lifetime of m itself. That is, no object captured by m can be pointed-to by an object captured by a method (transitively) calling m. Thus, the memory referenced by those objects can be safely reclaimed after m terminates.

[2] rv stands for return value.

Let cs be a creation site and m be a method such that $cs \in capture(m)$, that is, $m = mth(first(cs))$. We define $reclaim(cs)$ to be the subtree of the call tree of the program composed of those paths having cs as suffix, that is:

$$\begin{aligned} reclaim(cs) &= pred_{m_0}(trim(cs)) \\ &= \{pref(\rho, m) \mid \rho \in paths(CT_{m_0}) \wedge trim(cs) = suff(\rho, m)\} \end{aligned}$$

In words, $mth(\rho)$ is a call stack, and $mth(pref(\rho, m))$ is the portion of the stack that contains all methods where it is safe to allocate the memory required by cs. If an object o is allocated at line i of method n, where $n.i = last(cs)$, when the call stack is $mth(\rho)$, then o can be safely allocated in any region $r_{m'}$, where m' appears in the prefix of the call stack upto method m.

3.3.3. API and program transformation

In order to perform scoped-memory management at program level, we propose an API which differs from the RTSJ one, described in [BR01, GB00], in two major points. First, in our API memory scopes are not bound to runnable objects. In this point, our API is closer to the RC library [GA01]. Second, our API does not specify a unique region where an object is allocated, but rather a set of regions corresponding to methods in a prefix of the call stack. The actual region where the object will be allocated at runtime is left out to the implementation. We will discuss this issue in the next section. The API is shown in Table 3.1.

`enter(r)`	push r into the region stack
`exit()`	collect the objects in top region
`current()`	return the top region
`determineAllocationSite(CS)`	register creation sites in CS
`newInstance(l,c)`	create an object of class c
`newAInstance(l,c,n)`	same but for arrays of dimension n

Table 3.1: Scoped-memory API.

The program is transformed as follows. Let m be a method.

- The calls to *enter*(r_m) and *exit* are inserted at the beginning and at the end of the method.
- Let $l = m.i \in Label$ be the label of a *new C* (resp. *newA C[n]*) statement in the body of m. The statement in line i is replaced by an invocation to *newInstance(l,c)* (resp. *newInstance(l,c,n)*).
- Recall that creation sites are distinguished in the analysis by the paths in the call tree. Since a *newInstance* at label l only carries l as a parameter, and not the call chain, it is necessary to dynamically change the capture information to be able to compute $reclaim()$ at runtime. To do so, we register the set of creation sites captured by a method at the corresponding call site. Let l be such that $m = mth(l)$. If $register(l) \neq \emptyset$, an invocation to *determineAllocationSite(register(l))* is inserted just before l.

Thus, at *newInstance(l,c)*, where $mth(l) = m$, we have that $pref(\rho, m) \in reclaim(cs)$ iff $\sigma = mth(\rho)$ is the call stack, and $last(cs) \in register(l)$. Therefore, the object instance can be allocated in the region of any method in $pref(\sigma, m)$.

Example

Table 3.2 shows the instrumented code for the example. □

```
class RegisterExample
{
    final static String[] m0_2= {"m1_3","m2_3","m2_6","m2_7"};
    final static String[] m0_3= {"m2_3","m2_6","m2_7","m2_8"};
    final static String[] m1_5= {"m2_8"};
}
void m0(int mc) {
    ScopedMemory.enter(new Region("m0"));
    Ref0 h =(Ref0) ScopedMemory.newAInstance("m0_3", Ref0.class,1);
    Object[] a;
    ScopedMemory.determineAllocationSite(RegisterExample.m0_2);
    a = m1(mc);
    Object[] e;
    ScopedMemory.determineAllocationSite(RegisterExample.m0_3);
    e = m2(2 * mc, h);
    ScopedMemory.exit();
}
Object[] m1(int k) {
    ScopedMemory.enter(new Region("m1"));
    int i;
    Ref0 l =(Ref0) ScopedMemory.newAInstance("m1_2", Ref0.class, 1);
    Object b[] = (Object[]) ScopedMemory.newAInstance("m1_3", Object[].class, k);
    for (i = 1; i <= k; i++) {
        ScopedMemory.determineAllocationSite(RegisterExample.m1_5);
        b[i - 1] = m2(i, l);
    }
    Object c[] = (Integer[]) ScopedMemory.newAInstance("m1_6", Integer[].class, 9);
    ScopedMemory.exit();
    return b;
}
Object[] m2(int n, Ref0 s) {
    ScopedMemory.enter(new Region("m2"));
    int j; Object c, d;
    Object[] f = (Object[]) ScopedMemory.newAInstance("m2_3", Object[].class, n);
    for (j = 1; j <= n; j++) {
        if (j % 3 == 0) {
            c = (Integer[]) ScopedMemory.newAInstance("m2_6", Integer[].class, j * 2 + 1);
        } else {
            c = (Integer[]) ScopedMemory.newAInstance("m2_7", Integer[].class, 1);
        }
        d = (Integer[]) ScopedMemory.newAInstance("m2_8", Integer[].class, 4);
        s.ref = d;
        f[j - 1] = c;
    }
    ScopedMemory.exit();
    return f;
}
```

Table 3.2: Instrumented code for the example

3.3.4. Properties of the code instrumentation

In the instrumentation proposed in [DC02], which uses the RTSJ API [GB00], each creation site is statically assigned to a fixed region by accessing directly outer-scopes using the RTSJ method `getOuterScope()` at the allocation place. This means that, when a creation site is captured by different methods (in different call chains), the inferred scope is necessarily the one corresponding to the capturing method which is closer to the root of the call tree. Therefore, this approach tends to generate fewer regions with bigger sizes, specially near the call tree root, thus maximizing objects' lifetime.

On the contrary, our instrumentation fully exploits the result of the scope analysis in terms of call chains, by dynamically mapping creation sites to a prefix of the region

stack at runtime via the registering mechanism. The actual region where an object is allocated in is determined by the implementation. One possible strategy consists in allways allocating objects in the region of the method that captures them (that is, the last one in the prefix). This strategy produces regions which sizes tend to be bigger for the leafs of the call tree, that is for those methods with shorter lifetimes, rather than near the root. In other words, it minimizes the lifetime of allocated memory.

Example

Consider, for instance, creation site $m2.8$ in our example (see Fig. 3.3). The instrumentation of [DC02] will always allocate memory inside the region r_0 associated with method $m0$, independently of the caller. Our instrumentation will dynamically choose to allocate memory inside regions r_0 or r_1, depending on the caller m_0 or m_1, respectively. □

Our approach allows executing the same transformed program with different implementations of scoped-memory managers. In particular, our API can be implemented directly on top of the ones proposed by the RTSJ and RC. All these instantiations will be functionally equivalent. However, they may exhibit different performances with respect to different quantitative parameters, such as region size, allocation time and memory fragmentation. In the next section, we discuss several possible implementations and focus our analysis on the fragmentation problem.

3.4. Run-time analysis

In this section we describe a framework for analyzing the behavior at runtime of different region-based memory-allocation algorithms that can be used to implement the scoped-memory API. In particular, we consider allocation algorithms that handle variable-size regions composed of fixed-size memory blocks. These algorithms typically manage a linked list of blocks where objects are allocated according to a first-fit or best-fit strategy [WJNB95]. The former allocates the object in the first block where there is enough place to. The latter searches for the block with the smallest amount of free space. The interest of these algorithms resides in the fact that allocation time is linear in the number of blocks, while region deletion is linear in the number of allocated objects (because of the calls to methods' finalizers)[3]. However, they introduce memory fragmentation, that is, holes of (temporarily) unusable free memory. Predicting the number of blocks and objects in a region is difficult and out of the scope of this chapter. A static-analysis technique for over-approximating such numbers is described inchapter 2. Here we concentrate on the problem of analyzing the run-time behavior of the allocation algorithms regarding memory fragmentation.

3.4.1. Intra-region fragmentation

The unused space of a region after a sequence of allocations is considered to be an "intra-region fragmentation" if the next allocation is such that:

(1) no single empty fragment is bigger than the size of the object to be allocated, and a new memory block needs to be added to the region, and

(2) the total amount of empty space is bigger than the size of the object.

[3]The cost could be made constant if calls to finalizers are eliminated via static analysis.

Now, let $\omega = o_1 \cdots o_n$ be a sequence of objects to be allocated in region. We denote by R the set of blocks of the region, and by R_i the set of blocks associated to the region before allocating object o_i. The sequence $R_1, \cdots, R_{n+1}$ is computed as follows. Initially, $R_1 = \{B_1\}$. Now, suppose R_i be $\{B_1, \ldots, B_{m_i}\}$. Let $free_i^k$ be the empty space in block B_k and K_i be the set of indices of blocks that have enough empty space to allocate object o_i, that is,

$$K_i = \{k \in [1, m_i] \mid free_i^k - size(o_i) \geq 0\}.$$

Then,

$$R_{i+1} = \begin{cases} R_i \cup \{B_{m_i+1}\} & \text{if } K_i = \emptyset, \\ R_i & \text{otherwise.} \end{cases}$$

Let $\preccurlyeq_i$ be a total order over K_i that gives the ordering of blocks of R_i that have enough space to allocate o_i according to the search strategy. For instance, for first-fit, $\preccurlyeq_i$ is such that $a \preccurlyeq_i b$ iff $a \leq b$, for all $a, b \in K_i$, and for best-fit, $\preccurlyeq_i$ is such that $a \preccurlyeq_i b$ iff $free_i^a \leq free_i^b$, for all $a, b \in K_i$.

The value $free_i^k$ is computed as follows. Initially, $free_1^1 = size(B_1)$. For $i \geq 1$, if $K_i \neq \emptyset$,

$$free_{i+1}^k = \begin{cases} free_i^k - size(o_i) & \text{if } k = \min_{\preccurlyeq_i} K_i, \\ free_i^k & \text{otherwise,} \end{cases}$$

and if $K_i = \emptyset$,

$$free_{i+1}^k = \begin{cases} size(B_k) - size(o_i) & \text{if } k = m_i + 1, \\ free_i^k & \text{otherwise.} \end{cases}$$

We define $free_i = \sum_{k \in [1, m_i]} free_i^k$.

Let $f(R, \omega)$ be the intra-region fragmentation of R produced by ω. It is the sequence $f_1, \cdots, f_n$ such that:

$$f_i = \begin{cases} free_i & \text{if } K_i = \emptyset \wedge free_i - size(o_i) \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

3.4.2. Inter-region fragmentation

The region where an object will be actually allocated is chosen by an inter-region allocation strategy. Here we consider three possible ones: (1) always allocate in the one of the capturing method (that is, the one corresponding to the method that registers the creation site); (2) allocate in the first region backwards in the prefix (of the call stack) where there is enough free space for the object (inter-region first-fit); (3) allocate in the region (in the corresponding prefix of the call stack) that leaves the smallest possible remanent (inter-region best-fit).

Let $\Gamma = R^{m_{i_1}} \ldots R^{m_{i_p}}$ be the prefix of the region stack associated with a creation site. The unused memory in Γ is considered to be an "inter-region fragmentation" when the allocation of a new object in Γ requires allocating a new memory block to some region $R^{m_{i_j}}$, $1 \leq j \leq p$, while there is enough contiguous free space in some other region $R^{m_{i_k}}$, $1 \leq j \neq k \leq p$, for the newly created object.

The inter-region fragmentation of Γ produced by ω, denoted by $F(\Gamma, \omega)$, can be defined similarly to $f(R, \omega)$.

3.5. Prototype tool

We have developed a software prototype that provides almost fully automatic tool support for transforming Java programs into programs with controlled memory

management via our API, and for analyzing their run-time behavior for different allocation algorithms. Figure 3.4 shows the structure of the tool.

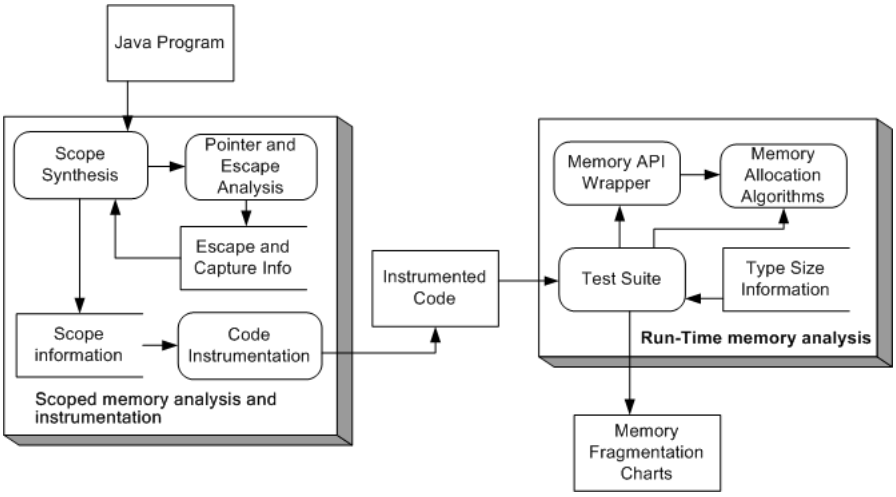


Figure 3.4: Tool suite

To generate the transformed program, we proceed as follows. We first use the Flex Harpoon Compiler [aG] to perform the escape analysis. The output of Flex is used to compute the capture function. We have developed an Eclipse plug-in that takes as input the original program and the capture function, traverses the syntax tree of the program, and generates the transformed one.

The transformed code can be easily integrated into a test suite that provides a software platform (Java classes) with the appropriate wrappers for executing the program. The test platform simulates the behavior of the different memory allocation algorithms by using the fragmentation models presented in the previous section. The classes have been developed in such a way they can be parameterized in many ways, in particular, by different allocation strategies, memory blocks' sizes, and analysis functions.

The output of the analysis is given as charts implemented with the JChart library. Figure 3.5 shows the intra-region fragmentation produced by a single run of the transformed program for a given block size and intra-region allocation strategy. The x-axis represents the sequence of memory accesses, that is, object allocations. The y-axis shows the intra-region fragmentation ratio, that is, the percentage of total intra-region fragmentation (i.e., the sum for all regions) for the total amount of allocated memory in all regions. It is also possible to run the transformed code several times with different memory blocks' sizes, but for the same sequence of allocations. In Figure 3.6, the x-axis represents the block sizes, and the y-axis the minimum, maximum and average intra-region fragmentation over all regions. The tool also provides functionality to count and output the number of operations performed by the algorithms.

3.6. Conclusions and Future Work

We presented a technique for program instrumentation at source code level which transforms a Java program with heap-based allocation into one with scoped-memory management. Our approach ensures scoping rules by construction and decreases run-time overhead by eliminating run-time checks.

Our instrumentation offers a light-weight mechanism for gathering information

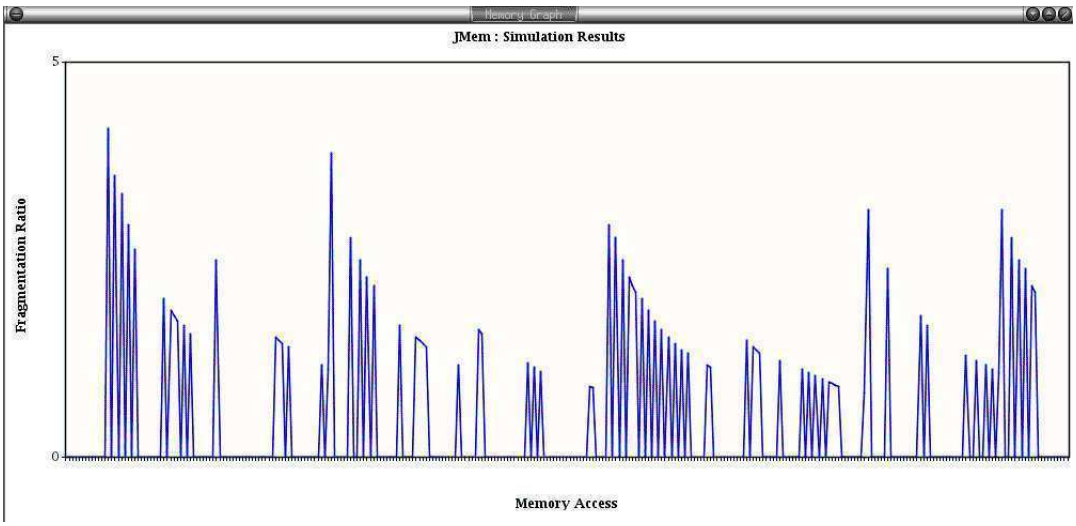


Figure 3.5: Intra-region fragmentation for a given block size

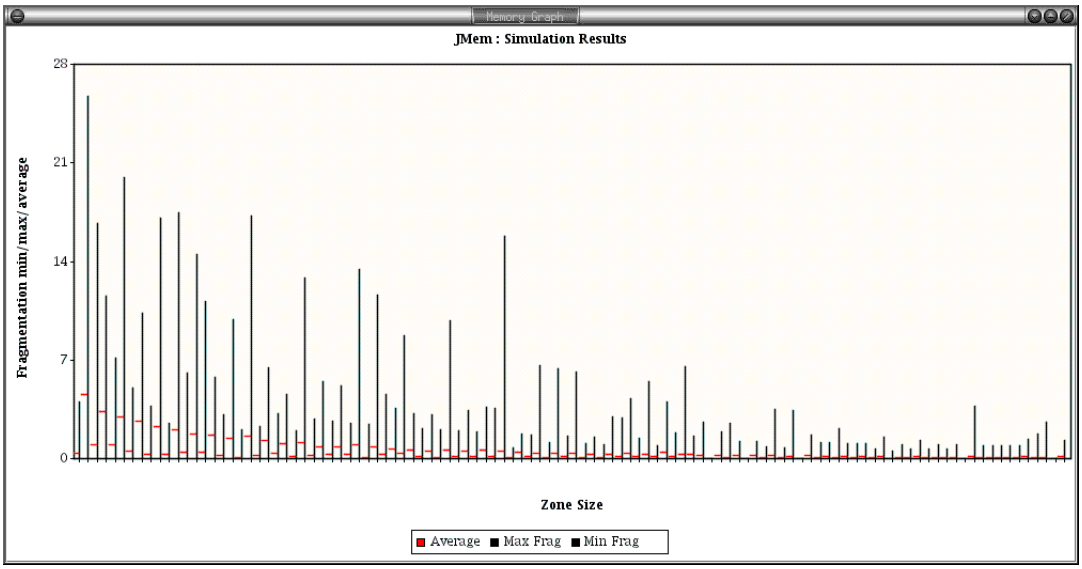


Figure 3.6: Max/Min/Avg intra-region fragmentation for different block sizes

about and controlling memory allocation at run-time. In this work, we have focused on using it for analyzing memory fragmentation for different allocation algorithms. Nevertheless, it can be used for other purposes such as measuring the number of object intances, region sizes, allocation time, etc.

The results of the runtime analysis allows customizing the parameters of the scoped-memory manager according to given performance criteria (e.g., minimize fragmentation ratio). It should be noted that this can be done without touching the transformed program at all.

We are currently working on implementing our API on top of the RTSJ and RC API, and integrating it into the TurboJ compiler [Ins]. Future work includes extending our approach to deal with multi-threading and recursion, and run-time validation of the static estimates given in [BGY04].

CHAPTER 4

A simple static analysis from region inference

We present an algorithm for escape analysis inspired by, but more precise than, the one proposed by Gay and Steensgaard [GS00]. The primary purpose of our algorithm is to produce useful information to allocate memory using a region-based memory manager. The algorithm combines intraprocedural variable-based and interprocedural points-to analyses. This is a work in progress towards achieving an application-oriented trade-off between precision and scalability. We illustrate the algorithm on severaltypical programming patterns, and show experimental results of a first prototype on a few benchmarks[1].

4.1. Introduction

Garbage collection (GC) [JL96] is not used in real-time embedded systems. The reason is that temporal behavior of dynamic memory reclaiming is extremely difficult to predict. Several GC algorithms have been proposed for real-time embedded applications (e.g., [Hen98, Sie00, RF02, HIB+02]). However, these approaches are not portable (as they impose restrictive conditions on the underlying execution platform), do require additional memory, and/or do not really ensure predictable execution times.

An appealing solution to overcome the drawbacks of GC algorithms, is to allocate objects in regions (e.g., [TT97]) which are associated with the lifetime of a computation unit (typically a thread or a method). Regions are freed when the corresponding unit finishes its execution. This approach is adopted, for instance, by the Real-Time Specification for Java (RTSJ) [GB00], where regions can be associated to runnables, and by [GA01], which implements a library and a compiler for C. These region-based approaches define APIs which can be used to explicitly and manually handle allocation and deallocation of objects within a program. However, care must be taken when objects are mapped to regions in order to avoid dangling references. Thus, programming using such APIs is error-prone, mostly because determining objects' lifetime is difficult.

An alternative to programming memory management directly using an API consists in *automatically* transforming a program so as (a) to replace (whenever possible)

[1] This chapter is based on the results published at the "First International Workshop on Abstract Interpretatation for Object Oriented Languages" (AIOOL'05) [SYG05].

"new" statements by calls to the region-based memory allocator, and (b) to place appropriate calls (i.e., guaranteeing absence of dangling references) to the deallocator. Such an approach requires to analyze the program behavior to determine the lifetime of dynamically allocated objects. In [DC02], analysis is based on profiling, while [GNYZ05, CR04] rely on static (points-to and escape) analysis.

Escape analysis aims at conservatively determining if an object *escapes from* or is *captured by* a method. Intuitively, an object escapes a method when its lifetime is longer than the method's lifetime, so it can not be collected when the method finishes its execution. An object is captured by the method when it can be safely collected at the end of its execution.

Several approaches to escape analysis for Java have been proposed, most of which aim at allocating objects on the stack, and removing unnecessary synchronizations. [Bla03] works on the bytecode, which brings in an additionnal complexity due to the stack-based model. [CGS+99, WR99] use points-to analysis to determine if an object escapes a method through a path in the points-to graph. [GS00] proposes a fast but very conservative escape analysis, based on solving a simple system of linear constraints obtained from a *Static Single Assignment* (SSA) form [CFR+91] of the program.

For region-based allocation in Java, we are aware of two works. [GNYZ05] exploits method-call chains and escape analysis to dynamically map allocation sites to regions associated with methods. [CR04] defines a points-to analysis to determine regions of objects with similar lifetimes (with instruction-level resolution, as opposed to method-level).

In this work, we present an algorithm for escape analysis inspired by, but more precise than, the one proposed in [GS00]. The primary purpose of our algorithm is to produce useful information to allocate memory using a region-based memory manager. The algorithm combines intraprocedural variable-based and interprocedural points-to analyses. This is a work in progress towards achieving an application-oriented trade-off between precision and scalability. We illustrate the algorithm on several typical programming patterns, and show experimental results of a first prototype on a few benchmarks.

4.2. The algorithm

In this section we describe our escape analysis algorithm in detail. We assume the program is in static single assignment form (SSA) [CFR+91], that is, every variable is assigned only once in the program. The transformation of the program into SSA comes at a cost, but gives to a flow-insensitive analysis the power of a flow-sensitive one. Our algorithm is mainly based on local variables, instead of on a complex points-to graph, which would be much more expensive to build and to work with. The analysis is based on abstract interpretation [CC77] and computes several properties for local variables and methods.

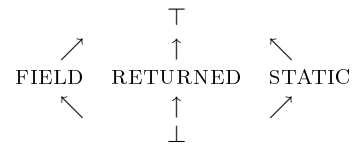
4.2.1. Properties

escape

For each local variable `v` of a method, escape(`v`)$\in$*Escape*, where *Escape* is the lattice in figure 4.1(a), says whether `v` may *escape* from its method, that is, if an object pointed to by `v` is referenced in a way such that its lifetime may exceed the method.

A variable v escapes because it is returned (escape(v)=RETURNED) or it is copied into a global variable (escape(v)=STATIC). When a variable is stored into an object field (escape(v)=FIELD), v may escape through a chain of references. Determining whether v escapes in this case, requires further analysis that will be explained later. The $\top$ value stands for variables that escape by *several* ways, or when the analysis cannot compute a tighter information (e.g., when v is used as a parameter in a non-analyzed method). For example, in the program shown on figure 4.1(b) escape(a)=STATIC, escape(b)=$\bot$, escape(c)=RETURNED, escape(d)=$\top$.

Notice that escape(v) = $\bot$ is not sufficient to say that the object pointed to by v is local to the method. It only means that the method does not create any *new* reference path from the outside of the method to the object, but the object may *already* be reachable from outside. This is the case for variable b in figure 4.1(b) which is an alias of the static variable s.



(a) The escape lattice

```
class Test01 {
  static Object s,t;
  void m0() {
    Object a=m1();
    s=a;
    Object b=m2();
  }
  Object m1() {
      Object c=new Object();
      return c;
  }
  Object m2() {
      Object d=new Object();
      s=d;
      return d;
  }
}
```

(b) the Test01 program

Figure 4.1: The *Escape* lattice and the Test01 program

mfresh

Let *MFresh* be the lattice: $\bot \leq$ RETURNED $\leq \top$. For each method m, mfresh(m) $\in$ *MFresh* describes how objects returned by m escape: mfresh(m)=$\bot$ when m does not return any object (it may be void, or return some primitive type value); mfresh(m)=$\top$ when returned values are already known to escape from m in a different way; mfresh(m) = RETURNED when m returns an object (or several objects) which does (do) not escape otherwise. If there is no other path leading to this object (see section 4.2.2), the caller of m can *capture* it.

sites

Let *Sites* be $\mathbb{P}$(*AllocationSites* $\cup$ {UNKNOWN}), where *AllocationSites* is the set of all allocation sites in the program. For each local variable v, sites(v)$\in$*Sites* contains all allocation sites that can create an object referenced by v. sites(v) can always be computed at the unique (thanks to SSA) statement where v is defined. To be conservative, if we cannot determine *all* the sites that v can point to (e.g., because of a not analyzed method call), a "fake" allocation site UNKNOWN is added to sites(v). In the program of figure 4.1(b), sites(a) = sites(c) = {[m1:c=new Object]}, and sites(b) = sites(d) = {[m2:d=new Object]}.

msites

For each method m, msites(m) is an element of *Sites*, saying where objects returned by m come from. In the program of figure 4.1(b), msites(m0) = $\emptyset$, msites(m1)

= {[m1:c=new Object]}, and msites(m2) = {[m2:d=new Object]}. Notice that, if mfresh(m) = RETURNED, then objects from msites(m) are possibly *captured* by callers of m, but it is not certain. In some complex situations, there can still be a path of references leading to these objects. For example in the program shown on figure 4.6(a), the e variable is *not* captured by m0.

isdereferenced

isdereferenced(v) is true iff v, or one of its aliases, is *dereferenced* in m. That is, is v.f appears in the right-hand side of an assignment.

usedasparameter

usedasparameter(v) is true iff v, or one of its aliases, is used as a concrete parameter in a method call.

def

For each variable v, def(v) says how v was defined.

fielduse

fielduse shows reference relations between local variables. For each v in m, fielduse(v) is the set of variables u in m such that v may be an alias of u.f (for some field f). fielduse is mainly useful when a variable v escapes by a FIELD: for example, if escape(v)=FIELD, but all variables of fielduse(v) are captured by m, then so is v.

the mrefs graph

When objects are passed through several methods, knowledge about local variables is often not sufficient to determine objects' lifetimes, that's why a reference graph is needed. Our reference graph is very simple, in order to minimize the algorithmic cost of the analysis. mrefs is a subset of $AllocationSites \times Fields \times AllocationSites$, where $(\alpha, f, \beta) \in$ mrefs means: "an object created in α, may point, with its f field, to an object created in β".

side

The main goal of our analysis is to determine in which regions to allocate objects. Each method m has an associated *region*, containing objects which do not escape m. To determine the region, we compute for each variable v of m, where objects pointed to by v live, namely, side(v):

- side(v)=INSIDE, when objects pointed to by v are *captured* by m. If they are created by m, they can be allocated in m's region. If they are created by callees, m can *ask* for them to be allocated in its region, as is described in [GNYZ05];
- side(v)=OUTSIDE, when objects pointed to by v live *longer* than m. If they are created by m, they must be allocated outside its stack frame. But such an object may be captured by a caller n of m, in this case m can allocate the object in n's region.

An example is presented on figure 4.6(a): the RefObject allocated by m2 is captured by m1. Our analysis detects this situation by computing side(a)=OUTSIDE and side(c)=INSIDE.

4.2.2. The rules

The algorithm works in two phases. First, it determines for each variable the values of escape, sites, isdereferenced, usedasparameter, fielduse, def, it builds the mrefsgraph and computes msites and mfresh values. To compute these values, the algorithm solves the least fixpoint in Figures 4.2 and 4.3.

In a second phase, the algorithm uses these values to compute, for each variable, its side value, as presented on figure 4.4. It is the combination of side and sites that will enable us to instrument the bytecode in order to use a region memory allocator for captured sites.

α: `v := new`
$\alpha \in \mathsf{sites}(v)$

`v:=` φ`(`$\mathtt{v}_1..\mathtt{v}_n$`)`
$\mathsf{def}(\mathtt{v}) = \text{PHI}$
$\forall i = 1..n$
$\mathsf{sites}(\mathtt{v}) \supseteq \mathsf{sites}(\mathtt{v}_i)$
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \mathsf{escape}(\mathtt{v}_i)$
$\mathsf{escape}(\mathtt{v}_i) \sqsupseteq \mathsf{escape}(\mathtt{v})$
$\mathsf{isdereferenced}(\mathtt{v}) \geq \mathsf{isdereferenced}(\mathtt{v}_i)$
$\mathsf{isdereferenced}(\mathtt{v}_i) \geq \mathsf{isdereferenced}(\mathtt{v})$
$\mathsf{usedasparameter}(\mathtt{v}) \geq \mathsf{usedasparameter}(\mathtt{v}_i)$
$\mathsf{usedasparameter}(\mathtt{v}_i) \geq \mathsf{usedasparameter}(\mathtt{v})$
$\mathsf{fielduse}(\mathtt{v}) \supseteq \mathsf{fielduse}(\mathtt{v}_i)$
$\mathsf{fielduse}(\mathtt{v}_i) \supseteq \mathsf{fielduse}(\mathtt{v})$

`v := `$\mathtt{v}_1$
$\mathsf{def}(\mathtt{v}) = \text{COPY}$
other properties: similar to φ-expression

$\mathtt{v}_1$`.f := v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{FIELD}$
$\mathsf{fielduse}(\mathtt{v}) \ni \mathtt{v}_1$
$\mathsf{mrefs} \supseteq \{\mathtt{s}_1 \xrightarrow{\mathtt{f}} \mathtt{s}_2,$
$\mathtt{s}_1 \in \mathsf{sites}(\mathtt{v}_1),\ \mathtt{s}_2 \in \mathsf{sites}(\mathtt{v}_2)\}$

`s := v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{STATIC}$
$\mathsf{mrefs} \supseteq \{\text{UNKNOWN} \longrightarrow \mathcal{S},\ \mathtt{s} \in \mathsf{sites}(\mathtt{v}_i)\}$

`v := s`
$\mathsf{def}(\mathtt{v}) = \text{STATIC}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

`v := p`
$\mathsf{def}(\mathtt{v}) = \text{PARAM}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$
other properties: similar to φ-expression

`v := constant`
$\mathsf{def}(\mathtt{v}) = \text{CONSTANT}$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

`v :=` $\mathtt{v}_1$`.f`
$\mathsf{def}(\mathtt{v}) = \text{FIELD}$
$\mathsf{isdereferenced}(\mathtt{v}_1) \geq \mathit{true}$
$\mathsf{sites}(\mathtt{v}) \supseteq \{\mathtt{s} \mid \exists\, \mathcal{S}' \in \mathsf{sites}(\mathtt{v}_1),\ \mathcal{S}' \xrightarrow{\mathtt{f}} \mathcal{S}\}$
If $\text{UNKNOWN} \in \mathsf{sites}(\mathtt{v}_i)$
$\mathsf{sites}(\mathtt{v}) \ni \text{UNKNOWN}$

$\mathtt{return}_m$ `v`
$\mathsf{escape}(\mathtt{v}) \sqsupseteq \text{RETURNED}$
$\mathsf{mfresh}(\mathtt{m}) \sqsupseteq \mathsf{escape}(\mathtt{v})$
$\mathsf{msites}(\mathtt{m}) \supseteq \mathsf{sites}(\mathtt{v})$

Figure 4.2: Escape analysis rules

First phase

Most of these rules are simple. They are only intraprocedural information propagation. The only complicated rule is the one on figure 4.3, which handles method calls. This is not trivial, because we do not want to perform a full points-to analysis, neither to be too conservative about method calls.

Our analysis is designed to process arbitrary portions of an application. That is why we have an istobeprocessed predicate, that tells if a method must be analyzed or not. If not, for example because the method is native, or unavailable, we must be conservative about it.

For a not analyzed method, we assume that all parameters escape, and are referenced by the UNKNOWN site.

On the other hand, if the method *is* analyzed, then we can be more precise. Obviously, we have $\mathsf{sites}(\mathtt{v}) \supseteq \mathsf{msites}(\mathtt{m})$, that is, v will point to any object returned by m. If these objects have escaped ($\mathsf{mfresh}(\mathtt{m}) \neq \text{RETURNED}$), then the return value is not capturable either. ($\mathsf{escape}(\mathtt{v}) \sqsupseteq \mathsf{mfresh}(\mathtt{m})$)

```
v := v_0.m(v_1..v_n)
    ∀ m that may be invoked here
    If istobeprocessed(m)
    |  sites(v) ⊇ msites(m)
    |  If mfresh(m) ≠ RETURNED
    |      escape(v) ⊒ mfresh(m)
    |  ∀i = 0..n
    |  usedasparameter(v_i) ≥ true
    |  Let p_i the i-th formal parameter of m
    |  isdereferenced(v_i) ≥ isdereferenced(p_i)
    |  If ¬escape(p_i) ∈ {RETURNED, ⊥}
    |      escape(v_i) ⊒ ⊤
    |      mrefs ⊇ {UNKNOWN ⟶ S, S ∈ sites(v_i)}
    |  If isdereferenced(p_i) = true
    |      mrefs ⊇ {UNKNOWN ⟶ s |∃ S' ∈ sites(v_i), S' ⟶ s)}
    else
    |  sites(v) ∋ UNKNOWN
    |  ∀i = 0..n
    |      usedasparameter(v_i) ≥ true
    |      isdereferenced(v_i) ≥ true
    |      escape(v_i) ⊒ ⊤
    |      mrefs ⊇ {UNKNOWN ⟶ S, s ∈ sites(v_i)}
```

Figure 4.3: Escape analysis rules (cont)

To process the parameters of m, we match the formal parameters (p_i) with the concrete ones (v_i): if p_i escapes from m, v_i is considered as escaping from the current method, and we put an edge from UNKNOWN to all sites pointed to by v_i. If p_i does not escape but isdereferenced in m, then we cannot be precise about those references without performing a points-to analysis. In this case, we conservatively consider that all children of v_i escape.

Second phase

	def(v)					
escape(v)	NEW	RETVAL	PARAM STATIC	COPY PHI	FIELD	CONSTANT
⊥	(3)	(3)	OUTSIDE	(1)	(2)	OUTSIDE
FIELD	(2)	(2)	OUTSIDE	(1)	(2)	OUTSIDE
RETURNED	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE
STATIC	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE
⊤	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE	OUTSIDE

(1)
```
v:= φ(v_1..v_n) or v := v_1
    side(v) ⊒ side(v_i)
∀i side(v_i) ⊒ side(v)
```

(2)
```
If ∃u ∈ fieldUse(v) s.t. side(u) = OUTSIDE
      or s.t.isdereferenced(u) ∧ usedasparameter(u)
      side(v) = OUTSIDE
else
      (3)
```

(3)
```
If ∃s ∈ sites(v) s.t. UNKNOWN ⇝ s
      side(v) = OUTSIDE
else
      side(v) = INSIDE
```

Figure 4.4: Computation of side(v)

Once the fixed point is reached, the algorithm computes side(v) for each variable using rules shown in figure 4.4. This is not a one-pass computation, but a second least fixpoint, because of the (1) and (2) rules:

- The (1) rule says that, if a variable may alias another, then those two variables cannot have different side values.

- Similarly, the (2) rules says that if a variable `v` is referenced by another variable's field (e.g. by a `u.f=v`), `v` cannot be captured unless `u` is.

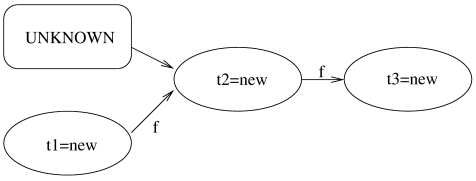
Examples

Let us consider the example presented in fig.4.5(a). First, `m0` builds a small chained structure, then it calls `m1` which makes the last element (`t3`) escape. As shown on fig.4.5(b), the analysis of `m0` understands the behavior of `m0`, but as we can only match `x` with `t1`, and not `a` with `t2`, we cannot keep track of `m1`. Nevertheless, to stay conservative, we put an edge from UNKNOWN to the site of `t2` because `x` is dereferenced in `m1`. Notice that, `t2` and `t3` are usedasparameter, because they are the `this` parameter of their constructor. That is why the only captured site is `[m0:t1 = new RefObject]`.

```
class RefObject {
    Object f;
}

class Test25 {
    void m0() {
        RefObject t1=new RefObject();
        RefObject t2=new RefObject();
        Object t3=new Object();
        t1.f=t2;
        t2.f=t3;
        m1(t1);
    }
    static Object s;
    void m1(RefObject x)
    {
        RefObject a=(RefObject)x.f;
        Object b=a.f;
        s=b;
    }
}
```

(a) The `Test25` program



(b) **mrefs** graph

	escape mfresh	def	IsD	uP	fieldUse	sites msites	side
m0	⊥					∅	
t1	⊥	NEW	true	true	[]	[m0:t1 = new RefObject]	INSIDE
t2	FIELD	NEW	false	true	[t1]	[m0:t2 = new RefObject]	OUTSIDE
t3	FIELD	NEW	true	true	[t2]	[m0:t3 = new java.lang.Object]	OUTSIDE
m1	⊥					∅	
x	⊥	PARAM	true	false	[]	[UNKNOWN]	OUTSIDE
a	⊥	FIELD	false	false	[]	[UNKNOWN]	OUTSIDE
b	STATIC	FIELD	false	false	[]	[UNKNOWN]	OUTSIDE

(c) analysis results

Figure 4.5: The `Test25` program

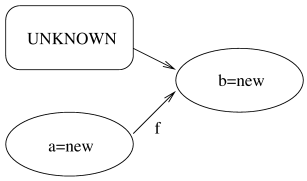
The second example, shown in figure 4.6(a), illustrates the msites property. The `m2` method allocates two objects and makes one (`a`) point to the other (`b`), which escapes. Then it returns `a`, which is captured by `m1` (side(`c`)=INSIDE). `m1` dereferences c to get the `Object` and returns it, but `m0` cannot capture it because of the edge from UNKNOWN to `[m2:b = new Object]`.

4.3. Empirical results

We have implemented a prototype version of this algorithm using the Soot framework [VRHS+99] v.2.2.1. Table 4.1 presents the results of our algorithm on the Jolden benchmarks [CM01]. The first two column are the size of the program in lines, and the number of allocation sites. The next three columns present the time spent by our escape analysis, in seconds, not including Soot's phases: class loading, transformation from bytecode to Jimple (Soot's three-address stackless code), and transformation into SSA form.

```
class Test30 {
    void m0() {
        Object e=m1();
    }
    Object m1() {
        RefObject c=m2();
        Object d=c.f;
        return d;
    }
    static Object s;
    RefObject m2() {
        RefObject a=new RefObject();
        Object b=new Object();
        s=b;
        a.f=b;
        return a;
    }
}
```

(a) the `Test30` program



(b) **mrefs** graph

	escape mfresh	def	IsD	uP	fielduse	sites msites	side
m0	⊥					∅	
e	⊥	RETVAL	false	false	∅	`[m2:b = new Object]`	OUTSIDE
m1	RETURNED					`[m2:b = new Object]`	
c	⊥	RETVAL	true	false	∅	`[m2:a = new RefObject]`	INSIDE
d	RETURNED	FIELD	false	false	∅	`[m2:b = new Object]`	OUTSIDE
m2	RETURNED					`[m2:a = new RefObject]`	
a	RETURNED	NEW	false	true:	∅	`[m2:a = new RefObject]`	OUTSIDE
b	⊤	NEW	true	true	`[r1]`	`[m2:b = new Object]`	OUTSIDE

(c) analysis results

Figure 4.6: The `Test30` program

Program	Lines	Allocation sites	Analysis time			INSIDE		G&S's analysis
			escape	side	total	variables	sites	*stackable* variables
bh	1128	41	9.430	23.51	32.481	34	21	23
bisort	340	10	7.876	11.509	19.385	7	7	7
em3d	462	26	8.551	15.706	24.257	13	11	11
health	562	28	8.454	19.414	27.868	18	13	10
mst	473	16	8.106	14.260	22.366	8	8	7
perimeter	745	13	11.357	23.944	35.301	7	7	7
power	765	21	3.628	1.159	4.787	9	9	5
treeadd	195	11	10.876	27.539	38.415	6	6	6
tsp	545	12	11.19	30.201	41.220	7	7	7
voronoi	1000	35	12.778	66.566	79.344	34	20	31

Table 4.1: Analysis results

The last three columns give the number of INSIDE variables and allocation sites, as computed by our algorithm, and the number of *stackable* variables, as computed by our implementation of G&S's analysis [GS00]. Our analysis is more precise than [GS00] as it subsumes all its rules. That is, all *stackable* variables in the sense of [GS00] are INSIDE variables, but the converse is not true. In our experiments, we did not use any inlining of analyzed code. It is interesting to remark that without inlining, [GS00] does not find any stackable variable in the programs of figures 4.5 and 4.6. As noted in [GS00], both analyses will benefit from method inlining.

We did not have enough time to use the computed information to actually instrument the benchmarks as described in [GNYZ05]. We count on doing this soon. Anyway, a preliminary implementation on another test program revealed a gain of 20% of total utilized memory, when using GC together with region-based manager, w.r.t. GC only, even if the actual region-allocated memory is about 5%.

Besides, only a subgraph of the whole call graph has been analyzed for each test case. The subgraph contains all application methods and a subset of library methods transitively invoked by the program. This explains why there are only a few allocation sites. Nevertheless, these results are interesting, because an important fraction of analyzed allocation sites are indeed computed to be captured. Our algorithm is

parameterized by the set of classes to be analyzed. This allows the user to fine-tune the analysis trading precision against performance according to specific application behaviors.

CHAPTER 5

Annotations for more precise points to analysis

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[1].

5.1. 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, [FLL+02, BLS05]. 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

[1] This chapter is based on the results published at the "International Workshop on Aliasing, Confinement and Ownership" (IWACO'07) [BFGL07a].

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. [SR05] to infer read and write effects for code targetting the .NET Common Language Runtime (CLR) [ECM06]. The CLR is the common infrastructure for languages such as C#, VB, 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 [Mey88]. 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 [BN04]. 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 [DM06].

The main contributions of this work 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 [Spe] to infer and verify method purity and for checking the admissibility of specifications in the Boogie methodology [BLS05].

5.1.1. The Problem

Consider the following simple, but realistic example. Figure 5.1 contains a method written by a programmer to copy a list of integers. In C#, the *foreach* is syntactic "sugar" which the compiler expands ("desugars") into the code shown

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

Figure 5.1: 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 5.2: "Desugared" version of the iterator example.

in Figure 5.2. (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 do not describe precisely the behavior of the method.

5.1.2. Structure

First, we review the essential ideas from Salcianu's analysis in Section 5.2 and present our extensions to deal with .NET memory model and non-analyzable calls. Section 5.3 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 5.4. Some related work is reviewed in Section 5.5 and our conclusions are presented in Section 5.6.

5.2. Salcianu's Analysis

Salcianu et al. [SR05] 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 joined 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 [2]. 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

[2] 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.

method's exit point in term of its parameters. It contains all reachable nodes from the (original) parameter nodes.

5.2.1. 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 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 5.3 shows the representation of object and

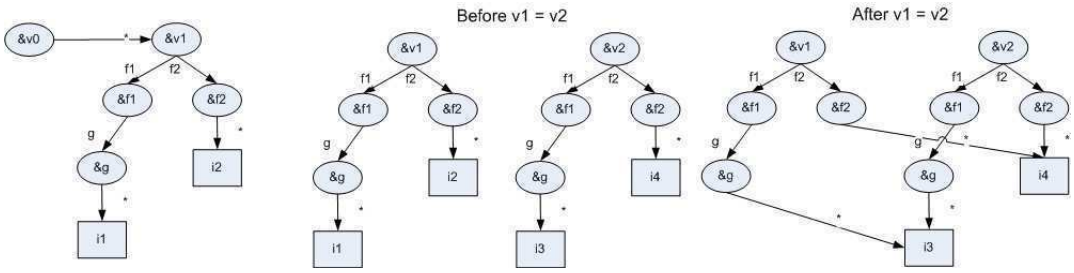


Figure 5.3: Modeling objects and structs. On the left v_0 is the address of v_1, which is a value of a struct type with two fields $f1$ and $f2$. (v_0 can be thought of as an object, e.g., if the struct is passed to a method that takes an object as a parameter then v_1 would be a *boxed* value.) The 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.

struct values and how the assignment of struct values is done. Address nodes are depicted as ovals, values as boxes.

In [BFGL07b] 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 5.4 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 $\mathcal{R}$.

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

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

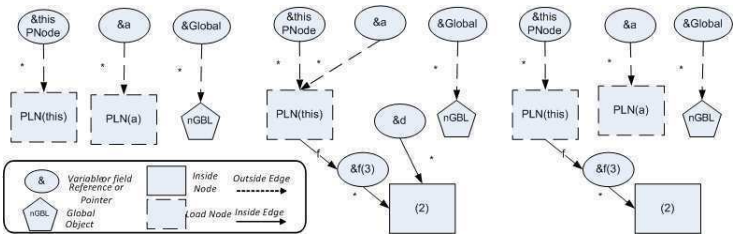


Figure 5.4: Three points-to graphs for the beginning, end, and summary of the method `A.m`.

5.2.2. 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 [SR05].

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 place holder 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.

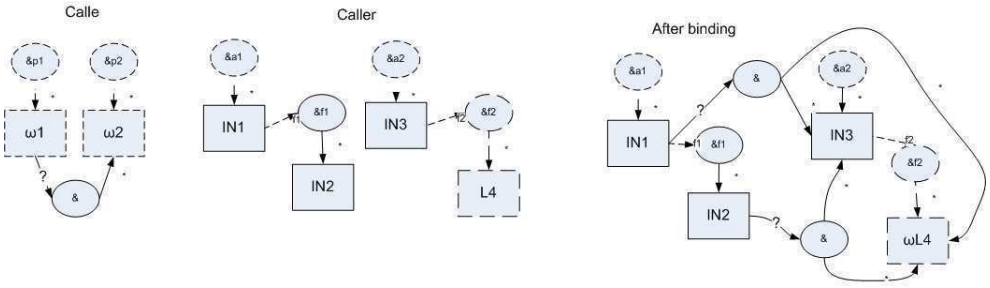


Figure 5.5: Effect of omega nodes in the inter-procedural mapping

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).

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 5.5 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 [BDF+04].) 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 5.3 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 5.3 presents our annotation language that helps eliminate some of these edges.

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[3] and ? edges. Note that we reject paths that contain $ edges.

Finally, we convert any load nodes, n^L_{pc}, contained in the set $\mu(n^{L\ \omega}_{pc})$ 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 instance in Figure 5.5, 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 [BFGL07b].

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.

5.3. Annotations

Table 5.1 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 [BFGL07b]. 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 5.7 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 5.6 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

[3] We mean "owned fields" as defined in the Boogie methodology [BDF+04].

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 5.1: 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]`.

(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 5.6. 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.

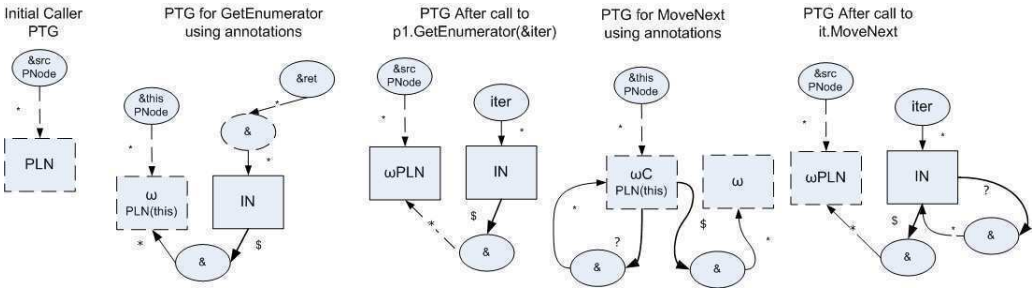


Figure 5.6: The evolution of *Copy*'s points-to graph after calling `src.GetEnumerator` and `iter.MoveNext`. We use the special field $ to indicate that *src* is reachable from *iter* but *iter* is able to mutate objects only using fields that *iter*'s class owns. For simplicity we do not show the evolution of the newly created objects pointed to by the list *l*.

```
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 5.7: The methods needed for analyzing *Copy* along with their annotations.

5.4. Experimental Results

Our implementation is integrated into the Spec# compiler pipeline and can also be run as a stand alone application. We analyze Boogie [BDJ+06], a program verification tool for the Spec# language [BDF+04]. 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 fix point):* 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 fix point 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 5.2 shows the time to analyze the annotated methods and the full application regardless or whether methods are annotated or not. One of the reasons why the full analysis takes more time is because it computes a partial call graph and the

fixpoint computation for mutually recursive methods.

Approach	Time (sec)
Intra-procedural	15.78
Inter-procedural (full)	89.00
Inter-procedural (depth 3)	22.83

Table 5.2: Analysis time for Boogie.

Table 5.4 show the number of method on each packet and how many are declared as pure. Table 5.4 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.

Project	#Methods	Declared Pure
AbsInt	348	66
AIFramework (AI)	15063	3514
Graph	97	20
Core	9628	1326
ByteCodeTrans (BCT)	5564	984
VCGeneration (VCG)	2050	187
Compiler Plugin (CP)	55	12

Table 5.3: Information about the different components of Boogie s showing the number of methods annotated as pure.

Project	Using Annotations						Without Annotations					
	Intra	%	Inter 3	%	IF	%	Intra	%	Inter 3	%	IF	%
AbsInt	66	100%	66	100%	66	100%	51	77%	51	77%	51	77%
AI	2702	77%	2725	77%	2730	78%	1631	46%	1688	48%	1688	48%
Graph	14	70%	14	70%	14	70%	10	50%	10	50%	10	50%
Core	1164	88%	1224	92%	1224	92%	709	53%	729	55%	729	55%
BCT	781	79%	845	86%	863	88%	255	26%	297	30%	297	30%
VCG	171	91%	171	91%	171	91%	155	83%	155	83%	155	83%
CP	10	83%	10	83%	10	83%	8	66%	8	66%	8	66%

Table 5.4:]
Results for Boogie showing the number of methods annotated as pure that were verified as pure by our analysis. IF stands for "Inter Procedural Full" bottom up analysis.

5.5. Related work

Our analysis is a direct extension of the points-to and effect analysis by Salcianu et al. [SR05]. We add support for a more complex memory model (managed point-

ers 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. [LG88] 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 [CD02] while Leino et al. use data groups [LPHZ02]. In [GB99], 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 [LBR99] and Spec# [BDF+04] 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 [TE05] 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 [BN04].

Points-to information has also been used to infer side effects [RR01, MRR05, CBC93, CR07]. Our analysis, as well as Salcianu's analysis [SR05], 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 [CR07] 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 trade-off 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 [AKC02] 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. [NX05] 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.

5.6. Conclusions and Future Work

We have implemented an extension to Salcianu's analysis [SR05] that works on the complete .NET intermediate language CIL. The extensions involve several non-trivial details that enable it to deal 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.

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*. It is also possible to use it for inferring and checking method *modifies clauses*.

CHAPTER 6

JScoper: Scoping and Instrumentation for region-based Java Applications

We present `JScoper`, an Eclipse plug-in which will help developers, researchers and students, to generate, understand, and manipulate memory regions in scoped-memory management setting. The main goal of the plug-in is to provide a tool that will transparently assist the translation of Java applications into Real-time Specification for Java (RTSJ) compliant applications. More accurately, its purpose is to enable automatic and semi-automatic ways to translate heap-based Java programs into scope-based ones, by leveraging GUI features for navigation, specification and debugging[1].

6.1. Introduction

Current trends in the embedded and real-time software industry are leading practitioners towards the use of object-oriented programming languages such as Java. From a software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces. Because programmer-controlled memory management hinders modularity, object-oriented languages like Java provide built-in garbage collection, i.e. the automatic reclaiming of heap-allocated storage after its last use by a program.

However, automatic memory management is not used in real-time embedded systems. The main reason for this is that the execution time of software with dynamic memory reclaiming is extremely difficult to predict. Therefore, in current industrial practices the use of garbage collection in real-time applications is simply forbidden. The typical alternative approach is to have programs allocate all memory during their initialization phase and free it upon termination. This leads to very inefficient memory use, usually resulting in over-dimensioning physical memory requirements at an unnecessary additional cost.

A automatic memory management techniques that meet real-time requirements would clearly have a huge impact on the design, implementation, and analysis of embedded software. These techniques would prevent programming errors produced

[1] This chapter is based on the results published at the "International Eclipse Technology eXchange at OOPSLA" (etX'05) [FGB+05].

by hazardous memory handling, which are both hard to find and to correct. As a result, they would drastically reduce implementation and validation costs while considerably improving software quality.

In order to overcome the drawbacks of current garbage collection algorithms, the Real-Time Specification for Java (RTSJ)[GB00] proposes the use of application-level memory management, based on the concept of "scoped memory", for which an appropriate API is specified. Scoped-memory management relies on the idea of allocating objects in regions associated with the lifetime of a computation unit (method or thread). Regions are deallocated when the corresponding computational units finish their execution [TT97, GA01, GB00, GNYZ05]. Unfortunately, the task of determining object scopes is left to the programmer.

Some techniques have been proposed to address this problem by automatically mapping sets of objects with regions[DC02, GNYZ05]. These techniques typically use Pointer and Escape Analysis [SR01, SYG05, Bla99] to conservatively approximate object lifetimes. Informally, an object escapes a method when its lifetime is longer than the method's lifetime, so it cannot be collected when the method finishes its execution. In contrast, an object is captured by the method when it can be safely collected at the end of the method's execution.

Our main goal is to provide developers with a tool that will assist the translation of Java applications into Java Real-time compliant applications. More accurately, the idea is to enable translation of heap-based Java programs into scoped-based ones, by leveraging GUI features for navigation, specification, translation, fine-tuning and debugging.

6.2. Scoped Memory Management

The aim of the Real-Time Specification for Java (RTSJ) [GB00] is to enable the development of real-time applications using Java. One of its most remarkable characteristics is a new memory hierarchy which incorporates several kinds of memory models: Heap memory (garbage collected), Immortal memory and Scoped memory. Neither Immortal nor Scoped memory use garbage collection. Objects allocated in Immortal memory are never collected and live throughout program lifetime. Scoped-memory management is based on the idea of allocating objects in **regions** associated with the lifetime of a runnable object. When a computational unit finishes its execution, its objects are automatically collected.

This approach imposes restrictions on the way objects can reference each other in order to avoid the occurrence of dangling references. An object $o1$ belonging to region r references an object $o2$ only if one of the following conditions holds: $o2$ belongs to r; $o2$ belongs to a region that is always active when r is active; $o2$ is in the Heap; $o2$ is in Immortal (or static) memory. An object $o1$ cannot point to an object $o2$ in region r if: $o1$ is in the heap; $o1$ is in immortal memory; r is not active at some point during $o1$'s lifetime.

	Heap	Inmortal	Scoped
Heap	Yes	Yes	No
Inmortal	Yes	Yes	No
Scoped	Yes	Yes	if active

Table 6.1: Scoped-memory reference rules.

At runtime, region activity is related to the execution of computational units (e.g., methods or threads). In a single-threaded program, if each region is associated with one method, then there is a region stack where the number and ordering of

active regions corresponds exactly to the appearances of each method in the call stack. In a multi-threaded program, where regions are associated with threads and methods, there is a region tree whose branches are related to each execution thread.

In order to perform scoped-memory management at program level, an API is proposed which differs from the RTSJ one, described in [GB00], in three main points. First, in the proposed API memory scopes are not bound to runnable objects. In this point, this API is closer to the RC library [GA01]. Second, the API does not specify the region where an object will be allocated, but rather a set of regions corresponding to methods in a prefix of the corresponding call stack. The actual region where the object will be allocated at runtime is left out to the implementation. To determine in which region an object will be allocated we use a registering mechanism. Basically, when regions are created, they are informed about the set of creation sites (*new* statements) it will allocate. When object instantiation is requested, the API allocates the object in the last region the creation site was registered in. Finally, there is no Immortal memory; instead, it is simulated by a "main" region with a global scope. The API is shown in Table 6.2.

`enter(r,lCSs)`	push r into the region stack and register the creation sites it will allocate
`exit()`	collect the objects in top region
`newInstance(cs,c)`	create an object identified by the creation site cs of class c
`newAInstance(cs,c,n)`	same but for arrays of dimension n

Table 6.2: Scoped-memory API.

6.3. Eclipse Plug-in: JScoper

The Eclipse Java Development Toolkit (JDT) is one of the most popular and feature rich platforms currently available to Java developers. Because Eclipse is not only an IDE but an extensible plug-in platform, it is the ideal framework to use for the development of tools aimed at transforming Java code. Currently there are few tools that can be used to assist in the conversion of standard Java code to scoped-memory code. An Eclipse plug-in called *JScoper* that fullfils this purpose is presented in this paper. This is a tool that can be used to support both automatic and semi-automatic translation of heap-based Java programs into scope-based ones. Although the resulting programs are not fully compliant with RTJS (this will be supported in the future), they also implement a scope-based memory management mechanism which replaces the garbage collector from the Java Virtual Machine [GNYZ05].

JScoper allows the user to visualize, debug and control the transformation process. Its GUI facilities provide a user-friendly way of gaining insight into the underlying concepts of controlled memory management.

6.3.1. Usage and Features

JScoper makes use of three main windows: the `CallgraphBrowser`, the `Scoped-Memory Java Editor`, and the standard `Java Editor` provided with the Eclipse Java Development Toolkit. It also features additional views that provide alternative representations of the callgraph and memory regions.

The *CallGraph Browser* is used for the visualization of the code callgraph and creation sites corresponding to dynamic memory allocation statements. It also has some editing capabilities: the manual creation of memory regions and the movement

of creation sites between different regions. These editing features are meant to allow for manual adjustment of the automated output of the tool.

The *Scoped-Memory Java Editor* is a source code editor with syntax highlighting support for scoped-memory Java code, as well as special marker icons which act as hyperlinks between the different plug-in windows. These markers will be discussed later.

The *Java Editor* is the standard editor provided with the Eclipse JDT, with additional support for special marker icons analogous to those of the Scoped-Memory Java Editor.

During a normal usage workflow, the user will start from regular Java source code, use the integrated tools to identify the creation sites, perform escape analysis [SYG05] (an optional step) and generate the callgraph, and then examine the resulting graph in the Callgraph Browser window. Memory region and creation site adjustments are possible at this stage. The user may also switch between the three editors (Callgraph, Instrumented and standard Java), using special marker icons which link related memory allocation sites. The final output of the plug-in will be stored as a series of XML files describing memory regions, creation sites and callgraph of the source code. These files are described with more depth in the following section, "Design and Implementation". The workflow consists of the following steps:

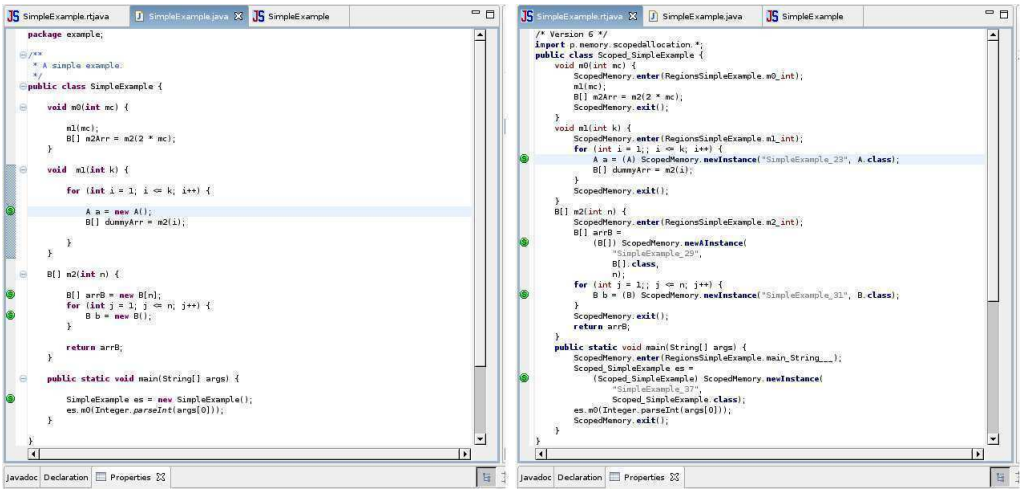


Figure 6.1: A side by side view of the two code editors. Left: the standard Java Editor. Right: the Scoped-Memory Java Editor.

1. Start from Java source: this is the program the developer originally coded, with no concern for real-time issues. Positioned in the package explorer of the Eclipse Java view, the user must select the appropriate options provided by JScoper in order to analyze the code and memory regions (optional) and generate the callgraph. This will create a series of XML files corresponding to the callgraph, memory regions and creation sites, the `rtjava` instrumented code file and a `jscoper` project file which links all the previous files together.
2. Output visualization: the user can now examine the result of the automated code analysis and instrumentation. The Scoped-Memory Java Editor (figure 6.1, right) is used to browse the instrumented code, which is a file with extension `rtjava`. Instrumented Java files contain an extension of Java code with special scoped-memory related statements. This editor can be used to switch to the relevant sections in the original source code, for comparison purposes. In

order to allow this, there are special icons called markers that connect dynamic memory allocation statements in the original Java code with the corresponding statements in the instrumented code. It also links the `java` and `rtjava` files with the callgraph. The user is able to inspect related locations in the original source code, the instrumented code and the callgraph.

The code callgraph is represented visually in a directed graph form (figure 6.2). Nodes represent Java methods and show their corresponding creation sites (dynamic memory allocation statements, like *new*). When a Java method calls another, an arrow with a label stating the line number is drawn to connect the corresponding two nodes in the graph. Each creation site lists the memory regions that capture it. Several filters that can reduce visual clutter and are useful to inspect the code flow are provided: for example, it is possible to trace a path from the root node (which represents the initial caller method) to any selected node in the graph, focus on the subgraph that spans from any given node or hide the region information so that only the code flow is shown. In addition, there are two side views that can also be inspected: a hierarchical tree view of the callgraph and a tree view of the current memory regions. Image snapshots of the callgraph may be exported at any time.

3. Manual adjustments: both the generated memory regions and the creation sites location within those regions may be manually adjusted. If the automatically generated regions are not satisfactory (for example, because they are too conservative), they can be deleted, modified or added at will using a region management window which can be accessed both from the toolbar and from a context menu. This manager also allows the reassigning of creation sites to different regions (figure 6.3).

All intermediate files are persisted to disk storage and can be inspected at any time with a text editor. JScoper can be used to explicitly write the current state of region/creation site mappings at any time.

6.3.2. Design and Implementation

JScoper was developed for the 3.x series of the Eclipse platform. Currently there is no support for versions 2.x or earlier. It was developed and tested in Linux and Windows XP. It has not been tested (yet) on other operating systems, but it should work on any platform supported by Eclipse and Java 1.4.x.

JScoper integrates 4 distinct modules which roughly correspond to the editors described in the previous section, "Usage and Features": the Callgraph Browser, the Scoped-Memory Java Editor, the standard Java Editor and the Backend (which is actually a collection of different tools itself). This work focuses on the frontend of the plug-in.

- The Callgraph Browser handles the visual representation of the program callgraph and allows the manual editing of memory regions and creation sites. This module uses an add-on for Eclipse called GEF, the Graphical Editor Framework [2], which is used to implement the graphical editor following the Model-View-Controller pattern.
- The Scoped-Memory Java Editor is used to inspect and edit the instrumented source code. Special Eclipse markers allow switching to and from creation sites in the regular Java source code and also to the corresponding nodes in the Callgraph Browser window.

[2] See the homepage at http://www.eclipse.org/gef/

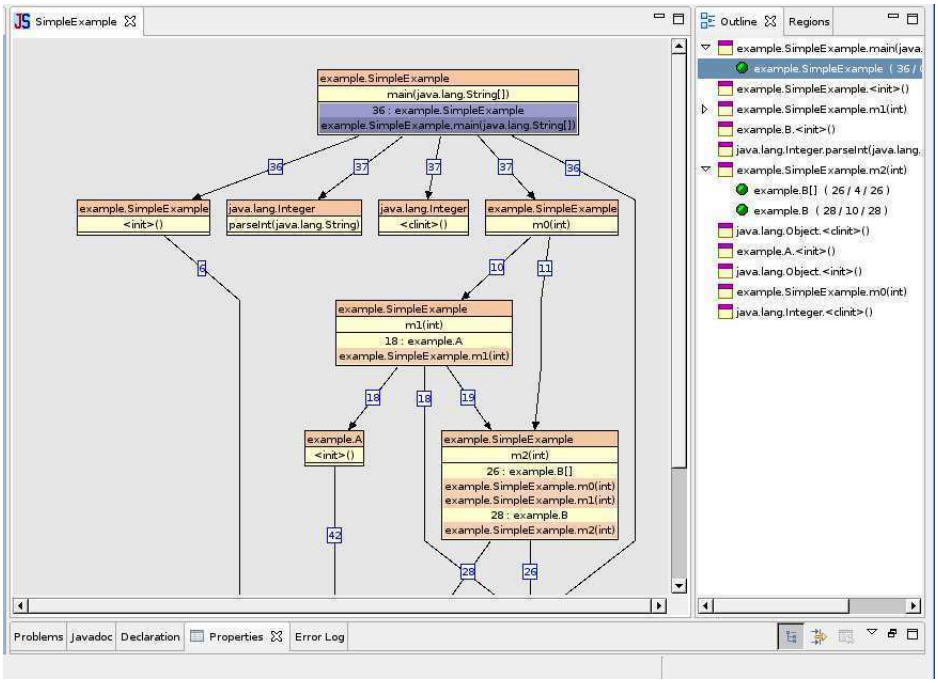


Figure 6.2: The callgraph browser window. The view on the right shows a tree outline of the callgraph.

- The Java Editor mimics the behavior of the standard source editor included with the Eclipse platform, and adds support for the special markers mentioned above.
- The Backend consists of a collection of tools that actually perform the code analysis, including a code instrumentator [GNYZ05], a callgraph generator based on *Soot* [VRHS+99], an escape analysis and region inferrer [SYG05] and a creation sites finder.

A sketch of the plug-in model is shown in figure 6.4. The original `Code Model` is the basis for establishing derived models (and their corresponding views), namely, `Call Graphs` and `Creation Sites`. The `Point of View` defines the abstraction parameters used to obtain call graphs and creation sites (e.g., root method for the analysis, whether or not to include standard Java API creation sites, etc.). The `Region Model` is a mapping from creation sites to sets of regions, and it is used as the input for the instrumentation procedure that generates a `Scoped Code Model`. The `Object Lifetime Model` is an escape analysis [SYG05] representation and holds the relationship between creation sites, the regions that contain them, and their paths within the call graph. This model can be used to either automatically synthesize a Region Model, and in the future it will also be used to validate a manually created one. Each of these models has a corresponding view in the plug-in, with the exception of the Point of View (which is currently unimplemented) and the Object Lifetime, whose graphical visualization, while currently unavailable, will be a call graph coloring.

The interface between the plug-in modules comprises several XML files. Assuming the original Java source file is named `MyClass.java`, then the XML files are:

- The callgraph file, `MyClassCallGraph.xml`. This is an XML that contains graph information in the form of nodes (class methods) with items (creation sites) linked to other nodes (method calls). Each node is identified by a classname and a fully qualified method name, and it has a list of all the "children"

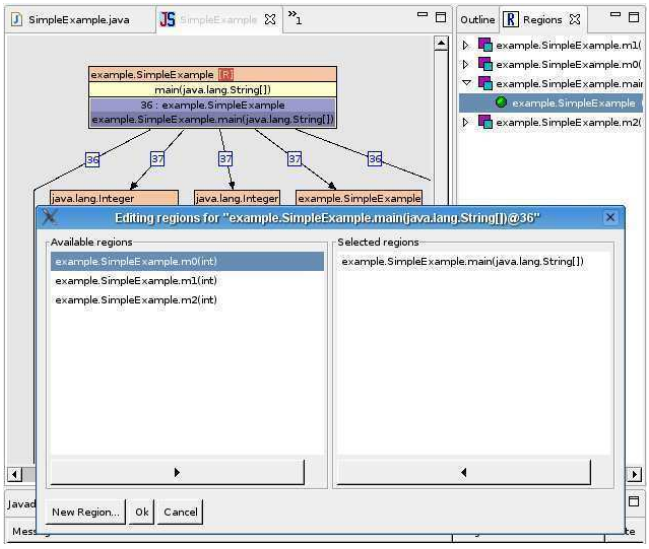


Figure 6.3: The Region Manager.

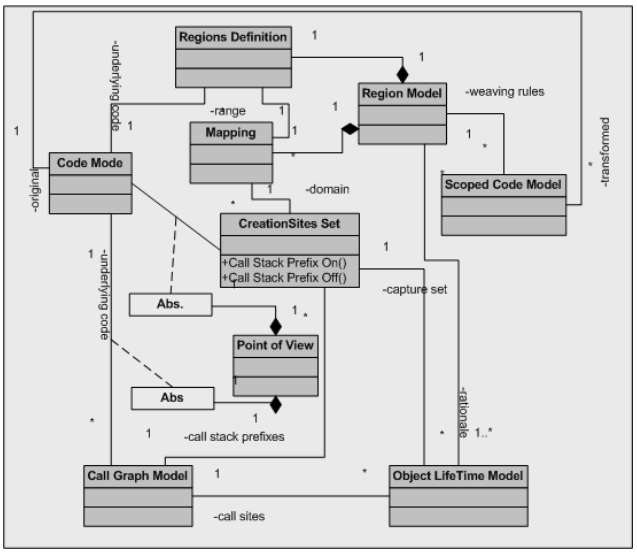


Figure 6.4: Modules of JScoper

or nodes it is linked to. Each child node represents a method that is called from the parent method at the line number specified by attribute *line*. Any arbitrary callgraph may be represented and cycles are possible.

- The creation sites file, `MyClassCreationSites.xml`. This is an XML that lists the line numbers of dynamic memory allocation statements within the Java source code. A simplified version looks like this:

```
<CreationSites id="example.SimpleExample">
    <CreationSite method="m0" line="26"/>
    <CreationSite method="m0" line="27"/>
    <CreationSite method="m1" line="29"/>
    <CreationSite method="m1" line="32"/>
</CreationSites>
```

  Method *m0* in the class *example.SimpleExample* has creation sites at lines 26 and 27, while method *m1* has its sites at 29 and 32.

- The memory regions file, `MyClassRegions.xml` (optional). This is an XML that stores the assignment of creation sites to scoped memory regions. There may

be more than one creation site within any given region. This file is optional; if it is not present when the user tries to visualize a callgraph, JScoper will simply generate default regions named after the corresponding method for each orphan site. A simplified version of this file has the following outline:

```
<Regions>
   <Region id="R1" scope="SimpleExample.m0"
           lineFrom="10" lineTo="28">
      <CreationSite method="m0" line="26" instancesExp="x"/>
      <CreationSite method="m0" line="27" instancesExp="x^2"/>
   </Region>
   <Region id="R2" scope="SimpleExample.m1"
           lineFrom="29" lineTo="50">
      <CreationSite method="m1" line="" instancesExp="2x"/>
      <CreationSite method="m1" line="" instancesExp="x"/>
   </Region>
</Regions>
```

A region description states its scope (essentially, the classname and method where it is located), the line numbers it spans and the creation sites which it contains. Currently, regions cannot cross method or class boundaries but there may be two or more regions within a given Java method.

- The Java to Scoped-Memory Java file, `MyClassCSR.xml`. This is an XML file (similar to the one containing the creation sites) which maps the line number of each creation site to the corresponding line in the instrumented code.

There are two additional files which are not XMLs and have special meanings:

- The instrumented code, `MyClass.rtjava`.
- The JScoper project file, `MyClass.jscoper`, which links all the previous files together.

6.4. Conclusions and Future Work

JScoper is an Eclipse plug-in that assists the automatic translation of standard Java code to a RTJS-like code. It provides a graphical call graph browser that helps ease program understanding, supports the generation and edition of memory regions, automatic code generation and code visualization. JScoper can be downloaded from `http://dependex.dc.uba.ar/jscoper/download.html`.

Future work plans include the implementation of debugging facilities such as runtime browsing of active regions, visualization of object-lifetimes, region-sizes and scoping-rules violations. It is also planned to include full RTSJ compatibility (automatic instrumentation and edition) and support for automatic generation of memory size annotations [BGY05].

CHAPTER 7

Computing memory requirements certificates

This chapter presents a technique to compute symbolic non-linear approximations of the amount of dynamic memory *required* to safely run a method in (Java-like) imperative programs. We do that for scoped-memory management where objects are organized in regions associated with the lifetime of methods. Our approach resorts to a symbolic non-linear optimization problem which is solved using Bernstein basis.

7.1. Introduction

In a previous chapter we presented a technique for computing a parametric upper-bound of the amount of memory dynamically *requested* by Java-like imperative programs [BGY06] (see chapter 2). The idea consists in quantifying dynamic allocations done by a method. Given a method m with parameters $p_1, \ldots, p_k$ we exhibit an algorithm that computes a parametric non-linear expression over $p_1, \ldots, p_k$ which over-approximates the amount of memory allocated during the execution of m. This bound is a symbolic over-approximation of the total amount of memory the application *requests* to a virtual machine via `new` statements, but not the *actual* amount of memory really consumed by the application. This is because memory freed by the garbage collector is *not* taken into account. We also showed that assuming a region-based memory management [GA01, GB00, GNYZ05, CR04] where objects are organized in regions associated with computation units, the same technique allows to obtain non-linear parametric bounds of the size of every memory region.

Here we propose a new technique to over-approximate the amount of memory *required* to run a method (or a program). Given a method m with parameters $p_1, \ldots, p_k$ we obtain a polynomial upper-bound of the amount of memory necessary to *safely* execute the method and all methods it calls, without running out of memory. This polynomial can be seen as a *pre-condition* stating that the method requires that much free memory to be available before executing, and also as a *certificate* engaging the method is not going to use more memory than the specified. To compute this estimation we consider memory deallocation that may occur during the execution of the method. Basically, assuming a region-based memory management we model all the potential configurations of regions stacks at run-time. Since region sizes are expressed as polynomials, this model leads to a symbolic non-linear optimization problem. This problem can be solved using a technique using Bernstein basis [CT04].

Applications of this set of techniques are manifold, from improvements in memory

management to the generation of parametric memory-allocation certificates. These specifications would enable application loaders and schedulers (e.g., [KNY03]) to make decisions based on available memory resources and the memory-consumption estimates.

Outline

In section 7.2 we present a definition of the problem we want to solve and some assumptions that we are making. In section 7.3 propose an effective definition of a function that predict memory requirements for a scoped-based memory management. In section 7.4 propose an approach to compute the memory requirements function. In section 7.6 we discuss some aspects of the technique that we would like to improve. In section 7.7 we discuss some related work and in section 7.8 we present our conclusions and future work.

7.2. Problem statement

```
void m0(int mc) {
1:  m1(mc);
2:  m2(3 * mc);
}
void  m1(int k) {
3:   B[][] dummyArr = new B[k][];
4:   for (int i = 1; i <= k; i++) {
5:     dummyArr[i-1]= m3(i);
  }
}
void  m2(int k2) {
6: B[] m3Arr=m3(k2);
}

B[] m3(int n) {
7: B[] arrB = new B[n];
8: N l  = new N();
9: for (int j = 1; j <= n; j++) {
10:   arrB[j-1] = m4(l,j);
   }
11: return arrB;
}

B m4(N l, int v)
{
12:  N c = new N();
13:  c.value = new B(v);
14:  c.next = l.next;
15:  l.next = c;
16:  return c.value;
}
```

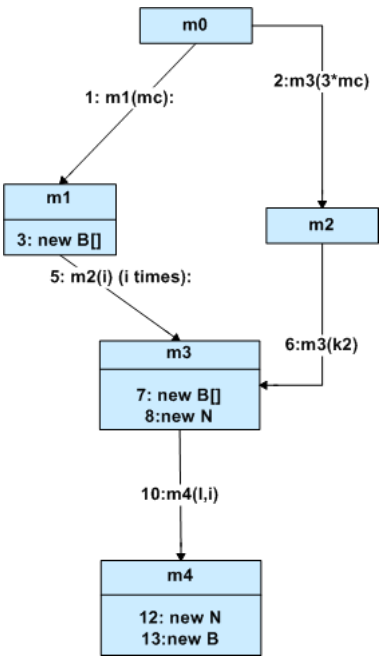


Figure 7.1: A sample program with his detailed call graph

Let us introduce the problem informally with an illustrative example (Fig. 7.1). Method $m0$ calls $m1$ and $m2$. $m1$ allocates an array of size k ($k = mc$ when called from $m0$) and calls k times a method $m3$. $m2$ also calls $m3$ but only once with a different parameter assignment ($k2 = 2mc$ when called from$m0$). $m3$ allocates an array of size n (n ranges from 1 to k when called from $m1$ or $k2$ when called from $m2$), allocates also a node of a list and calls n times to $m4$ that add a node to the list and returns a newly created object of type B.

The objects allocated in method $m4$ at locations 12 and 13 (denoted as $m4.12$ and $m4.13$) cannot be collected when the method finishes its execution because they are referenced from outside (the object of type N is annexed to the list referenced

by l and the object of type B is returned). Thus, we say that both objects *escape* the scope of $m4$. $m4.12$ can be collected when $m3$ finishes its execution. However, $m4.13$ also escapes $m3$ and can be collected just at the end of $m2$ or $m1$ when the last reference is removed. In a similar fashion, $m3.7$ can be collected at the end of $m2$ or $m1$ but $m3.8$ can do it at the end of $m3$. Finally $m1.3$ can be collected by $m1$. Observe that, at $m0$ all objects created during the execution of $m1$ can be collected before the call to $m2$ is performed.

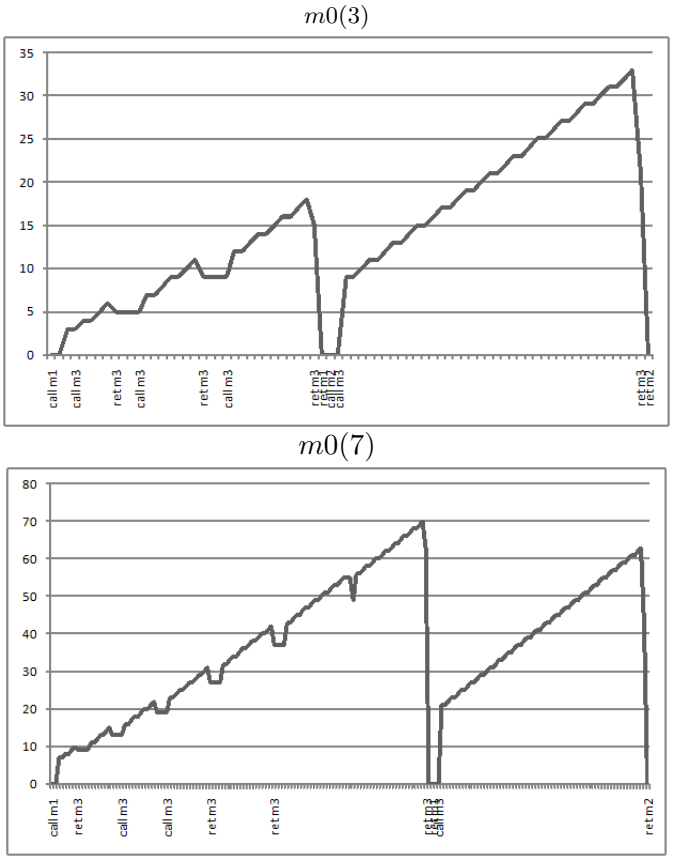


Figure 7.2: Two traces: $m0(3)$ (above) and $m0(7)$ (below).

Fig. 7.2 depicts the effects of allocations and deallocations on memory occupancy for different executions ($m0$ invoked with $mc = 3$ and $mc = 7$, respectively) using an "ideal" garbage-collection scheme where objects are collected as soon as they are no longer referenced. The figure only shows memory occupation generated by explicit requests at allocation statements, that is, there is no allocation overhead introduced by the memory manager. The *peaks* represent the *maximum* amount of memory occupied by the runs. Notice that in the first run the peak is reached in some point after $m0$ calls $m2$, and in the second one, the peak occurs somewhere after $m0$ calls $m1$.

Determining the peak consumption of a run in advance would enable to know "a priori" the amount of memory required to safely execute the run. However, predicting this peak is hard for different reasons. First, it is difficult to estimate the amount of memory requested by a program. Second, it is also challenging to determine when objects will be collected. In fact, the figure shows that different runs may exhibit different peak consumptions on different program states (i.e. variable valuation and program control location).

This example pinpoints the issues that need to be considered in order to formulate in a precise way the problem of computing the memory requirements.

Assume a Java-like program $Prog$ which semantics is given by a transition system $[\![Prog]\!]^{\mathtt{M}} = \langle \Sigma, \sigma^I, \rightarrow^{\mathtt{M}} \rangle$ where Σ set of states, σ^I a set of program initial states and $\rightarrow^{\mathtt{M}}$ a transition relation according to the language semantics using a memory manager $\mathtt{M}$. In a few words, the state of a program in run-time is given by the variable values ($\sigma(v)$ yields the value associated to v), a control location is associated to a special variable pc and a call stack ($\mathsf{stack}(\sigma)$ yields the stack). Given program location $l = \sigma(pc)$, $\mathsf{stm}(l)$ yields the statement to be executed at that program location (i.e., pc pinpoints the next statement that will be executed from σ).

We denote $memUsed^{\mathtt{M}} :: \Sigma \mapsto \mathbb{N}$ the function that returns the amount of dynamic memory occupied in a given program state. We only consider objects created by the program under analysis. Let `ideal` be the collector which frees objects as soon as they are no longer alive (i.e. not reachable from local or stack variables). In this case, $memUsed^{\mathtt{ideal}}$, yields the memory occupied by live objects.

Let $r = \sigma_0, \sigma_1, \ldots$ such that $\sigma_i \rightarrow \sigma_{i+1}$ be a run. We denote r_i to the state corresponding to the i^{th} element of the run. Let $R([\![Prog]\!]^{\mathtt{M}})$ be the set of runs for $Prog$.

We define the *maximum amount of memory* consumed by a method m in a particular run r at particular state r_i corresponding to an invocation to method m as follows:

$$\begin{aligned} peakForRun^r_m(r,i) \;=\; & \max\{memUsed^{\mathtt{M}}(r_k) \mid \; i+1 \leq k \leq t \\ & \qquad \wedge\, \mathsf{stm}(r_i(pc)) = \mathtt{call}\ m \\ & \qquad \wedge\, \mathsf{stm}(r_{t_i}(pc)) = \mathtt{ret}\ m\} \\ & - memUsed^{\mathtt{M}}(r_i) \end{aligned}$$

where r_{t_i} is the corresponding return statement of the invocation to m at r_i. For the sake of simplicity we assume that m does terminate every time is invoked.

The amount of memory consumed by a method m with formal parameters $\vec{P_m}$ when invoked with arguments $\vec{x} : \vec{T_m}$, denoted $Peak^{\mathtt{M}}_m(\vec{x})$, is defined as the maximum of $peakForRun^{\mathtt{M}}_m$ over all traces that invoke m with arguments $\vec{x}$:

$$\begin{aligned} Peak^{\mathtt{M}}_m(\vec{x}) \;=\; & \max\{peakForRun^r_m(r,i) \mid r \in R([\![Prog]\!]^{\mathtt{M}}) \\ & \qquad \wedge\, \mathsf{stm}(r_i(pc)) = \mathtt{call}\ m \\ & \qquad \wedge\, r_i(\vec{P_m}) = \vec{x}\} \end{aligned}$$

In this definition we are assuming that the peak function has the same input parameters as the method definition: $Peak^{\mathtt{M}}_m : \vec{T_m} \mapsto \mathbb{N}$. In section 7.6 we discuss how we can support a more liberal definition that allows the use of different expressions as parameters of memory requirements.

$Peak^{\mathtt{ideal}}_m(\vec{x})$ gives the *least upper-bound* of the amount of memory needed to run method m with parameters $\vec{x}$. Any other garbage collector $\mathtt{M}$ will not liberate memory earlier that this ideal policy:

$$\forall m, \vec{x} : \vec{T_m} \cdot Peak^{\mathtt{ideal}}_m(\vec{x}) \leq Peak^{\mathtt{M}}_m(\vec{x})$$

Our aim is to get a parametric upper bound of the amount of memory required to safely run a method under ideal conditions. Thus, the goal of this work is to approximate the above mentioned least upper-bound by a function $\mathtt{memRq} : \vec{T_m} \mapsto \mathbb{N} \cup \infty$ such that:

$$\forall \vec{x} : \vec{T_m} \cdot Peak^{\mathtt{ideal}}_m(\vec{x}) \leq \mathtt{memRq}_m(\vec{x})$$

7.3. A Peak Overapproximation for Scoped-memory

The `ideal` memory manager is optimal in terms of memory consumption. This collector is used in works that verify memory usage certificates such as [CNQR05, BPS05], etc. However, it is not well understood how to infer memory consumption for it, especially if the expression is not linear in terms of method parameters or object are not deallocated manually.

In this work we follow a different strategy: we assume the presence of a scoped-memory manager to over approximate memory requirements. Thus, this will not only lead to a solution to the original general problem (an over approximation of `ideal`), but it will provide the memory requirements for a predictable garbage collection in embedded applications.

More specifically, our proposal is the use of a scoped-based memory collection mechanism that reclaims memory only at the end of the execution of every method. Besides, the collector is only allowed to claim for non-live created during the execution of the method (and the method it transitively calls). Objects created in an outer scope cannot be collected by the current method and may be reclaimed by some of the methods in the call stack.

In particular, we will choose a scoped-based memory management where objects are organized in regions and each method has an associated region (denoted as an m-region) whose lifetime corresponds with its associated method's lifetime [GNYZ05]. To be safe, objects in a region can point to objects in the same region or a parent region (corresponding to a method that is in the call stack). This scoping restriction can be satisfied inferring the regions at compile-time by performing escape analysis [GNYZ05, SYG05].

We will assume that two parametric memory-consumption specifications are given for each method\region: `memCaptured` and `memEscapes`. Given a method m, `memCaptured`(m) yields an over approximation, in terms of method m parameters, of the size of the region associated to m. It can also be seen as the amount of dynamic memory temporally occupied by the objects created during the execution of m that can be safely collected when m finishes it execution. `memEscapes`(m) yields an over approximation, in terms of method m parameters, of the amount of dynamic memory allocated by objects created during the execution of m that cannot be released, meaning that they has be allocated in other callers regions. `memEscapes` provides useful information to the callers of that method as they must consider that the call to that method will require some additional space of their own regions. In [GNYZ05, SYG05] we proposed techniques to automatically infer memory regions and in [BGY06] we proposed a technique to automatically infer `memCaptured` and `memEscapes` .

Example For instance, we can compute the following escape and capture information for our motivating example:

	`memCaptured`	`memEscapes`
$m0$	0	0
$m1$	$\texttt{size}(B[])k + (\texttt{size}(B[]) + \texttt{size}(B)).(\frac{1}{2}k^2 + \frac{1}{2}k)$	0
$m2$	$(\texttt{size}(B[]) + \texttt{size}(B)).k2$	0
$m3$	$\texttt{size}(N) + \texttt{size}(N).n$	$(\texttt{size}(B[]) + \texttt{size}(B)).n$
$m4$	0	$\texttt{size}(B) + \texttt{size}(N)$

□

Given a method m we know how to compute the size of its associated region. But this is not enough. To compute the amount of memory required to run a method we need to consider also the size of the regions of every method that may be called during the execution of m.

There are two important facts to consider:

1. There are some region stack configurations that cannot happen at the same time.

2. Although a method can be potentially invoked several times with the same configuration stack, there will be at most one active region instance per method. Its size may change according to the calling context (the value assigned to its parameters each time it is invoked).

To illustrate the first fact consider method $m0$ in the example of Fig. 7.1. At location $m0.1$, $m0$ calls $m1$ which calls $m3$ and then calls $m4$. Similarly, at location $m0.2$, $m0$ calls $m2$ which calls $m3$ and then calls $m4$ (see figure 7.3). Under our region-based memory management there will be two independent region stacks. One stack will be formed by the regions in the call chain $m0 \xrightarrow{1} m1 \xrightarrow{5} m3 \xrightarrow{10} m4$, and the another stack with the regions in the call chain $m0 \xrightarrow{2} m2 \xrightarrow{6} m3 \xrightarrow{10} m4$. These two chains are independent as they cannot be simultaneously active. In particular the region stack for the call chain $m1 \xrightarrow{5} m3 \xrightarrow{10} m4$ is completely collected before calling method $m2$.

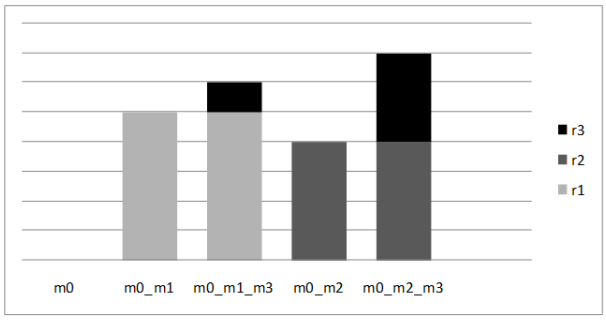


Figure 7.3: Potential region stacks for the sample

Now, To illustrate the second fact consider the call chain $m0 \xrightarrow{1} m1 \xrightarrow{5} m3$. The the method $m3$ will the called k times with its parameter n assigned with an argument varying from 1 to k. Each time $m3$ is called an $m3$-region is created which is completely collected when the method returns the control to its caller. Since there will be only one region active for $m3$ it suffices to consider the maximum size the region can reach according to its calling context (e.g. all calls from $m0.1.m1.5$). In this case, the region is maximized when $n = k$. Since we need to compute the requirements for $m0$ (the MUA) we need a way to represent the maximum region for $m3$ in terms of $m0$ parameters instead of $m3$ parameters. In Fig. 7.4 we show the evolution of $m3$-regions when $m0$ is called with $mc = 3$.

7.3.1. Memory required to run a method

Given a MUA mua, let $\texttt{rSize}^{\pi.m}_{mua}$ be a function from mua parameters which yields the size of the largest m-region instance created by any call to m with control stack π in a program which starts with an invocation to mua with `rSize` arguments.

Suppose we can compute `rSize` for each method in each call chain. Then, to compute the amount of memory required to run a method mua, we basically need

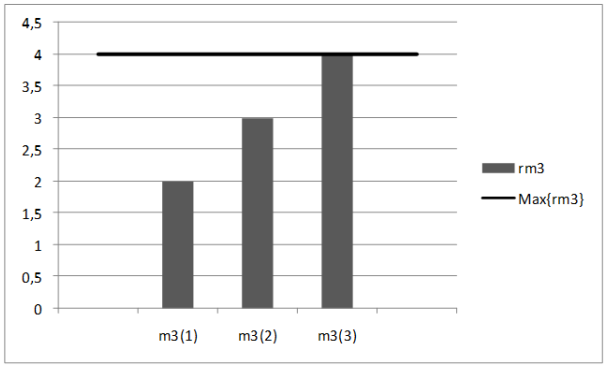


Figure 7.4: Evolution of $m3$-region sizes for a when $m0$ is called with $mc = 3$.

to consider the size of its own region and add the amount of memory required to run every method it calls. Since every branch will launch an independent region stack, we can select only the branch the would require the maximum amount of memory. In general, this function can be defined as follows:

$$\begin{aligned}\texttt{memRq}^{\pi.m}_{mua}(p_{mua}) \;=\; & \texttt{rSize}^{\pi.m}_{mua}(p_{mua}) + \\ & \max\{\texttt{memRq}^{\pi.m.l.m_i}_{mua}(p_{mua}) \mid (m,l,m_i) \in edges(CG_{mua} \downarrow \pi.m)\}\end{aligned}$$

where $CG_{mua} \downarrow \pi.m$ is a projection over the path $\pi.m$ of the call graph of method mua and $edges$ is the set of its edges.

Note that this recursive definition leads to an *evaluation tree* where the leaves are related with `rSize` operations and nodes with max or sum operations. We will show later some options on how to reduce and evaluate this evaluation tree.

Observe also that in order to properly define $\texttt{memRq}^{\pi}_{m}$ we must rule out recursive calls. Mutually recursive components has to be removed by program transformation or manually provide a requirement specification for every strongly connected component.

Example The amount of memory required to run $m0$ can be modeled as:

$$\begin{aligned}
\texttt{memRq}^{m0}_{m0}(mc) &= \texttt{rSize}^{m0}_{m0}(mc) \\
&\quad + max\{(\texttt{memRq}^{m0.1.m1}_{m0}(mc), \texttt{memRq}^{m0.2.m3}_{m0}(mc)\} \\
\texttt{memRq}^{m0.1.m1}_{m0}(mc) &= \texttt{rSize}^{m0.1.m1}_{m0}(mc) + \texttt{memRq}^{m0.1.m1.5.m3}_{m0} \\
\texttt{memRq}^{m0.1.m1.5.m3}_{m0}(mc) &= \texttt{rSize}^{m0.1.m1.5.m3}_{m0}(mc) \\
&\quad + \texttt{memRq}^{m0.1.m1.5.m3.10.m4}_{m0}(mc) \\
\texttt{memRq}^{m0.1.m1.5.m3.10.m4}_{m0} &= \texttt{rSize}^{m0.1.m1.5.m3.10.m4}_{m0}(mc) \\
\texttt{memRq}^{m0.2.m2}_{m0} &= \texttt{rSize}^{m0.2.m2}_{m0}(mc) + \texttt{memRq}^{m0.2.m2.6.m3}_{m0}(mc) \\
\texttt{memRq}^{m0.2.m2.6.m3}_{m0}(mc) &= \texttt{rSize}^{m0.2.m2.6.m3}_{m0}(mc) \\
&\quad + \texttt{memRq}^{m0.2.m2.6.m3.10.m4}_{m0}(mc) \\
\texttt{memRq}^{m0.2.m2.6.m3.10.m4}_{m0}(mc) &= \texttt{rSize}^{m0.2.m2.6.m3.10.m4}_{m0}(mc)
\end{aligned}$$

These expressions can be reduced to:

$$\begin{aligned}\texttt{memRq}^{m0}_{m0}(mc) \;=\; & \texttt{rSize}^{m0}_{m0}(mc) + \\ & \max\{\texttt{rSize}^{m0.1.m1}_{m0}(mc) + \texttt{rSize}^{m0.1.m1.5.m3}_{m0}(mc) \\ & \qquad + \texttt{rSize}^{m0.1.m3.5.m3.10.m4}_{m0}(mc), \\ & \quad \texttt{rSize}^{m0.1.m2}_{m0}(mc) + \texttt{rSize}^{m0.2.m2.6.m3}_{m0}(mc) + \\ & \qquad \texttt{rSize}^{m0.2.m2.6.m3.10.m4}_{m0}(mc)\}.\end{aligned}$$

□

$\texttt{memRq}^{\pi.m}_{mua}$ computes an over approximation of the amount required to be able to allocate all the regions that can be active at the same time.

Recall that in our model $\texttt{memCaptured}(m)$ over approximates the size of the m-region and $\texttt{memEscapes}(m)$ is an over approximation of the amount of memory that is allocated during the execution of m and cannot be released. Thus, we still need to consider the amount of memory that is not collected. However, we only need to take into account the amount of memory escaping mua as escape information is absorbent. By absorbent we mean that any object escaping the scope of a method m, transitively called by mua, is eventually captured by some method in the call stack prefix defined by $mua, \ldots, m$ and, thus, it will be considered in $\texttt{memRq}^{mua}_{mua}$, or in the worst case, it will escape mua. Therefore it suffices to consider the amount of memory escaping mua.

Finally, a function that can be used to predict the amount of memory required to run a method is defined as follows:

$$\boxed{\texttt{memRq}_{mua}(p_{mua}) = \texttt{memEscapes}(mua)(p_{mua}) + \texttt{memRq}^{\pi.m}_{mua}(p_{mua})}$$

7.3.2. Defining the function rSize

Now, we will focus on how to define the function rSize. To do that we will introduce the idea using the example of Fig. 7.1.

In the example method $m0$ calls method $m1$ which calls k times method $m3$. At each invocation the size of the $m3$-region changes because it is defined in terms of the parameter n. Then, the expression for $\texttt{rSize}^{m0.1.m1.5.m3}_{m0}$ has to be the maximum region size for method $m3$ among all the k possible ones. In order to obtain such an expression, it is necessary to provide some sort of information about the calling context that constraints the instantiation of the invoked method (in this case $m3$) when called from the MUA (in this case $m0$) with a given call-stack (in this case given by $m0.1.m1.5$).

To provide this information we resort to *binding invariants*. A binding invariant is used to (transitively) bind the MUA parameters with the parameters of method m following a call chain. It constrains the possible valuation of variables stored in stack frames when method m is invoked from the MUA following that call chain. Binding invariants can be obtained from local invariant as described in [BGY06].

For instance a valid binding invariant for the call chain $m0.1.m1.5.m3$ is

$$\{k = mc, 1 \leq i \leq k, n = i\}$$

Since the $m3$-region is defined by the expression $size(N) + size(N).n$, the largest region instance is produced by the assignment $n = i = k = mc$ which respects the invariant and maximizes the value of the expression. Then,

$$\texttt{rSize}^{m0.1.m1.5.m3}_{m0}(mc) = size(N) + size(N).mc$$

As we mentioned, $\texttt{rSize}^{\pi.m}_{mua}$ is a function (in terms of mua parameters) representing the size of the largest region created by any call to m with a control stack π

considering a program starting with mua. It can be defined as follows:

$$\begin{aligned}\texttt{rSize}^{\pi.m}_{mua}(P_{mua}) \quad = \quad & (\text{Maximize } \texttt{memCaptured}(m)(P_m) \\ & \text{subject to } \mathcal{I}^{mua}_{\pi.m}(P_{mua}, P_m, W))\end{aligned}$$

Notice that $\mathcal{I}^{mua}_{\pi.m}$ is treated as a function over three set of variables: P_{mua} (method mua parameters), P_m (method m parameters), and W are local variables appearing in the methods belonging to the call chain π. It is a binding invariant for the call chain $\pi.m$ and it models the admitted valuations of variables for a call stack prefix given by π (valid call stack configurations when mua calls m passing through π). $\texttt{memCaptured}(m)$, is the parametric expression for the memory captured by m, which is only in terms of P_m. Their parameters are related with mua parameters using the binding invariant.

In the example, we can approximate the maximum size of the region for $m3$ considering it is call from $m0.1.m1.5$ as follows:

$$\begin{aligned}&\texttt{rSize}^{m0.1.m1.5.m3}_{m0}(mc) = \\ &= \max\{\texttt{size}(N) + \texttt{size}(N) \cdot n\} \text{ s.t.}\{k = mc, 1 \leq i \leq k, n = i\} \\ &= \max\{\texttt{size}(N) + \texttt{size}(N) \cdot n\} \text{ s.t.}\{1 \leq n \leq mc\} \\ &= \texttt{size}(N) + \texttt{size}(N) \cdot mc\end{aligned}$$

To calculate $\texttt{rSize}^{m0}_{m0}$, no maximization is required since the region for the root method is activated only once and it is already expressed in terms of its parameters. In table 7.1 shows the resulting expressions for $\texttt{rSize}^{\pi.m}_{m0}$ for every possible region for the example of Fig. 7.1.

$\pi.m$	$\mathcal{I}^{m0}_{\pi.m} \setminus \texttt{rSize}^{\pi.m}_{m0}(mc)$
$m0$	true 0
$m0.1.m1$	$\{k = mc\}$ $(\texttt{size}(B[]) + \texttt{size}(B)).(\frac{1}{2}mc^2 + \frac{1}{2}mc) + \texttt{size}(B[])mc$
$m0.1.m1.5.m3$	$\{mc \geq 1, k = mc, 1 \leq i \leq k, n = i\}$ $\texttt{size}(N) + \texttt{size}(N)mc$
$m0.1.m1.5.m3.10.m4$	$\{mc \geq 1, k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, v = j\}$ 0
$m0.2.m2$	$\{k_2 = 3mc\}$ $(\texttt{size}(B[]) + \texttt{size}(B))3mc$
$m0.2.m2.6.m3$	$\{k_2 = 3mc, n = k_2\}$ $\texttt{size}(N) + \texttt{size}(N)3mc$
$m0.2.m2.6.m3.10.m4$	$\{mc \geq 1, k_2 = 2mc, n = k_2, 1 \leq j \leq n, v = j\}$ 0

Table 7.1: Expression for function `rSize` for the example

Using the resulting `rSize` expressions we can reduce $\texttt{memRq}_{m0}$ to:

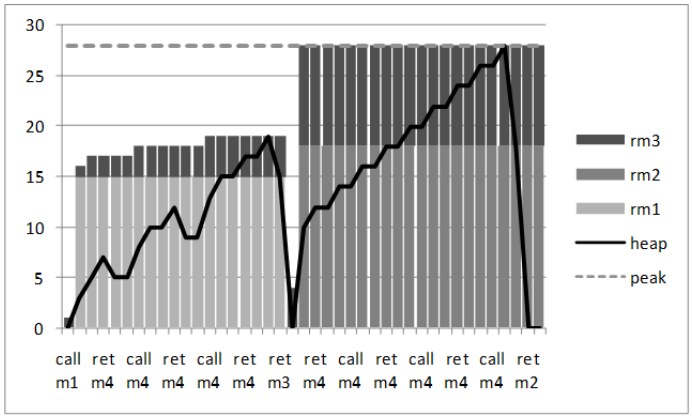


Figure 7.5: Consumption for $m0(3)$ together with the estimated memory requirements.

$$\begin{aligned}
\texttt{memRq}_{m0}(mc) &= 0 + \texttt{memRq}^{m0}_{m0}(mc) \\
&= 0 + 0 + \max\{(\texttt{size}(B[]) + \texttt{size}(B))(\frac{1}{2}mc^2 + \frac{1}{2}mc) \\
&\qquad + \texttt{size}(B[])mc + \texttt{size}(N) + \texttt{size}(N)mc, \\
&\qquad (\texttt{size}(B[]) + \texttt{size}(B))3mc \\
&\qquad + \texttt{size}(N) + \texttt{size}(N)3mc\} \\
&= \max\{(\texttt{size}(B[]) + \texttt{size}(B))(\frac{1}{2}mc^2 + \frac{1}{2}mc) + \texttt{size}(B[])mc \\
&\qquad + \texttt{size}(N) + \texttt{size}(N)mc, \\
&\qquad (\texttt{size}(B[]) + \texttt{size}(B))3mc + \texttt{size}(N) + \texttt{size}(N)3mc\}
\end{aligned}$$

Actual sizes of types are machine or language specific. We will assume their sizes are known at compile time and for our analysis can be considered as constants. Nevertheless, our technique can treat types as parameters and let the decision of assigning a particular size to a type to run-time. Here, for simplicity, we will assume that $\texttt{size}(T) = 1$ for all T.

Under this assumption $\texttt{memRq}_{m0}(mc)$ can be reduced to:

$$\begin{aligned}
\texttt{memRq}_{m0}(mc) &= 0 + \max\{mc^2 + 2mc + 1 + mc, 6mc + 1 + 3mc\} \\
&= 1 + \max\{mc^2 + 3mc, 9mc\} \\
&= 1 + 3mc + \max\{mc^2, 6mc\} \\
&= 1 + 3mc + \begin{cases} mc^2 & \text{if } mc < 0 \vee mc > 6 \\ 6mc & \text{if } 0 \leq mc \leq 6 \end{cases}
\end{aligned}$$

In figure 7.5 we show that the $\texttt{memRq}_{m0}$ is an upper-bound of the actual memory requirements of the example of Fig. 7.1. $rm1$, $rm2$, $rm3$, stand respectively for the region sizes for methods $m1$, $m2$, $m3$. *ideal* stands for the ideal consumption when $m0$ is invoked with $mc = 3$. $\texttt{memRq}(3)$ is the parametric prediction instantiated with $mc = 3$. This figure also shows how regions are created when methods are invoked and released when methods return control to their callers.

The formulation of `rSize` characterizes a non-linear maximization problem whose solution is an expression in terms of *mua* parameters. To avoid expensive run-time computations we need to perform off-line reduction as much as possible at compile

time. Off-line calculation also means that the problem must be stated parametrically. As a consequence, it is not adequate the use of standard non-linear optimization techniques.

7.4. Computing `rSize` and `memRq`

In this section we will show an effective method to solve the previously stated maximization problems and some strategies to evaluate the memory requirement expressions provided by the presented technique.

7.4.1. Computing `rSize`

Recall that `rSize` is a function in terms of the MUA that over approximates the largest size of the region associated with a method invocation and a given control stack. Once arguments are given, `rSize` is a non-linear maximization problem where the polynomial `memCaptured` represents the input and the binding invariant for the control stack represents the restriction.

As we stated, getting off-line a parametric easy-to-evaluate solution of `rSize` would avoid expensive run-time computations. Taking this into account, we based our approach in a work presented by Clauss et al. in [CT04]. It proposes an extension of Bernstein expansion [Ber52, Ber54] for handling parameterized multivariate polynomial expressions. Bernstein expansion allows simbolically bounding the range of a multivariate polynomial over a linear domain. Bernstein polynomials are special polynomials that form a basis for the space of polynomials. Expressing a polynomial in that basis gives minimum and maximum bounds on the polynomial values, represented by particular coefficients (in the new basis). Involved calculation is symbolic, and it could be calculated through a direct formula. Thus, this approach can be used to solve our optimization problem.

In this work we are not going into the details of this technique. An interested reader can find them in C. We use the approach as a "black box" meaning that we assume the existence of a function

$$bernstein : \mathbb{Q}[x_1, \ldots, x_k] \times \mathbb{Q}^{|x_1,\ldots,x_k,p_1,\ldots,p_n|} \mapsto \mathbb{P}(\mathbb{Q}^{|p_1,\ldots,p_n|} \times \mathbb{P}(\mathbb{Q}[p_1, \ldots, p_n]))$$

that is, given a polynomial $pol(x_1, \ldots, x_k)$ and a parametric domain given as a convex polytope I over variables $\{x_1, \ldots, x_k\}$ and parameters $p_1, \ldots, p_n$, yields a set of pairs $(D_i, CanSet_i), i \in [1, l]$ where D_i is a domain defined in terms of $p_1, \ldots, p_n$ and $CanSet_i$ is a set of "candidate" polynomials also in terms of $p_1, \ldots, p_n$ such that, for all $\vec{p}$:

$$\max_{I(\vec{p},\vec{x})} pol(\vec{x}) \leq \begin{cases} \max_j\{q(\vec{p}) \in CanSet_1\} & \text{if } D_1(\vec{p}) \\ \ldots & \\ \max_j\{q(\vec{p}) \in CanSet_l\} & \text{if } D_l(\vec{p}) \end{cases}$$

Example Applying Bernstein's to the polynomial n with the parametric polytope $\{1 \leq i \leq P_1 + P_2, i \leq 3P_2, n = i\}$ yields the following result:

Domain: $\{P_1 + P_2 \geq -1, 3P_2 \geq 1\}$ Candidates: $\{P_2 + P_1\}$

Domain: $\{P_1 \geq 2P_2, 3P_2 \geq -1\}$ Candidates: $\{P_2 + P_1\}$

Domain: Otherwise Candidates: $\{0\}$

□

We compute `rSize` by applying the Bernstein expansion to `memCaptured` (the input polynomial), constrained by a binding invariant I which requires to be a linear parametric invariant. The parameters are *mua* parameters. As we mentioned, the output is a list of domains $D_1, \ldots, D_l$ and for each D_i several polynomials (in terms of *mua* parameters) representing candidates for symbolic upper and lower bounds of `memCaptured` in the domain D_i. For instance, in Table 7.2 we show the results of computing `rSize` for the regions of the example in figure 7.1.

$\texttt{rSize}_{m0}^{m0.1.m1.5.m3} =$	$\texttt{bernstein}(\mathcal{I}_{m0.1.m1.5.m3}^{m_0}, \texttt{memCaptured}(m3))$
Domain:	$\{mc \geq 1\}$
Candidates:	$\{mc + 1\}$
Domain:	$\{mc < 1\}$
Candidates:	$\{0\}$
$\texttt{rSize}_{m0}^{m0.2.m2.6.m3} =$	$\texttt{bernstein}(\mathcal{I}_{m0.2.m2.6.m3}^{m_0}, \texttt{memCaptured}(m3))$
Domain:	true
Candidates:	$\{3mc\}$
$\texttt{rSize}_{m0}^{m0.1.m1} =$	$\texttt{bernstein}(\mathcal{I}_{m0.1.m1}^{m_0}, \texttt{memCaptured}(m1))$
Domain:	true
Candidates:	$\{mc^2 + 2mc\}$
$\texttt{rSize}_{m0}^{m0.2.m2} =$	$\texttt{bernstein}(\mathcal{I}_{m0.2.m2}^{m_0}, \texttt{memCaptured}(m2))$
Domain:	true
Candidates:	$\{6mc\}$

Table 7.2: Computing the function `rSize` using Bernstein basis

Although this solution is a promising approach to cope with our maximization problem, still has a drawback: the result is not simply a polynomial representing the maximum value. It may yield a set of different domains and for each domain a set of candidate polynomials. This means that, in order to evaluate this expression, it is necessary to decide first which domain holds for the input values. Thus, the cost of evaluation is related with the number of domains obtained by the method. Another problem is that given a domain, in general it is not easy to decide (symbolically) which of the candidates polynomials is actually the greatest one within that domain. This problem is similar to the maximum for `memRq` and can be handled analogously. That is, adding polynomials into the evaluation tree for run-time evaluation.

7.4.2. Evaluating `memRq`

We will discuss in this section how to deal with the formula of `memRq` presented in 7.3.1. Recall that `memRq` is defined recursively by traversing the application call graph:

$$\begin{aligned}\texttt{memRq}_{mua}^{\pi.m}(p_{mua}) \quad = \quad & \texttt{rSize}_{mua}^{\pi.m}(p_{mua}) + \\ & \max\{\texttt{memRq}_{mua}^{\pi.m.l.m_i}(p_{mua}) \mid (m, l, m_i) \in edges(CG_{mua} \downarrow \pi.m)\}\end{aligned}$$

Applying this recursive procedure leads to an evaluation tree where expressions in the tree are in terms of the MUA parameters. Nodes in the tree represents operations like maximums and sums between expressions the leaves are operations that yields expression in terms of method parameters.

An evaluation tree for our example is presented in Fig. 7.6. The tree has a direct relation with the application call graph. The *max* node is associated with a branch

in the call graph (i.e. independent regions). The *sum* node is related with adjacency relation in the call graph (i.e. regions that can live at the same time). Finally the leaves in the tree are associated with the nodes the in the unfolded version of call graph (i.e. potential memory regions) using `rSize` as the operation to obtain the largest region size.

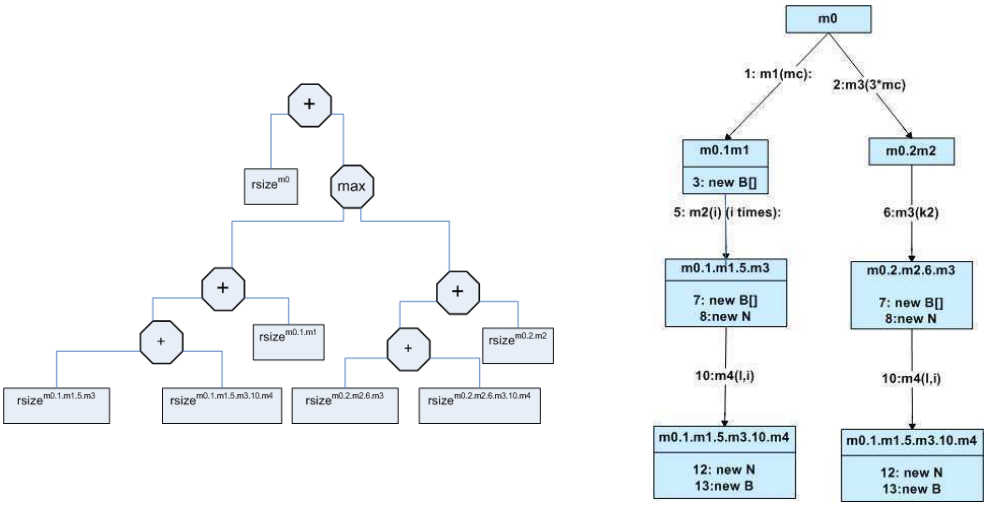


Figure 7.6: Evaluation tree showing the operations involved in the computation of the amount of memory required to run $m0$ and it correlation with the application (unfolded) call graph

```
data ET<T1,..,TK> = Max [ET<T1,..,TK>] | Sum [ET<T1,..,TK>] | Pol P<T1,..,TK>
                  | Cases [(<T1,..,TK> -> Bool), ET<T1,..,TK>)]

eval::  ET<T1,..,TK> -> Nat ∪ ∞
eval Max e1...en = max { eval(e1),..., eval(en) }
eval Sum e1...en = sum { eval(e1),..., eval(en) }
eval Pol p args = evalPol p a1..an
eval Cases (c1,e1)..(cn..en) args = if(c1 args) eval e1 args
                          else if...
                          else if(cn args) eval en args
```

Figure 7.7: Function for evaluating an evaluation tree

To reduce the number of involved variables, we assume that `size(Type)` expressions where replaced by the corresponding size of the type for the underlying architecture. For simplicity, we choose `size`(T)=1 for all T.

In order to compute `memRq`, for each node in the call graph, we have to select the maximum polynomial among the polynomials that represent the requirements for each branch. That selection can be easily done in run-time when MUA actual parameters are available and polynomials can be evaluated. However, when trying to reduce the tree off-line, the maximization need to be handled symbolically, possibly splitting the domains into sub-domains where a polynomial always is larger than the others. For instance, consider $P_1(n) = n^2$ and $P_2(n) = 3n + 1$. Then $\forall n \in \mathbb{N} \cdot P_1 > P_2 \iff n > 3$.

Thus, in order to keep precision at the expense of some run-time calculations, it is possible to leave some (unsolved) maximum expressions for runtime evaluation or

at least generate a function that evaluates to different polynomials depending on the function arguments. In any case, the number of calculation to perform is known at compile time and, in the worst case, we will have to perform a number of evaluations proportional to the number of edges of the call graph.

In Fig. 7.7 we show a Haskell like code defining evaluation trees and a function to evaluate them. The constructor `Pol` defines a leaf in the tree and is used to construct a polynomial. They can be, for instance, the output of the `rSize` function when it yields a simple polynomial or a user provided estimation. A `Case` constructor provides a more general construction and models a set of pairs $(condition, expression)$. Only the first evaluation tree whose condition is satisfied is actually evaluated. This can be used to model the output of `rSize` when it is split into several domains. Since *expression* is an evaluation tree we can also codify a maximization operation for the case when *bernstein* yields more than one candidate for a domain or the case when a maximum operation can be partially solved by splitting the domain into several parts.

At the end, we can automatically translate evaluation trees to Java code that can be evaluated at runtime. In this way, we can obtain the numerical prediction of the memory requirements just before running the chosen method when the method's arguments are available. Although the evaluation may lead to some overhead, the worst case complexity of the number of evaluations is known a priori ($O(egdes(CG_{mua}))$) and in practice the size of the evaluation tree is much smaller, since most of the maximum comparisons can be solved off-line.

The reduction can be achieved, for instance, by applying powerful symbolic techniques or by assuming some loss of precision of the upper-bounds. We can think as a function that takes an evaluation tree and yield a new, ideally easier to evaluate, evaluation tree.

If precision were not an issue, a new polynomial, larger than everyone involved, can be derived for instance by taking the largest coefficient for each degree of the polynomials. For example, given $P_1(n) = n^2$ and $P_2(n) = 3n+1$ we can safely choose $P3(n) = n^2 + 3n + 1$ whose evaluation will be an over-approximation of $P1$ and $P2$.

Fortunately, in some cases, it is known how to symbolically obtain the maximum between the polynomials (yielding directly the largest one, or a segmented function of polynomials). Some typical cases are:

1. Candidates are linear expressions. In this case, we select the largest one by solving a set of linear equations.

2. Candidates are polynomials expressed in terms of only one parameter. In this case, we can apply techniques like Sturm [Hei71] which given two polynomials yields the domains where the first is larger than the second.

There are cases where we can obtain the maximum by simply comparing the degree and the sign of the polynomials or by checking whether the difference is always positive in the analyzed domain. In the general case, we cannot apply these ideas. There are techniques like [Ped91] to deal with this problem, but we have not analyzed them to know if they are suitable for our problem. We are testing another approach which is based on applying the Bernstein expansion recursively in order to gradually reduce the number of variables until we could apply the solutions mentioned before.

In Fig. 7.8 we show the evolution of the evaluation tree for the example Fig. 7.1. The first tree is the evaluation tree after applying Bernstein for solving the maximization problem for `rSize`. The next two trees are successive simplifications.

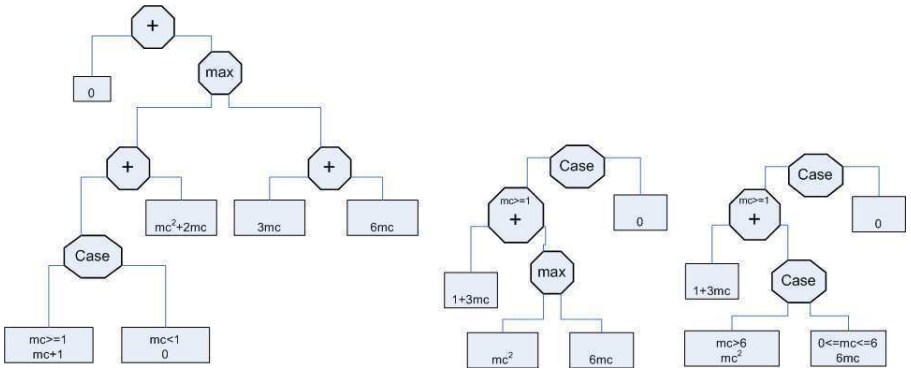


Figure 7.8: Evaluation tree after computing `rSize`. Then two successive reductions.

To go from the first tree to the second one, we start by removing the `Case` node by taking directly the case $m+1$ by using the fact that the binding invariant forces $mc \geq 1$. Then, we sum the nodes in the left part of the `max` node getting the expression $1{+}3mc{+}mc^2$. Since $3mc$ appears also in the right side ($6mc = 3mc{+}3mc$) we can factor that node. Then, to move from the second tree to the third we convert the `max` node to a `Case` node after finding the interval where one polynomial mc^2 is above $6mc$ and vice versa. In Fig. 7.9 we show how the resulting evaluation tree can be translated into code for runtime evaluation.

```
Class Requirements {
   long m0(int mc)
   {
      long required = 0;
      if(mc>=1)
      {
         required = 1 + 3*mc;
         if(mc>6)
            required += mc*mc;
         if(mc<=6)
            required += 6*mc;
      }
      return required;
  }
  ...
}
```

Figure 7.9: Code generated from an evaluation tree

7.5. Experiments

The initial set of experiments were carried out on a subset of programs from JOlden [CM01] benchmarks. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications the target application classes we had in mind when we devised the technique. In order to make the result more readable, the tool computes the number of object instances created when running the selected method, rather than the actual memory allocated by the execution of the method. Table 7.3 shows the computed peak

expressions, and the comparison between real executions and estimations obtained by evaluating the polynomials. The last column shows the relative error ((#Objs - Estimation)/Estimation).

Table 7.3: Experimental results

Example	`memRq`	Param.	#Objs	Estimation	Err%
`MST`(nv)	$1+\frac{9}{4}nv^2+3nv+5+\max\{nv-1,2\}$	10	253	270	6%
		20	943	985	4%
		100	22703	22905	1%
		1000	2252003	2254005	0%
`Em3d`(nN,nD)	$6nN.nD+2nN+14+\max\{6,2nN\}$	(10,5)	344	354	3%
		(20,6)	804	814	1%
		(100,7)	4604	4614	0%
		(1000,8)	52004	52014	0%
`BiSort(n)`	$6+n$	10	13	16	19%
		20	21	26	19%
		200	69	135	45%
		64	69	70	1%
		128	133	134	1%
`Power()`	32656	-	32420	32656	1%

These experiments show that the technique produced quite accurate results, actually yielding almost exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of allocations associated with exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the bisort example, the reason of the over-approximation is that the actual number of instances is always bounded by 2^i-1, with $i=\lfloor log_2 n \rfloor$. Indeed, the estimation was exact for arguments power of 2.

7.6. Discussion

7.6.1. Sources of imprecision

The way we define `memRq` introduces an additional source of over-approximation because it sums the maximum of each m-region along a call chain. However, it can be the case that two regions cannot both reach the maximum size at the same time.

Consider the example in Fig. 7.10 assuming that $N \leq 10$. As we have shown previously, to compute `memRq` we need to compute the `rSize` estimator for every call chain. In this case $m1$, $m1 \xrightarrow{2} m2$ and $m1 \xrightarrow{2} m2 \xrightarrow{5} m3$. The call graph for this sample is just a list, then we just need to sum the obtained expressions for the three possible chains.

`rSize`$(m1)=0$ because $m1$ does not capture any object. The size of an $m2$-region depends on the expression $11-k$. That means that the maximum size is reached when $k=1$. Thus, the maximum of the $m2$-regions constrained by the call chain $m1 \xrightarrow{2} m2$ is `rSize`$(m1.2.m2)=10$ and it is obtained when assigning $h=1$. On the other hand, the size of an $m3$-region is proportional to the value of c. However, the variable c reaches its maximum value when j does. That is exactly when $j=k$ being the maximum value of k reached when $h=n$. Thus, `rSize`$(m1.2.m2.5.m3)=2N$ and is obtained assigning $h=N \wedge k=h \wedge j=k \wedge c=j$. However, as we have seen, the assignment $h=N$ does not maximize the size of $m2$-regions. Thus, both situations cannot happen at the same time and summing up the resulting `rSize` expressions leads to an over-approximation. This problem is shown graphically in Fig. 7.11 and Fig. 7.12.

Notice that there are also other factors that may impact in the precision of the bounds. For instance, in the `rSize` function the region's sizes are given by the

```
void m1(int N) {
1:  for(int h = 1; h <= N; h++) {
2:    m2(h);
  }
}
void m2(int k) {
3:  B[] b = new B[11 - k];
4:  for(j = 1; j <= k; j++) {
5:    m3(j);
  }
}
void m3(int c) {
6:  for(int i = 1; i <= 2*c; i++) {
7:    A a = new A();
  }
}
```

Figure 7.10: An example that shows the over-approximation caused by `memRq`.

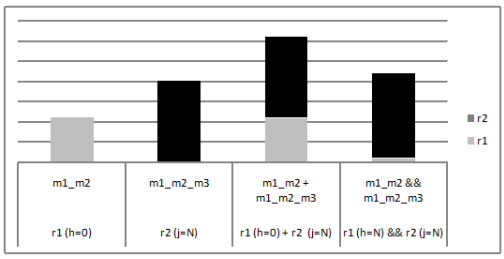


Figure 7.11: Region stack: regions in the same stack that cannot reach their maximum at the same time.

`memCaptured` estimator which can be computed using our technique presented in [BGY06]. That technique may obtain an over-approximation of actual region sizes whose accuracy depends on the precision of escape analysis or manually inferred regions.

Another source of approximation may come from the binding invariant. If it is too weak it may allow calling contexts that are not actually feasible leading inclusive to the impossibility of finding a maximum. Consider, for instance, the call chain $m0 \xrightarrow{1} m1 \xrightarrow{5} m3$ in the example presented in Fig. 7.1. If the invariant did not include the constrain $0 \leq i \leq k$ it would allow the values of i to be above the values of k, and therefore, leaving the variable n unbounded. Since n determines the size of the $m3$-region, that maximum would not be determined.

7.6.2. About the parameterization of `memRq`

In the definition of $Peak_m^{\mathtt{M}}$, we assumed that its signature is the same than the MUA signature. Actually, consumption may be more directly related with other expressions derivable from the parameters.

For instance, suppose that we want to know the amount of memory required to run a method `clone(`c`: Collection)` that returns a fresh copy of a collection c. We know that the size of the collection is relevant for computing the memory requirements. Thus, we can use a new variable *size* for the peak calculation and

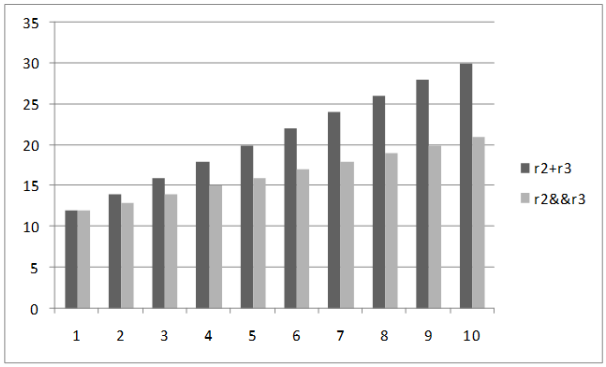


Figure 7.12: Actual region stack size vs. approximated sizes for different values of n.

relate it with the actual parameter of `clone` using a predicate $size = c.size()$.

To allow this kind of situation we propose an alternative definition of $Peak$ that allows the definition of new variables and a uses a predicate relating these variables with the method's formal parameters. We introduce a predicate $\phi \in \mathbb{P}(T_1 \times \ldots T_k, \Sigma)$ that relates the new variables with a program state and define ϕ-dependent version of $Peak$ as follows:

$$\begin{aligned} Peak^{\mathtt{M}}_{m,\phi}(a_1, \ldots, a_k) \quad = \quad & \max\{peakForRun^{r}_{m}(r, i) \mid r \in R([\![Prog]\!]^{\mathtt{M}}) \\ & \wedge\, \mathsf{stm}(r_i(pc)) = \mathtt{call}\ m \\ & \wedge \phi(a_1, \ldots, a_k, r_i)\} \end{aligned}$$

In practice, this definition is supported by relating these new variables with the formal parameters using the binding invariant.

7.6.3. Dealing with recursion and complex data structures

We do not allow recursion because our technique relies on having a finite evaluation tree. Although we believe that this restriction is acceptable for embedded systems, we are trying to overcome it. For instance, it is possible to provide and use peak memory-requirements specification for a set of mutually recursive methods considering them as being only one method.

Regarding the support of more complex data structures in [BGY06] we present some solutions to deal with some typical iteration patterns in collections. We are also studying the possibility of combining our technique with approaches like [CKQ+05, CNQR05] that seem to be suitable for the verification of Presburger expressions accounting for memory consumption annotations for class methods.

We believe it is possible to devise a technique integrating our analysis together with those mentioned type-checking based ones. The approach would be as follows. While methods for data container classes (like the ones provided by standard libraries) are annotated and verified by type-checking techniques, loop-intensive applications built on-top of those verified libraries may be analyzed using our approach. Benefits are twofold: first, work done by our technique would be reduced since we would have to deal with significantly smaller call graphs, and second, our ability to synthesize non-linear consumption expressions would entail an increase of expressive power and usability of type-checking based techniques.

7.7. Related Work

The problem of dynamic memory estimation has been studied for functional languages in [HJ03, HP99, USL03]. The work in [HJ03] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching). The computed expressions are linear constraints on the sizes of various parts of data. Our technique is suited for region-based memory manager and is able to compute non-linear parametric expressions. In[HP99] a variant of ML is proposed together with a type system based on the notion of sized types [HPS96], such that well typed programs are proven to execute within the given memory bounds.

The technique proposed in [USL03] consists in, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does. Our technique generates an evaluation tree which evaluation cost is known at analysis time.

For imperative object-oriented languages, solutions have been proposed in [CKQ+05, CNQR05, Ghe02]. The technique of [Ghe02] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper-bound. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, three points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables and third, it does not consider any memory collection mechanism. The method proposed in [CKQ+05, CNQR05] relies on a type system and type annotations, similar to [HP99]. It does not actually synthesize memory bounds, but statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear. Their type system allows aliasing and object deallocation (dispose) annotations. Our technique does not allow such annotations and indeed our memory model is more restricted. But as a counterpart we we can infer non-linear bounds. The reason we do not support individual object deallocation is our current impossibility of computing lower bounds which are required for safely compare the difference between allocations and deallocations.

To our knowledge, the technique used to infer non-linear dynamic memory requirements under a region-based memory manager and its effective computation using Bernstein basis is a novel approach to memory requirements calculus.

7.8. Conclusions and Future work

We presented a novel technique to compute non-linear parametric upper-bounds of the amount of dynamic memory required by a method. The technique is more suited for region-based dynamic memory management, when regions are directly associated with methods, but it can be used safely to predict memory requirements for memory management mechanism that free memory by demand.

The inputs of the technique are the application call graph enriched with binding invariant information to constraint calling contexts, a set of parametric expressions

that bound the size of every region and a mapping from creation sites to regions (we can compute this information using the technique proposed in [BGY06]) and yields a parametric certificate of the memory required to run a method (or program).

These certificates are given in the form of evaluation trees that can be easily translated to code that can be evaluated in runtime. The size of the evaluation trees is known at compile time and can be reduced either using mathematical tools to symbolically solve maximums between polynomials or by compromising some accuracy of run-time calculations.

The precision of the technique relies on several factors: the precision of the inputs (regions size and invariants), the structure of the program that may allow or do not allow two active regions get its maximum size at the same time, the precision of the Bernstein approximation and eventual trade-offs made to reduce the evaluation tree. We still need to perform more benchmarks to really asses how well this technique works in practice.

As we mention in the discussion, we will try to enhance our technique to support recursion and we plan to explore combining our approach with other which are better suited for more complex or recursive data structures.

CHAPTER 8

Conclusions

8.1. Concluding remarks

We have developed a series of techniques aiming at the automatic synthesis of parametric certificates of dynamic memory consumption for Java like programs in embedded and real-time environments.

First, we have developed a method to synthesize non-linear parametric estimations of dynamic memory utilization. The analysis is general in the sense that it may be used in for different applications since it is based on counting the number of times a selected set of statements is executed. Thus, it can be applied to obtain bounds on the usage of other resources by selecting, for instance, statements involved in communication, message passing, database access, etc.

Then, we have presented our approach for automatically inferring scoped-memory regions that are used to replace conventional garbage collectors. We have also implemented a tool that allows manipulation of the inferred regions together with an API for supporting our region-based memory management. We have presented a technique to produce region-based code out of conventional Java code. This transformation ensures scoping rules by construction, thus, eliminating the need for run-time checks. Under this setting, we have shown how we can predict the size of memory regions to reserve enough space to allocate objects into them.

Finally, we have presented a technique to compute parametric upper bounds of the amount of dynamic memory required by a method. The technique is better suited for a region-based memory manager such us the one we have implemented, but it can be safely used to predict memory requirements for any other memory management mechanism that collects unused memory on demand.

We have developed a prototype tool that covers the complete chain of techniques and allows us to evaluate experimentally the efficiency and accuracy of the method on several Java benchmarks. The results are very encouraging. We are aware that the precision of our technique depends of several factor such as the ability of finding strong linear invariants, discover small sets of inductive variables, precise escape analysis information, etc. Therefore, we are working in providing new facilities for obtaining these data from other sources.

8.2. Future Work

There are several aspects of our techniques that we would like to improve. Some of them were discussed in the respective chapters. However, here we want to point out the most important issues we would like to deal with in the near future. Specifically, we aim at improving the precision of the techniques, and at making it applicable to a wider spectrum of applications (usability and scalability).

8.2.1. Improving Precision

Some sources of imprecision depend on external factors (e.g. invariants) but some others are intrinsic to our techniques.

In 2.6.3 we have shown that the precision of the technique to compute memory allocations can be improved for the case of some conditional branches (e.g. `if` statements and virtual calls). For instance, notice that `then` and `else` branches of an `if` statement cannot happen in a same loop iteration. Thus, the idea is instead on relying only on strong invariants consider also the control structure of the program. This will avoid counting visits to set of statements that are impossible to happen at the same time.

Another source of imprecision is introduced when modeling `memRq`. In chapter 7 we have shown how the model may introduce over approximations since it allows some potential region stack configurations that may not be possible in runtime. We are thinking about how to refine this notion by introducing new constrains in the computation of `rSize`.

Our analysis leverages on the use of a region-based memory manager in order to model objects deallocation. That decision comes with the price of having a less flexible memory model which may consume more memory. Other techniques like [CNQR05, HJ03] allow individual object deallocation and, therefore, more precise bounds, but are limited only to linear upper bounds. To extend our techniques to add support for individual object deallocation, we must be able to infer precise lower bounds of the number of times a given set of statement is executed. Therefore, this is another challenge we would like to address in the near future.

There are also improvements that can be made when inferring memory regions. In particular, most escape analysis techniques (including ours) abstract away sets of objects using only one representative (e.g, allocation site, creation site, etc.). However, sometimes this approach can be overly conservative. Consider the following example:

```
A m1()
{
1: for(int i=0; i<100000; i++) {
2:    Object a = new A();
3:    if(i==100)
4:      return a;
   }
5: return new A();
}
```

In this case, all objects created at $m1.2$ but only one is actually captured by $m1$. However, since the escape analysis techniques uses $m.2$ as a representative of all objects created at that program location, the analysis has to consider that all object escapes. One possible approach to solve this problem is trying to split the set

of objects leveraging on program invariants and then use the counting mechanism to determine how many objects actually escape.

We also plan also to keep working in the generation on program invariants. In particular, we would like to specialize our static invariant generation tool `JInvariant` [PG06] to try focus only in discover relationships between inductive set of variables. We also like to try other approaches mixing static and dynamic approaches in order to try to get stronger invariants.

8.2.2. Usability and Scalability

Although it is fed with local information, the use of creation sites and call chains make the analysis non modular. The advantages of modularity are manifold: reuse of specifications, better scalability, analysis of applications that calls non-analyzable methods, integration with other techniques, etc. However, a straightforward approach to modularity in this setting would mean that we will have to compute the polynomial consumption specifications of a method m out of the polynomial specifications of its callees. This is challenging because these calls may be performed into complicated iteration patterns that would required more complicated mathematical machinery.

Nevertheless, we would like to support some degree modularity. We can easily incorporate linear specifications since they can be encoded in the invariants as we have done in for array creation in chapter 2. In that direction we plan to integrate approaches like [CKQ+05, CNQR05] suitable for the verification of Presburger expressions accounting for memory consumption annotations for class methods.

Another approaches allow verification (not inference) of non-linear consumption expression [Š07, AM05]. In those cases, we can try to model those expressions using some tricks. For instance, we conjecture that polynomial specifications can be modeled as the number of solutions of a parametric polyhedron and thus, reuse our technique. To deal with non-polynomial dynamic memory consumption we suggest the use of a fresh parameter that represents the non-polynomial expression. They would be treated as non-interpreted symbols (program parameters).

We also want to extend the approach to support recursive method calls. Our first approach is to treat those cases as non-analyzable calls and manually provide specifications. Nevertheless, we plan to evaluate and try to incorporate some ideas like the one presented in [AAG+07] which infers recurrence equations modeling the cost of executing recursive functions.

Bibliography

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007.

[ABH+04] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2004.

[aG] MIT. Program analysis and Compilation Group. The flex compiler infraestructure. http://www.flex-compiler.csail.mit.edu/.

[AKC02] 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.

[AM05] David Aspinall and Kenneth MacKenzie. Mobile resource guarantees and policies. In Gilles Barthe, Benjamin Grégoire, Marieke Huisman, and Jean-Louis Lanet, editors, *CASSIS*, volume 3956 of *Lecture Notes in Computer Science*, pages 16–36. Springer, 2005.

[Bak92] Henry G. Baker. The Treadmill: Real-Time Garbage Collection without Motion Sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

[BB00] Jakob Berchtold and Adrian Bowyer. Robust arithmetic for multivariate bernstein-form polynomials. *Computer-aided Design*, 32:681–689, 2000.

[BCG04] D. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. *In EMSOFT'04*, 2004.

[BCGV05] David F. Bacon, Perry Cheng, David Grove, and Martin T. Vechev. Syncopation: generational real-time garbage collection in the metronome. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[BCR03] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92, New York, NY, USA, 2003. ACM Press.

[BDF+04] 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.

[BDJ+06] 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 *Lectures Notes in Computer Science*, pages 364–387. Springer, September 2006.

[Ber52] S. Bernstein. *Collected Works*, volume 1. USSR Academy of Sciences, 1952.

[Ber54] S. Bernstein. *Collected Works*, volume 2. USSR Academy of Sciences, 1954.

[BFGL07a] Mike Barnett, Manuel Fändrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) precise points-to analysis. In *IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Berlin, Germany, jul 2007.

[BFGL07b] 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.

[BFGY07] Victor Braberman, Federico Fernadez, Diego Garbervetsky, and Sergio Yovine. Dynamic memory requirement inference using berstein basis. Research Report 07-01, Departamento de Computación. FCEyN. Universidad de Buenos Aires, Argentina, 2007.

[BFK02] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symb. Comput.*, 33(1):1–12, 2002.

[BGY04] V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. *Internal Report TR-2004-03. Verimag, France*, 2004.

[BGY05] Víctor Braberman, Diego Garbervetsky, and Sergio Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTfJP'2005: 7th Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, July 26, 2005.

[BGY06] Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[BHMS04] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *LPAR*, pages 347–362, 2004.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA 99*, volume 34, pages 20–34, 1999.

[Bla03] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[BLS05] 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 *Lectures Notes in Computer Science*, pages 49–69. Springer, 2005.

[BN04] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, Lectures Notes in Computer Science, pages 54–84. Springer, July 2004.

[BPS05] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[BR00] P. Boulet and X. Redon. Sppoc: fonctionnemen et applications. Research Report 00-04, LIFL, 2000.

[BR01] W. S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *LNCS*, 2211:289–??, 2001.

[Bro84] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Symposium on LISP and functional programming*, pages 256–262. ACM Press, 1984.

[Bro85] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Conference on Functional programming languages and computer architecture, LNCS 201*, pages 273–288, 1985.

[CBC93] 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.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL 77*, pages 238–252, 1977.

[CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC 02*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304.

[CD02] Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Notices*, 37(11):292–310, November 2002.

[CEI+07] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. *ACM Trans. Program. Lang. Syst.*, 29(5):28, 2007.

[CFGV06] Philippe Clauss, Federico Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. Technical Report 06-04, Université Louis Pasteur, oct 2006.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, October 1991.

[CGS+99] J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78*, pages 84–97, Tucson, Arizona, 1978.

[CJPS05] David Cachera, Thomas P. Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2005.

[CKQ+05] W. Chin, S. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE 2005*, 2005.

[CL98] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):Kluwer Academic, 1998.

[CL05] B. Chang and K. Rustan M. Leino. Infering object invariants. In *AIOOL'05*. ENTCS, 2005.

[Cla96] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS'96*, pages 278–285, 1996.

[Cla97] P. Clauss. Handling memory cache policy with integer points counting. In *Euro-Par'97*, pages 285–293, 1997.

[CM01] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001.

[CNQR05] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.

[CR04] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004.

[CR07] 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.

[CT04] Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In Evelyn Duesterwald, editor, *13th International Conference on Compiler Construction, CC 2004*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004.

[DC02] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM 02*, pages 25–35, 2002.

[DHPW01] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *Java Grande*, pages 106–115, 2001.

[DM06] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[ECGN99] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE99*, pages 213–224, 1999.

[ECM06] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/-ecma-335.htm, Ecma International, 2006.

[Ehr77] E. Ehrhart. Polynômes arithmetiques et m´ethode des polyedres en combinatorie. *Series of Numerical Mathematics*, 35:25–49, 1977.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[Fah98] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *TJS*, 12(3), 1998.

[Far93] G. Farin. *Curves and Surfaces in Computer Aided Geometric Design.* Academic Press, San Diego, 1993.

[Fer06] Federico Fernández. Obtención de cotas del consumo de memoria requerido para ejecutar un método bajo el modelo de memoria por alcance a través de bases de bernstein. Master's thesis, Departamento de Computación. FCEyN. UBA, sep 2006.

[FGB+05] Andrés Ferrari, Diego Garbervetsky, Victor Braberman, Pablo Listingart, and Sergio Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 50–54, New York, NY, USA, 2005. ACM Press.

[FL01] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. *LNCS*, 2021, 2001.

[FLL+02] 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.

[FR87] R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, 1987.

[GA01] D. Gay and A. Aiken. Language support for regions. In *PLDI 01*, pages 70–80, 2001.

[Gar05] D. Garbervetsky. Using daikon to automatically estimate the number of excecuted instructions. *Internal Report. UBA, Argentina*, 2005.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. *Lecture Notes in Computer Science*, 1628:205–??, 1999.

[GB00] James Gosling and Greg Bollella. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[GBD98] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *CODES/CASHE '98*, pages 145–149. IEEE, 1998.

[Ghe02] O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002., 2002.

[GNYZ05] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in java. *Electr. Notes Theor. Comput. Sci.*, 113:105–121, 2005.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93, London, UK, 2000. Springer-Verlag.

[Hei71] Lee E. Heindel. Integer arithmetic algorithms for polynomial real zero determination. *J. ACM*, 18(4):533–548, 1971.

[Hen98] R. Henriksson. Scheduling garbage collection in embedded systems. *PhD. Thesis, Lund Institute of Technology*, 1998.

[HIB+02] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J-Ph. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[HJ03] M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, SIGPLAN, New Orleans, LA, January 2003.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*, pages 70–81. ACM, 1999.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996.

[Ins] Silicomp Research Institute. Turbo j. Java to native compiler. http://www.ri.silicomp.fr/adv-dvt/java/turbo/index.htm.

[IS97] A. Ireland and J. Stark. The automatic discovery of loop invariants. Fourth NASA Langley Formal Methods Workshop. Conference Publication 3356., 1997.

[JL96] R. Jones and R. Lins. *Garbage collection. Algorithms for automatic dynamic memory management.* John Wiley and Sons, 1996.

[KNY03] Ch. Kloukinas, Ch. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *EMSOFT'03*, Philadelphia, USA, October 2003.

[LBR99] 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.

[LG88] J. M. Lucassen and D. 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.

[Lis03] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET 03*, 2003.

[LLP+00] G.T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.

[LMC02] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *TJS*, 21(1):37–76, 2002.

[Loe99] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, March 1999.

[LPHZ02] 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.

[Mét88] Daniel Le Métayer. Ace: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[Mei04] Benoit Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization.* PhD thesis, December 2004.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Series in Computer Science. Prentice-Hall International, New York, 1988.

[MRR05] 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.

[NE01] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants:integrating Daikon and ESC/Java. In *RV 2001,ENTCS*, volume 55, 2001.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NX05] 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.

[Ped91] Paul Pedersen. Multivariate sturm theory. In *AAECC91*, pages 318–332, London, UK, 1991. Springer-Verlag.

[PFHV04] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.

[PG06] Diego Piemonte and Diego Garbervetsky. Descubrimiento automático de restricciones lineales entre variables de programas mediante análisis estático. Master's thesis, Departamento de Computación. FCEyN. UBA., mar 2006.

[Pol] The PolyLib polyhedral library. http://icps.u-strasbg.fr/PolyLib/.

[Pug94] W. Pugh. Counting solutions to presburger formulas: How and why. In *PLDI 94*, pages 121–134, 1994.

[Rab06] Tilmann Rabl. Volume calculation and estimation of parameterized integer polytopes. Master's thesis, Universität Passau, January 2006.

[RF02] T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. *In EMSOFT'02, LNCS 2491*, 2002.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM Press.

[RR84] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Ellis Horwood, 1984.

[RR01] 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.

[Sal] Alexandru Salcianu. Pointer analysis and its applications for java programs. SM Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2001.

[SHM+06] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

[Sie99] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. *rtcsa*, 00:96, 1999.

[Sie00] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. *CASES'00*, 2000.

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

[SR01] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP 01*, volume 36, pages 12–23, 2001.

[SR05] 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.

[SYG05] Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. Fast escape analysis for region-based memory management. *Electronic Notes Theoretical Comput. Sci.*, 131:99–110, 2005.

[TE05] 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.

[TT97] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[USL] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical report, Computer Science Department, Indiana University. To appear.

[USL03] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Optimized live heap bound analysis. In *VMCAI 03*, volume 2575 of *LNCS*, pages 70–85, January 2003.

[Ver07] Sven Verdoolaege. *barvinok*, a library for counting the number of integer points in parametrized and non-parametrized polytopes.
Available at http://freshmeat.net/projects/barvinok, April 2007.

[VRHS+99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - A java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[Š07] Jaroslav Ševčík. Proving resource consumption of low-level programs using automated theorem provers. *Electron. Notes Theor. Comput. Sci.*, 190(1):133–147, 2007.

[VSB+04] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04*, pages 248–258. ACM, 2004.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

[ZG98] M. Zettler and J. Garloff. Robustness analysis of polynomials with polynomial parameter dependency using bernstein expansion. *IEEE Transactions on Automatic Control*, 43(3):425–431, 1998.

[ZM99] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC '99*, pages 811–816. ACM Press, 1999.

APPENDIX A

Tool Support

Now, we will discuss some technical aspects of a tool that we have developed to evaluate our approach. As we mentioned in the introduction, the tool has three main components:

- Dynamic utilization analyzer
- Region inferencer
- Dynamic memory requirements analyzer.

A.1. Dynamic utilization analyzer

The *Dynamic utilization analyzer* is the part of the tool that has required more work. It is responsible for the computation of the set of creation sites, the generation and manipulation of invariants, estimation of inductive set of variables and interfacing with other tools that deal with polyhedra and polynomials manipulation.

An schematic diagram of the components that composed this tool showed in Fig. A.1.

The main components are:

- Application Instrumentator: Instruments the application's source code or bytecode to produce a new functionally equivalent code that provides explicit information we would like to make it appear in local invariants.
- `Daikon` [EPG+07]: Third party dynamic analysis tool that produces likely invariant.
- Invariant Globalizer: Generates control state invariants out from local ones.
- Symbolic polyhedral calculator: Simplifies invariants and generates the parametric expressions that counts the number of solutions of given invariants.
- Polynomial Evaluator:It is a tool that manipulates and evaluates polynomials.

Most of the components were implemented in Java using `soot` [VRHS+99] which is a framework designed to facilitate program analysis. We use the framework to generate call graphs, to implement several dataflow analyses and for code generation. The *Symbolic Polyhedral calculator* integrates different tools such as `SPPoC`

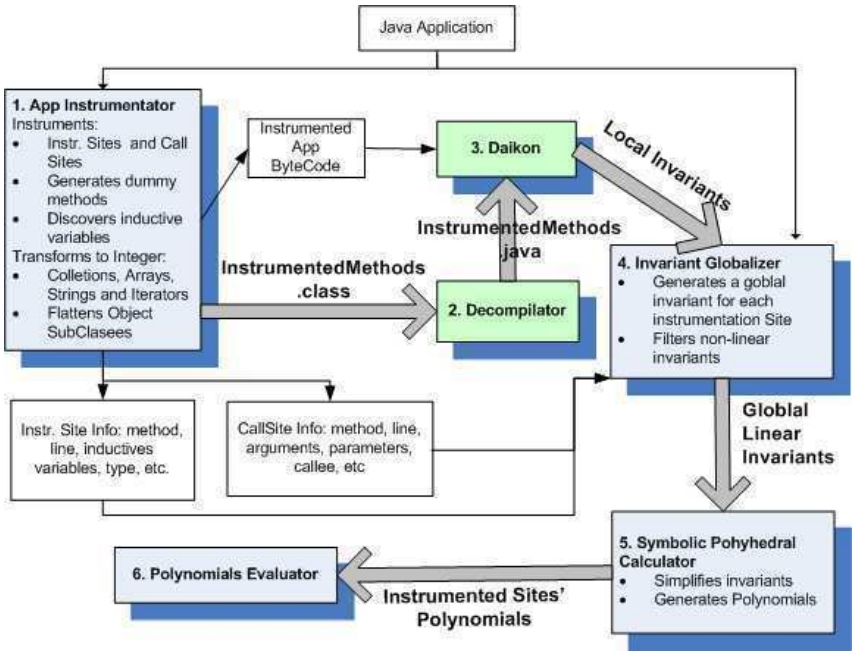


Figure A.1: Components of the Dynamic Utilization Analyzer

[BR00] and `Barvinok` [VSB+04] and the `Polylib` library [Pol] that are useful for the manipulation of polyhedra and generation of Ehrart quasi-polynomials [Ehr77].

A.1.1. Application Instrumentator

The goal of this component is to automatically produce code that is enriched with information that can help `Daikon` in the generation of local invariants that we need.

The component performs the following tasks:

- Identification of allocation sites and call sites
- Identification of "sizeable" variables and parameters
- Identification of inductive set of variables
- Instrumentation of the code

An important role of the tool is the identification of the variables and expressions that have to appear in the invariant. Basically, it creates a new variable for any "sizeable" expression. By sizeable we mean variables or expressions of integer type that can be relevant in a linear invariant. Examples of expressions that we want to capture are: length of arrays and strings, size of collections, instance and static fields, etc. For each one of those expressions we create a new variable and introduce code that binds the variable with that expression.

We also include new variables in order to try to "linearize" common iterations patterns. For instance, for each variable of type `iterator` we create a new associated counter. We introduce code to update the counter when the iterator is updated. For instance, `it.next` increments the counter `it`.

Since Java programs use a passing by value convention, method parameters are local variables which have a copy of the arguments passed by the caller. As a consequence, parameters can be updated as any other variable. Since we need to discover

"parametric" we introduce new "meta"-variables and additional code to make these variables conserve the initial value of the parameters (see Table B.1). Since these new variables are fresh and are only updated at the beginning of the method, they can be interpreted, at any moment, as the original value of the parameter before the execution of the method (precondition).

`Daikon` is able to generate a pre and postconditions of every method that analyzes. Since we need invariants for other program points such as allocation sites (i.e. codenew statements) and call sites since both are necessary to generate creation-site invariants, at those control points, we introduce calls to dummy (and empty) methods whose arguments are the variables we need `Daikon` to consider when trying to infer the invariants.

Summarizing we perform the following tasks:

1. Create a new variable for each sizeable variable or expression.
2. At method's entry: Add code to take a snapshot of the initial value of the parameters
3. Before every point of interest:
   a) Add code to store the current value of variables and expressions of interest.
   b) Add code to call a dummy method passing as arguments all the variables that `Daikon` should analyze.

In Table A.1 we show the pseudo-code of the algorithm we have implemented to instrument each method. The function `codeForInitParams` yields the code necessary to record the initial value of method's parameters. `gen` is a function that, given a program location, yields (if it is a location that requires instrumentation) a fresh dummy method whose formal parameters are associated with the set of relevant (inductive) variables and expressions for that method location. It may also yields a potentially augmented set of variables (e.g. a new artificial variable introduced for an iterator) which needs to be considered when computing invariants. Finally, the function `instrument` returns the code necessary to record the values of the relevant variables and expressions (the ones that we want to appear in an invariant for that program location) and to call the generated dummy method.

In appendix B we present a full example where most of the interesting aspects of the instrumentation technique are applied. Applying this instrumentation procedure, we ensure that we the variables and expressions of interest will appear in the runtime traces that `Daikon` is going to analyze.

Computing set of inductive variables

Since we relate invariants with number of visits to control states, we need invariants to bound in some way all variables for a given control state. However, some of the variables may not have any real impact in the number of visits of the analyzed control state. If we apply the counting technique without considering that fact, we may count valuations of variables that are not connected with number of visits which will lead to a very pessimistic over-approximation (see 2.6.3). To cope with this problem, we must identify the set of inductive variables of the location under analysis.

Our tool automatically (and conservatively) discovers inductive variables for all instrumentation and call sites. Up to the moment we implemented a dataflow analysis that combines a live variables analysis (but augmented with field sensitivity) with a classic loop inductive analysis [NNH99].

```
instrumentMethod(m)
// instruments method m
// returns the set of created dummy method
ε = ∅;
IMs = ∅;
initCode:=codeInitParams(m);
insert(m,initCode);
for each l ∈ Body_m do
    (im_l, ε')= gen(l, ε);
    code = instrument(l, im_l, ε');
    insertBefore(m,l,code);
    IMs = IMs ∪ im
    ε = ε';
end for;
return IMs;
```

Table A.1: Pseudo-code showing how we instrument the code

Since our analysis is conservative the tool allows manual edition of the inferred set of inductive variables for each program location. This allows programmer to "fine-tune" the sets in order to produce tighter bounds at the risk of losing soundness.

Notice that the minimal set of variables that must be considered when instrumenting the code to guide `Daikon` should include at least the set of inductive variables and method's parameters.

Local Invariant Generation

As we mentioned, we use `Daikon` to infer local invariants. The output is basically a set of specifications containing pre and postconditions of the analyzed dummy methods. In our case, we only ask `Daikon` to analyze the class that contains the artificially created dummy methods which are conveniently codified to refer to the original program location in such a way that the precondition of the dummy method is the obtained invariant for that program location.

A.1.2. Invariant Globalizer

Once we obtain invariants using the procedure mentioned previously or by another means (e.g. manually or by static analysis [PG06]) we need to generate what we informally call "control state invariants" which refers to invariants predicating about variables which belongs to several methods along a call stack. Instances of control state invariants are creation sites invariants which are necessary for the computation of `memalloc` (see 1.4) and binding invariant required for the computation of `rSize` (see 1.6).

To compute the binding invariants, we generate all possible call chains by traversing the application call graph starting by the MUA. Then, for every call chain we iteratively compose the caller's local invariant with each callee's by conjoining them and adding a set of equalities that reflect the actual binding of caller's arguments with callee's parameters. Since a creation site is simply a call chain finalizing in a program location that performs a `new` statement, creation site invariants are basically an extension of a binding invariant conjoined with the local invariant associated with that program location.

The output of this tool is a set of global invariants that is passed to the *Symbolic Polyhedra Calculator* to produce the polynomials.

A.1.3. Symbolic Polyhedra Calculator

This component takes a set of global invariants and produce a set of polynomials representing the number of integer solution for those invariants.

We rely on a tool called *Symbolic Parameterized Polyhedral Calculator* [BR00] (`SPPoC`) which is library written in OCaml and allows symbolic manipulation of parametric polyhedra. To use this tool we first produce an OCaml program which incorporates the global invariant and perform calls to `SPPoC` in order to convert the global invariants in parametric polyhedra. In the conversion phase we simplify, linearize and project only relevant (inductive) variables and parameters of the invariants. `SPPoC` provides and API to interface with `Polylib` that provides an algorithm for counting the number of solution of parametric polyhedra. But, instead of using that algorithm we use another one implemented in a tool called `Barvinok` [VSB+04] because it is more effective in practice.

A.2. Region inference

This goal is this component is the generation of memory regions and automatic generation of code that use our region-based memory mechanism.

In Fig. A.2 we show an schematic diagram of the components of this part of the tool.

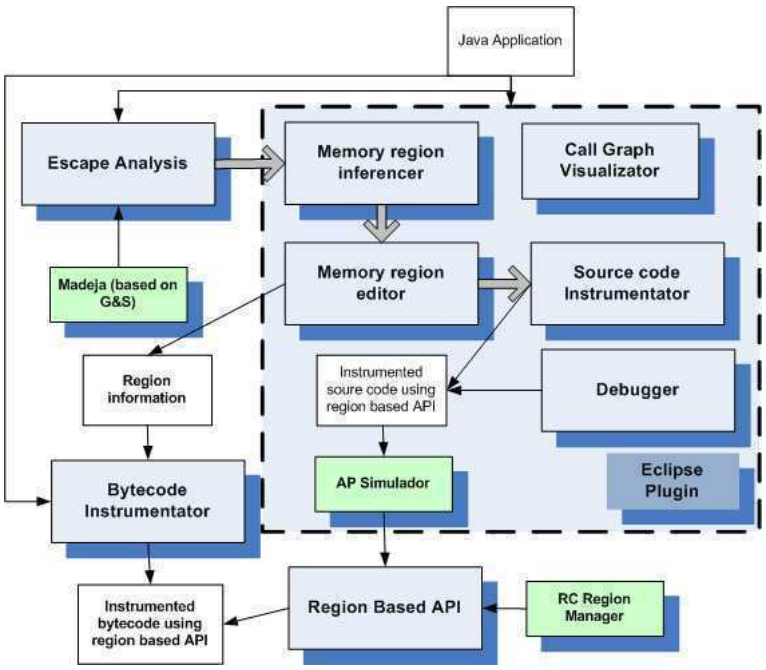


Figure A.2: Components of the memory region inferencer and region-based code generator

As mentioned, we have implemented two algorithms for escape analysis (see 4 and 5). The former was implemented in Java using the soot framework and is integrated in the tool. The latter was implemented in C# and it is currently integrated in the Spec# compiler [BLS05].

A component to edit memory regions was implemented as a Java Eclipse plug-in called `JScoper` [FGB+05] (see chapter 6). We used this component as a front-end to produce region-based Java code starting from a conventional Java application. The component automatically calls our escape analysis component, process it, and produce a file with the inferred memory regions. Then, using the original source code and the regions produces the region-based code by instrumenting the source application as explained in chapter 3.

In chapter 3 we presented an API to support a region-based memory management in Java and a tool to automatically generate code that uses the API. We implemented two versions of this API. The first one is simply a simulator we use for checking the feasibility of our approach, for accounting and debugging purposes. In fact, we use this simulator to contrast the memory consumption prediction against actual executions. The simulator is also useful when debugging applications because it is completely implemented in Java without using any native method. Thus, we can easily access to the internal representation of the API and display and manipulate the regions.

We also implemented a "real" memory manager based on the region-based memory manager `RC` developed by David Gay [GA01]. In this case, instead of generating Java bytecode, we generate native code which includes the instrumented code, the API and the `RC` memory manager.

A.3. Memory requirements calculation

The goal of this component is to implement the memory requirements technique presented in chapter 7.

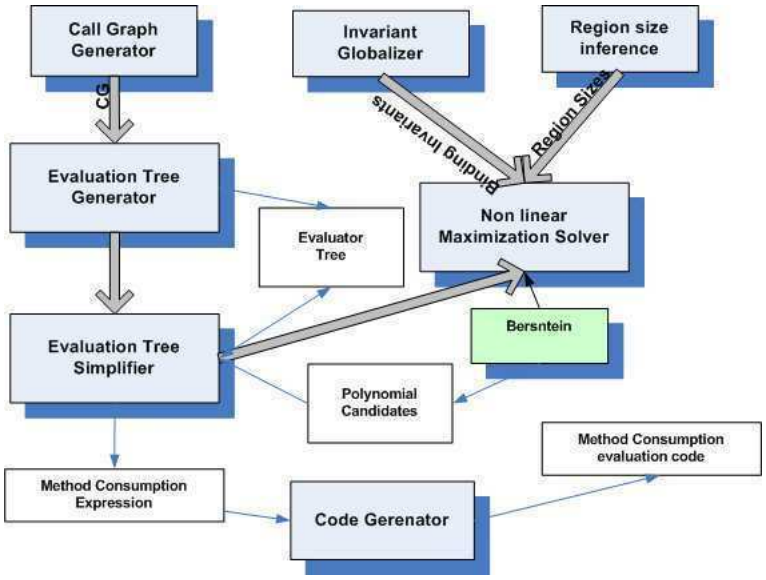


Figure A.3: The main subcomponents of the components for predicting memory requirements

The most important subcomponent is the one that implements an algorithm to solve the non-linear symbolic maximization problem. As we mentioned in chapter 7 we decided to follow the approach presented by Clauss in [CT04]. Since, at that moment there were no implementations of the Bernstein transformation over multiple variable's polyhedra, we started the development of the first implementation. This work was mainly due to Federico Fernández as part of his work for obtaining with

M.Sc. thesis [Fer06] that I co-advised. This tool was capable of obtaining the set of bound candidates as explained in chapter 7. It was implemented in C++ relaying on libraries like `GiNaC` [BFK02] for manipulating polynomials and `Polylib` [Pol] for dealing with polyhedra.

Lately, our implementation of Bernstein transformation has been incorporated in the `Barvinok` library [Ver07] which features better support for determining the maximums between the candidates and important performance and interfacing improvements.

The other subcomponents are implemented in Java using soot and reuse part of the functionality implemented for A.1. For instance, call chains and the binding invariants are generated using the components described for the invariant globalization (see A.1.2).

APPENDIX B

Instrumentation for Daikon: An example

Here we present an example that shows several aspects about the instrumentation technique. We instrument programs in order to generate code that is useful to guide Daikon in obtaining local linear invariants that are useful for counting the number of visits of selected set of statements.

B.1. Example

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  public ArrayDim() {
1:  list= new Vector();
2:  len = 0;    }
  public void add(Object o) {
1:  Object[] block;
2:  if (len % BSIZE == 0)
3:    block = newBlock(BSIZE);
    else
4:    block=(Object [])
            list.lastElement();
5:  block[len % BSIZE] = o;
6:  len++;
}
  Object[] newBlock(int how) {
7: Object[] block=new Object[how];
8: list.add(block);
9: return block;
  }
  void addAll(Collection c) {
10: for(Iterator it=c.iterator();
            it.hasNext();)  {
11:   add(it.next());
      }
  }
}
```

Figure B.1: Motivating example

In Fig. B.1 we present an example program for which we want to obtain creation sites invariants. It is a (very simple) implementation of a dynamic array using a list of fixed sized nodes. We are interested in the allocation statement located at *newBlock*.7. The number of times this statement is executed when execution start by method *addAll* depends on the size of the collection c passed as a parameter. The execution of this statement takes place in the method where a new block of memory is request because the previous block is full.

The Call Graph and Call Tree starting from method *addAll* are depicted in Fig. B.2. □

We instrumented the code using the algorithm presented in Table A.1. Table B.1 shows part of the instrumented code for the example. The code that has been added to the original example can be distinguish since it is in italic font.

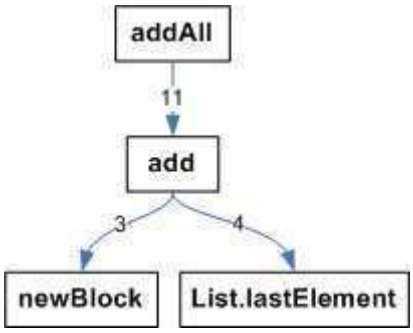


Figure B.2: Call Graph for method *ArrayDim.addAll* of the proposed example

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  public void add(Object o) {
    Object[] block;
    ArrayDim this_init=this;
    int this_init_list_size, this_init_len;
    int this_list_size, this_len;
    int ArrayDim_BSIZE;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
    this_init_len = 0; }
    if (len % BSIZE == 0) {
      ArrayDim_BSIZE = BSIZE;
      if(this!=null) {
        if(this_list!=null)
          this_list_size = this_list.size();
        else this_list_size = 0;
        this_len = this.len; }
      else { this_list_size = 0;
        this_len = 0; }
      IM.ArrayDim_3(this_list_size,this_list_len,
        this_init_list_size, this_init_len,
        ArrayDim_BSIZE);
      block = newBlock(BSIZE);
    }
    else {
      block=(Object [])list.lastElement(); }
      block[len % BSIZE] = o;
      len++; }
  }
  Object[] newBlock(int how) {
    int how_init = how;
    ArrayDim this_init=this;
    int this_init_list_size,this_init_len;
    int this_list_size,this_len;
    int ArrayDim_BSIZE;
    this_init = this;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
      this_init_len = 0; }
    ArrayDim_BSIZE = BSIZE;
    if(this!=null) {
      if(this_list!=null)
        this_list_size = this_list.size();
      else this_list_size = 0;
      this_len = this.len; }
    else { this_list_size = 0;
      this_len = 0; }
    IM.ArrayDim_7(how, how_init,
      this_list_size,this_list_len,
      this_init_list_size, this_init_len,
      ArrayDim_BSIZE);
      Object[] block=new Object[how];
      list.add(block);
      return block;
  }
  void addAll(Collection c) {
    Collection c_init = c;
    int c_size, c_init_size;
    if(c_init!=null) c_init_size = c_init.size();
    else c_init_size = 0;
    ArrayDim this_init;
    int this_init_list_size, this_init_len;
    int this_list_size, this_len;
    int ArrayDim_BSIZE;
    this_init = this;
    int it_count;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
      this_init_len = 0; }
    it_count = 0;
    for(Iterator it=c.iterator();
        it.hasNext();) {
      if(c!=null) c_size = c.size();
      else c_size = 0;
      it_count++;
      ArrayDim_BSIZE = BSIZE;
      if(this!=null) {
        if(this_list!=null)
          this_list_size = this_list.size();
        else this_list_size = 0;
        this_len = this.len; }
      else { this_list_size = 0;
        this_len = 0; }
      IM.ArrayDim_11(it_count, c_size, c_init_size,
        this_list_size,this_list_len,
        this_init_list_size, this_init_len,
        ArrayDim_BSIZE);
      add(it.next());
    }
  }
}
```

Table B.1: Instrumented code for the example

At the beginning of every method we automatically generate code used to keep the initial value of the method parameters (recorded in special variables named with the `_init` suffix). Since `this` is a complex parameter (an instance of `ArrayDim` class) we generate several variables to record the value of each of its component. We also apply an special treatment to the field `list` which is a "sizeable" object because it is an instance of the `Collection` type. In particular, we generate a fresh variable to represent the expression `list.size()`.

At every instrumentation site (i.e. call sites, allocation sites), we introduce code to store, in local variables, the value of relevant variables and expressions at that program location and to make a call to a generated dummy method using that local variables. We apply an special treatment for some iteration patterns. Notice, for instance, that in method `AddAll` we introduce a new variable `it_count` which is associated with the iterator `it`. The idea is that, every time `it.next` is executed, `it_count` is incremented.

In order to generate local invariants we run `Daikon` over a test harness to try to ensure a good coverage. Table B.2 shows some obtained invariants for our example. They correspond to the instrumentation site `newblock.7` and the call sites `addAll.7` and `add.3` that belong to its call chain.

label	invariant
addAll.11	$BSIZE = 5, size_{f\_this\_init\_list} = 0, f\_this\_init\_len = 0, size_{c\_init} = size_c, size_{f\_this\_list} >= 0, size_{f\_this\_list} < size_c, f\_this\_len >= 0, f\_this\_len < size_c, size_{f\_this\_list} <= f\_this\_len, count\_it = f\_this\_len + 1, count\_it >= 1, count\_it <= size_c$
add.3	$BSIZE = 5, size_{f\_this\_list} = size_{f\_this\_init\_list}, f\_this\_len = f\_this\_init\_len, f\_this\_len\%5 = 0, size_{f\_this\_list} <= f\_this\_len, size_{f\_this\_list} < 5, f\_this\_len = (size_{f\_this\_list} * 5)$
newBlock.7	$BSIZE = 5, size_{f\_this\_list} = size_{f\_this\_init\_list}, f\_this\_len = f\_this\_init\_len, how = how\_init, f\_this\_len\%5 = 0, how = 5, size_{f\_this\_list} <= f\_this\_len, size_{f\_this\_list} < how, f\_this\_len = (size_{f\_this\_list} * how)$

Table B.2: Local invariants found by `Daikon` for two call sites and one instrumentation site

Invariant:

$\mathbf{l11@size_{f\_this\_init\_list} = size_{f\_this\_list}, l11@f\_this\_init\_len = f\_this\_len,}$
$\mathbf{l11@size_{c\_init} = size_c}$
$BSIZE = 5, l11@size_{f\_this\_init\_list} = 0, l11@f\_this\_init\_len = 0, l11@size_{c\_init} = l11@size_c, l11@size_{f\_this\_list} >= 0, l11@size_{f\_this\_list} < l11@size_c, l11@f\_this\_len >= 0, l11@f\_this\_len < size_c, l11@size_{f\_this\_list} <= l11@f\_this\_len, l11@count\_it = l11@f\_this\_len + 1, l11@count\_it >= 1, l11@count\_it <= l11@size_c$
$\mathbf{l3@size_{f\_this\_init\_list} = l11@size_{f\_this\_list}, l3@f\_this\_init\_len = l11@f\_this\_len,}$
$BSIZE = 5, l3@size_{f\_this\_list} = l3@size_{f\_this\_init\_list}, l3@f\_this\_len = l3@f\_this\_init\_len, l3@f\_this\_len\%BSIZE = 0, l3@size_{f\_this\_list} <= l3@f\_this\_len, l3@size_{f\_this\_list} < BSIZE, l3@f\_this\_len = (l3@size_{f\_this\_list} * BSIZE),$
$\mathbf{l7@size_{f\_this\_init\_list} = l3@size_{f\_this\_list}, l7@f\_this\_init\_len = l3@f\_this\_len,}$
$\mathbf{l7@how\_init = 5,}$
$BSIZE = 5, l7@size_{f\_this\_list} = l7@size_{f\_this\_init\_list}, l7@f\_this\_len = l7@f\_this\_init\_len, l7@how = l7@how\_init, l7@f\_this\_len\%BSIZE = 0, l7@how = BSIZE, l7@size_{f\_this\_list} <= l7@f\_this\_len, l7@size_{f\_this\_list} < l7@how, l7@f\_this\_len = (l7@size_{f\_this\_list} * l7@how)$

Simplified invariant:

$BSIZE = 5, count\_it >= 1, count\_it <= size_c\, count\_it = f\_this\_len + 1, f\_this\_len\%BSIZE = 0$

Number of solutions:

$\mathtt{C}(\mathcal{I}^{addAll}_{newBlock.7}) = \frac{1}{5} size_c + (per(size_c, [0, \frac{4}{5}, \frac{3}{5}, \frac{2}{5}, \frac{1}{5}])$

Table B.3: Original and simplified global invariant for the call chain and the counting expression for `addAll.11.add.3.newBlock.7`

Finally, combining the generated local invariants and binding information ob-

tained from method calls we produce control state invariants. In Table B.3 we show a control state invariant for `addAll.11.add.3.newBlock.7` after the binding.

APPENDIX C

Symbolic Bernstein Expansion over a Convex Polytope

This section explains the theory behind Bernstein expansion. We first recall the classical Bernstein expansion of a univariate polynomial over an interval and then show how it can be extended to multivariate parametric polynomials over parametric convex polytopes.

C.1. Bernstein Expansion over an Interval

There are many ways to represent a (rational) univariate degree-d polynomial $p(x) \in \mathbb{Q}[x]$. The canonical representation of $p(x)$ is as a $\mathbb{Q}$-linear combination of the power base, i.e., the powers of x,

$$p(x) = \sum_{i=0}^{d} a_i x^i, \tag{C.1.1}$$

with $a_i \in \mathbb{Q}$. The polynomial $p(x)$ can also be represented as a $\mathbb{Q}$-linear combination of the degree-d *Bernstein base polynomials* [Ber52, Ber54, FR87, BB00]:

$$p(x) = \sum_{k=0}^{d} b_k^d B_k^d(x), \tag{C.1.2}$$

where the Bernstein polynomials $B_i^d(x)$ are defined by:

$$B_k^d(x) = \binom{d}{k} x^k (1-x)^{d-k} \quad k = 0, 1, ..., d \quad \binom{d}{k} = \frac{d!}{k!(d-k)!}, \tag{C.1.3}$$

and $b_i^d \in \mathbb{Q}$ are the Bernstein coefficients corresponding to the degree-d basis.

Example C.1. Here is an example of a univariate polynomial in its power form and in its Bernstein form:

$$p(x) = x^3 - 5x^2 + 2x + 4 = 4B_0^3(x) + \frac{14}{3}B_1^3(x) + \frac{11}{3}B_2^3(x) + 2B_3^3(x)$$

where $B_0^3(x) = (1-x)^3, B_1^3(x) = 3x(1-x)^2, B_2^3(x) = 3x^2(1-x)$ and $B_3^3(x) = x^3$. We will explain below how to compute the Bernstein coefficients in this expression. □ □

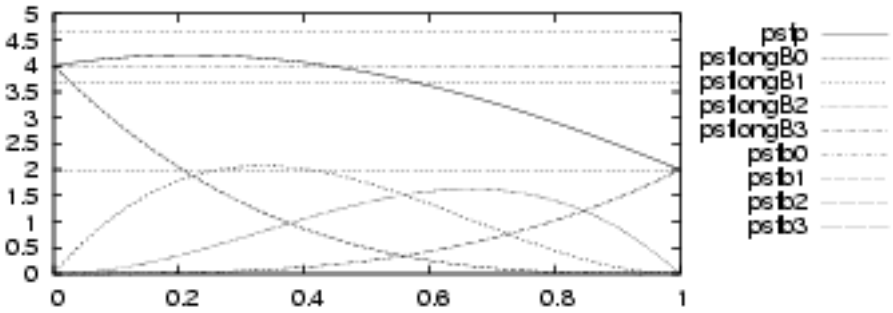


Figure C.1: Decomposition of the polynomial $p(x) = x^3 - 5x^2 + 2x + 4$ in the Bernstein basis

The Bernstein expansion of a polynomial has many interesting properties. The properties that will interest us most here is that the sum of the Bernstein base polynomials (C.1.3) is 1 and that, on the interval $[0, 1]$, $0 \leq B_k^d(x) \leq 1$. The first property follows from the identity:

$$1 = (x + (1 - x))^d = \sum_{k=0}^{d} B_k^d(x).$$

On the interval $[0, 1]$, Equation (C.1.2) expresses the polynomial $p(x)$ as a convex combination (with coefficients $B_i^d(x)$) of the Bernstein coefficients b_i^d. On this interval, the polynomial $p(x)$ is therefore bounded by its Bernstein coefficients, i.e.,

$$\min_{0 \leq i \leq d} b_i^d \leq p(x) \leq \max_{0 \leq i \leq d} b_i^d.$$

Moreover, if the minimum or maximum of the b_i^d is b_0^d or b_d^d then this bound is exact, since they correspond to values taken by $p(x)$ at the vertices as is clear from (C.1.3). These coefficients where the bound is exact are sometimes referred to as *sharp coefficients*.

[RR84] proved that the estimation error can be made smaller as the degree d is elevated. Hence, tighter bounds can be obtained by expressing the polynomial $p(x)$ in terms of higher degree ($> d$) Bernstein base polynomials.

Example C.2. Figure C.1 shows the polynomial $p(x) = x^3 - 5x^2 + 2x + 4$ from the previous example, the terms $b_i^3 B_i^3(x)$ of its Bernstein form and the constants b_i^3. On the interval $[0, 1]$, the polynomial is bounded by the minimal and maximal Bernstein coefficients, $b_3^3 = 2$ and $b_1^3 = 14/3$. The first of these coefficients is sharp; the second is not. □

To compute the Bernstein coefficients b_i^d from the power form coefficients a_i, we write the point x on the interval $[0, 1]$ in terms of its barycentric coordinates,

$$x = \alpha_0 v_0 + \alpha_1 v_1,$$

with

$$\alpha_i \geq 0 \quad \text{for } i \in \{0, 1\} \qquad \text{and} \qquad \alpha_0 + \alpha_1 = 1$$

and where $v_0 = 0$ and $v_1 = 1$ are the vertices of the interval $[0, 1]$. We see that $\alpha_1 = x$ and $\alpha_0 = 1 - x$ and that the Bernstein base polynomials (C.1.3) are homogeneous polynomials of degree d in α_0 and α_1. To write $p(x)$ (C.1.1) as a homogeneous

polynomial in α_0 and α_1, we simply substitute $x = \alpha_0\, 0 + \alpha_1\, 1 = \alpha_1$ and multiply each degree-i homogeneous component of $p(\alpha_0, \alpha_1)$ $(i \leq d)$ by $1 = (\alpha_0 + \alpha_1)^{d-i}$, i.e.:

$$\begin{aligned} p(\alpha_0, \alpha_1) &= \sum_{i=0}^{d} a_i \alpha_1^i (\alpha_0 + \alpha_1)^{d-i} \\ &= \sum_{i=0}^{d} a_i \alpha_1^i \left(\sum_{j=0}^{d-i} \binom{d-i}{j} \alpha_0^{d-i-j} \alpha_1^j \right) = \sum_{k=0}^{d} \left(\sum_{i=0}^{k} a_i \binom{d-i}{k-i} \right) \alpha_1^k \alpha_0^{d-k}. \end{aligned}$$

Comparing with (C.1.2) and noting that

$$B_k^d(x) = B_k^d(\alpha_0, \alpha_1) = \binom{d}{k} \alpha_1^k (\alpha_0)^{d-k}, \tag{C.1.4}$$

we obtain:

$$b_k^d = \sum_{i=0}^{k} \frac{\binom{d-i}{k-i}}{\binom{d}{k}} a_i = \sum_{i=0}^{k} \frac{\binom{k}{i}}{\binom{d}{i}} a_i,$$

where the last equality follows from the identity:

$$\binom{d-i}{k-i}\binom{d}{i} = \binom{d}{k}\binom{k}{i}.$$

Bounds on the values attained by a polynomial over an arbitrary interval $[a, b]$ can be obtained using essentially the same technique. We write:

$$x = \alpha_0\, a + \alpha_1\, b,$$

with

$$\alpha_i \geq 0 \quad \text{for } i \in \{0, 1\} \qquad \text{and} \qquad \alpha_0 + \alpha_1 = 1,$$

substitute this expression in $p(x)$ to obtain a polynomial $p(\alpha_0, \alpha_1) \in \mathbb{Q}[\alpha_0, \alpha_1]$, multiply each term with the appropriate power of $1 = \alpha_0 + \alpha_1$ and compute the coefficients b_k^d with respect to the basis formed by the terms in the expansion

$$1 = (\alpha_0 + \alpha_1)^d = \sum_{k=0}^{d} B_k^d(\alpha_0, \alpha_1).$$

The terms $B_k^d(\alpha_0, \alpha_1)$ are defined as in (C.1.4). They are then again the coefficients in the expression of $p(\alpha_0, \alpha_1)$ as a convex combinations of the b_k^d and so

$$\min_{0 \leq i \leq d} b_i^d \leq p(x) \leq \max_{0 \leq i \leq d} b_i^d$$

on the interval $[a, b]$.

C.2. Bernstein Expansion over a Convex Polytope

In this section, we generalize the so-called Bernstein-Bezier form of a polynomial defined over a triangle [Far93], and apply the same principles to multivariate parametric polynomials defined over parametric polytopes of any dimension.

A (rational) convex polytope $P \subset \mathbb{Q}^n$ is the convex hull of a set of points $\vec{v}_i$,

$$P = \left\{ \vec{x} \mid \exists \alpha_i \in \mathbb{Q} : \vec{x} = \sum_i \alpha_i \vec{v}_i, \alpha_i \geq 0, \sum_i \alpha_i = 1 \right\}.$$

If no $\vec{v}_i$ is a convex combination of the other $\vec{v}_i$ and then these $\vec{v}_i$ are called the vertices of the polytope.

To compute lower and upper bounds on a (rational) multivariate polynomial $p(\vec{x}) \in \mathbb{Q}[\vec{x}] = \mathbb{Q}[x_1, \ldots, x_n]$,

$$p(x_1, x_2, \ldots x_n) = \sum_{i_1=0}^{d_1} \sum_{i_2=0}^{d_2} \cdots \sum_{i_n=0}^{d_n} a_{i_1,i_2,\ldots,i_n}\, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \tag{C.2.1}$$

over a polytope $P \subset \mathbb{Q}^n$, we essentially follow the procedure from the previous section. We first write $\vec{x}$ as a convex combinations of the vertices

$$\vec{x} = \sum_i \alpha_i \vec{v}_i$$

and substitute this expression in the polynomial $p(\vec{x})$. We then multiply each term in the result with the appropriate power of $1 = \sum_i \alpha_i$ to obtain a homogeneous polynomial in the α_i of degree d, where d is the maximum of the d_i. Finally, we compute the coefficients $b^d_{\vec{k}}$, for $\vec{k} = (k_1, \ldots, k_n)$, $0 \le k_i$, $\sum k_i = d$, in terms of the *generalized Bernstein base polynomials* $B^d_{\vec{k}}$. These generalized Bernstein base polynomials are the terms in the expansion of

$$\begin{aligned} 1 &= (\alpha_1 + \alpha_2 + \cdots + \alpha_n)^d \\ &= \sum_{\substack{k_1,k_2,\ldots,k_n \ge 0 \\ k_1+k_2+\cdots+k_n=d}} \binom{d}{k_1, k_2, \ldots, k_n} \alpha_1^{k_1} \alpha_2^{k_2} \cdots \alpha_n^{k_n} = \sum_{\substack{k_1,k_2,\ldots,k_n \ge 0 \\ k_1+k_2+\cdots+k_n=d}} B^d_{\vec{k}}(\vec{\alpha}), \end{aligned}$$

where

$$\binom{d}{k_1, k_2, \ldots, k_n} = \frac{d!}{k_1! k_2! \ldots k_n!} \tag{C.2.2}$$

are the multinomial coefficients. Note that, again, the $B^d_{\vec{k}}(\vec{\alpha})$ are nonnegative and sum to 1 and so can be considered to be the coefficients in the expression of $p(\vec{x})$ as a convex combination of the $b^d_{\vec{k}}$. We therefore have

$$\min_{\substack{k_1,k_2,\ldots,k_n \ge 0 \\ k_1+k_2+\cdots+k_n=d}} b^d_{\vec{k}} \le p(\vec{x}) \le \max_{\substack{k_1,k_2,\ldots,k_n \ge 0 \\ k_1+k_2+\cdots+k_n=d}} b^d_{\vec{k}} \tag{C.2.3}$$

on the polytope $P \subset \mathbb{Q}^n$. The generalized Bernstein base polynomials we use here are different from the multivariate Bernstein polynomials [ZG98, CT04], which are products of standard Bernstein polynomials.

Note that the algorithm outlined above does not require the points $\vec{v}_i$ to be the vertices of the polytope P. They may instead be any set of generators for the polytope P.

We may also consider *parametric polytopes* $P : D \to \mathbb{Q}^n : \vec{q} \mapsto P(\vec{q})$,

$$P(\vec{q}) = \left\{ \vec{x} \mid \exists \alpha_i \in \mathbb{Q} : \vec{x} = \sum_i \alpha_i \vec{v}_i(\vec{q}), \alpha_i \ge 0, \sum_i \alpha_i = 1 \right\}, \tag{C.2.4}$$

where $D \subset \mathbb{Q}^r$ is the parameter domain and $\vec{v}_i(\vec{q}) \in \mathbb{Q}[\vec{q}]$ are arbitrary polynomials in the parameters $\vec{q}$. Note that some of these generators may be vertices for only a subset of the values of the parameters. The coefficients $a_{\vec{i}}$ of the polynomial $p(\vec{x})$ (C.2.1) may also themselves be polynomials in the parameters $\vec{q}$, i.e., $p(\vec{x}) \in (\mathbb{Q}[\vec{q}])[\vec{x}]$ and

$$a_{\vec{i}} = \sum_{j_1=0}^{m_1} \sum_{j_2=0}^{m_2} \cdots \sum_{j_r=0}^{m_r} b_{j_1,j_2,\ldots,j_r}\, q_1^{j_1} q_2^{j_2} \cdots q_r^{j_r}.$$

Applying the algorithm outlined above, we obtain parametric generalized Bernstein coefficients $b^d_{\vec{k}}(\vec{q})$ and parametric bounds

$$\min_{\substack{k_1,k_2,\ldots,k_n\geq 0\\ k_1+k_2+\cdots+k_n=d}} b^d_{\vec{k}}(\vec{q}) \leq p(\vec{q})(\vec{x}) \leq \max_{\substack{k_1,k_2,\ldots,k_n\geq 0\\ k_1+k_2+\cdots+k_n=d}} b^d_{\vec{k}}(\vec{q}). \tag{C.2.5}$$

The removal of redundant bounds in this expression is discussed in Section C.3.

Example C.3. Consider the polynomial $p(x_1,x_2)=\frac{1}{2}x_1^2+\frac{1}{2}x_1+x_2$ over the parametric polytope generated by the points $\begin{pmatrix}0\\0\end{pmatrix}$, $\begin{pmatrix}N\\0\end{pmatrix}$ and $\begin{pmatrix}N\\N\end{pmatrix}$. Hence any point $\begin{pmatrix}x_1\\x_2\end{pmatrix}$ in the polytope is a convex combination of these points:

$$\begin{pmatrix}x_1\\x_2\end{pmatrix}=\alpha_1\begin{pmatrix}0\\0\end{pmatrix}+\alpha_2\begin{pmatrix}N\\0\end{pmatrix}+\alpha_3\begin{pmatrix}N\\N\end{pmatrix}\qquad 0\leq\alpha_i\leq 1\qquad \sum_{i=1}^{3}\alpha_i=1$$

By replacing $\begin{pmatrix}x_1\\x_2\end{pmatrix}$ with this convex combination, a new polynomial is obtained whose variables are $\alpha_1,\alpha_2,\alpha_3$:

$$\frac{1}{2}N^2\alpha_2^2+N^2\alpha_2\alpha_3+\frac{1}{2}N^2\alpha_3^2+\frac{1}{2}N\alpha_2+\frac{3}{2}N\alpha_3$$

Monomials of degree less than 2 are transformed into sums of monomials of degree 2:

$$\begin{aligned}\frac{1}{2}N\alpha_2 &= \frac{1}{2}N\alpha_2(\alpha_1+\alpha_2+\alpha_3)\\ \frac{3}{2}N\alpha_3 &= \frac{3}{2}N\alpha_3(\alpha_1+\alpha_2+\alpha_3).\end{aligned}$$

The final polynomial is:

$$\begin{aligned}p(\alpha_1,\alpha_2,\alpha_3)=\left(\frac{1}{2}N^2+\frac{1}{2}N\right)\alpha_2^2+\left(\frac{1}{2}N^2+\frac{3}{2}N\right)\alpha_3^2\\ +\frac{1}{2}N\alpha_1\alpha_2+\frac{3}{2}N\alpha_1\alpha_3+(N^2+2N)\alpha_2\alpha_3.\end{aligned}$$

The basis is built from the expansion of $(\alpha_1+\alpha_2+\alpha_3)^2$ providing the following monomials:

$$\begin{aligned}B_{2,0,0} &= \alpha_1^2\\ B_{0,2,0} &= \alpha_2^2\\ B_{0,0,2} &= \alpha_3^2\\ B_{1,1,0} &= 2\alpha_1\alpha_2\\ B_{1,0,1} &= 2\alpha_1\alpha_3\\ B_{0,1,1} &= 2\alpha_2\alpha_3.\end{aligned}$$

Rewriting $p(\alpha_1,\alpha_2,\alpha_3)$ in terms of this basis, we obtain

$$\begin{aligned}0\,B_{2,0,0}+\left(\frac{1}{2}N^2+\frac{1}{2}N\right)B_{0,2,0}+\left(\frac{1}{2}N^2+\frac{3}{2}N\right)B_{0,0,2}\\ +\frac{1}{4}NB_{1,1,0}+\frac{3}{4}NB_{1,0,1}+\left(\frac{1}{2}N^2+N\right)B_{0,1,1}.\end{aligned}$$

It can then be concluded that the polynomial varies between 0 and $\frac{1}{2}N^2+\frac{3}{2}N$. Since both of these coefficients are sharp coefficients, the bounds are exact bounds. The graph of the polynomial and the corresponding Bernstein coefficients are shown in Figure C.2 for $N=10$. □ □

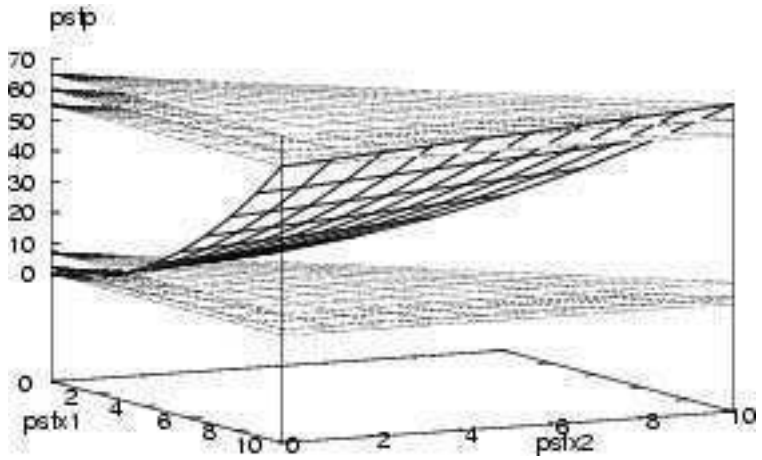


Figure C.2: The polynomial $p(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{2}x_1 + x_2$ and the corresponding Bernstein coefficients

C.3. Bounding a Polynomial over a Parametric Domain

We already explained in Section C.2 that given a polynomial and a set of parametric points, we can compute the Bernstein coefficients of the polynomial over the parametric convex polytope generated by the parametric points and that for any value of the parameters the minimum and maximum values over all Bernstein coefficients evaluated for this particular value of the parameters, provide a lower and an upper bound for the value of the polynomial over the convex polytope associated to these parameter values. However, in many situations where we wish to find a bound for a polynomial, the domain over which we wish to compute this bound is not given by a set of generators, but rather by a set of constraints. Also, when evaluating the lower or upper bound, we want to evaluate as few of the Bernstein coefficients as possible. We discuss these two issues in this section.

For example, suppose we want to compute an upper bound for the polynomial

$$-\frac{1}{2}i^2 - \frac{3}{2}i - j - n^2 + 4n + 2in \tag{C.3.1}$$

over the domain

$$D(n) = \{\,(i,j) \mid 0 \le i \le 3n-1 \wedge 0 \le j \le n-1 \wedge 3n-1 \le i+j \le 4n-2\,\}, \tag{C.3.2}$$

where n is a parameter. The first step is to compute the (parametric) vertices of $D(n)$. If the domain is bounded by linear constraints in the variables and the parameters, i.e.,

$$P : D \to \mathbb{Q}^n : \vec{q} \mapsto P(\vec{q}) = \{\,\vec{x} \in \mathbb{Q}^d \mid A\vec{x} \ge B\vec{q} + \vec{d}\,\}, \tag{C.3.3}$$

with $D \subset \mathbb{Q}^r$, $A \in \mathbb{Z}^{m \times n}$, $B \in \mathbb{Z}^{m \times r}$ and $\vec{d} \in \mathbb{Z}^m$, then we can use `PolyLib` [Loe99] to compute these vertices. The result is a subdivision of the parameter space in polyhedral cells, called *chambers*, each with an associated set of parametric vertices [CL98]. Note that we mentioned in Section C.2 that the generators of a parametric polytope do not need to be vertices for all values of the parameters. However, they do obviously have to be inside the parametric polytope. Vertices associated with one subdomain that are not also associated with another subdomain will lie *outside* of this other subdomain. We therefore need to treat each subdomain separately. In the

example, there is only one parameter domain and we find the vertices

$$\left\{\binom{2n}{n-1}, \binom{3n-1}{0}, \binom{3n-1}{n-1}\right\} \qquad \text{if } n \geq 1.$$

If the constraints describing the domain are only linear in the variables (and not in the parameters), then we may still compute the vertices of the domain, but the subdomains of the parameter space that have a fixed set of parametric vertices will no longer be polyhedral [Rab06].

The next step is to compute the Bernstein coefficients as explained in Section C.2. For our example we obtain the coefficients

$$n^2 - \frac{n}{4} + \frac{5}{4}, \frac{n^2}{2} + \frac{n}{2} + 1, \frac{n^2}{2} + \frac{3}{2}, n^2 + 1, \frac{n^2}{2} - \frac{n}{2} + 2, n^2 - \frac{3n}{4} + \frac{7}{4}.$$

An upper bound $u(n)$ for the value of the polynomial over $D(n)$ is therefore

$$u(n) = \max\left\{n^2 - \frac{n}{4} + \frac{5}{4}, \frac{n^2}{2} + \frac{n}{2} + 1, \frac{n^2}{2} + \frac{3}{2}, n^2 + 1, \frac{n^2}{2} - \frac{n}{2} + 2, n^2 - \frac{3n}{4} + \frac{7}{4}\right\} \quad \text{if } n \geq 1. \tag{C.3.4}$$

To compute the upper bound for any particular value of n, we therefore need to evaluate these 6 polynomials at this value and take the maximum. However, it is clear that some of these polynomials are redundant in the sense that for any value of the parameters in the domain the polynomial always evaluates to a smaller number than some other polynomial.

The simplest way to eliminate redundant Bernstein coefficients, is to examine the sign of the difference between two polynomials. If the sign is constant over the domain (where a zero sign may be treated as either positive or negative), then one of the two is redundant. Some easy ways of determining the sign of a (difference) polynomial are as follows.

1. If the difference is a constant, the check is trivial.

2. If the difference is linear in the parameters, we add the constraint that the difference be strictly larger than zero to the domain and check whether it becomes empty. For example, the polynomial $\frac{n^2}{2} + \frac{3}{2}$ is redundant since

$$\left(\frac{n^2}{2} + \frac{3}{2}\right) - \left(\frac{n^2}{2} + \frac{n}{2} + 1\right) = \frac{1}{2} - \frac{n}{2}$$

   and this difference polynomial is never greater than zero for $n \geq 1$. The polynomial $\frac{n^2}{2} - \frac{n}{2} + 2$ is eliminated for the same reason, while $n^2 - \frac{n}{4} + \frac{5}{4}$ and $n^2 - \frac{3n}{4} + \frac{7}{4}$ are eliminated because they are redundant with respect to $n^2 + 1$. If it turns out that the sign of the difference varies over the domain, we could in principle further subdivide the domain along the above constraint.

3. If the domain over which we want to determine the sign is bounded, we can apply Bernstein expansion again on the difference over this domain, which is now considered to be a fixed domain without parameters. The resulting Bernstein coefficients are therefore constants. If all the non-zero Bernstein coefficients have the same sign, then so will the difference over the whole domain. For example, if we assume that there is an upper bound on n, say 1000, then we can perform Bernstein expansion on

$$\left(\frac{n^2}{2} + \frac{n}{2} + 1\right) - \left(n^2 + 1\right) = -\frac{n^2}{2} + \frac{n}{2} \tag{C.3.5}$$

over $1 \leq n \leq 1000$. The resulting Bernstein coefficients are

$$\left\{0, -499500, \frac{-999}{4}\right\}$$

and so we can conclude that $\frac{n^2}{2}+\frac{n}{2}+1$ is redundant with respect to n^2+1. Note that if the polynomial is univariate of degree d with coefficients c_i then we know that all real roots lie in the interval $[-M, M]$ with $M = 1+\max_{0\leq i\leq d-1} |c_i|/|c_d|$ (Cauchy's bound). It is therefore sufficient to consider the intersection of a strict superset of this interval with the possibly unbounded domain of interest. In the example, it would be sufficient to consider the domain $1 \leq n \leq 3$.

4. If the domain over which we want to determine the sign is not bounded, but there is a lower bound on one of the parameters, we can write the Taylor expansion of the difference about this lower bound and determine the signs of the coefficients in the Taylor expansion. Note that we can easily compute these coefficients using synthetic division. If all signs are constant and equal, then also the difference will have this constant sign. For example, we can write (C.3.5) as

$$-\frac{1}{2}(n-1) - \frac{1}{2}(n-1)^2$$

and the coefficients are clearly negative, so we can again conclude that $\frac{n^2}{2} + \frac{n}{2}$ is redundant, over the whole domain $n \geq 1$.

In our example we have now been able to simplify (C.3.4) to

$$u(n) = n^2 + 1 \qquad \text{if } n \geq 1. \tag{C.3.6}$$

In general, we will however not be able to identify all but one polynomial as redundant. Still, it may be desirable in some cases to have only a single polynomial associated to every subdomain, such that for a given subdomain only this single polynomial needs to be evaluated. If the difference between two polynomials is linear then this can easily be accomplished by splitting the domain along the hyperplane where the difference is zero. For example, suppose we have two polynomials $n^2 + 3n - 500$ and $n^2 + n$ in the maximum expression associated to the domain $n \geq 4$. The difference between these two polynomials $2n - 500$ is zero along $n = 250$ and so we would split the domain into, say, $4 \leq n < 250$ and $250 \leq n$. If the differences between pairs of polynomials is not linear, but they are univariate, then we may not be able to easily split the domain into subdomains where only a single polynomial remains, but based on Cauchy's bounds, we can identify and split off a region of "big" values where the upper bound is given by a single polynomial.

List of Figures

List of Tables