Università degli Studi di Genova

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale

EveC: an object-oriented programming language for efficient computation

Candidati:
Luca Roverelli
Gabriele Zereik
Nicoló Zereik

Relatore:
Char.mo Prof. Mauro Migliardi

Correlatore:
Prof. Alessio Merlo

Anno Accademico 2011-2012

Abstract

In the last years, the *Java* programming language has reached a very high popularity. Its portability across architectures and operating systems, the automatic garbage collection and the huge library system available are only few of the motivations of this global acceptance. However, all these features comes with some drawbacks. In particular, portability implies that every *Java* application needs a *Java Virtual Machine* to run. The goal of this thesis is to develop a new programming language, *EveC*, similar to *Java* in the syntax but that does not need an intermediate layer, like the *JVM* for *Java*, and the consequent drawbacks. In particular, the system developed consists of a compiler that translates the *EveC* code into *ANSI-C* code. Moreover, the compiler includes a high-level optimization module, whose task is to optimize the produced *C* code. The optimized code can be, finally, compiled into native code by any *ANSI-C* compiler. In a period in which the development and the optimization of the *JVM* layer seems to be the leading direction, this work aims at demonstrating that a completely different approach can lead to comparable, or even better results, both in terms of memory usage and execution time.

Contents

Chapter 1

Introduction

The *Java* programming language [1] was firstly released in 1996 and, since then, its popularity has tremendously grown. The reasons for the widespread acceptance achieved by this programming language throughout the variety of the existing ones are multiple. For example, unlike most of other programming languages, *Java* provides many features like an execution environment that supplies services such as security, portability across operating systems and architectures, automatic garbage collection and a huge library that let the programmers easily use programming characteristic that were hard to use before, such as networking and multi-threading.

As the authors have explained in the *Java White Paper* [2], the *Java* programming language (hereafter *Java*) was born with precise intents and characteristics. For the goal of this thesis, three of them have been considered as particularly important: *Architecture Neutrality*, *Portability* and *Robustness*.

Architecture Neutrality and *Portability* allow programmers to "write once, run everywhere", in the sense that the same compiled code can be distributed and executed on every architecture in which is available a *Java Virtual Machine* (JVM). To achieve this, the *Java* compiler produces *ByteCode* instead of the traditional native code. The *ByteCode* is a condensed representation of the original source code, containing architecture-independent instructions, that has to be interpreted by the *JVM*. This means that every instruction is executed via-software using a virtual processor emulated by the *JVM* and not directly by the hardware processor.

Moreover, the *Java* programming language provides a pointer model that eliminates the possibility of overwriting memory and corrupting data: *Java* programmers are always sure that every single memory access is safe.

All these features come with a counterweight: it is reasonable to think that the *JVM* represents a weight as for execution time for the application,

as the application does not only have to be emulated, but, for some spots, it is compiled into native code at *run-time*. In addition it is reasonable to think that the *JVM* also affects memory usage [3]: in fact, in memory resides not only the data needed by the program, but also the one needed by the *Java Virtual Machine*. Moreover the pointer model that *Java* uses adds implicit check instructions in order to ensure the safeness of every memory access: this clearly decreases performances too.

After twenty years of research and development of the *JVM*, the negative effects of these and other issues have been dramatically reduced. The most important improvement is the*Just-in-time* (*JIT*) compilation [4], a technique that allows to compile into native code portions of the executed program. The *JVM* starts interpreting the program; once it detects that a portion of executed code is important (that is an "hot spot"; e.g. a function called many times), the *JVM* can decide to compile it to native code and directly execute it through the hardware processor [5].

Other optimizations are introduced by the *JIT* compiler, some of which aim at removing the implicit check instructions [5], when not required, without loosing safeness. This leads to an increase in the performances related to the execution time in some cases.

Nevertheless the same downsides are partially still present: the overhead introduced by the *JVM* layer on the execution time is still remarkable; in addition, it also affects the memory usage, a part of which is imperatively occupied by the virtual machine [3].

When executing a program, the *JVM* has to strike a balance between the portion of *ByteCode* to be interpreted and the one to be firstly compiled into native code and then executed; the optimizations that the *JIT* performs must be limited because the compilation process is performed at *run-time*; on the other hand, the *JIT* has more information than a static compiler.

As for memory usage, since nowadays in most of the architectures are available gigabytes of memory, the effects of the presence of the *JVM* layer are not so relevant; however, a better use of the memory can be exploited for embedded systems application, for which the memory usage is critical (this assertion has to be validated by a deeper study, which has not been conducted yet).

The goal of this thesis arises from all these considerations. Instead of using a *JIT* compilation technique, we statically compile the whole source program directly into native code, allowing optimizations that cannot be integrated in an environment such as the *JVM*. We maintain portability in the sense that the same source code, without changes, can be compiled theoretically for any architecture where an *ANSI-C* compiler is available. This approach originates from the idea that a direct translation from an

object-oriented programming language to native code, made at compile time, can avoid many of the presented issues and may generate smaller overhead at *run-time*. In literature, something similar has been tried; e.g. the *Marmot Compiler* [6], the *Excelsior Jet* [7] and the *IBM High Performance Compiler for Java* [8] translate *ByteCode* to native code; the approach proposed in *Java-through-c compilation* [9] and in *Translating java to c without inserting class initialization tests* [10] is to translate *ByteCode* into *ANSI-C* and than an *ANSI-C* compiler is used to produce native code.

The approach proposed in this thesis is to translate an high-level object-oriented language, similar to *Java*, to *ANSI-C*, performing high-level optimizations targeted to object-oriented languages, and leaving all the low-level and architecture-depending optimization to an *ANSI-C* compiler to produce optimized native code.

In addition, in order to allow users to exploit the new programming language without the necessity of the command line, an *Eclipse* plug-in has been implemented. This integrates in the *Eclipse* environment [11] and provides the user a graphical interface to create, build and run programs written with the new language. For more detailed information about it see appendix B.

Chapter 2

System Overview

The presented work consists of the definition of a new programming language, *EveC*, that appears to be very close to *Java* as regards syntax (see Appendix A for the language specifications). However, despite this apparent similarity, the *EveC* source code is translated into *ANSI-C* code, optimized through some high-level optimization targeted to object-oriented languages, and finally compiled by an *ANSI-C* compiler that produces optimized native code. *EveC* inherits some features of *Java*. For example, it ensures to have a similar robustness and portability between operating systems and architectures to *Java*, with the only difference that it is necessary to rebuild the source code for the target architecture and operating system, without the need to change any line of code. The *Runtime System* is provided with a concurrent garbage collector, an exception system and concurrent primitives. *EveC* provides a full generic class support and, unlike *Java*, it is possible to determine at *run-time* the type of every object. Moreover, *EveC* supports also unsigned primitive types and arithmetic. The full definition of the language is specified in appendix A.

Building an entire compiler from scratch is an enormous project, especially if different architectures and operating systems must be supported. Different architectures mean different instruction sets and different low level architecture-dependent optimizations are absolutely necessary to achieve good performances. For this reason, the *EveC* source code is translated into *ANSI-C* code, that is compiled by a state of the art *C* compiler in order to achieve the best possible performance. Because *C* is extremely popular, an *ANSI-C* compiler is available for a huge number of architectures and operating systems, allowing to tremendously reduce the effort to port the *EveC* compiler to different architectures. The compiler used in this work, is the *GNU C Compiler* (*GCC*) [12], even if it could be switched, without any effort, with other compilers, like the *Clang* [13]. The use of an *ANSI-C*

compiler as back-end of the *EveC* compiler has some drawbacks. Some constructs, like the *try-catch* statement and the exception system, are difficult to be implemented in *ANSI-C*. For example, the *Marmot compiler* [6] uses a program-counter-based exception handling mechanism, with no *run-time* cost for the expected case in which no exception is thrown during the execution of the program. This mechanism cannot be implemented in this work because there is no access to the program counter and, moreover, there is no control over the native code generated by the *ANSI-C* compiler (more detailed explanations about exception handling will be presented in chapter 3).

As mentioned before, some works that translate the *Java* programming language to native code, using static compilation techniques, are present in literature. The work that is closest to our thesis is [9]: in such a project the *ByteCode* generated by the standard *Java* compiler is translated into *C* that is compiled by the *GNU C Compiler* to produce native code. The produced *C* code is similar to the one produced by the *EveC* compiler in terms of class hierarchy management, and the garbage collector used is the same used in this project [14]. However, the interface system is managed in a completely different way with respect to the solution presented. The most important difference between this work and Varma's work is that the second focuses on producing the smallest possible *C* code, targeting embedded systems. Moreover, in the result that they present, their *C* code does not perform array bound checks and thus their code is unsafe.

The presented approach is different to all the other works: in fact, all of these works translate the *ByteCode* directly into native code or into *C* code. On the contrary, the *EveC* source code is translated directly into *C*, without any intermediate representation like *ByteCode*. This approach has some benefits but also some drawbacks. Parsing the *ByteCode* is easier than developing a complex lexer and parser system to manage a complex language like *Java* or *EveC*. Moreover, working at *ByteCode* level allows to use existing libraries, accessing to a huge amount of code without the necessity to rebuild anything. Nevertheless, the translation process from *ByteCode* to an higher representation level like *C* is not trivial. [9] uses *Soot* [15], a sophisticated *ByteCode* analysis and optimization framework to transform the *ByteCode* into a typed 3-address intermediate representation called *Jimp* [16]. Another important problem is that some information that are present in the source code are not included in *ByteCode*. For example, in *ByteCode*, the local variables do not have type information and, according to [6], one natural formulation of the type inferring problem for the local variables is NP-complete. The translation process from *EveC* or *Java* source code into *C* is in many cases trivial, because both languages share an important number of construct

with *C*. Moreover, translating from source code instead of *ByteCode* offers the opportunity to implement a superset of the *Java* programming language, that could be difficult otherwise.

This work is fully implemented using the *Java* programming language. This choice has two motivations. First, *Java* is very similar to *EveC*. Therefore, without much effort, it is possible to completely translate the compiler into *EveC* language, making the language self-compiling. The second motivation is that using *Java*, the *EveC* compiler can be used in various architectures and operating systems.

2.1 System Architecture

The system developed for this project is a compiler for the *EveC* programming language. The compilation process can be conceptually divided into three phases, as described in figure 2.1:

- Translation phase.
- Optimization phase.
- Compilation phase.

The first phase concerns the translation of the *EveC* source code into *ANSI-C*: an *ANTLR* grammar (see appendix C) is appointed to perform this translation, handling all those object-oriented features, like objects and classes, that do not exist in the target code (for more details see Chapter 3).

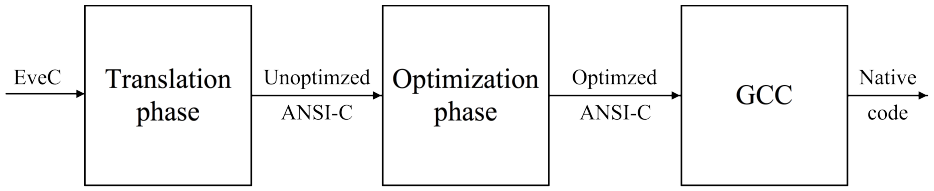


Figure 2.1: The different phase that characterize the compilation process.

The second phase consists of a set of optimizations, most of which are targeted for object-oriented programming languages and, therefore, cannot be performed by an *ANSI-C* compiler. In this phase the compiler manipulates the *C* code already generated in order to make it more efficient. The purpose of the last phase is to compile the optimized *C* code generated into native code. This task can be performed by any *ANSI-C* compiler, therefore the

EveC programming language is portable on a huge number of architectures and operating systems.

Chapter 3

EveC

3.1 Introduction

As said in the previous chapter, the first phase of the compilation process consists in the translation of the *EveC* source code into *C* code. This phase can be divided into two different subphases. In the first subphase, the source code is parsed for the first time and information about the classes or the interfaces, the methods and the attributes of the class or the interface are collected. During the second subphase, the source code is parsed for a second time and the translation from *EveC* to *C* is performed. This first phase of the compilation process is performed using an ANTLR grammar, differently annotated in the two phases. In the first subphase, the ANTLR grammar is annotated to collect all the necessary information. In the second subphase, the ANTLR grammar is annotated to directly translate the *EveC* code into *C*, performing all the necessary type checking, verification of existence of called methods and so on, using the information gathered by the previous subphase. The translation is performed directly, without using an intermediate representation. In the following, we describe how all the features of the *EveC* programming language are implemented or translated in C code.

3.2 Translate from EveC to C

For every class in the *EveC* source code, a corresponding set $Class$ is generated. Every set $Class$ is defined as follows:

$$Class \Rightarrow < \xi, \epsilon, \beta, \gamma, \varphi > \tag{3.1}$$

ξ is a set of `C global variables`. For each static attribute defined in the class, a corresponding element of ξ is generated.

ϵ is a set of `C functions`. For each static method defined in the class, a corresponding element of ϵ is generated.

β is a set of `C functions`. For each non-static and non-abstract method defined in the class, a corresponding element of β is generated. Moreover, each element of β is pointed by an element of the table γ.

γ is a table containing all the addresses of the non-static methods defined in the class and in all its ancestor classes. It is implemented as a *C struct*, where all members are *function pointer*s, with the exception of the first member, that is a pointer to the γ of the father class. All the *function pointer*s point to an element of β or to an element of the β set of the ancestor classes. For each $Class$, an instance of γ is created and shared between all the instances of the $Class$.

φ is a *C struct* that contains all the non-static attributes that the class and all its ancestor classes define. Moreover, the first element of the *struct* is a pointer to the corresponding γ while the second is a *string* describing the type of the instance.

In the code in listing 3.1 an example of a simple *EveC* class is presented. To clarify the translation process, the description of the corresponding set $Class$ is provided while, in the next sections, the detailed description of how all the features of the language are implemented is depicted.

Listing 3.1: Example of *EveC* class.

```
abstract class Animal {
  private int age;
  public Animal(int age) {
    this.age = age;
    count++;
  }
  public int getAge() {
    return age;
  }
  public abstract int getNumberOfPaws();
  private static int count = 0;
  public static int getCount() {
    return count;
  }
}
```

For the class `Animal`, the set ξ is formed by an only *C global variable*, named `AnimalStaticcount`, of type `_evec_int`, that corresponds to the static variable `count` of the class `Animal`.

Listing 3.2: Correspective set ϵ for the *EveC* class `Animal`.

```
_evec_int MethodAnimalStaticgetCount0(__evec_env *
   _evec_env) {
  return AnimalStaticcount;
}
```

The set ϵ is formed by an only element, that is the `C function` defined in listing 3.2. An argument of type `__evec_env*` is added, and it is neccesary to handle the stack trace and the exception system.

Listing 3.3: Correspective set β for the *EveC* class `Animal`.

```
Animal* MethodAnimalAnimal0(__evec_env *_evec_env,
   Animal *pointer, _evec_int age) {
  //some code initialization added by the compiler
  ...
  pointer->age = age;
  AnimalStaticcount++;
  return pointer;
}

_evec_int MethodAnimalgetAge0(__evec_env *_evec_env,
    Animal *this) {
  return this->age;
}
```

The set β is formed by two elements, described in listing 3.3. It can be noticed that the respective *C function* of method `getNumberOfPaws()` is not present, because it is defined as `abstract`, and no implementation is defined. Moreover, all the *C function*s have two more arguments compared to the equivalent *EveC* methods. The first one is a pointer of type `__evec_env`, having the same role as in static methods of the ϵ set. In the constructor, a pointer to a memory address is added. This is the heap-space allocated for the instance of the class `Animal`. In the non-constructor method, the second argument is the implicit pointer to the caller object.

The first element of the table γ for the class `Animal` is a pointer to the table γ of its father class; since the class `Animal` does not extend explicitly any class, it extends the `Object` class. Thus, the first element points to the respective `Object` γ table. Assuming that the class `Object` has n non-static methods, from the 2^{nd} element to the $n + 1^{th}$ element of the `Animal` γ table, this elements point to *C function*s that can be defined either in the

β set of the class `Animal`, or in the β set of one of the ancestor class of `Animal`. `Animal` does not override any method of the `Object` class, hence the elements from the 2^{nd} to the $n + 1^{th}$ point to *C functions* of the `Object` β. The last three elements of `Animal` β point respectively to the *C functions* `AnimalMethodAnimal0`, `AnimalMethodgetAge0`, and to `NULL`.

Listing 3.4: φ struct for the class `Animal`.

```
struct Animal {
  AnimalGamma *MethodTable;
  char *_evec_type;
  _evec_int age;
};
```

Finally, the φ `C struct` is described in listing 3.4. The first element is a pointer to the γ table of the class `Animal`. The second element is a pointer to `char` that defines the type of the instance. Since the class `Object` does not have any non-static attribute and the class `Animal` defines only the attribute `age`, the last field of φ is `age` of type `_evec_int`, that corresponds to the attribute defined in `Animal`.

3.2.1 Primitive Types

To ensure platform-independence, every *EveC primitive type*s are mapped into *C* types that are defined into the *EveC Runtime*. The *EveC* type α is mapped into the corresponding *C* type `_evec_`α. For example, the *EveC* type `long` is mapped to `_evec_long`.

3.2.2 Method Overloading

The *EveC* programming language allows to overload a method, defining two or more methods with the same name but with different arguments signature, letting the compiler choose the correct method to call. However, the *C* programming language does not allow the definition of two functions with the same name. Every *EveC* method name is translated in the following *C function* name:

$$MethodClassNameOriginalNameSuffix \tag{3.2}$$

where *ClassName* is the name of the *EveC* class in which the method is defined with an hash code that represents the *package* name of the class; *OriginalName* is the original method name defined in *EveC*; *NameSuffix* is an integer value added by the compiler to ensure that two methods with the

same name in *EveC* corrispond to two different *C function*s. For the first method defined with a certain name, the corresponding *NameSuffix* will be *0*; for the second method *NameSuffix* will be *1*, and so on.

When a certain method `A` is called, the compiler checks if the method is overloaded. In this case, it observes the type of arguments with which the method `A` is called, and it chooses the one having a signature that is the "closest" to the signature of the called method. The compiler infers the correct *NameSuffix* used to call the correct *C function*.

The "closest" signature means that the compiler first searches for a method with exactly the same signature of the called one. At most, only a method with the exact signature can be find. Let us assume that the method `A` is overloaded as described in listing 3.5, and that `SonClass` extends `SomeClass` and that `SomeClass` extends `FatherClass`. If the called method signature is `SonClass`, than the compiler will choose to call the method defined with `SomeClass` as argument, because it is the "closest" to the signature of the called method. The inheritance mechanism arranges all the classes in a strict hierarchy.

Listing 3.5: The method `A` overloaded.

```
public void A(FatherClass obj) {
   ...
}
public void A(SomeClass obj) {
   ...
}
```

3.2.3 Inheritance

Inheritance is a mechanism which allows a class to inherit the methods and the attributes of another class. A class that inherits the characteristics of another class is called *subclass*, while the class that provides the inheritance is called *superclass*. A class can have only a *superclass*, but each class can have an unlimited number of *subclass*es. In the *EveC* programming language, the inheritance mechanism arranges all the classes in a strict hierarchy. The class `Object` is on top of the hierarchy; thus, all the other classes extend implicitly or explicitly the `Object` class.

The class in listing 3.6 defines a new *EveC* class, called `Cat`, that extends the class `Animal` defined in the listing 3.1. The correspective set *Class* for the class `Cat` is defined as follows:

- Sets ξ and ϵ are empty, because no static attributes or methods are defined in class `Cat`.
- The set β is formed by two *C functions*, defined in listing 3.7.
- The table γ has the same identical elements of table γ of class `Animal` plus one more element. Moreover, the first element points to the γ table of class `Animal`. The last element points to the element of set β `MethodCatCat0`. The second-last element for table γ of class `Cat` does not point to `NULL` as the last element of the table γ for class `Animal`, but it points to the element `MethodCatgetNumberOfPaws0` of the set β.
- The φ is quite similar to the φ for class `Animal`. The only two differences are that the first element points to the set γ of the class `Cat`, and the second element points to a different C string that encapsulates the type of the instance.

Listing 3.6: The `Cat` class.

```
class Cat extends Animal {
  public Cat(int age) {
    super(age);
  }
  public int getNumberOfPaws() {
    return 4;
  }
}
```

Listing 3.7: The set β for the class `Cat`.

```
Cat * MethodCatCat0(__evec_env *_evec_env, Cat *
   pointer, _evec_int age) {
  ....
  MethodAnimalAnimal0(_evec_env,pointer,age);
  return pointer;
}
int MethodCatgetNumberOfPaws0(__evec_env *_evec_env,
    Cat *this) {
  return 4;
}
```

3.2.4 Polymorphism and Abstract Classes

Thanks to the particular layout chosen for all the class skeleton, the implementation of the polymorphism is trivial. Assuming the classes `Cat` and `Animal` as defined in the previous sections, the code in listing 3.8 can be directly translated, as described in listing 3.9.
The crucial point is the translation of the `obj.getNumberOfPaws` statement into `obj->MethodTable->getNumberOfPaws0`. At first, the *C* compiler calculates the offset from the start of `obj` to get the address of the `MethodTable` (that is the γ table corresponding to the object). Since the `MethodTable` for classes `Animal` and `Cat` is at the same offset, the address of the γ table of `Cat` is taken. Then, the offset of `getNumberOfPaws0` is calculated from the γ table of `Cat`, and thus the *C function* `MethodCatgetNumberOfPaws0` is called. This is exactly the expected behavior. In the example presented the overriding of an `abstract` method is analysed. If the method is not declared as `abstract`, the only change is that the last element of the γ table of class `Animal` does not point to `NULL`, but to the *C function* that implement the respective method. Nevertheless, the method overrided by the class `Cat` is still called, because the *function pointer* is taken from the γ table of class `Cat`. The differences between an `abstract` class and a standard class are that the `abstract` one can have one or more `abstract` method and thus one or more elements of the respective γ table can point to `NULL`, while in a standard class no element of the respective γ table can be `NULL`. It is important to notice that the order of the element in the γ table is absolutely important. For example, if the method `getNumberOfPaws` for class `Cat` was not placed with the same offset as the same element in class `Animal` in the respective γ table, then the method call would result in an unpredictable behavior.

Listing 3.8: An example of polymorphism with classes `Animal` and `Cat`.

```
...
Animal obj = new Cat(3);
int n = obj.getNumberOfPaws();
...
```

Listing 3.9: The translation of the *EveC* code in listing 3.8.

```
...
Animal *tmp = SuperMalloc(sizeof(Animal));
Animal *obj = (Animal*)MethodCatCat0(_evec_env,tmp
   ,3);
_evec_int n = obj->MethodTable->getNumberOfPaws0(
   _evec_env,obj);
...
```

3.2.5 Nested and Inner Classes

The *EveC* programming language allows the programmer to define *nested class*es and *inner class*es, with a behavior close to the equivalent construct in *Java*. A *nested class* is a class defined inside an *outer class*. In listing 3.10 an example is presented. This construct can be very useful to define a class which is necessary to perform some behaviour of the outer class, but there is no need to have another *package* visibility class.

The implementation of the *nested class*es is trivial and it is the same of the standard class defined in the previous sections.

Listing 3.10: An example of nested class.

```
public class Outer {
  static class Complex {
    double real;
    double imm;
  }
  static void init() {
    ...
    Complex a = new Complex();
    a.real = 4;
    a.imm = 3;
    ...
  }
}
```

As *nested class*es, the *inner class*es are classes defined inside an another class, called the *outher class*. The difference between the two is that the *inner class*es are sensitive to the context in which they are instantiated.

In listing 3.11 an example of *inner class* is presented. The method `other` of class `Outer` creates a new instance of the *inner class* `Inner`. Because the *inner class*es are sensitive to the context, the new instance of `Inner` is associated with the instance of `Outer` that calls the method `other`. Thanks to this association, the method `a` of class `Inner` can modify attributes and call methods of the instance of the *outer class*.

Listing 3.11: An example of *inner class*.

```
public class Outer {
  public void method() {
    ...
  }
  int k;
  class Inner {
    SomeData obj;
    public Inner() {
      ...
    }
    public void a() {
      k = 5;
      method();
    }
  }
  public void other() {
    Inner i = new Inner();
    i.a();
  }
}
```

The implementation of the *inner class*es is not trivial as for the *nested class*es. The translated *C* code for the methods of the example in listing 3.11 is presented in listing 3.12. In the φ for class `Inner` an implicit element is added. This element points to the instance of the outer class that creates the instance of `Inner`. If an attribute (or a method) is accessed (called) from a method of the inner class, the compiler searches its definition in the *inner class*. If no definition is found, then the compiler searches it in the definition of the *outer class*. Thus, in method `a` of class `Inner`, the attribute `k` of `Outer` is accessed through the implicit pointer to the outer class added in the φ *C struct* of class `Inner`. In the same way, the correct call to method `method` is performed.

Listing 3.12: The translated code for classes of listing 3.11.

```
void MethodOutermethod0(__evec_env *_evec_env, Outer
   * this) {
  ...
}
Inner *MethodInnerInner0(__evec_env *_evec_env,
   Inner *pointer, Outer *out) {
  //some code initialization addd by the compiler
  pointer->outer = out;
  ...
  return pointer;
}
void MethodInnera0(__evec_env *_evec_env, Inner *
   this) {
  this->outer->k = 5;
  this->outer->MethodTable->method0(_evec_env,this->
     outer);
}
void MethodOuterother0(__evec_env *_evec_env, Outer
   *this) {
  Inner *tmp = SuperMalloc(sizeof(Inner));
  Inner *i = MethodInnerInner0(_evec_env,tmp,this);
  i->MethodTable->a0(i);
}
```

3.2.6 Packages and class names

EveC allows to include one or more classes into a *package*. A *package* is defined as a name and it is equivalent to a folder. A *package* can contain classes, interfaces and also other *packages*, which are called subpackages. Let us assume the existence of a package `house`, its subpackage named `furniture` and a class `Table` which belongs to `furniture`. To import the class `Table`, it is necessary to specify its complete name and package, that is `house.furniture.Table`.

The presence of *package*s involves that two different classes, belonging to two different packages, can have the same name. Therefore, it is not enough to use only the name of the class without consider the package to define the corresponding set *Class*.

The class name is then translated in `ClassNameHash`, where `ClassName` is the original class name, and `Hash` is an hash code generated starting from the *package* name. This is very useful, because in the vast majority of *C* compiler the maximum length of a struct name is fixed. Conversely, *package* names can be very huge, and using the entire *package* name instead of an hash can generate problem with the length of the struct name.

3.2.7 Interfaces

Interfaces are sets of methods defining the behaviour of a class. An interface can be defined as a "contract", and if a class implements the interface, it "signs" the contract, in the sense that the class commits to implement all the methods defined in the interface. As described in the previous sections, a class can extend only a *superclass*. Despite of that, a class can implement none, one or more interfaces.

As mentioned before, interfaces are a set of methods. More precisely, an interface is formed by a set of method definitions (implicitly non-static and with a public visibility), a set of static attributes and a set of static methods. The set of static attributes and the set of static methods are translated in *C* as the static attributes and static methods of standard classes. For each interface, a *C struct* δ is defined. δ has the same layout of γ for the classes, in the sense that the first element points to a synthetic methods table (shared between all the instances of every interface) and the second element encapsulates the type of the instance in a *C* string. The presence of the synthetic methods table is necessary because all interfaces implicitly extend the `Object` class. Therefore, it is also possible to call the methods defined in the `Object` class from an interface. The third element of δ is a pointer to the instance of a class that implements the interface. From the 4^{th} element, δ contains *function pointer*s to the implementation of the methods defined in the interface.

For every interface implemented by a class, a pointer to the respective δ is added to the φ of the class. For each instance of the class, an unique δ is created, and the elements of δ point to the respective implementation of the methods defined in the class. Let us assume the existence of an interface `Inter`, the existence of a class `SomeClass` that implements `Inter` which define a method `method`. The code in listing 3.13 shows an example of interface and a method call through the interface. The translated *C* code is presented in listing 3.14. When the object `obj` is assigned to `Int`, the compiler adds the access to the `MyInter` field, that corresponds to the δ instance of `obj`. Then, the method call is translated in this way: the offset of the address for method `method0` is calculated from the `Inter` *C struct*,

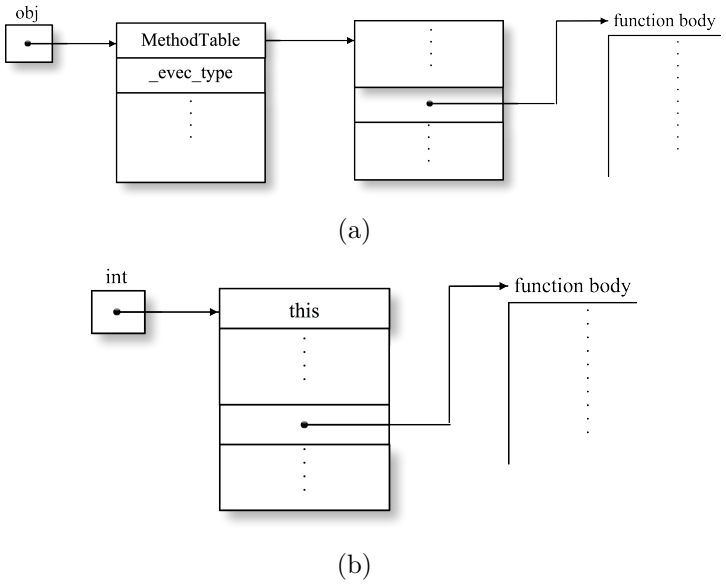


Figure 3.1: The indirect method call for classes and interfaces.

and the function is called through the respective *function pointer*.

In the scheme presented in figure 3.1, the difference between a class method call and an interface method call is compared. The class method call has one more indirection level than the interface method call, hence the interface method call is faster than the class method call.

Listing 3.13: Example of a method call through an interface.

```
...
SomeClass obj = new SomeClass();
Inter Int = obj;
Int.method();
...
```

Listing 3.14: Translated code for the listring 3.13.

```
...
SomeClass *tmp = SuperMalloc(sizeof(SomeClass));
SomeClass *obj = MethodSomeClassSomeClass0(_evec_env
   ,tmp);
Inter *Int = obj->MyInter;
Int->method0(_evec_env,Int->this);
...
```

Inheritance in Interface

Unlike classes, an interface can extend none, one or more interfaces. If a class `SomeClass` implements the interface `Inter` and the interface `Inter` extends the interface `Father`, than `SomeClass` must implement not only the methods defined in `Inter`, but also the methods defined in `Father`. Since an interface can extend more than one interface, the use of a strict layout for δ, similar to the one used for the γ table of classes, is not enough. In fact, when an interface extends two or more interfaces, it is not possible to define an unambiguous order of the elements for δ. On the contrary, since a class can extend only another class, the order of the elements in the γ table is unique, allowing trivial implementation of inheritance.

To solve this problem, when an interface extends two interfaces, a list of methods definition is created, merging first the methods defined in the first extended interface, than the methods defined in the second extended interface, and finally adding the methods defined in the new interface. δ of the new interface will have an entry for any method in the list of methods. Moreover, for every interface directly extended by the interface, an element is added to δ. For example, if the interface `A` extends the interface `B`, than a pointer to an instance of the δ belonging `B` is added to the δ belonging the interface `A`.

Let us assume the definition of three interfaces, `A`, `B` and `C`, and that `C` extends `A` and `B`, and the definition of a class `SomeClass` that implements `C`. In listing 3.15 an example of the assignment of an object of class `SomeClass` to interfaces of type `A` and `B` is presented. Thanks to the presence in δ belonging to interface `C` of elements that point to instance of δ `A` and `B`, the code can be trivially translated, as presented in listing 3.16.

Listing 3.15: Example of assignament to interface.

```
...
SomeClass obj = new SomeClass();
A some = obj;
B some1 = obj;
...
```

Listing 3.16: Translation of the code in listing 3.15.

```
...
SomeClass *tmp = SuperMalloc(sizeof(SomeClass));
SomeClass *obj = MethodSomeClassSomeClass(_evec_env,
   tmp);
A *some = obj->MyC->MyA;
B *some1 = obj->MyC->MyB;
...
```

3.2.8 Generics

The *EveC* programming language allows to define a class or an interface with one or more arguments, that are called *place-holder*. A class (or interface) with arguments is called a *Generic* class (interface). If a class or an interface is generic, it is possible to use a *place-holder* instead of a standard *EveC* non-primitive type. In listing 3.17 an example of a generic class is presented. The class `Container` defines a *place-holder* named `V`. The attribute `element` has not a specified type; instead it has a generic type, with the only assumption that the type is at least `Object` or one of `Object` subclass. When a variable of type `Container` is defined or when an instance of `Container` is created, it is necessary to explicitly define the type of the *place-holder*. In the listing 3.18 an example of variable declaration named `obj` of type `Container<String>` and an instantiation of `Container<String>` is presented. Since the type of `obj` is a `Container<String>`, the *place-holder* `V` is replaced with the type `String`, and thus all occurrences of `V` are replaced with `String`. The attribute `element` for `obj` has type `String`, and the method `put` receives only an argument of type `String`. This feature is tremendously useful in classes and interfaces that implement data structure like lists, queues, trees.

Listing 3.17: Example of a generic class.

```
class Container<V> {
  V element;
  Container<V> next;
  ...
  void put(V element) {
    ...
  }
}
```

Listing 3.18: Example of instance of generic class.

```
...
Container<String> obj;
obj = new Container<String>();
String t = obj.element;
obj.put(t);
...
```

Inheritance and implementations with generics

A generic class can be extended by a generic or a non-generic class. In any case, the subclass must choose the *place-holder*s for the superclass, that can be both *place-holder* defined in the subclass or defined type. For example, a class `SubClass<T>` can extend `Contaier` both as `Container<String>` or `Container<T>`.

In exactly the same way, a generic interface can be extended by a generic or a non-generic interface.

Finally, a generic interface can be implemented by a generic or a non-generic class. The *place-holder*s of the generic interface can be replaced by *place-holder*s of the generic-class or by defined type.

Implementation of generics in Java

The *Java* programming language added the support for generics in 2004 as part of *J2SE 5.0*. Generics are implemented using *type erasure*. For example, if an attribute has a *place-holder* instead of a type, the *place-holder* is replaced by the `Object` type. Furthermore, at runtime, the type `Container<String>` is simply replaced by the type `Container`. The code in listing 3.18, after *type erasure*, is transformed in the code in listing 3.19. At *compile-time*, the *Java Compiler* still verifies that, in the example of listing 3.18, the attribute `element` of `obj` is assigned to a variable of type `String`. The *type erasure* used for generics implementation has both positive and negative aspects. In fact, the *ByteCode* instruction set and the object layout used by *JVM*s did not need to be changed. On the other hand, at *run-time*, it is not possible to distinguish a `LinkedList<String>` from a `LinkedList<Integer>`. The code in listing 3.20 exploits this problem. In

this example, the *Java compiler* will raise only a warning message[1], anyway compiling the snipped. If executed, this code will not raise any error or exception, and the `LinkedList t` and `i` contain both objects of type `String` and `Integer`, instead of only `String` and `Integer` respectively.

Listing 3.19: Example of instance of generic class after *type erasure*.

```
...
Container obj;
obj = new Container();
String t = (String)obj.element;
obj.put(t);
...
```

Listing 3.20: The problem with *type erasure* in generics.

```
...
LinkedList<String> t = new LinkedList<String>();
t.add("Hello, world");
Object o = t;
LinkedList<Integer> i = (LinkedList<Integer>)o;
i.add(new Integer(12));
...
```

Implementation of generics in EveC

Unlike *Java*, *EveC* does not use *type erasure* to implement generics. When a generic class is translated into *C* code, all *place-holder*s are replaced by pointer to `Object`. This can be done because all classes and all interfaces implicitly extend the `Object` class. Despite that, when an instance of a generic class is created, the type of *place-holder*s used to instantiate are recorded and used to form the *complete type*, available also at *run-time*. Since two instances of the same generic class may have different *complete types*, it is not possible to use the γ table to identify the original type. Therefore, the type information is stored into the φ struct, so that if an instance of `LinkedList<String>` is created, then it is possible to know at *run-time* that the instance is of type `LinkedList<String>`, and not only `LinkedList`.

[1]Using a very simple dataflow analysis, the *Java compiler* could realize that a `LinkedList<String>` is assigned to a `LinkedList<Integer>`, because it still has the complete knowledge of objects type.

Thanks to this arrangement, the code in listing 3.20 will raise a `ClassCastException` when the cast to `LinkedList<Integer>` for an object of type `LinkedList<String>` is executed, unlike *Java*, which would continue the program execution without reporting any error or exception.

3.2.9 Exception System

Exceptions are the customary way in *EveC* to indicate to a calling method that an abnormal condition has occurred. When a method encounters an abnormal condition (an "exception condition" that can not handle itself), it may throw an exception. Exceptions are caught by handlers positioned along the thread's method invocation stack. If the calling method is not prepared to catch the exception, it throws the exception up to its calling method, and so on. If an exception is thrown and is not caught by any method along the method invocation stack, the thread in which the exception is thrown will expire, signalling that an exception has occurred and that no method caught has it.

In *EveC*, exceptions are objects. It is possible to throw only instances of classes that descend from the `Throwable` class. The `Throwable` class has two direct subclasses, `Exception` and `Error`. Objects of type `Exception` or its subclasses are thrown to signal abnormal conditions that can often be handled by some catcher. Objects of type `Error` are usually thrown for more serious problems. The code in listing 3.21 shows an example of a method that throws an exception. In the example, in the case of abnormal condition (i.e., the file does not exist), the method throws an `IOException`, creating a new instance of the class `IOException`, which extends the `Exception` class. Since the method does not catch the exception, it must signal that an `IOException` can be thrown during its execution, adding the `throwing` statement.

Listing 3.21: Example of a method throwing an exception.

```
static void openFile(File f) throwing IOException {
  if (!f.exist()) {
    throw new IOException("File not found");
  }
  ...
}
```

In listing 3.22 an example of a call to the method described in listing 3.21, using the `try-catch` statement, is presented. To catch an exception, the code that can throw the exception must be surrounded by the `try` block; if, during the execution an exception raises, the execution path is transferred in

the `catch` block. The thrown exception is examined and if the `catch` block can catch the exception, the code inside the `catch` block is executed. It is possible to define a series of `catch` blocks, that can catch different exceptions. In addition, it is important to notice that the `catch` blocks are examined in order and thus, if for instances the first block catches `Throwable` exception, all the other blocks will never catch any exception, because all exceptions must be subclasses of the `Throwable` exception. After the series of `catch` blocks, it is possible to define a `finally` block. The `finally` block will be executed after the `try` block if no exception is thrown, or at the end of the execution of a `catch` block. If none of the defined `catch` blocks can handle the exception, the `finally` block is executed before the exception is thrown up in the calling method.

Listing 3.22: Example of the `try-catch` statement.

```
...
try {
  openFile(file);
} catch (IOException e) {
  ...
}
finally {
  ...
}
```

In listings 3.23 and 3.24 the translated *C* code for the example of listing 3.21 and 3.22 is presented. The throwing statement is simply translated into the calling of a macro function defined in the *Runtime system*. The `try-catch` translated is more complex, but it uses, as the `throw` statement, some macro functions defined in the *Runtime System*. The macro function in *Unix* systems are implemented with the *C* functions `setjmp/longjmp`, that provide "non-local jumps" facilities.

Listing 3.23: Tranlsation of the code in listing 3.21.

```
void openFile(__evec_env* _evec_env, File *t) {
  if (!f->MethodTable->exist(_evec_env,f)) {
    IOException *tmp = SuperMalloc(sizeof(
       IOException));
    Throwable *t = MethodIOExceptionIOexception(
       _evec_env,tmp,"...");
    THROW(t,"LineNumber,ClassName,...");
  }
  ...
}
```

Listing 3.24: Tranlsation of the code in listing 3.22.

```
...
try {
  openFile(file);
} catch (IOException e) {
  ...
}
finally {
  ...
}
```

3.2.10 Synchronization

EveC defines two synchronization constructs. The first is that a method can be defined as synchronized. Let us assume the definition of a class named `Synch`, which defines a method called `produce` that is `synchronized`, and the existence of two threads, respectively *Thread1* and *Thread2*, that share an instance of `Synch`, called `obj`. The compiler will ensure that the method `produce` of the object `obj` can not be executed concurrently by *Thread1* and *Thread2*. Also a static method can be defined as `synchronized`. In this case, the compiler will ensure that the method is executed by only a thread per time.

To implement these features, every object has an additional field in the φ struct, of type `_evec_ThreadMonitor`. `_evec_ThreadMonitor` is a *C struct*, defined in the *Runtime System*, that depends on the operating

system and on the architecture. It is an implementation of the monitor construct. Furthermore, the *Runtime System* defines also two macros, `_evec_StartSynchBlock` and `_evec_EndSynchBlock`, that implement the entry and the exit of the monitor. When a synchronized method is defined, an `_evec_StartSynchBlock` is added as the first instruction of the method. Moreover, before any return, an `_evec_EndSynchBlock` call is added. Both the macro are called using the `_evec_ThreadMonitor` of the called object if the method is non-static. Otherwise, if the method is defined as static, the compiler will add an implicit variable of type `_evec_ThreadMonitor` to the ξ set of the class, and will use it to call the monitor macro.

The second construct is the synchronized block. In listing 3.25 an example is present. The synchronized block is similar to the definition of a synchronized method. Referring to the example in listing 3.25, and assuming the existence of two threads sharing the same object called `object`, than only a thread per time can execute code inside a synchronized block using `object` as lock. This construct is the equivalent of a traditional semaphore and can be used to delimit critical sections. It is implemented using the `_evec_ThreadMonitor` field added to the φ of every object. When the execution path enters the synchronized block, then the macro `_evec_StartSynchBlock` is called. Dually, when the execution of the synchronized block ends, then the macro `_evec_EndSynchBlock` is called.

Listing 3.25: Example of synchronized block.

```
...
synchronized (object) {
  ...
}
```

3.2.11 Arrays, matrices and volumes

EveC allows to define *arrays, matrices* and *volumes*. *Matrices* and *volumes* are respectively two and three dimensional arrays. As for any other programming language, the type of elements for *arrays*, *matrices* and *volumes* must be the same. An example of the use of an *array* is described in listing 3.26. In this example, an *array* of `int` with `k` elements is instantiated and it is initialized. Every *array* has an implicitly attribute, `size`, that contains the number of elements that forms the *array*. Therefore, it is possible to know at *run-time* the length of the *array*. Every access to an element of any *array* is safe, in the sense that before accessing the element, the offset that corre-

sponds to the element is verified to be between 0 and the length of the *array* - 1. If the offset is outside the range, an `ArrayIndexOutOfBoundsException` is thrown. The code in listing 3.27 shows an example of a *matrix* and a *volume*. As for *array*, every *matrix* and every *volume* has two and three implicit attributes respectively, named `sizex`, `sizey` and `sizez`, that contain the number of elements respectively in the first, the second and the third dimension. Also for *matrices* and *volumes* every access to any element is safe, checking if the accessed element is valid or not.

Listing 3.26: Example of an *array*.

```
...
int[] vect = new int[k];
for (int i=0; i<vect.size; i++) {
  vect[i] = i;
}
...
```

Listing 3.27: Example of a *matrix* and a *volume*.

```
...
int[,] matrix = new int[p,w];
for (int i=0; i<matrix.sizex; i++) {
  for (int k=0; k<matrix.sizey; k++)
    matrix[i,k] = i+k;
}
int[,,] volume = new int[p,w,g];
for (int i=0; i<volume.sizex; i++) {
  for (int k=0; k<volume.sizey; k++)
      for (int j=0; j<volume.sizez; j++)
        volume[i,k,j] = i+k+j;
}
...
```

The translations of the example in listings 3.26 and 3.27 are presented respectively in listing 3.28 and 3.29. It should be noted that for every access to an *array* element, a conditional statement is added by the compiler to check the validity of the accessed element. The conditional statements become respectively two and three when an element of *matrix* and *volume* is accessed.

Listing 3.28: Translation of the code in listing 3.26.

```
...
_safeArrayint vect = _safeArrayint(k);
for (int i=0; i<vect->size; i++) {
  int tmp = i;
  if (tmp>=vect->size || tmp<0) {
    //throw the exception
  }
  vect->data[tmp] = i;
}
...
```

Listing 3.29: Translation of the code in listing 3.27.

```
...
_safeMatrixint matrix = _safeMatrixint(p,w);
for (int i=0; i<matrix->sizex; i++) {
  for (int k=0; k<matrix->sizey; k++) {
    int tmp1 = i;
    if (tmp1>=matrix->sizex || tmp1<0) {
      //throw the exception
    }
    int tmp2 = k;
    if (tmp2>=matrix->sizey || tmp2<0) {
      //throw the exception
    }
    matrix->data[tmp1][tmp2] = i+k;
  }
}
_safeVolumeint volume = _safeVolumeint(p,w,g);
for (int i=0; i<volume->sizex; i++) {
  for (int k=0; k<volume->sizey; k++) {
    for (int j=0; j<volume->sizez; j++) {
      int tmp1 = i;
      if (tmp1>=volume->sizex || tmp1<0) {
        //throw the exception
      }
      int tmp2 = k;
      if (tmp2>=volume->sizey || tmp2<0) {
        //throw the exception
      }
      int tmp3 = j;
      if (tmp3>=volume->sizey || tmp3<0) {
        //throw the exception
      }
      volume->data[tmp1][tmp2][tmp3] = i+k+j;
    }
  }
}
...
```

Arrays, *matrices* and *volumes* implicitly extend the `Object` class. Thus, it is possible to assign to an `Object` variable an *array*, for example an `int[]`.

For this reason, an instance of any *array*, *matrix* or *volume* must have the same layout as φ struct of all the other objects. The compiler generates the necessary code to implement the correct methods of the `Object` class. Hence, the first element of the φ struct of the *array*, *matrix* or *volume* is a pointer to the γ table generated by the compiler, while the second element is the C string containing the type of the instance. The successive element of the φ struct is an attribute, called `data`, that points to a memory region that contains the data of the *array*, *matrix* or *volume*, implemented ad a `C` array, array of arrays, and array of arrays of arrays respectively for *array*, *matrix* and *volume*. Finally, the compiler adds the implicit attribute `size` for *array*, `sizex` and `sizey` for *matrix* and *volume*, and `sizez` for *volume*.

Of course, the compiler generates the necessary data structures only for the used *arrays*, *matrices* and *volumes*. For example, if a `int[]` object is created, the compiler will generate the data structures for it, but it does not generate the data structures for `long[]`, if it is never declared or used.

EveC allows to assign a row of a *matrix* to an *array*. In listing 3.30 an example is reported. The compiler will add the necessary code to make sure that the new `int[]` shares the elements of the matrix. The translation of this example is reported in listing 3.31.

Listing 3.30: Assignament of a *Matrix* row to an *Array*.

```
...
int[,] matrix = new int[w,i];
int[] vect = matrix[k];
...
```

Listing 3.31: Translation of the code in 3.30.

```
...
_safeMatrixint* matrix = new _safeMatrix(w,i);
int tmp = k;
if (k>=matrix->sizex || k<0) {
  //throw an exception
}
_safeArrayint *vect = SuperMalloc(sizeof(
   _safeArrayint));
vect->data = matrix->data[tmp];
vect->size = matrix->sizex;
...
```

3.2.12 NullPointerException and Stack Trace

Since the *EveC* programming language is defined as a safe language, every access to any object must be safe. An object can be referenced by a local variable, that is implemented as a standard *C* pointer to the φ struct. If a local variable is not initialized, it points to the address 0, also called `NULL`. If an access to an object though a non-initialized local variable is performed, a `lang.NullPointerException` must be thrown. To throw correctly the `lang.NullPointerException` without encountering in performance penalty, in the actual implementation the *posix* signals are used, which allow to define a particular *C* function, called a *signal handler*, that is executed when particular events occur. More precisely, when the `NULL` address is accessed, a `SIGSEGV` signal is raised, and a custom signal handler is executed. Then, the custom signal handler throws the `lang.NullPointerException` and the execution of the application continue. If the correct exception catcher is defined, than the exception can be catch. Otherwise, the application will abort. This implementation for the `lang.NullPointerException` is extremely fast because if no memory access to the `NULL` address is executed, there is no performance penalty. Otherwise, the behaviour of signal is defined in the *posix* standard, and so it is necessary that the target machine defines a *posix* environment to allow a correct execution of *EveC* program.

Another important feature of the *EveC* programming language is the *Stack Trace*. The *Stack Trace* allows to know at any moment of execution the entire sequence of method that has been called. Any method records the method from which it is called in a stack data structure. The *Stack Trace* is implemented in an straight manner. Since any method defined in *EveC* is translated into a *C* function, the first instruction in the *C* function is a push of an element to a stack of string that is passed to any method called. The element is a string that contains the *EveC* name of the method and the line number and file in which it is defined. The last instruction before a return statement is, in contrary, a pop instruction to the stack. This methods has some computational costs. The stack is implemented as a preallocated array; the push instruction is a store for an element of the array and an integer increment (the index to the top of the stack); the pop instruction is only an integer decrement.

3.2.13 The library system

The *EveC* programming language defines libraries, as described in appendix A. As basis of the *EveC* class library, the *GNU Classpath* [17] is used. The *GNU Classpath* provides an open source implementation of the standard li-

brary of *Java*. A substantial portion of the *GNU Classpath* has been successfully ported to *EveC*, which include all the basis classes like `lang.Object`, `lang.String`, `lang.Number` and its subclasses. A significant part of the *Collections Framework* is also ported. To provide feedback to the user, the `io.PrintStream` class has been ported, but classes for the complete input/output subsystem are not yet ported.

3.2.14 Runtime System

Besides the macro functions used in the synchronized statements, the *Runtime System* defines two other important functions. The first is used while any cast is executed, while the second implements the `instanceof` statement. Moreover, the *Runtime System* is responsible for managing all the type system at *run-time*.

To implement the *Runtime System*, the *SQLite* database [18] is exploited. This is used as integrated database in a lot of commercial level products. The intent is not to store information between two runs of a program, but to have a database to record data that do not need to survive between different runs of the program. Thus, the `INMEMORY` options is used to store all the data of the database on the main memory and not in a file as usual, achieving also better performances.

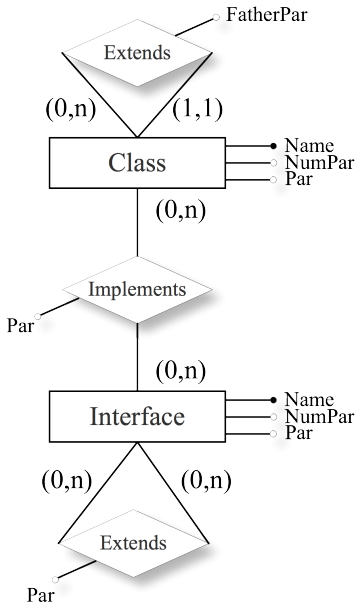


Figure 3.2: The entity-relationship diagram of the database used for the *Runtime System*.

The database is used to store all information about the type hierarchy. For any class, information about the class name, number of its parameters and their name, the name of the father class and the parameter used to extend it are recorded. Also the information about all the interfaces implemented by the class, the argument used to implement them and the offset, calculated from the start of the φ of the class, that correspond to the φ element relative to the implemented interface, are recorded. For any interface, the information about the interface name, number of its parameters and their name, a list of extended interfaces with the parameters used to extend them are stored. In figure 3.2 the entity-relationship diagram of the database is presented.

Thanks to all information stored in the database, it is possibly to easily implement the function that performs the `instanceof` statement and the cast statement.

3.3 Garbage Collection

Garbage collection is the automatic reclamation of computer storage [19]. While in many systems programmers must explicitly reclaim heap memory at some point of the program, e.g. by using a *free* statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find objects[2] that are no longer in use and make the space they occupied available for reuse by the running program. An object is considered *garbage* if it is not reachable by the running program via any path of pointer traversals. An object that can be reached by the running program is *live*.

Garbage collection (GC) is useful for fully-modular programming, in order to avoid the introduction of unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure (unless there are good reasons to coordinate their activities). When an object is explicitly deallocated by a module, this last one must know that there are no other modules interested in that object. In addition, the necessity of explicitly release objects no more needed may introduce programming mistakes such as the maintenance of no more used objects or the release of objects later used by the running program. Failing to reclaim memory at the proper point may lead to slow memory *leaks*, with unclaimed memory gradually increasing until the process terminates or the swap space is exhausted. Releasing memory too soon can lead to very strange behaviour, because an object's space can be reused to store a completely different object while an older pointer still exists. In

[2]The term object refers to any structured data record, as well as the very objects about the object-oriented programming.

this way, the same memory may be interpreted as two different objects simultaneously, with updates to one causing unpredictable mutations of the other.

Because *EveC* has been thought as an high-level programming language, it has been thought that it was useful to equip it with a *garbage collector*, able to eliminate the previously described issues. In the next paragraphs an overview about the *garbage collection* method is depicted, while in the last paragraph of this chapter, the GC used in the *EveC* programming language is described.

3.3.1 Garbage Collection techniques

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such objects are referred to as *garbage*. The basic functioning of a GC consists in two phases:

- Distinguishing *live* object from the *garbage* (*garbage detection*)
- Reclaiming the *garbage* object storage

The *liveness* criterion of a *garbage collector* is based on two terms: *root set* and *reachability*. Each time garbage collection occurs, all globally visible variables of active procedures are considered *live*, as well as the local variables of any active procedure. The *root set* therefore consists of the global variables, the local ones and any register used by the active procedures. Every *heap* object directly reachable from any of these variables can be accessed by the running program; hence it must be preserved. Consequently, any object reachable by a *live* object is also *live*. Hence, the set of *live* objects is simply the set of objects on any directed path of pointers from the *root set*. Any object not reachable by the *root set* is *garbage*, because there is no way that would allow the program to reach that object. Although the basic idea is always the same, there are different types of *garbage collection*; in the next sections a brief overview on the main *garbage collection* techniques is depicted.

Reference Counting

Reference counting is a technique that consists in counting how many reference have an object. In this context, to every object is associated a number which counts from how many objects it is reachable. Each time a reference to an object is created, its count is incremented, while when an existing reference is eliminated, the count is decremented. If the count reaches 0, then

the object is no more reachable by the running program and the memory can be reclaimed. When an object is reclaimed, every object that can be reached by this has its counter decremented, since references from a *garbage* object do not count in determining *liveness*. One advantage of *reference counting* is the incremental nature of most of its operation; *garbage collector* work, in fact, is interleaved with the running program's execution. It can be easily made completely incremental and *real time*, thus performing at most a small amount of work per unit of program execution. There is, however, a great issue involving *reference counting*: it fails to reclaim circular structures. If the pointers in a group of objects create a cycle, the objects' reference counts never reach 0, even if there is no path to the objects from the *root set*. Hence, if an object is not pointed to by any variable or other object, it is clearly *garbage*, but the converse is not always true.

Mark-Sweep Collection

Mark-sweep garbage collectors take the name from the two phases (the same seen in general) they implement: the first consists in distinguish (*mark*) *live* objects and the second in *sweeping* the memory to find the unmarked (*garbage*) objects and reclaim their space.

There are three main issues involving this technique: first, it is difficult to handle objects of different size without fragmenting the available memory; the second problem is that the cost of a collection id proportional to the size of the heap, including both *live* and *garbage* objects; the third involves locality of reference, in fact, since objects are never moved, the *live* objects remain in place after a collection, interspersed with free space.

An increase in this technique has been made to avoid the fragmentation of *mark-sweep* collectors; the first phase, that consists in *marking* the *live* objects, is always the same, while the second phase *compacts* the *live* objects in such a way that they are all in a contiguous part of the memory. This is often by a linear scan through memory, finding a marked object and "sliding" it down so as to be adjacent to the previous one. This leaves one contiguous occupied area at one end of the heap memory, and implicitly a unique contiguous free area at the other end. This method, called *mark-compact*, as said, avoids the fragmentation of memory. In addition, as the *garbage* spaces are simply pressed without disturbing the original order of objects in memory, this technique can reduce locality problems, because the allocation order is usually more similar to subsequent access order than an arbitrary order imposed by a GC.

Copying Garbage Collection

Like *mark-compact*, *copying garbage collection* does not really "collect" *garbage*. Rather, it moves all of the *live* objects into one area and the rest of the heap is then known to be available, because it contains only *garbage*. While compacting collectors use a separate marking phase that traverses the *live* data, *copying* collectors integrate the traversal of the data and the copying process, in such a way that most objects need to be traversed only once.

A very common kind of *copying garbage collector* is the *semispace* collector using the Cheney algorithm for copying traversal. In this scheme, the space devoted to the heap is divided into two contiguous *semispaces*. During program execution, only one of these spaces is currently in use. Memory is allocated linearly upward through the current *semispace*: there is no fragmentation problem. When the running program demands an allocation that will not fit in the unused area of the current *semispace*, the program is stopped and the *copying garbage collector* is called to reclaim space (hence the name *stop-and-copy*). All of the *live* data are copied from the current space to the other one. Once the copy is completed, the second space becomes the current one and program execution is resumed. Thus the roles of the two *semispaces* are reversed each time the *garbage collector* is invoked.

3.3.2 The EveC Garbage Collector

As already said, it was important to integrate *garbage collection* in the *EveC* programming language in order to enhance its high-level structure. However, building up a *garbage collector* from scratch is a difficult and long work, not practicable inside the work of this thesis for reasons of time. For this motive, it has been searched a working *garbage collector* that could fit in this project. The chosen GC is the *Boehm-Demers-Weiser conservative garbage collector*[3] [14]. The reason why this has been chosen relies in its integration simplicity in the *C* language (it can be used also in *C++*): in fact, it is sufficient to substitute the *malloc* statement [20] with a *GC_MALLOC* statement. The rest of the work is completely executed by the GC. It is possible that building up a custom-made *garbage collector* would lead to better performances. However, this GC is a good compromise, as confirmed by the fact that it is used in many important projects, such as "The Mozilla project" [21] (used as leak detector), "The Mono project" [22], an open source implementation of the .NET development and many others. In the next section it is described

[3] A garbage collector is conservative if it has only partial information about the location of pointers. A conservative garbage collector assumes that any bit pattern in memory could be a pointer if, interpreted as a pointer, it would point into an allocated object.

the algorithm used in this *garbage collector*.

The Boehm-Demers-Weiser garbage collector

The garbage collector uses a modified *mark-sweep* algorithm. Conceptually it operates roughly in four phases:

- *Preparation*, in which all the mark bits are cleared, meaning that all objects are potentially reachable.
- *Mark phase*, in which all the reachable objects are marked.
- *Sweep phase*, which returns the unmarked objects to an appropriate free list for reuse.
- *Finalization*, in which unreachable objects which had been registered for finalization are enqueued for finalization outside the collector.

The collector includes its own memory allocator, which obtains memory from the system in a platform-dependent way; e.g. under UNIX it uses either *malloc*, *sbrk* or *mmap*.

An important characteristic to take into account is that the GC creates a number of threads equals to the number of cores of the architectures on which it is running, as it exploits parallelism. For more information about this *garbage collector* see [14].

Chapter 4

Optimizations

4.1 Introduction

As already mentioned, *Java* achieves portability by translating source code into platform-independent *ByteCode* [23]. The execution of a *Java* program starts in the interpreter, which steps the *ByteCode* of a method and executes a code template for each instruction. Only the most frequently called methods, referred to as *hot spots*, are scheduled for *Just-in-time (JIT)* compilation. As most classes used in a method are loaded during interpretation, information about them is already available at the time of *JIT* compilation. This information allows the compiler to generate better optimized machine code. The *Java Hotspot VM* has two alternative *Just-in-time* compilers: the *server* and the *client compiler*. The *server compiler* [24] is a highly optimizing compiler tuned for peak performance at the cost of compilation speed. A low compilation speed is acceptable for long-time running server applications, because compilation impairs performance only during the warm-up phase and can usually be done in background if multiple processors are available. For interactive client programs with graphical user interfaces, however, response time is more important than peak performance. For this purpose, the *client compiler* [5] was designed to achieve a trade-off between the performance of the generated machine code and the compilation speed.

In order to achieve better performances, optimizations can be implemented also in the *EveC* compiler. The intent of the thesis is to produce a *C-to-C* translator that transforms the original *C* code, obtained after the translation phase, into an optimized *C* code. In order to understand which optimization can be performed and which not in *EveC*, it is useful to have a general overview of the optimizations performed by the *JIT compilers*.

4.1.1 Compilers Overview

In the *client compiler* the compilation of a method is split into three phases, allowing more optimizations to be done than in a single pass over the *ByteCode*. All information conveyed among the different phases is stored in intermediate representations of the program. First, a *high-level intermediate representation* (*HIR*) of the compiled method is built via an abstract interpretation of the *ByteCode*. It consists of a control-flow graph (CFG), whose basic blocks are linked lists of instructions (i.e. longest possible sequences of instructions without jumps or jump target in the middle; only the last instruction can be a jump to one or more successor blocks). The *HIR* is in *static single assignment* (SSA) form, which means that for every variable there is just a single point in the program where a value is assigned to it. Both during and after the generation of the *HIR*, several optimizations are performed. These optimizations will be explained later in this section.

The back end of the compiler translates the optimized *HIR* into a *low-level intermediate representation* (*LIR*). The *LIR* is conceptually similar to machine code, but still mostly platform-independent. *LIR* instructions use explicit operands that can be virtual registers, physical registers, memory addresses, stack slots or constants. The *LIR* is more suitable for low-level optimizations than the *HIR*, because all operands requiring a machine register are explicitly visible. The register allocation is performed with a linear scan algorithm [5], simpler and faster than the most common algorithms based on graph coloring, usually employed in this task. After the register allocator has replaced all virtual registers with physical registers or stack slots, each *LIR* instruction can be mapped to a pattern of one or more machine instructions. The *LIR* does not use *SSA* form.

The *server compiler* proceeds through the following traditional phases: parser, machine-independent optimization, *Instruction Selection*, *Global Code Motion* and *Scheduling*, *Register Allocation*, *Peephole* optimization and code generation. The parser phase, divided into two steps, translates, like the previously described *client compiler*, the *ByteCode* into compiler's *high-level intermediate representation* (*HIR*), which is in *SSA* form. This intermediate representation is used throughout optimizations, conversion to machine instructions, scheduling, and register allocation. After parsing has completed, some optimizations, later described in this section, are performed. *Instruction Selection* is the stage of the *Jit* compiler that transforms the *HIR* into *LIR*. Moreover, *Global Code Motion* and *Scheduling* are two optimizations used to improve instruction-level parallelism, as they try to rearrange the order of instructions avoiding pipeline stalls without changing the meaning of the code. Finally, the last phase is *Register Allocation*, in which each of

the virtual register is mapped to a physical one. The used approach is based on graph-coloring algorithms [25], whose execution is computationally more complex than the one used in the *client compiler*. As already said, in order to perform optimizations, these compilers need an *SSA* form code. The *SSA* enables optimizations such as *Constant Propagation*, *Dead Code Elimination* and *Global Value Numbering*, which will be discussed later in this section. The *SSA* transforms the *HIR* in such a way to assign a value to every variable only once.

Listing 4.1: simple assignment instructions

```
int a = 1;
a = 2;
int b = a;
```

Existing variables in the original *HIR* are renamed if necessary, in order to make sure assignments are unique. In listing 4.1, a few instructions with simple assignments are shown, while listing 4.2 represents its *SSA* transformation.

Listing 4.2: *SSA* transformation

```
int a_1 = 1;
a_2 = 2;
int b_1 = a_2;
```

Method Inlining

Thanks to the *Method Inlining* step, the body of the invoked method is inserted into the corresponding location where it has been invoked; with simple words, every call is replaced by the body of that method. This procedure cannot be applied to all situations, since it is not valid for cases in which recursion is exploited. The code shown in listing 4.3 is then transformed, without changing the semantics, into the code shown in listing 4.4.

Listing 4.3: Function call example

```
int counter = 0;
public void main()
{
  increment();
  increment();
}

public void increment()
{
  counter++;
}
```

Listing 4.4: Method inlined

```
int counter = 0;
public void main()
{
  counter++;
  counter++;
}
```

This processing is useful to save space on the stack, as well as to eliminate the call overhead, caused by redirections. This overhead includes the cost of passing arguments, saving and restoring registers, building return linkage information and branching to the procedure body. On the other hand, duplicating large pieces of code many times could pollute the memory.

Devirtualization

For the *Method Inlining* optimization to be effective, the method must be statically bounded, i.e. the compiler must be able to unambiguously determine the actual target method; this action is complicated because most methods in *Java* are defined as virtual. A class hierarchy analysis is used to detect virtual call points where currently only one suitable method exists; this method is then optimistically inlined. Such an algorithm takes the name of *Devirtualization*. If a class that adds another suitable method is loaded later and, therefore, the previous optimistic assumption no longer holds, the method is deoptimized.

Constant Propagation

The *Constant Propagation* procedure takes the variable values, whenever known, and replaces the usage of such variables by their values. As an example, the expression `return id + 1;` shown in listing 4.5 can be easily replaced with `return 4;`.

Listing 4.5: Constant sum example

```
public static final int id = 3;
int incrId()
{
  return id + 1;
}
```

However, for non-final variables, reassignments should be taken into account.

Dead Code Elimination

Dead code elimination is aimed to detect pieces of code that are unreachable. As an example, a possible situation to be taken into account is represented by the paths in the control flow of the code shown in listing 4.6.

Listing 4.6: Dead code example

```
boolean ascending = true;
if(ascending)
  counter++;
else
  counter--;
```

Combining the *Dead Code Elimination* procedure with the *Constant Propagation* process, the compiler can easily decide that only one path can be taken and it also knows which one. Therefore the `else` branch can be removed.

Loop Unrolling

Sometimes statements can be moved out of a loop, just before it, if they are loop invariant. By performing this, they are only executed once instead of needlessly being processed at each stage. Hence, the example code in listing 4.7 is transformed into the one shown in listing 4.8.

Listing 4.7: A loop with invariant statements

```
while(i < n - 1)
{
  j += (n+1) * array[i] * (pi + 2);
  i++;
}
```

Listing 4.8: The loop invariant parts are moved before the loop

```
int max = n - 1;
int tmp = (n + 1) * (pi + 2);
while(i < max)
{
  j += tmp * array[i];
  i++;
}
```

Common Subexpression Elimination

Common Subexpression Elimination (CSE) detects identical expressions that are used different times and replaces them with a common variable, so that the expression is computed only once. An example of this tranformation is shown in listings 4.9 and 4.10.

Listing 4.9: Instructions with a common subexpression

```
int x = v * w + z;
int k = y * v * w;
```

Listing 4.10: Elimination of the common subexpression

```
int tmp = v * w;
int x = tmp + z;
int k = y * tmp;
```

Global Value Numbering

The *Global Value Numbering* (GVN) technique can optimize the use of variables, values and calculations, as well as it can improve the effect of constant

propagation. The performed optimization involves the elimination of redundant code by looking at the values stored in the different variables. While common subexpression evaluation can eliminate some pieces of redundant code, GVN can detect cases that CSE would not have noticed. In the same way CSE optimizes code pieces the GVN would not process, thus creating the need for both. The word "Global" means that value-number mappings hold across basic block boundaries as well.

Global Value Numbering works by assigning a specific value to variables and expressions. When it can be proven that certain expressions are equivalent, the same number is assigned to them, as the transformation from the code shown in listing 4.11 to the code of listing 4.12.

Listing 4.11: Assignments using different variables with the same value

```
int v = 2;
int w = 2;
int x = v + 3;
int y = w + 3;
```

As can be seen, a correct GVN algorithm not only detects that `v` and `w` are the same, but also that `x` and `y` are equivalent.

Listing 4.12: Previous assignments transformed by GVN

```
int v = 2;
int w = v;
int x = v + 3;
int y = x;
```

After the transformation is applied, the *Constant Propagation* and the *Dead Code Elimination* can easily remove the `w` and `y` if they are not reassigned.

The difference between CSE and GVN lies in the fact that CSE matches lexically identical expression while GVN tries to determine an underlying equivalence. An example where CSE would not give an optimal result is shown in listing 4.13: this algorithm would not be able to eliminate the recomputation of `z`.

Listing 4.13: Instructions with a common subexpression

```
int x = v * w;
int y = v;
int z = y * w;
```

Range Check Elimination

As previously said, each time an array is accessed the *JVM* executes an instruction to verify that the access is within the array valid range. For example, if a method accesses `a[i]` in more than one statement, as long as `i` is the same and `a` is not reassigned, only the first access needs a bound check. The *Range Check Elimination* (RCE) procedure eliminates the useless checks, also the ones repeated inside a loop.

4.1.2 Optimizations in EveC

In order to reach better performances, it has been decided to add to the *EveC* compiler some of the previously described optimizations. In order to execute only useful optimizations it is important to remember the phases of the compilation in our programming language. In particular it is essential to remind that the *EveC* language is translated into *C* code and then compiled by a *C* compiler. Hence it is not useful to perform the optimizations that are already executed by the selected *C* compiler. In this work the referred *C* compiler is the so called *GNU Compiler Collection* (*GCC*[1]). *Method Inlining*, *Constant Propagation*, *Dead Code Elimination*, *Loop Unrolling*, *Common Subexpression Elimination*, *Global Value Numbering* are already performed by *GCC*. On the other hand, *Devirtualization* is not implemented because the *C* programming language is not object-oriented; hence there are no classes and there is no need for that. Moreover, as the *C* language is not typesafe and does not perform any array bound check, there is no need for the *Range Check Elimination* too. *EveC* is different in this sense: it is an object-oriented programming language that, as already mentioned, supports class inheritance, and is typesafe; hence these optimizations may be useful and are implemented within the compiler. Furthermore, the implementation of a *Method Inlining* procedure was established, even if it is already performed by *GCC*; this choice permits the enhancing of the other optimizations executed by the *EveC* compiler (see the next sections). Additionally, other two optimizations were implemented and are particularly useful for the *EveC* work context: *Array Explosion* and *Escape Analysis*. The first issue is needed because of the particular structure that *EveC* arrays take after the translation; the second may enhance performances allocating volatile objects on the *stack* instead of allocating them on the *heap*. Both procedures will be explained in the next sections.

[1] We based our research on *GCC* because it is the most widespread. Other *C* compilers may implement different optimizations.

As already said, the *Java* compiler performs the optimizations on an intermediate representation of the code: it would be too complex to perform optimizations on the original code. For this reason, beside the *EveC* compilers, it has been created a module (an ANTLR grammar) able to build up, starting from the original *C* code, an intermediate representation. The used intermediate representation is the *Abstract Syntax Tree* (AST) [26], a structure able to represent the whole original code (see Appendix C for more information about ANTLR). In this way, before the execution of the optimizations, the system translates the original code into the AST; the optimizations are performed exploiting this form. The optimizations are performed starting from the *C* code because they could not be performed in another way. In fact in the *EveC* code all the methods are virtual (as seen in chapter 3), while in the *C* language all the methods are unambiguous; hence *Devirtualization* and *Method Inlining* can reach better improvements if performed on the *C* language. In addition, in the *EveC* code there are no range checks; then the *Range Check Elimination* and the *Array Explosion* cannot be applied. Once the *AST* is built up, the optimizations are performed on it, visiting and moving nodes and subtrees. When all the optimizations have been performed, another module (another ANTLR grammar) recreates the *C* optimized code that will be compiled by the *GCC*.

4.2 Devirtualization

4.2.1 Introduction

In the *EveC* programming language, every method is implicitly defined as *virtual*; thus, every method call that is performed must be executed using a look-up table instead of a direct call to the corresponding function. Since the language allows a class to extend only one father class, the indirect call is not as expensive as in other languages, for instance *C++*. Nevertheless, the indirect method call does not allow further optimizations, such as *Method Inlining*; moreover [27] shows that indirect method call can break hardware mechanisms like *speculative execution* and *prefetching* because they rely on *control flow prediction*, which is harder to analyse in presence of indirect method calls.

For this reason, a lot of work has been done in the past, both for statically typed languages as *C++* and for dynamically typed languages as *Modula-3* [28], *Cecil* [29] and *SELF* [30]. There exists two different types of techniques: *Dynamic* and *Static Analysis*. Since *EveC* is statically compiled, the focus of *Devirtualization* optimization in the present work lays on *Static Analysis* techniques; for the sake of completeness, a brief summary on *Dynamic Analysis* is given too.

4.2.2 Dynamic Analysis

One of the most important techniques of *Dynamic Analysis* is the *Type Feedback*: in [31] such a method is applied to the *SELF* programming language. The basic idea is first to compile the program in an unoptimized way and then to run it in order to gather information about the type of method receivers. This preliminary run of the program is called *profiling* and the retrieved information can be used to produce an optimized code. Listing 4.14 shows an example of source code, while in listing 4.15 the optimized version of the code is depicted. It is worthy to note that in the optimized code an *if*-statement is introduced to check the type of the receiver. This is necessary because from the profile information is not possible not ensured that the type of `a` is always `A`.

Listing 4.14: Example of original Code.

```
void foo(A a) {
  a.method();
}
```

Listing 4.15: Optimized Code.

```
void foo(A a) {
  if (a.type == A) {
    method();
  else {
    a.method();
  }
}
```

In the *EveC* context, the execution time of the introduced *if*-statement is greater than the one introduced by the look-up in the method table. Moreover, the result of the profiling phase is strictly related to the input data provided by the user. For this reason, the effectiveness of this optimization technique cannot be properly set up and the result achieved cannot even be correctly estimated. From this consideration, the *Type Feedback* technique was discarded and the *Static Analysis* is adopted.

4.2.3 Static Analysis

About *Static Analysis* techniques, two big fields of research can be identified. The first one uses information from the *Class Hierarchy* to estimate in a conservative, and thus safe, way a set of possible types for a method call site; this technique is very efficient, and some efforts have been made to improve this analysis.

The second branch exploits pointer-related analysis that has been originally developed for the *C* language. These techniques use flow-insensitive or flow-sensitive analysis to estimate the type of an object, and then employ intra-procedural or inter-procedural analysis to propagate the information. For this reason, the *Point-to Analysis* leads to more precise results than the *Class Hierarchy Analysis*, but the computational cost is tremendously high.

In the following sections, these techniques will be examined in more details.

Class Hierarchy Analysis

Class Hierarchy Analysis, from now on referenced as *CHA*, has been widely studied for the *C++* language [32, 33, 34, 35]. The basic idea is very simple. Firstly, the *Class Hierarchy Graph* is build. An example of this graph is depicted in figure 4.1. Such a graph summarizes the relationships between all the classes of the program. Each class is represented by a node; if a class `B` extends a class `A`, then an edge links node `B` to node `A`.

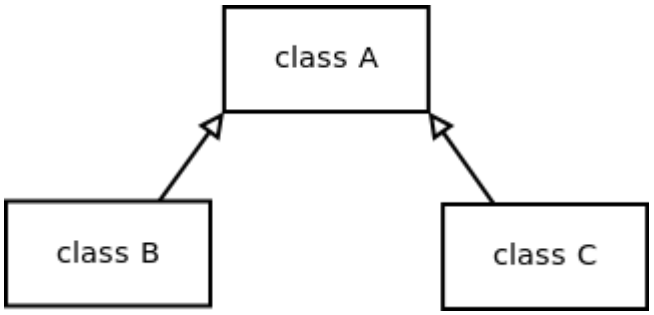


Figure 4.1: Example of Class Hierarchy Graph.

Considering listing 4.14 and figure 4.1, it should be clear that the variable `a` can be an instance of classes `A`, `B` or `C`. In the case that the method `method` is defined only in class `A` and is not overridden in classes `B` and `C`, it is possible to directly call the method, without looking it up in the method table. Otherwise, if the method is defined in class `B` or class `C` too, it is not possible to directly call it. *CHA* resolve this issue and it is characterized by a very low computational cost. The work in [32] shows that *CHA* can resolve an average of 51% of virtual method calls in its nine "real" benchmarks of *C++* programs.

Unfortunately, *CHA* achieves the best performance only when the whole source code is available. In fact, it is not possible to apply *CHA* to build up a standard library: it is not possible to know both if the final user will extend a library class, and which are the overridden methods of the extended class.

Rapid Type Analysis

Rapid Type Analysis [32], from now on referenced as *RTA*, represents an improvement over *CHA*; the basic idea is to track when a new class is instantiated. As it is shown in the code of listing 4.14 and in the *Class Hierarchy Graph* in figure 4.1, if `method` is overridden both in class `B` and `C`, and if no objects of both classes `A` and `C` are instantiated, then it is possible to directly call `method` choosing the overridden one by class `B`. Work in [32] demonstrates that *RTA* can resolve an average of 71% of virtual method calls, thus achieving better performances than *CHA*, without adding much computational cost in the algorithm.

Unfortunately, *RTA* suffers of the same problem described above for *CHA*.

Point-to Analysis

In the literature, a lot of *Point-to* algorithms can be found; specifically, these techniques are developed for the *C* programming language. For example, [19] proposes a *Point-to* algorithm based on *Datalog*.

Another very well-known algorithm is described in [36]; it uses a flow-insensitive and intra-procedural analysis to estimate a conservative set of types for each variable.

The biggest problem of [36] is that the algorithm does not scale well as the size of the program increases. Many efforts have been made in order to achieve better scalability performances; for details refers to [37, 38].

In our opinion, a flow-insensitive approach is not enough precise and, if including also the intra-procedural analysis, the computational cost is too high compared to the benefits.

Listing 4.16: Flow-insensitive point-to analysis fail.

```
1 void foo() {
2   A object = new B();
3   object.SomeMethod();
4   object = new C();
5   object.SomeMethod();
6 }
```

Consider the code in listing 4.16. In line 3 the method call can be done directly, because the variable `object` is trivially of type `B`, though `object` is declared as `A`. Also the method call in line 5 can be done directly, because `object` is trivially of type `C`. However, a flow-insensitive analysis infers that `object` can be either `B` or `C` and so none of the method calls can be resolved. It can be noticed that all the presented analyses fail in resolving the method calls of code in listing 4.16.

4.2.4 Fast and Precise Analysis

Within the present thesis, a new algorithm is proposed, i.e. *Fast and Precise Analysis* (*FPA*), which is able to resolve all the method calls pointed out in listing 4.16 and, at the same time, is cheaper than a complete *Point-To Analysis*. Moreover, it allows to directly call some methods in compiled library code.

A combination of *FPA* and *RTA* have been exploited to resolve as many method calls as possible, keeping a low computational cost.

A flow-sensitive point-to analysis is locally performed for each method by *FPA*, without any inter-procedural or intra-procedural analysis. Thus, *FPA*

can resolve only method calls for locally declared objects inside the method being examined, and it never resolves the following issues:

- Method calls on objects that are arguments of the considered method.
- Method calls on the *this* pointer.
- Method calls on attributes of the *this* pointer.
- Method calls on attributes of any object.

RTA attempts to resolve all the cases in which *FPA* fails, particularly all the cases described above.

FPA Algorithm

The *FPA* algorithm is extremely simple; when a new object is declared inside a method, in addition to the declaration type, the object *real type* is stored too. At first, the *real type* is left blank and it can be modified only in two case:

- When a new instance is created and assigned to the local object.
- When an object is assigned to another object.

The code in listing 4.17 shows an example of the both cases. In line 2, the *real type* of `object` is set to `B`, as a consequence of the *new* statement. In line 3, the *real type* of `other` is set to *B*, as a consequence of the assignment. In line 5, a non-local object is assigned to the local object `object`. Since the *real type* of `arg` is unknown, the *real type* of `object` becomes `null`. Thanks to the *real type* information, *FPA* can resolve the method calls in line 4 and line 7.

Listing 4.17: *FPA* simple example.

```
1 void foo(C arg) {
2   A object = new B(); //realType[object] = B
3   A other = object; //realType[other] = B
4   object.SomeMethod();
5   object = arg; //realType[object] = null
6   object.SomeMethod();
7   other.SomeMethod();
8 }
```

Because of *FPA* flow-sensitiveness, every statement which includes implicitly branch, such as the *if* and *if-then-else* statements and all the loops statements, deserves particular considerations.

Conditional Statement

It is important to distinguish the *if* statement and the *if-then-else* statement. In the first case, when the conditional branch returns, the gathered information must be merged with the information prior the conditional statement. This is necessary since there is no certainty that the conditional branch is executed. For example, in line 6 of code in listing 4.18, the *real type* of `object` becomes `B` or `C`; thus *FPA* cannot resolve the method call in line 7, but a more precise information can be passed to *RTA*. Without *FPA*, the type of `object` would be `A` or any class extending `A`; with *FPA*, the `object` type is more precise, and this can help *RTA* to achieve better performances.

Listing 4.18: *FPA* and conditional statement.

```
1 void foo(C arg) {
2   A object = new B(); //realType[object] = B
3   A other = object; //realType[other] = B
4   if (someConditions) {
5     object = new C(); //realType[object] = C
6   } //realType[object] = B,C
7   object.SomeMethod();
8   other.SomeMethod();
9 }
```

In the second case, the execution path is divided into two different paths, one executed if the *if* condition is verified, one executed otherwise. Both paths must start with the information present before that the conditional statement is executed and the information at the end of both the paths must be merged. In listing 4.19 an example is presented. At line 4, the *real type* of object `object` becomes `C`. On the contrary, in line 6, the *real type* is reverted to `B`, to allow the correct analysis of the second execution path. In line 7, the *real type* of `object` is set to `A` and thus, from line 8, the *real type* for object `object` can be `A` or `C`, but not `B`.

Listing 4.19: *FPA* and conditional statement.

```
1  void foo(C arg) {
2    A object = new B(); //realType[object] = B
3    if (someConditions) {
4      object = new C(); //realType[object] = C
5    }
6    else {  //realTye[object] = B
7      object = new A(); //realType[object] = A
8    } //realType[object] = A,C
9    object.SomeMethod();
10 }
```

Loop Statements

The code inside any loop can be executed none, one or more times, depending on the state of the program. In listing 4.20 an example of loop is presented. Before entering the loop, the *real type* of object `object` is `B`. In line 6, the *real type* of `object` is assigned to `C`. At the end of the loop is not possible to know, at *compile-time*, how many times the loop is executed and thus the type information on `object` must be merged with the information known before the loop. Thus, the *real type* for object `object` from line 8 is the set `B` and `C`. It is important to notice that the method call in line 5 can not be simply resolved. In fact, in the first execution of the loop, the effective type of `object` is `B`, but from the second execution the effective type becomes `C`. *FPA* can infer that the *real type* of `object` in line 5 is `B` or `C`, but not more.

Listing 4.20: *FPA* and loops.

```
1  void foo(C arg) {
2    A object = new B(); //realType[object] = B
3    while (someConditions) {
4      ...
5      object.SomeMethod();
6      object = new C(); //realType[object] = C
7      ...
8    } //realType[object] = B,C
9    object.SomeMethod();
10 }
```

Cast statements

Another important construct to be taken into account is the cast statement. In listing 4.21 an example is reported. In line 2, the *real type* of object `obj` is trivially `B`. In line 3, the object `obj` is casted to a `A` object. The *real type* of object `obj` must be propagated to the *real type* of `other`. At *run-time*, the compiler will add the necessary code to verify the legality of the cast. If the cast fails, an exception raises. Otherwise, the execution continues and the *real type* of `obj` is correctly setted to `B`.

Listing 4.21: *FPA* and the cast statement.

```
1 void foo() {
2   B obj = new B(); //realType[obj] = B
3   A other = (A)obj; //realType[other] = B
4 }
```

Managing Interfaces

Interfaces are managed in the same manner as classes in the *FPA* algorithm. Hence, an object that is defined as an interface has the *real type* information as any other class object. The difference between the *real type* of an interface object and the *real type* of a class object is that the *real type* of the interface object is always different to the interface type.

4.2.5 Implementation

As introduced before, a combination of the presented *FPA* and *RTA* algorithms is implemented. The *FPA* algorithm is executed during the translation phase of the compilation process. This is necessary because some type information are not available at level of *C* code. Moreover, during the translation phase, the location of all method calls are recorded in such a way that it is possible to easily find the translated *C* function calls equivalent to each method call. In chapter 3 the way in which an *EveC* method call is translated into a *C* function call is defined. An example of a method call and its translation into *C* is presented in the example of listings 4.22 and 4.23 respectively.

Listing 4.22: Example of method call in *EveC*.

```
1 ...
2 SomeClass obj;
3 ...
4 obj.foo();
5 ...
```

Listing 4.23: The example in Listing 4.22 translated into *C*.

```
1 ...
2 SomeClass *obj;
3 ...
4 obj->MethodTable->foo0(_evec_env,obj);
5 ...
```

Listing 4.24: The example in Listing 4.22 translated into *C* with *Devirtualization* enabled.

```
1 ...
2 SomeClass *obj;
3 ...
4 $MethodCall12$(_evec_env,obj);
5 ...
```

When the *Devirtualization* optimization is enabled, every method call is translated in a different way: instead of performing the function call through the `MethodTable`, a place-holder is used. Listing 4.24 presents the *C* translation of the code in listing 4.22 when this optimization is performed. It can be noted that the method call is translated to `$MethodCall12$`, that is a place-holder. The *Devirtualization* optimization tries to transform as many as possible method calls into direct function call using the information gathered both from the *FPA* algorithm and from the *RTA* algorithm. For example, if *Devirtualization* succeeds, the place-holder `$MethodCall12$` is substituted with the code for the direct function call (in this case, it could be something like `MethodSomeClassfoo0`); otherwise, the place-holder is substituted with the code `obj->MethodTable->foo0`, as present in listing 4.23.

The *RTA* algorithm is executed at the end of the translation phase. First, a *Class Hierarchy Graph* is built using the information gathered during the translation phase. It is strictly similar to the one described in figure 4.1. This graph considers only the classes used in the program, but not the interfaces. Moreover, each node of the *Class Hierarchy Graph* contains a boolean values

to indicate if at least an instance of the respective class is created. To manage the interfaces, a second graph, called the *Interface Hierarchy Graph*, is built. This second graph is similar to the first, but each node, that represents an interface and not a class, contains an additional pointer to a node of the *Class Hierarchy Graph* for each class that implements the interface. In this way, it is possible to simply manage also the method call through interfaces.

Listing 4.25: Pseudo code for the *Devirtualization* optimization.

```
for each MethodCall in Source {
  if (size[realType[MethodCall]]==1) {
    translate into direct function call;
    continue;
  }
  if (size[realType[MethodCall]]>1) {

    if (HasSameImplementation(realType[MethodCall])
       {
      translate into direct function call;
      continue;
    }
  }
  Type = definedType[MethodCall];
  if (isClass[Type]) {
    if (!HasSubclasses(Type)) {
      translate into direct function call;
      continue;
    }
    if (CanBeDirect(MethodCall,Type)) {
      translate into direct function call;
      continue;
    }
  }
  else {
    ImplementedType = FindInterfaceImplementation(
       Type);
    if (CanBeDircet(MethodCall,ImplementedType)) {
      translate into direct function call;
      continue;
    }
  }
  translate into an indirect function call;
}
```

Once the two graphs are built, all the place-holders are substituted with direct or indirect function calls. The general algorithm is described in listing 4.25. First, if *FPA* succeeds, the information contained in the *real type* of the corresponding target object of the method call is used. If the *real type*

contains only a class, than it is possible to translate the method call to a direct function call. Otherwise, if the *real type* contains two or more classes, then it is verified that all of this classes share the same implementation of the called method. Also in this case, the method call can be translated into a direct function call. Let us assume the existence of a *Class Hierarchy Graph* as described in figure 4.2, and consider a method call to the method `toString`. The defined type of the target object is `AbstractList` and *real type* of the target object contains two classes, `ArrayList` and `LinkedList`. If the method `toString` is not overridden (and thus implemented) in both the two classes, then it is possible to infer that the two classes share the same implementation of the `toString` method and thus it is possible to translate the method call to a direct function call. On the other hand, if at least one of the two classes has got its own implementation of `toString` method, then it is not possible to translate the method call into a direct function call.

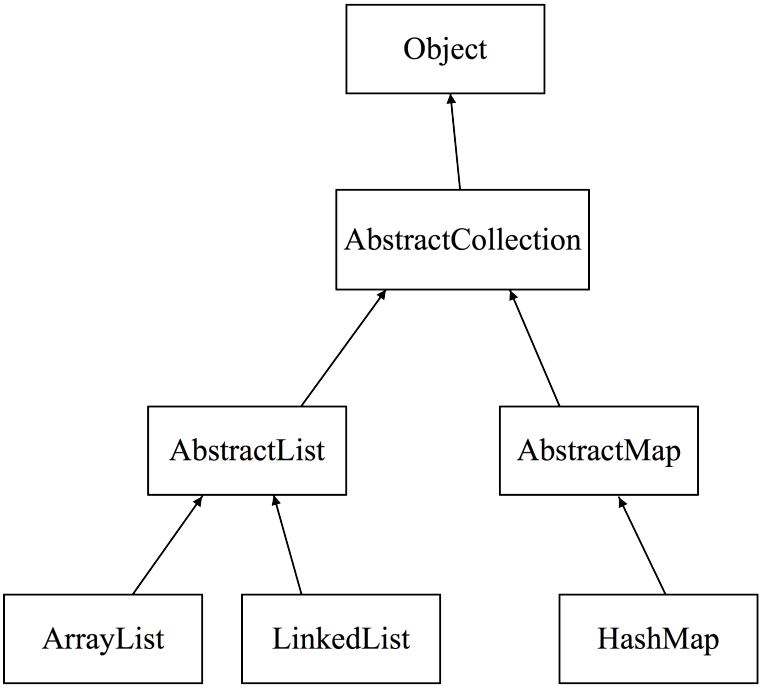


Figure 4.2: A more complex example of Class Hierarchy Graph.

If the *FPA* algorithm fails, the *RTA* algorithm can still be used. The second part of the algorithm described in listing 4.25 explains *RTA*, distinguishing the case in which the target object is declared as a class or as an interface. Let us consider the first case for now. If the class of the target object has no *subclasses*, than the method call can be translated into a direct function call. Referring to the figure 4.2 that represents a *Class Hierarchy*

Graph, any method call targeting an object declared as `ArrayList` can thus be translated into direct function call. On the contrary, if the target object has one or more `subclasses`, a longer analysis is needed, which is performed by the function `CanBeDirect` in the algorithm of listing 4.25. The function `CanBeDirect` is described in detail in listing 4.26: starting from the type definition, it recursively searches if any of its *subclsses* implements the method, pruning the *Class Hierarchy Graph* if a class is never instantiated.

Listing 4.26: Pseudo code for the `CanBeDirect` function.

```
boolean CanBeDirect(MethodCall, Type) {
  for each (class  subclasses[Type]) {
    if (!CanBeDirect(MethodCall,class)) return FALSE
       ;
  }
  if (NeverCreated(Type)) return TRUE;
  if (Type implement MethodCall) return FALSE;
  return TRUE;
}
```

As instance, let us consider the *Class Hierarchy Graph* in figure 4.2 and, as before, the method call `toString`. Let us assume also that some objects of `ArrayList` and `HashMap` are instantiated. If the target object of the method call is defined as `AbstractCollection`, the algorithm infers that it is not possible to translate into a direct call the method, because two different implementations of the method can be called.

Let us now consider the case in which the target object is defined as an interface. In this case, the algorithm firstly retrieves all the possible classes that implements the interfaces. Now, it can be used the same algorithm used for classes to discover if the called method is univocally determinable and thus translated into a direct function call.

4.3 Method Inlining

4.3.1 Introduction

As introduced in the previous section, *Method Inlining* trades code space for time by replacing a procedure call with the called procedure's body. More in details, such a trade-off constists in the reduction of the time overhead generated by the call of a method, at the cost of a simultaneous code growth that can lead to a loss of locality of reference and excessive code growth. For this reason, the effectiveness of an inlining algorithm is determined not only by its ability to recognize inlining opportunities but also by its discretion to exercise those opportunities [39]. Inlining has two important benefits: first of all, it eliminates procedure call overhead, including the cost of passing arguments, saving and restoring registers, building return linkage information and branching to the procedure body. In addition, by merging the procedure body with its calling context, it enables other optimizations to specialize the caller and the callee (i.e. the method called) together [40]. After summarizing the state of the art, this chapter introduces the inlining algorithm used by the *EveC* compiler, the policy adopted, and finally how the algorithm is implemented.

4.3.2 State of the art

Method inlining is a very important optimization in Java-like languages, and, more in general, in object-oriented, high-level languages. The reasons are multiple, but one of the most important is that *Method Inlining* dramatically increases effectiveness of various optimizations, such as *Escape Analysis* or *Loop Unrolling*. However in this kind of languages, mechanisms such as inheritance, method overloading and overriding make inlining algorithms less effective, if they are not helped by advanced techniques of program's control-flow analysis [40]. In fact, in order to be inlined, a method must be statically bounded, i.e. the compiler must be able to unambiguously determine the actual target method, which is complicated because most methods in object-oriented languages are virtual methods. As instance, the *JVM* perform a class hierarchy analysis, *CHA* [35], in order to detect virtual call sites where currently only one suitable method exists [5]. Our approach is different since *Method Inlining* is performed on a lower-level language, *C*, and the analysis made to devirtualize method is not complicated by features like overloading and overriting, which does not exists. Moreover, in the *EveC* compiler, the phisical replacement of a function call with its body does not even need to be implemented, since it is performed directly by the preprocessor part of the *C*

compiler. The next paragraph will explain this assertion, and will illustrate the mechanisms exploited to implement *Method Inlining*.

4.3.3 Method Inlining in EveC

In the *EveC* compiler the algorithm appointed to perform *Method Inlining* is implemented exploiting two characteristics of our system: since this optimization manipulates a representation of *C* code, produced by the *EveC* compiler during the translation phase, there is no need to handle features of an object-oriented language like method overloading and overriding. Moreover, since all the solvable method calls have already been resolved by the previously described *Devirtualization* optimization, *Method Inlining* has only to decide whether or not a *C* function can be inlined. The objectives of this optimization are multiple: it does not only reduce the function call overhead, but it also enhances the effectiveness of other implemented optimizations, like *Bounds Checks Optimization* and *Escape Analysis*.

Listing 4.27: A function called within a loop.

```
void Method()
{
  for(int i = 0; i <= 10; i++)
  {
    A(i);
  }
}

void A(int i)
{
  array[i] = 5;
}
```

Listing 4.28: The resulting code after inlining function A.

```
void Method()
{
  for(int i = 0; i <= 10; i++)
  {
    array[i] = 5;
  }
}
```

Listings 4.27 and 4.28 show a sample code where *Bound Check Elimination* is improved by performing inlining: in the first case, without inlining, the bounds checks $i \geq 0$ and $i \leq length(vett)$ are always performed even if they are redundant; on the contrary, if the `A` function, as in the second case, is inlined, redundant bound checks on the access of `array` can be found and removed.

Listing 4.29: An object allocated and passed as an argument to a function.

```
void Method()
{
  struct Object* o = SuperMalloc(sizeof(Object));
  B(o)
}
```

Method Inlining is even more useful for *Escape Analysis*, because if an object allocated on the heap memory is passed as an argument to a non-inlined function call, as in listing 4.29, the *EveC* compiler cannot know what kind of operations the `B` function will perform on the object `o`; hence the object will not be allocated on the stack because it is not sure whether or not there will be other references to `o` within other functions throughout `B`. However, if `B` is inlined, the *Escape Analysis* algorithm can decide, analyzing the operations executed on it, if `o` is used only in `Method`: in that case, the object is allocated on the method stack. For this very reason an *EveC* own inlining algorithm has been implemented, instead of letting *GCC* perform it: in this way other optimizations can be performed, with a considerable gain of effectiveness, on the inlined code. Detailed results of inlining effectiveness on other optimizations will be presented in chapter 5 .

The algorithm

The algorithm implemented for this optimization exploits a simple feature of *C* programming language: the *macro expansions*. A macro is a piece of text that is expanded by the preprocessor part of a *C* compiler, it is invoked through the *C* keyword `♯define` [41], and its purpose is to define a fragment of code labelled with a name. The *C* preprocessor (*cpp*) is, indeed, a macro processor, that is used automatically by the *C* compiler to transform a program before compilation. The preprocessor performs a series of textual transformations on its input, that happens before all other processing. Conceptually, they happen in a rigid order, and the entire file is run through each transformation before the next one begins. These transformations correspond roughly to the first three "phases of translation" described in the *C*

standard [41].

Listing 4.30: A macro definition.

```
#define PI_PLUS_ONE (3.14+1)
double multiply(double a)
{
  return a*PI_PLUS_ONE;
}
```

Listing 4.30 shows a simple object-like macro definition and a function employing it; when the code is compiled it is firstly passed to the compiler preprocessor, which replaces the macros with their own definitions. The actual code that will be compiled is shown in listing 4.31

Listing 4.31: Resulting code of the preprocessor pass.

```
double multiply(double a)
{
  return a*(3.14+1);
}
```

Moreover, in *C* function-like macros can also be defined, whose usage is similar to a function call. As shown in listing 4.32 the only difference from object-like functions are the brackets, within which there can be arguments, just like a "real" function, but without any type. In listing 4.32 is shown an example of a function-like macro; note that each single row has to end with the "\" character in such a way that the following row is considered as part of the macro.

Listing 4.32: A function-like macro.

```
#define SUM(a,b,c){\
 a + b + c;\
}
```

Therefore, our implementation of *Method Inlining* exploits these characteristics of macro definition: the idea is to transform into macros all the functions that can be inlined, so that the preprocessor of the *C* compiler can perform itself the inlining of this macros. However the original versions of the functions are stored too, for two reasons: firstly because they have to be linked to the γ table of the class (see chapter 3); secondly, there can be spots in the code where a function call cannot be inlined, thus the original version has to be called. In the following, the method to transform a function into a macro

as well as the problems encountered, and the parameters used to determine if a function can be inlined are presented.

From functions to macros

As already said in paragraph 4.1.2, *Method Inlining*, just like all the other optimizations implemented in the *EveC* compiler, works on an abstract representation of *C*, the *Abstract Sintax Tree* (AST), which summarizes the essential information of the *C* instructions. The first change to be made is the name of the macro: it simply consists in adding the suffix "MACRO" to the original name of the function; then, the type of the function return must be removed and replaced with the ♯*define C* keyword. Finally, the types of all arguments passed to the function must be removed too. Listing 4.33 shows an example of all the basic differences between a macro and a function definition.

Listing 4.33: From the function definition to the macro definition.

```
void initToFive(int a)
{
  a = 5;
}
#define initToFiveMACRO(a){\
 a = 5;\
}
```

However, it is not sufficient to edit this nodes of the AST: in fact, unlike functions, macros cannot return any value. Therefore, in order for the macro generated by non-void function to return a value, a simple trick, shown in listing 4.34, is used: a new argument, whose purpose is to store the return value, is added to the macro, and whenever is encountered a return statement, it is replaced by two different instructions: the assignment of the value returned to the just added argument, and a break statement.

Listing 4.34: From non-void function to macro.

```
int sum(int a, int b)
{
  return a+b;
}
#define sumMACRO(a,b,retValue)do{\
 retValue = a + b;\
 break;\
}while(0)
```

A macro definition has a few different characteristics with respect to its original function definition: all macros' bodies are encapsulated in a *do-while* loop that is always performed once and the return statement is replaced by a *break* instruction; moreover if the function has a non-void return, the *break* statement is preceded by an assignment of the return value to a parameter added to the function, in order to store it. The name of the parameter added is unique, so that it is granted that there were no other variables with the same name. In this way, after saving this value, the *break* statement interrupts the loop, giving the same results of the original *return* statement. The new structure definition implies that, before every single call to `sumMACRO`, differently to the call to a function (listing 4.35), a variable used for storing the return has to be defined, as shown in listing 4.36; listing 4.37 shows the code produced after the expansion of the macro performed by the compiler preprocessor. It should be clear that the type of the variable instantiated to store the return value has to be of the same type of the value itself.

Listing 4.35: Call to sum function.

```
...
int a = sum(5,7);
...
```

Listing 4.36: Call to sumMACRO.

```
...
int a;
int tmp;
sumMACRO(5,7,tmp);
a = tmp;
...
```

Listing 4.37: The expansion of the call to the macro in listing 4.36.

```
...
int a;
int tmp;
do{
 tmp = 5 + 7;
 break;
}while(0)
tmp = a;
...
```

However, this kind of structure for a macro can create few problems after the program preprocessing. Consider the sample code shown in listing 4.38: if `addFiveMACRO` was just a function, the printed number would have been 0, because the integer parameter, like all non-pointer variables, is passed to the function by copy; instead, using a macro means that the code that is actually compiled is the one shown in listing 4.39, with the result that the printed number is 5.

Listing 4.38: Arguments passed to a macro.

```
#define addFiveMACRO(int n) do{\
n+=5;\
}while(0)
void main()
{
  int i = 0
  addFiveMACRO(i);
  printf("%i",i);

}
```

Listing 4.39: Compiled code of 4.38.

```
void main()
{
  int i = 0
  do{
  i+=5;
  }while(0)
  printf("%i",i);
}
```

Even though this is not a useful sample code, and `addFiveMACRO` is an unuseful macro, it is clear that the process of transforming functions into macros always has to preserve the program original behaviour. This implies that each variable passed by copy to a function through a call has to be passed by copy also to the corresponding macro call. Listing 4.40 shows the correct form that the call to `addFiveMACRO` should have.

Listing 4.40: Correct form of code in 4.38.

```
void main()
{
  int i = 0
  int tmp1 = i;
  addFiveMACRO(tmp1);
  printf("%i",i);

}
```

Another issue to be fixed is generated by the way the *return* statements within a macro are handled. As already said, each macro body is surrounded by a *do-while* loop always performed once, and the *break* statements make the loop exit when it is necessary. Considering the function and its corresponding macro defined in listing 4.41, whose purpose is to check if a certain number is contained in the array, it should be clear that they have not the same behaviour. In fact, the macro does not act as expected: when the check response is positive, the *break* statement does not exit the *do-while loop*, but the inner one; hence the value returned by this macro is wrong.

Listing 4.41: A particular function definition with its corresponding macro.

```
bool arrayContainsNumber(int * array,int n){
for(int i =0;i<length(array);i++)
{
  if(a[i]==n)
  {
    return true;
  }
}
return false;
}


#define arrayContainsNumberMACRO(array, n, ret) do{\
  for(int i =0;i<length(array);i++)\
  {\
    if(a[i]==n)\
    {\
      ret =true;\
      break;\
    }\
  }\
  ret =false;\
  break;\
}while(0)
```

Such an issue has been fixed not allowing inlining of functions that present *return* statements inside a loop. This is clearly a limit for the optimization, and is to be removed in future, but, for now, it can be considered not relevant. The last problem encountered in the development of *Method Inlining* derives directly from the rough text replacement performed by the *C* compiler preprocessor. Just because it works on text, and not code, the preprocessor is not aware that `this->name` is a different object from `name` (listing 4.42).

Listing 4.42: The translation of a Java constructor.

```
#define PersonMACRO(name, surname,ret) do{\
  this->name = name;\
  this->surname = surname;\
  ret = this;\
  break;\
}while(0)\

void main()
 {
  ...
  struct * Person p = SuperMalloc(sizeof(struct
     Person));
  struct * Person retValue;
   PersonMACRO(n,s,retValue);
   p = retValue;
 }
```

Thus, as it can be seen in listing 4.43, when a macro is called *cpp* replaces it with its definition, roughly substituting the occurrences of the arguments with the parameter's names. This leads to errors at *compile-time*, since the compiler does not recognize `n` as a member of `struct Person`, which it is trying to access through the expression `this->n`.

Listing 4.43: The macro inlined in code.

```
void main()
 {
  ...
  struct * Person p = SuperMalloc(sizeof(struct
     Person));
  struct * Person retValue;
  do{
  this->n = n;
  this->s = s;
  retValue = this;
  break;
}while(0)
p = retValue;

 }
```

In this case the problem has been solved by changing the names of the parameters of each function: during translation from *EveC* to *C*, each parameter name of every function defined in *EveC* is replaced by a unique string. In this way, the produced *C* code presents no more ambiguity in variable names and the inlined macros replacing the original functions act exactly the same way.

The policy

As previously hinted, *Method Inlining* permits to avoid costs introduced by the call to a function, e.g. argument setting, volatile register saving and the call itself; on the other hand, it increases the code size that, whenever too high, can affect performance. The first functions to be inlined should be the ones whose estimated compiled code size is equal or less then the size of the code sequence necessary to call that function. Since invocation and frame allocation costs outweigh the execution costs of the bodies of these methods, and since inlining them is considered completely beneficial without causing any harmful effects in either compilation time or code size expansion, they always have to be inlined [42]. On the contrary, duplicating large pieces of code many times could pollute the memory [43]. For this reason the code size can be considered one of the most important parameters in order to set up an inlining policy, as in the *HotSpot JVM*, whose *client compiler* inlines methods with a size (of the *ByteCode*) less than 35 bytes [5]. Therefore,

the first parameter considered in building our own inlining policy is the code size of the function body. However, since our optimization is not performed directly on the code, but on an abstract representation of it (the AST), the code size can only be estimated by the number of existing nodes in the block sub-tree of a function definition.

Opposed to the static inlining policy performed by the *HotSpot client compiler*, the *server compiler* takes into account some other parameters, such as the number of calls to a certain call site or the frequency of these calls (measured as the ratio between the number of times this call site was executed and the number of times the callee was called) [44] [45]. In the *EveC* compiler the estimation of this parameters is not currently implemented: for now the adopted strategy exploits, besides code size, the number of call sites of a specified function present in the program. The first steps to be done to improve this, still not definitive, policy, include an effort to gain as much information as possible about the calling context (for instance understanding if the call is performed within a loop, or an *if* statement); however, so far, the implemented inlining policy takes into account only an estimation of the code size and the number of total calls to a function in the whole program.

Implementation

After introducing the basic idea and the features exploited by *Method Inlining*, in this paragraph the implementation of the algorithm will be examinated more in details; it consists of two different phases:

- Analysis phase.
- Transformation phase.

The purpose of the analysis phase is to detect which functions can be inlined according to the parameters explained in the previous section. The transformation phase is the one in which *Method Inlining* is actually performed.

The input of the optimization is the AST representation of a single file with a sequence of all the definitions of function generated by the *EveC* compiler, starting from the original *EveC* code written by the developer. Actually it is not a single tree, but there is a tree for every function definition, besides the trees for the import instructions (that are ignored). Figure 4.3 shows the structures of the two possible types of input trees.

The first pass through the tree is made in order to collect some useful information about the function: the number of nodes included in the `BLOCK` sub-tree (in order to estimate the size of the function body), the number of

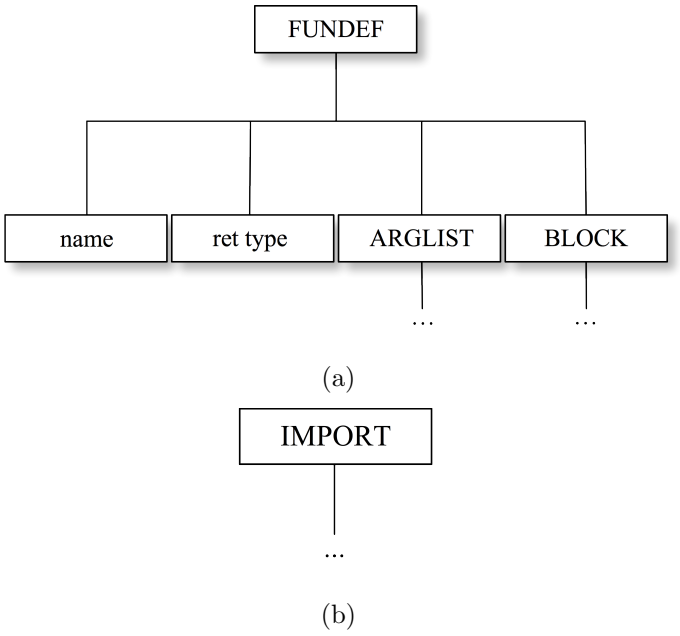


Figure 4.3: Tree structure of the two possible input of *Method Inlining*.

calls to any non-library function and their names (for now, the library functions cannot be inlined, since their source code is not available to programs; for more details see chapter 6.4, the types and the positions of the arguments that are not pointers.

As a result of this first visit to the tree, each function definition is associated with the following metadata:

- An estimation of the function code size.
- A list, accessible through the function name, of the functions called within it.
- A list of types and positions of non-pointer arguments.

Afterwards, the algorithm checks, for each function, whether it is recursive or not, exploiting the meta-data just built: as the pseudo-code illustrated in listing 4.44 explains, for each function the algorithm searches a recursive call in the function call list; if it does not find the recursion, it searches for it, recursively, in each list of each entry of the original list. The algorithm stops either when it finds the recursion, thus labelling the function definition as not inlineable, or when it completes the check of each called function, directly or indirectly, in the body of the definition.

Listing 4.44: Pseudo code of the algorithm that checks recursion.

```
boolean recursionCheck(fundefName)
{
  for each (string s in list(fundefName))
  {
    if(s == fundefName)
      return false;
    else
    {
      for each(t in list(s))
      {
        if(list(fundefName) not contains t)
          add t to list(fundefName)
      }
    }
  }
  return true;
}
```

After checking recursion, a method for bounding inlining only to those functions whose code size is not too much large has to be set up. The quickest way to fulfil this intent is to build a tree that summarizes the structure of the calls of each function, starting from the *main* function (which is not inlineable for definition). Thus, the next step of the inlining algorithm analyzing phase is to build up such a tree, exploiting and exploring all the lists of the function calls, for all the function definitions: this results in a tree in which the root node is the *main* function, the parent of a node is its caller and each child represents a function called in the body of the function embodied by that node.

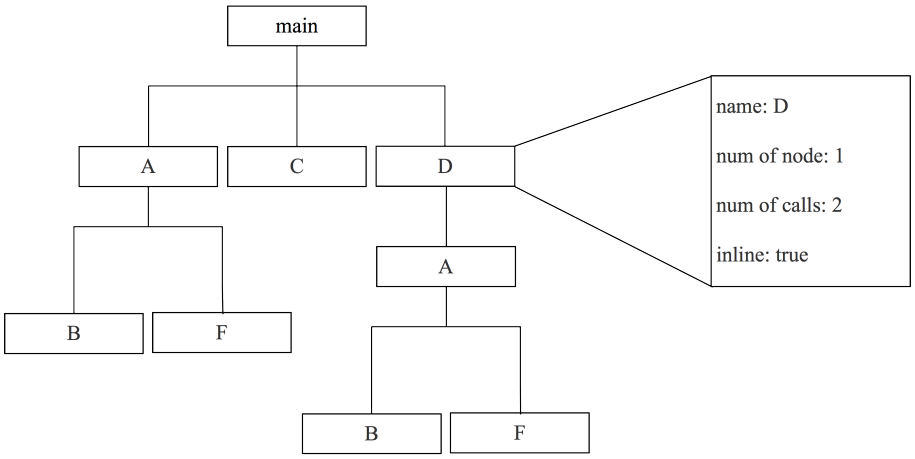


Figure 4.4: The tree built starting from code in listing 4.45.

Listing 4.45: Function calls.

```
void main()
{
  a();
  c();
  d();
  d();
}
void d()
{
  a();
}
void a()
{
  b();
  f(1);
  f(2);
}
```

Figure 4.4 shows the tree built up from the code in listing 4.45 and, in particular, the metadata conveyed by a node, representing a certain function call. Besides the already examined name and number of nodes of the call, each node conveys two more information: how many times the parent called the function and a boolean value that states if that function can be inlined

or not. The former is important in order to correctly calculate the number of nodes that the parent body would consist of if the algorithm decided to perform inlining on the currently examined function; the latter stores the final decision about inlining or not that function. Once the tree is built up, a post-order visit is performed, in order to decide which functions can be inlined. As it is shown in listing 4.46, the algorithm visits at first all the children of the node, from left to right, and, then, the node itself. For each child, if its number of nodes is less or equal a certain threshold, the function corresponding to that child is labelled as inlineable and the number of nodes of its parent is updated, adding the number of nodes of the just inlined function. Moreover, in updating the number of nodes, the number of times that the function is called by its parent is also considered, just because if the function is called twice, then it is inlined twice. However, before all the operations performed on the children, the algorithm checks whether the current function body is too big, without inlining any function called inside it: in this case, the visit in that branch of the tree stops in that node and no functions directly or indirectly called inside it are inlined.

Listing 4.46: Pseudo code of the post-order visit algorithm

```
void postOrderVisit(treeNode)
{
  if(treeNode->numOfNodes >=threshold)
  {
    treeNode->inlineable = false;
  }
  else
  {
    for each(node in treeNode->children)
    {
      postOrderVisit(node);
      if(node->numOfNodes * nodes->numOfCalls <
         threshold)
      {
        treeNode->numOfNodes+= node->numOfNodes *
           nodes->numOfCalls;
      }
    }
  }
}
```

Once the analysis phase is completed, the compiler knows, for each function,

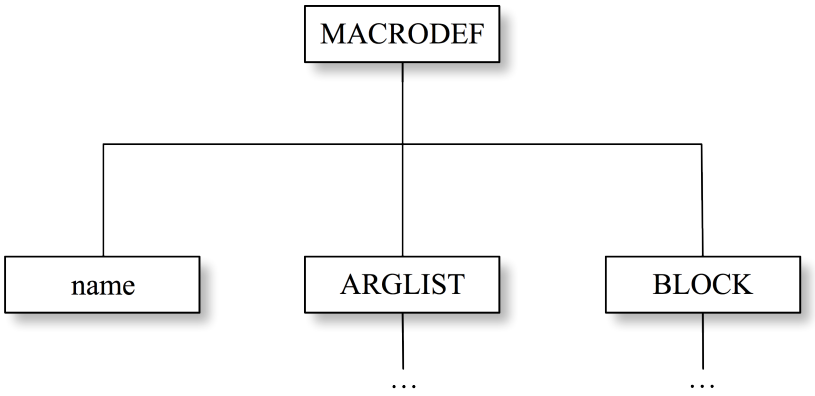


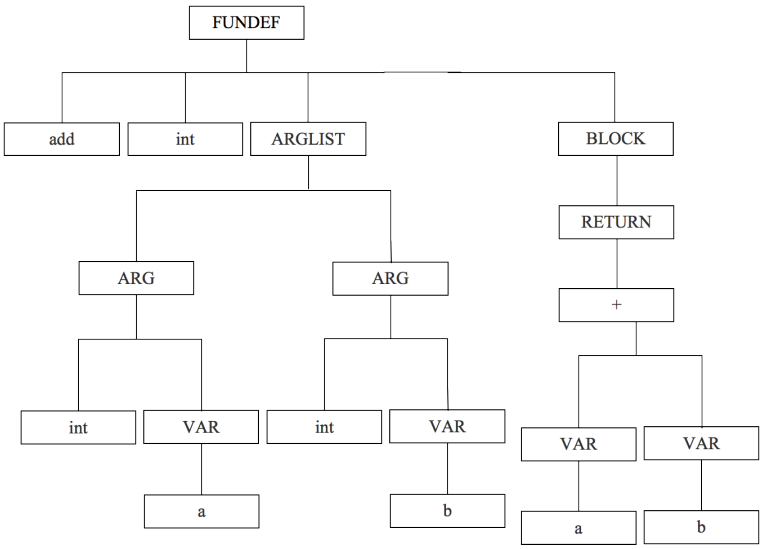
Figure 4.5: Structure of the tree representing a macro definition.

if it will be inlined or not and the procedure for transforming those functions into macro can start. This procedure is split into two stages: the first one has the purpose to transform the definition of the chosen functions into macro definitions, while the second one aims at transforming, whenever possible, the function calls into calls to macros. The tree representation of the macro definition is shown in figure 4.5: it is notable to see that, in this intermediate representation of the macro definitions, some information, like the *do-while* statement and the *define* instruction, are missing, because they are implicitly conveyed by the `MACRODEF` node of the tree.

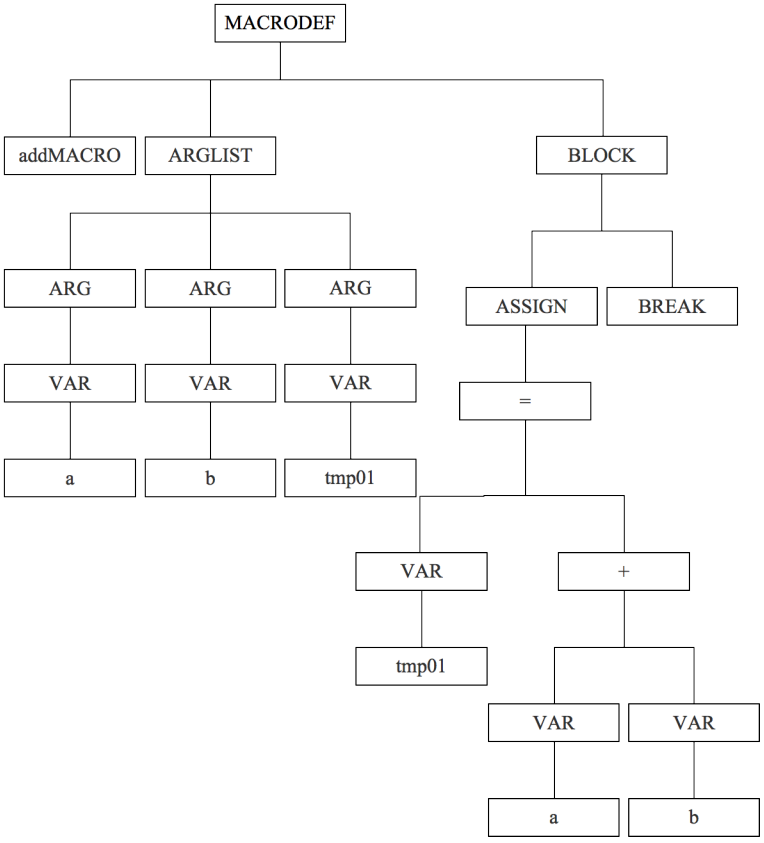
Figure 4.6 shows a concrete example of the trees representing respectively the function (whose code is illustrated in listing 4.47) and the macro definition of a simple algorithm that calculates the sum of two integers. As it can be seen, the performed changes are the ones already explained: the name of the macro is edited, in order to distinguish between a function call and its corresponding macro call; the types of all the arguments are removed and a further argument is added to store the return value (because the function has a non-void return); finally, the return statement is replaced with the break statement and it is preceded by an assignment of the return value to the argument in charge of storing it.

Listing 4.47: A function performing the sum of two integers.

```
int sum(int a, int b)
{
  return a+b;
}
```



(a)



(b)

Figure 4.6: Tree of a macro and a function definitions.

The task of the second stage of the transformation phase is to collect all the calls to inlineable functions and transform them into calls to macros. The changes to be adopted in the tree structure are shown in figure 4.7. Besides adding the definition of the variable used to store the return value, the `FUNCALL` sub-tree is moved outside the instruction sub-tree in which it was found to constitute a newly built instruction, and it is substituted in the original statement with its return value. In this way, the preprocessor of the compiler will inline this macro and the program will behave as the original one.

However, the challenge is that a function call can be found in quite all possible statements, even as an argument of another function call, or as a part of it. Listing 4.48 shows a sample statement in which there are three function calls while in listing 4.49 its translation in order to call the macros instead of the original functions is depicted; in figure 4.8 the tree structure corresponding to the original sample code is represented.

Listing 4.48: A statement with various function calls.

```
...
k = f(g(a),1+h());
```

Listing 4.49: The translation of the code in listing 4.48.

```
int tmp01;
int tmp02;
int tmp03;
gMACRO(a,tmp03);
hMACRO(tmp02);
fMACRO(tmp03,1+tmp02,tmp01);
k = tmp01;
...
```

The algorithm is able to identify all the existing function calls and to divide them in different instructions positioning them in the correct order, as shown in figure 4.9, also adding the definition of all the variables used to store the return value of each function (some variable definitions are omitted in the figure and replaced by the dots for reasons of space).

The only kind of function calls that the optimization algorithm is not able to inline are those that are called through function pointers, as shown in listing 4.50; this is because it does not know the function the pointer is pointing to. However, as explained in section 4.2, this type of situation

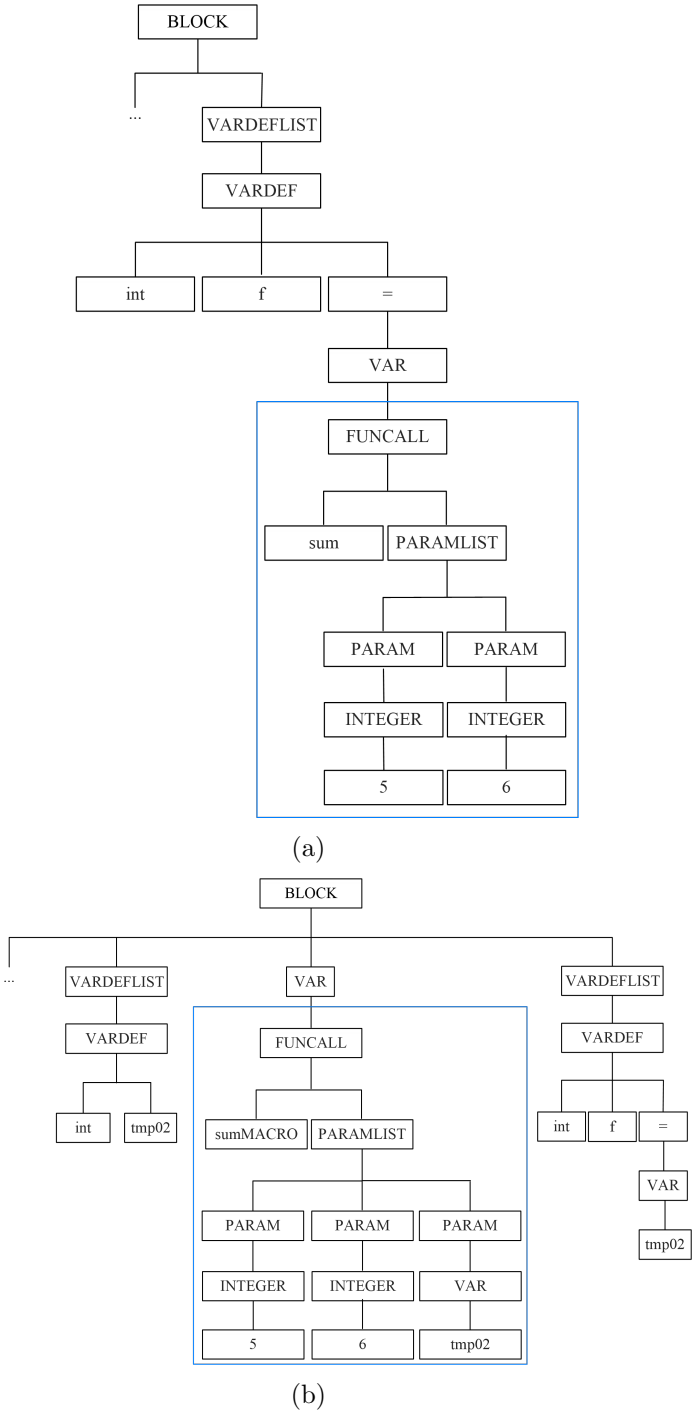


Figure 4.7: Tree structures of a function definition (a) and of a macro definition (b).

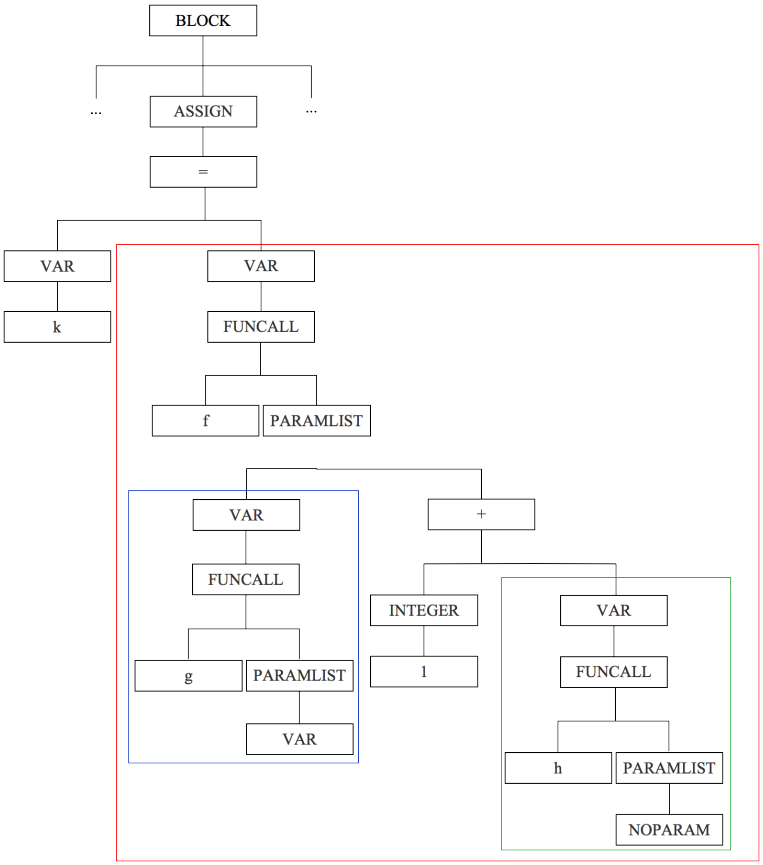


Figure 4.8: Tree structure of the original statement represented in listing 4.48.

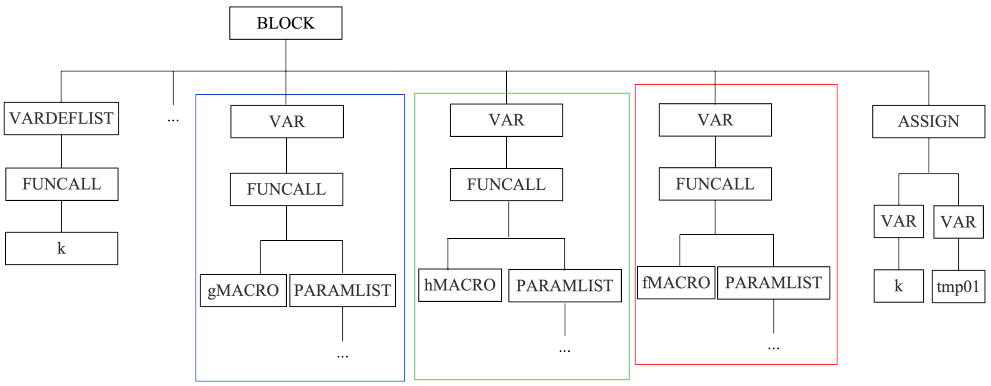


Figure 4.9: Tree structure of the modified code represented in listing 4.49.

is, whenever possible, eliminated by the *Devirtualization* optimization, with good results; such results, as well as the *Method Inlining* ones, are illustrated in chapter 5.

Listing 4.50: A function called through a pointer.

```
...
obj->funcall();
...
```

4.4 Escape Analysis

4.4.1 Introduction

When an object is created through the *new* statement in *EveC*, the compiler translates it in a call to a function defined in the *Runtime System*, *SuperMalloc*, whose aim is to allocate the object on the heap. Therefore, each object in *EveC* is allocated on the heap, and so, it can be deallocated only by the garbage collector. *Escape analysis* is a modern compiler optimization technique for the identification and optimization of objects that are accessible only within a method. Such objects are said *not to escape* the method [5]. When it is verified that a certain object does not *escape* the method in which it was declared, it can be allocated on the method's stack frame. This has two important implications. First of all, stack allocation is inherently cheaper than heap allocation; moreover it reduces garbage collector overhead, since the storage on the stack is automatically reclaimed when the method returns [46].

4.4.2 State of the art

Escape Analysis is a quite new optimization technique and it is introduced, as a default optimization, in the *Java Development Kit 7* (*JDK7*) of the *Java HotSpot Virtual Machine*. The technique implemented in the *JVM* is able to detect objects that not only do not escape from a method, but also from a thread. In fact, if an object does not escape from a thread, the synchronization operations on it can be safely omitted because it will never be locked by another thread [5]. This feature is not yet implemented in *EveC*, just because, for now, it does not completely support multi-threading. More detailed explanations will be illustrated in chapter 6.4. Moreover, according to [5], the *JVM* performs another optimization on stack-allocated objects: the *scalar replacement*. If an object does not escape the creating method, its fields can be replaced by scalar variables (for more detailed explanations see chapter 6.2). Thus, the compiler can eliminate both the allocation of memory on the heap and the initialization of the object. The access to a scalar variable is cheaper than the access to an object field, because it does not require any dereferencing [5]. In the *EveC* compiler this feature is not yet implemented too, but it is certainly one of the first optimizations to be implemented in future. Moreover, according to the available documentation, the *JVM* implementation of the *Escape Analysis* relies on a heavy interprocedural analysis in order to detect objects that do not escape methods. This information is computed during the compilation of a method and stored

together with the machine code. As long as the method is still interpreted, no inter-procedural escape information is available for it. When the *JIT* compiler reaches a call of a method that has not yet been compiled, it does a quick analysis of the its *ByteCode* to determine if a parameter escapes the method [5]. The choice for the implementation of *Escape analysis* in the *EveC* compiler is to avoid inter-procedural analysis and to singularly examine, at compile time, the *C* code of all the function definitions, searching for the allocated objects and verifying if they can be allocated on the stack. This is more conservative than inter-procedural escape analysis, but it is faster to compute because the analysis is much faster since the control flow is ignored [5].

4.4.3 Escape Analysis in EveC

Just like other optimizations presented in the previous sections, *Escape Analysis* works on a representation of the *C* code generated by the *EveC* compiler, the *AST*. The aim of this optimization is to determine if an object does not escape the method in which has been created and, in this case, to allocate it on the method's stack frame. According to the adopted policy, an object escapes a method if one of the following situations take place:

- The object, or one of its members, is returned by the method.
- The object is passed as an argument to a function call.
- The object is assigned to another object that escapes from the method.

The algorithm

If the object is returned by the method there is no way to allocate it on the stack frame because it would stop existing in the very moment the method returns. However, if it is passed to another method, a more complex algorithm could check if the object escapes that function, and, if not, allocate it on the stack of the outer method. The implemented algorithm is conservative in that way: it labels as *escaped* an object passed to another method, independently to what that method will perform on it. This choice has been made after considering that *Method Inlining* is performed before this optimization and the possibility of this situation to happen is reduced. In fact, if that function call is inlined, the object does not escape through the argument of that call any longer. Finally, if the object is assigned to another object, this second object is checked in order to establish whether it escapes the method and, if not, the allocation of the first one can be made on the stack. When

the analysis on a certain object has determined that it does not escape the method, the call to the `SuperMalloc` function can be replaced with the assignment of the object to another variable, as shown in the *C* code samples illustrated in listings 4.51 and 4.52.

Listing 4.51: Object allocated on heap.

```
void method()
{
  ...
  struct Object * obj = SuperMalloc(sizeof(Object));
  ...
}
```

Listing 4.52: Object allocated on stack.

```
void method()
{
  struct Object tmp;
  struct Object * obj = &tmp
  ...
}
```

In such a way, the memory needed by `tmp` is allocated on the stack, and the pointer `obj` is forced to point to the address of `tmp`.

Implementation

The *Escape Analysis* implementation can be conceptually divided into two phases:

- Analysis phase.
- Transformation phase.

The analysis phase consists in searching, in every function's body, for objects allocated on the heap, and in verifying if the object escapes the method. Listing 4.53 shows the pseudo-code of the analyzer algorithm. It takes as input the name of the variable to be analyzed, the sub-tree of the entire block in which it was declared and an index, storing the position in the block of the variable allocation instruction. This because, the algorithm has to search for all the other occurrences of that variable, except the allocation one. The index and the sub-tree of the block are passed as arguments in order to limit the research of the occurrences only in the right spots: if a

variable is declared at a certain point of a scope, it exists only from that point until the closure of the scope itself.

Listing 4.53: Pseudo code of the algorithm that checks if an object escapes a function.

```
boolean isEscaped(String var, int index, Tree block)
   {
  while((stmt = found(var in block))!= null){
    if(stmt is definition of var) {
      ignore all the occurences of var from this
         point until the end of the current block;
        }
    if(stmt is return){
      return true;
    }
    if(var is parameter of funcall){
      return true;
    }
    if(var is assigned to anotherVar){
      if(stmt is an assignment){
        newStmt = searchDefinition(anotherVar);
        if(newStmt == null)
          return true;
        block = block of newStmt;
        index = index of anotherVar definition
           statement in block;
        if(isEscaped(anotherVar, index, block))
          return true;
      }
      else if(stmt is the definition of anotherVar){
        block = block of anotherVar;
        index = index of anotherVar definition
           statement in block;
        if(isEscaped(anotherVar, index, block))
          return true;
      }
    }
  }
  return false;
}
```

Once an occurrence of `var` is found, a series of checks is performed in order to understand what kind of statement involves this occurrence. If it is a definition of the variable, this is a new definition that obscures the first one: in this case the algorithm ignores all the occurrences of the variable since that point, until the end of the scope in which it was found. This because the possible occurrences found within the current scope are not occurrences of the variable under consideration, but of the new one with the same name, and they must not be considered in the analysis. To better understand this, listing 4.54 shows a sample code in which the variable `p` of type `Person` is obscured, in the loop scope, by an `int` variable with the same name. The algorithm is able to understand that the variable `p` passed as an argument to `otherFunction` is actually the `int` variable defined in the inner scope and to label the variable of type `Person` as not escaping the method, if this is the case.

Listing 4.54: A local variable definition.

```
void main()
{
  ...
  struct Person * p = SuperMalloc(sizeof(Person));
  for(int p = 0; p<size(array);p++)
  {
    otherFunction(p);
  }
  ...
}
```

Moreover, when the analyzed object `a` is assigned to another object `b`, the algorithm must distinguish between two cases: the statement could be a definition of `b`, or it could be an assignment. In both cases the algorithm has to check if `b` escapes the method, in order to decide whether or not `a` can be allocated on the stack; however in the first case the algorithm has only to determine the root node of the sub-tree of the scope in which the declaration of `b` was found. On the contrary, in the second case it has to determine the position of the declaration statement of `b`, and the scope in which it was found. This because, if `b` is declared in an outer scope, there can be occurrences of this object out of the scope in which the assignment of `a` was found. An instance of this case is shown in listing 4.55: in this case the variable `a` (and the variable `b` too) cannot be allocated on the stack. To reach this result, the algorithm, once found the assignment of `a` to `b`, has to search for the declaration of `b`; if the algorithm only considered the scope in

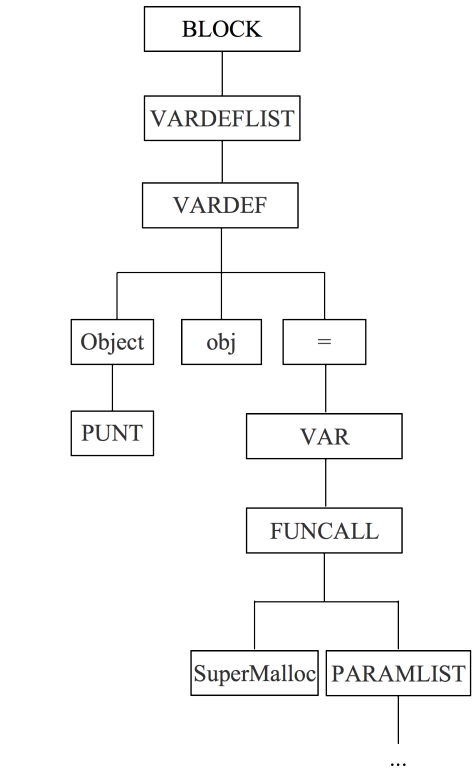
which the assignment statement was found, it would have labelled `a` as not escaping the function.

Listing 4.55: Assignment of a variable declared in an outer scope.

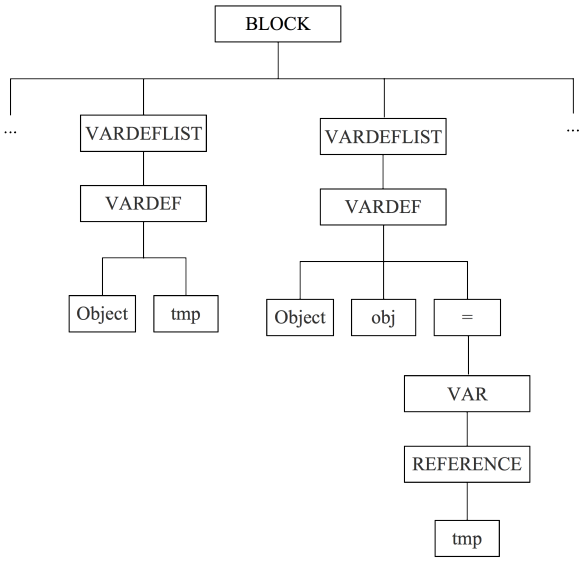
```
void function()
{
  ...
  struct Person * b = SuperMalloc(sizeof(Person));
  {
    struct Person * a = SuperMalloc(sizeof(Person));
    a = b
  }
  ...
  return b;
}
```

The last case to be considered is when the declaration statement of `b` is not found: in this case b could be either a global variable or a parameter of the function; hence, `a` is labelled as escaping the function. Once the analysis stage has verified that a certain object does not escape the creating function, the transformation phase replaces the call to `SuperMalloc` with an assignment of the object to another object allocated on the function stack. As shown in figure 4.10, this happens through a sub-tree replacement, with the addition of the corresponding definition of the new variable. In this way `obj` (see listing 4.52), that is actually a pointer, points to the object `tmp` which is not defined as a pointer object and, allocated on the stack frame of the method.

The *Escape Analysis* results, in terms both of objects no more allocated on heap and of increasing performance are presented in 5 chapter, which will show also the strong influence of the *Method Inlining* on the performance of this optimization technique. Due to this consideration, the performance of *Escape Analysis* can be improved even more by permitting the inlining of functions whose body size exceeds the threshold, if this allows to allocate a good number of objects on the stack.



(a)



(b)

Figure 4.10: Tree structures of the statements that allocate the object respectively on the heap (a) and on the stack (b).

Listing 4.56: Example of accessing on *array* elements.

```
int[] vect = new int[k];
for (int i=0; i<k; i++) {
  vect[i] = i;
}
```

Listing 4.57: Translation of the code in listing 4.56.

```
_safeArrayint vect = _safeArrayint(k);
for (_evec_int i=0; i<k; i++) {
  _evec_int tmp = i;
  if (tmp>=vect->size) {
    //throw exception
  }
  vect->data[tmp] = i;
}
```

4.5 Array Explosion

As described in chapter 3, for every *arrays*, *matrices* and *volumes* of different type used in a program, the compiler generates a synthetic class in which encapsulates as an attribute the memory space allocated respectively for manage *arrays*, *matrices* and *volumes*. Let us consider the usual example, described in listing 4.56, in which all the elements of an *array* are assigned. In listing 4.57 the usual translation in *C* code is presented. Following the example, it can be noted that, for every access to an *array* element, firstly the `data` offset is calculated to get the correct pointer; than the offset to access the correct element of the *array* is computed starting from the address of `data`. In an application that uses intensively *arrays* and that performs a huge number of access to *arrays* or *matrices* the overhead can be significant.

The *Array Explosion* optimization is implemented in order to decrease the overhead introduced by the compiler for any access to an element of any *array*, *matrix* or *volume*. The following considerations are made for the *arrays*, but they are also valid for *matrices* and *volumes*.

The *Array Explosion* optimization tries to reduce the overhead of accessing *arrays* only inside a loop. The elements of the *arrays* are typical accessed using a loop, so this justifies that the optimization is performed only in such case. The optimization consists in accessing the elements of the *array* using a variable that points directly to the memory space allocated for the accessed

Listing 4.58: Translation of the code in listing 4.56 with the *Array Explosion* optimization.

```
_safeArrayint vect = _safeArrayint(k);
_evec_int *tmp1 = vect->data;
for (_evec_int i=0; i<k; i++) {
  _evec_int tmp = i;
  if (tmp>=vect->size) {
    //throw exception
  }
  tmp1[tmp] = i;
}
```

array instead that using the original variable. In listing 4.58 is presented the previous code of listing 4.56 translated using the *Array Explosion* optimization. It can be noted that a new variable, `tmp1`, is declared before the loop and it is assigned to the `data` attribute of `vect`. Thus, `tmp1` points to the memory address in which the data of the *array* is stored. Inside the loop, the access to `vect->data` is translated into `tmp1`, that is cheaper than the unoptimized access presented in 4.57.

4.5.1 Implementation

For the *Array Explosion* optimization is not necessary to rewrite much code as in other optimization and it can be also performed easily during the translation. For this reason, the optimization is implemented during the translation of the *EveC* source code into *C* code, with no additional computational cost. If inside a loop an *array* is accessed, then the new variable is added immediately before the loop and it is assigned to the `data` attribute of the accessed *array*. Until the end of the loop, any other access to that *array* is performed using the new variable. If another *array* is accessed, the same procedure is performed.

4.6 Bound Check Optimization

4.6.1 Definitions

An array A is defined by a lower bound $lo(A)$ and an upper bound $up(A)$. An explicit declaration of the lower bound is allowed even if in *Java* as in *EveC* it is always 0, in the sense that the algorithm for *Bound Check Optimization* described below can be applied to arbitrary lower bounds of arrays. An array element reference is denoted by $A[\sigma]$; for the reference to be valid, σ must be within the valid range of indices for A: $lo(A)$, $lo(A) + 1$, ..., $up(A)$. The for loop shown in listing 4.59 is defined for simplicity as $L(i, l, u, B(i))$, where i is the loop index, l defines the lower bound of the loop, while u represents its upper bound. The iteration space is defined by the range of values acquired by i: l, $l + 1$, ..., u. $B(i)$ is the body of the loop, which typically contains references to the loop index variable i.

Listing 4.59: The loop defined by $L(i, l, u, B(i))$

```
for(int i = l; i <= u; i++)
{
  B(i);
}
```

4.6.2 Introduction

Whenever an array element is accessed, the *Java Virtual Machine* executes a comparison instruction to ensure that the index value is within the valid range; this is made to guarantee a typesafe execution: if a reference is invalid, an exception must be thrown. The same mechanism has been implemented in *EveC*: whenever the compiler finds an array access, a check is added to determine if the access is safe. This clearly reduces execution speed, since there are more instructions to be executed. In particular, when there is an array access inside a loop, the check is executed many times. In this case the check may be redundant and the processor would execute many useless instructions. For example, listing 4.60 shows a simple loop in which checks are totally useless.

Listing 4.60: Rendundant bounds checks

```
for(int i = 0; i < A.length; i++)
{
  if(i >= A.length || i < 0)
    RaiseException();
  A[i] = i;
}
```

Even within loops containing a non-safe array access, there are many redundant checks. In this case we talk about *partial redundancy*. Listing 4.61 shows one of this case: only in the last iteration of the loop the check is effectively needed.

Listing 4.61: Partially redundant bounds checks

```
for(int i = 0; i <= A.length; i++)
{
  if(i >= A.length || i < 0)
    RaiseException();
  A[i] = i;
}
```

A first simple optimization permits to halve the number of necessary bounds checks. In fact a test to verify that an index is within the array range can be implemented with a single comparison. Because $lo(A) = 0$ always, it suffices to test if $i < up(A) + 1$ using unsigned arithmetic. A negative value appears, in unsigned arithmetic, as a very large positive number which is always greater than the largest possible array extension.

In addition to this optimization, redundant bounds check elimination can be applied to increase performances as for execution time. Different approaches to this problem can be applied and are discussed in the next section.

4.6.3 State of the art

During the last years many different algorithms for array bound check elimination have been developed. Many of them exploits the *SSA* form. The algorithms described in [47, 48] build a so called *inequality graph* to represent the numerical distance between variables, in order to find constraints. The *inequality graph* is a weighted directed graph, where nodes represent variables or constants and edges have a weight to represent the difference

from the source to the destination node. The fundamental idea is that each program point of interest has a graph to reflect the constraints among variables at that program point. Once the graphs are built, they are inspected by searching the *shortest path* between variable values in order to find and resolve constraints, determining the unsafe array accesses. In [49] a constraint system is built up, always starting from the *SSA* form, and then resolved to determine which bound check can be avoided. Another approach used in literature is based on code duplication [50]. For each loop found (with some constraints explained later) it creates three loop with the same meaning of the original; in two of these loops there are still bound checks while in the other no checks appear: in this last the array accesses are surely safe. Within the present thesis, this method has been implemented because it needs neither the *SSA* form nor any other transformation ans also because it is simple to be implemented. This method will be discussed in the next sections.

4.6.4 The algorithm

The basic idea of this technique is to partition the iteration space of a loop into *regions*, defined as contiguous subsets of the iteration space. In particular it is useful to find a *safe regions*, where for sure there can be no violations in array references. If an array reference is guaranteed not to generate a violation, explicit tests are unnecessary. Consider the loop *L(i, l, u, B(i))* and let $A[i]$ be the only array reference in the loop body $B(i)$. For the array reference to be valid we must have $lo(A) \leq i \leq up(A)$. We can split the iteration space into three regions $R[1]$, $R[2]$ and $R[3]$ defined as follows:

$$R[1] : (l \leq i \leq u) \wedge (i < lo(A)) \tag{4.1}$$

$$R[2] : (l \leq i \leq u) \wedge (lo(A) \leq i \leq up(A)) \tag{4.2}$$

$$R[3] : (l \leq i \leq u) \wedge (i > up(A)) \tag{4.3}$$

Region $R[1]$ corresponds to those iterations of the loop for which the index i into array A is too small with respect to $lo(A)$. Therefore a check is required before each array reference in this region. No tests are needed in region $R[2]$ since the index i falls within the bounds of A. Finally, a test is required in region $R[3]$, in which the value of the index i is too large with respect to $up(A)$. Using equation 4.2 the lower bound $\mathfrak{L}$ and the upper bound $\mathfrak{U}$ of the safe region can be computed:

$$\mathfrak{L} = max(l, lo(A)) \tag{4.4}$$

$$\mathfrak{U} = min(u, up(A)) \tag{4.5}$$

$$R[2] : i = \mathfrak{L}, ..., \mathfrak{U} \tag{4.6}$$

Similarly equations 4.1 and 4.3 can be exploited to compute the lower and upper bounds of regions $R[1]$ and $R[3]$, respectively:

$$R[1] : i = l, ..., min(u + 1, lo(A)) - 1 \tag{4.7}$$

$$R[3] : i = max(l - 1, up(A)) + 1, ..., u \tag{4.8}$$

Note that $min(u + 1, lo(A)) = \mathfrak{L}$, except when $lo(A) < l$ or $lo(A) > u + 1$. In the former case $R[1]$ is empty; in the latter case $R[2]$ is empty. To handle these cases and the symmetric upper bound cases, we redefine:

$$\mathfrak{L} = min(u + 1, max(l, lo(A))) \tag{4.9}$$

$$\mathfrak{U} = max(l - 1, min(u, up(A))) \tag{4.10}$$

Now the bounds of each region can be expressed just in terms of l, u, $\mathfrak{L}$ and $\mathfrak{U}$:

$$R[1] : i = l, ..., \mathfrak{L} - 1 \tag{4.11}$$

$$R[2] : i = \mathfrak{L}, ..., \mathfrak{U} \tag{4.12}$$

$$R[3] : i = \mathfrak{U} + 1, ..., u \tag{4.13}$$

Figure 4.11 shows the values of $\mathfrak{L}$ and $\mathfrak{U}$ for different relative positions of iteration bounds and array bounds. Figure 4.11(a) has empty regions $R[1]$ and $R[3]$, while $R[2]$ comprises the entire iteration space. Region $R[1]$ is empty in figure 4.11(b), while region $R[3]$ is empty in figure 4.11(c). All three regions are nonempty in figure 4.11(d). Region $R[1]$ is the only nonempty one in figure 4.11(e), while in figure 4.11(f) only region $R[3]$ is nonempty.

We have considered in this section the simplest case with a one-depth nested loop with only one array reference in the body. The next section will discuss a method working on a loop with multiple array references inside.

The exact method

This method performs only the strictly necessary checks to detect array reference violations that occur during the execution of a loop. In this section we deal with references of the form $A[i]$ that allow the computation of safe

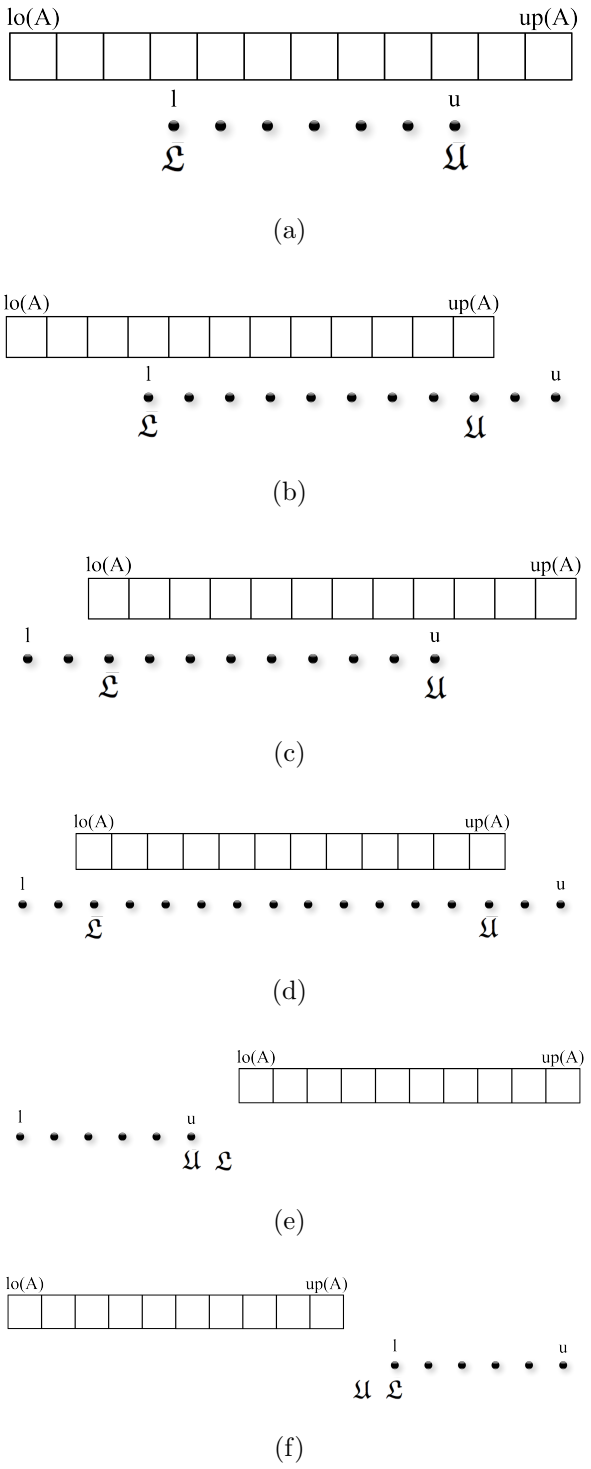


Figure 4.11: Relationship between loop bounds and array bounds.

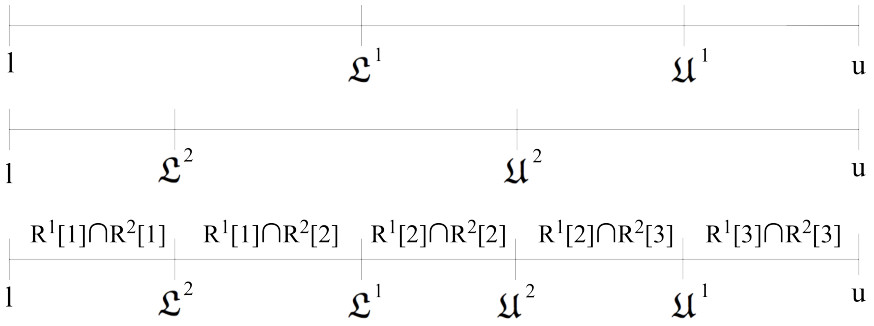


Figure 4.12: Regions generated by the intersection of *simple regions* obtained by two array accesses.

bounds as described in the previous section; further on a discussion on how to handle more complex subscript functions (i.e. the function with which the access is made, e.g. $i+1$) is provided.

Let $L(i, l, u, B(i))$ be a loop on index variable i and body $B(i)$. Let there be ρ array references of the form $A_j[i]$, $j = 1, ..., \rho$ in body $B(i)$. Each array reference partitions the iteration space into three, each possibly empty, regions:

$$R^j[1] : i = l, ..., \mathfrak{L}^j - 1 \tag{4.14}$$

$$R^j[2] : i = \mathfrak{L}^j, ..., \mathfrak{U}^j \tag{4.15}$$

$$R^j[3] : i = \mathfrak{U}^j + 1, ..., u \tag{4.16}$$

These regions, defined by a single array reference, are called *simple regions*. As before, $R^j[1]$ and $R^j[3]$ are unsafe, hence checks are needed, while $R^j[2]$ is safe, hence it requires no checks. For each region $R^j[k]$ we denote its lower bound by $R^j[k].l$ and its upper bound by $R^j[k].u$ where $k = 1, 2, 3$. With this definition $R^j[1].l$ is always l, $R^j[1].u$ is $\mathfrak{L}^j - 1$, $R^j[2].l$ is equals to $\mathfrak{L}^j$, $R^j[2].u$ is $\mathfrak{U}^j$, $R^j[3].l$ is $\mathfrak{U}^j + 1$ and $R^j[3].u$ is u.
Two references $A^1[i]$ and $A^2[i]$ partition the iteration space into 5 regions defined by the 4 points $\mathfrak{L}^1$, $\mathfrak{U}^1$, $\mathfrak{L}^2$ and $\mathfrak{U}^2$, as shown in figure 4.12. Note that some of the resulting regions may be empty. They are the subset of the $3 * 3 = 9$ regions formed by all combinations of intersections of two simple regions, one from each array reference.

At *compile-time*, when the optimization is performed, the numerical order of $\mathfrak{L}^1$, $\mathfrak{U}^1$, $\mathfrak{L}^2$ and $\mathfrak{U}^2$ is not known; it is only known that $\mathfrak{L}^1 \leq \mathfrak{U}^1$, and $\mathfrak{L}^2 \leq \mathfrak{U}^2$ (except for particular cases, such as when the *safe region* is empty

- see figures 4.11(e) and 4.11(f)). In general, given ρ references $A_j[i]$ it is possible to create a vector of all 3^ρ regions formed by intersecting the simple regions from different array references:

$$R[x_1, x_2, ..., x_\rho] = R^1[x_1] \cap R^2[x_2] \cap ... \cap R^\rho[x_\rho] \tag{4.17}$$

Each index x_k is 1, 2 or 3 and refers to the *simple regions* of reference k. The lower bound of a region $R[x_1, x_2, ..., x_\rho]$ is the maximum of the lower bounds of the forming *simple regions*. Correspondingly, its upper bound is the minimum of the upper bounds of the forming *simple regions*:

$$R[x_1, x_2, ..., x_\rho].l = max(R^1[x_1].l, R^2[x_2].l, ..., R^\rho[x_\rho].l) \tag{4.18}$$

$$R[x_1, x_2, ..., x_\rho].u = min(R^1[x_1].u, R^2[x_2].u, ..., R^\rho[x_\rho].u) \tag{4.19}$$

In the same way, the checks that have to be performed inside a region $R[x_1, x_2, ..., x_\rho]$ are the combinations of the checks that each *simple region* requires. Hence reference A_1 requires checks in those regions $R[x_1, x_2, ..., x_\rho]$ in which x_1 is 1 or 3; at the same time in those regions in which x_1 is 2 no checks are required for reference A_1. Using this partitioning of the iteration space into 3^ρ regions, the loop $L(i, l, u, B(i))$ can be transformed into 3^ρ loops, each one implementing one of the regions $R[x_1, x_2, ..., x_\rho]$. The body of each region can be specialized to perform exactly the checks needed. We use $B_{x_1 x_2 ... x_\rho}$ to denote the body $B(i)$ specialized with the checks needed for region $R[x_1, x_2, ..., x_\rho]$. Hence, the transformation of $L(i, l, u, B(i))$ is:

$$
\begin{array}{c}
for(i = R[1,1,...,1].l; i < R[1,1,...,1].u; i++)\{B_{11...1}(i)\} \\
for(i = R[1,1,...,2].l; i < R[1,1,...,2].u; i++)\{B_{11...2}(i)\} \\
for(i = R[1,1,...,3].l; i < R[1,1,...,3].u; i++)\{B_{11...3}(i)\} \\
... \\
for(i = R[1,2,...,1].l; i < R[1,2,...,1].u; i++)\{B_{12...1}(i)\} \\
for(i = R[1,2,...,2].l; i < R[1,2,...,2].u; i++)\{B_{12...2}(i)\} \\
for(i = R[1,2,...,3].l; i < R[1,2,...,3].u; i++)\{B_{12...3}(i)\} \\
... \\
for(i = R[3,3,...,1].l; i < R[3,3,...,1].u; i++)\{B_{33...1}(i)\} \\
for(i = R[3,3,...,2].l; i < R[3,3,...,2].u; i++)\{B_{33...2}(i)\} \\
for(i = R[3,3,...,3].l; i < R[3,3,...,3].u; i++)\{B_{33...3}(i)\}
\end{array}
\tag{4.20}
$$

Note that the order of the resulting loops is important, although there are many correct orders. The requirement is that, for any value of j, region $R[x_1, ..., x_{j-1}, 1, x_{j+1}, ..., x - \rho]$ has to precede $R[x_1, ..., x_{j-1}, 2, x_{j+1}, ..., x_\rho]$, which in turn has to precede $R[x_1, ..., x_{j-1}, 3, x_{j+1}, ..., x_\rho]$.

Out of the 3^ρ possible regions formed by the intersection of the *simple regions*, no more than $2\rho + 1$ are nonempty and are actually executed at *run-time*. Which of the 3^ρ regions are nonempty depends on the relative positions of the 2ρ safe bounds $\mathfrak{L}^1, ..., \mathfrak{L}^\rho, \mathfrak{U}^1, ..., \mathfrak{U}^\rho$. There is no way to know at *compile-time* the order of these bounds (except for particular cases) and then it is necessary to write all the 3^ρ loops.

Computing safe bounds for different functions

In this section it is described how to compute the safe bounds $\mathfrak{L}$ and $\mathfrak{U}$ with respect to an array reference $A[f(i)]$. The regions are defined as follows:

$$R[1] : (l \leq i \leq u) \wedge (f(i) < lo(A)) \tag{4.21}$$

$$R[2] : (l \leq i \leq u) \wedge (lo(A) \leq f(i) \leq up(A)) \tag{4.22}$$

$$R[3] : (l \leq i \leq u) \wedge (f(i) > up(A)) \tag{4.23}$$

We first consider the case of $f(i)$ monotonically increasing. Using the fact that

$$lo(A) \leq f(i) \Rightarrow \lceil f^{-1}(lo(A)) \rceil \leq i \tag{4.24}$$

$$up(A) \geq f(i) \Rightarrow \lfloor f^{-1}(up(A)) \rfloor \geq i \tag{4.25}$$

the safe region $R[2]$ can be defined by:

$$R[2] : i = max(l, \lceil f^{-1}(lo(A)) \rceil), ..., min(u, \lfloor f^{-1}(up(A)) \rfloor) \tag{4.26}$$

Hence we can compute:

$$\mathfrak{L} = min(u + 1, max(l, \lceil f^{-1}(lo(A)) \rceil)) \tag{4.27}$$

$$\mathfrak{U} = max(l - 1, min(u, \lfloor f^{-1}(up(A)) \rfloor)) \tag{4.28}$$

and write again equations 4.11-4.13.

Similarly, if $f(i)$ is monotonically decreasing we can compute

$$\mathfrak{L} = min(u + 1, max(l, \lceil f^{-1}(up(A)) \rceil)) \tag{4.29}$$

$$\mathfrak{U} = max(l - 1, min(u, \lfloor f^{-1}(lo(A)) \rfloor)) \tag{4.30}$$

In the particular case of a linear subscript function of the form $f(i) = ai + b$, the inverse function $f^{-1}(i)$ can be easily computed:

$$f^{-1}(i) = \frac{i - b}{a} \tag{4.31}$$

Also the monotonicity of $f(i)$ is determined from the value of a: if $a > 0$ then $f(i)$ is monotonically increasing, while if $a < 0$ then $f(i)$ is monotonically decreasing. Note that the values of a and b need not to be known at *compile-time* since they can be efficiently computed at *run-time*.

There are others subscript functions for which bounds can be computed but they will not be discussed in this thesis: they are treated in [50].

An example

Consider the following loop to be transformed:

```
double A1[ ] = new double[n];
double A2[ ] = new double[n];
for(int i = l; i ≤ u; i++)
{
    A2[i + 2] = A1[i - 2]
}
```

It has two array references, $A1[i-2]$ and $A2[i+2]$, each generating three *simple regions*: $R^1[1:3]$ and $R^2[1:3]$, respectively. We can compute, as shown in equations 4.27 and 4.28:

$$\mathfrak{L}^1 = min(u+1, max(l, 2)) \tag{4.32}$$

$$\mathfrak{L}^2 = min(u+1, max(l, -2)) \tag{4.33}$$

$$\mathfrak{U}^1 = max(l-1, min(u, n+2)) \tag{4.34}$$

$$\mathfrak{U}^2 = max(l-1, min(u, n-2)) \tag{4.35}$$

Hence the loop is transformed in the code shown below:

```
for(i = l; i ≤ min(𝔏¹ - 1, 𝔏² - 1); i++){
    B11(i); //checks for A1 and A2
}
for(i = max(l, 𝔏²); i ≤ min(𝔏¹ - 1, 𝔘²); i++){
    B12(i); //checks for A1
}
for(i = max(l, 𝔘² + 1); i ≤ min(𝔏¹ - 1, u); i++){
    B13(i); //checks for A1 and A2
}
for(i = max(l, 𝔏¹); i ≤ min(𝔘¹, 𝔏² - 1); i++){
    B21(i); //checks for A2
}
for(i = max(𝔏¹, 𝔏²); i ≤ min(𝔘¹, 𝔘²); i++){
    B22(i); //no checks
}
for(i = max(𝔏¹, 𝔘² + 1); i ≤ min(𝔘¹, u); i++){
    B23(i); //checks for A2
```

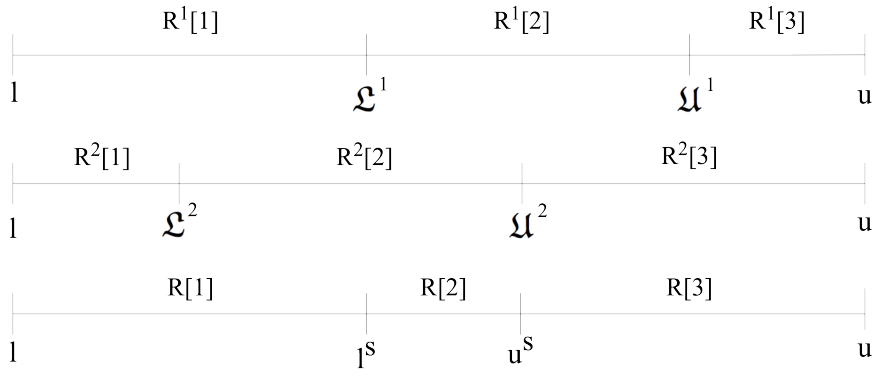


Figure 4.13: Partitions generated by two array references using the general method.

```
}
for(i = max(𝔘¹ + 1, l); i ≤ min(u, 𝔏² - 1); i++){
    B31(i); //checks for A1 and A2
}
for(i = max(𝔘¹ + 1, 𝔏²); i ≤ min(u, 𝔘²); i++){
    B32(i); //checks for A1
}
for(i = max(𝔘¹ + 1, 𝔘² + 1); i ≤ u; i++){
    B33(i); //checks for A1 and A2
}
```

The general method

The general method works by partitioning the iteration space of a loop always into three regions, independently of the number of array references in the body of the loop. One of the regions is safe: no array references in this region can cause violations. This region is defined by the range of values of $i = l^s, l^s + 1, ..., u^s$.

Consider the loop $L(i, l, u, B(i))$, with ρ references of the form $A_j[f_j(i)]$ in its body. We can compute its safe region as the intersection of all the safe regions of each reference:

$$R[2] = R^1[2] \cap R^2[2] \cap ... \cap R^\rho[2] \tag{4.36}$$

Concerning the other two regions, they are computed as:

$$R[1] = R^1[1] \cup R^2[1] \cup ... \cup R^\rho[1] \tag{4.37}$$

$$R[3] = R^1[3] \cup R^2[3] \cup ... \cup R^\rho[3] \tag{4.38}$$

Hence the safe bounds are computed as follows:

$$l^s = max(\mathfrak{L}^1, \mathfrak{L}^2, ..., \mathfrak{L}^\rho) \tag{4.39}$$

$$u^s = min(\mathfrak{U}^1, \mathfrak{U}^2, ..., \mathfrak{U}^\rho) \tag{4.40}$$

An example of this partitioning procedure is shown in figure 4.13. Region $R[2]$ is the safe region and its body can be implemented without any check. We denote this version of the loop body by $B(nocheck(i))$. Conversely, the bodies of the other two regions need checks on all the array references, as now we are no more discriminating for which reference the region is unsafe. These versions of the body are defined as $B(check(i))$. The transformation performed is:

$$\begin{array}{l} for(i = l; i \leq u; i++)\{ \\ \quad B(i) \\ \} \end{array} \quad \Longrightarrow \quad \begin{array}{l} for(i = l; i \leq l^s - 1; i++)\{ \\ \quad B(check(i)) \\ \} \\ for(i = l^s; i \leq u^s; i++)\{ \\ \quad B(nocheck(i)) \\ \} \\ for(i = u^s + 1; i \leq u; i++)\} \\ \quad B(check(i)) \\ \} \end{array}$$

or in shorthand:

$$L(i, l, u, B(i)) \quad \Longrightarrow \quad \begin{array}{l} L_1(i, l, l^s - 1, B(check(i))) \\ L_2(i, l^s, u^s, B(nocheck(i)) \\ L_3(i, u^s + 1, u, B(check(i))) \end{array}$$

With this method some redundant checks are still present; but, on the other hand, it is very useful to avoid loop explosion when there are too many array accesses inside a loop.

Handling nested loops

If the body B of a loop contains other loops, then the same transformation can be applied to each of the loops in B. The transformation can be applied individually to each and every loop in a general loop nest. Consider the case of a two-dimensional loop nest $L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j)))$. By first applying the transformation to the L loop, we generate a region without any check

on array references indexed by i:

$$L(i, l_i, u_i, L'(j, l_j, u_j, B(i,j)) \implies \begin{array}{l} L_1(i, l_i, l_i^s - 1, L'(j, l_j, u_j, B(check(i), j)) \\ L_2(i, l_i^s, u_i^s, L'(j, l_j, u_j, B(nocheck(i), j)) \\ L_3(i, u_i^s + 1, u_i, L'(j, l_j, u_j, B(check(i), j)) \end{array}$$

We can now apply the transformation to each of the three L' loops:

$$L_1(i, l_i, l_i^s - 1, \left| \begin{array}{l} L'_1(j, l_j, l_j^s - 1, B(check(i), check(j))) \\ L'_2(j, l_j^s, u_j^s, B(check(i), nocheck(j))) \\ L'_3(j, u_j^s + 1, u_j, B(check(i), check(j))) \end{array} \right|)$$

$$L_2(i, l_i^s, u_i^s, \left| \begin{array}{l} L'_1(j, l_j, l_j^s - 1, B(nocheck(i), check(j))) \\ L'_2(j, l_j^s, u_j^s, B(nocheck(i), nocheck(j))) \\ L'_3(j, u_j^s + 1, u_j, B(nocheck(i), check(j))) \end{array} \right|)$$

$$L_3(i, u_i^s + 1, u_i, \left| \begin{array}{l} L'_1(j, l_j, l_j^s - 1, B(check(i), check(j))) \\ L'_2(j, l_j^s, u_j^s, B(check(i), nocheck(j))) \\ L'_3(j, u_j^s + 1, u_j, B(check(i), check(j))) \end{array} \right|)$$

As can be seen there are now three versions of the L' loop for each version of L loop; also in this case if there are too many nested loops, there can be a loop explosion, i.d. to many loops are generated. The next section will discuss another method to handle nested loops avoiding the loop explosion.

Handling nested loops - The compact method

The exponential expansion on the number of loops caused by the application of the general method can be highly undesirable in some cases. The method proposed in this section results in only a linear increase of the number of regions. The difference resides in how the transformation is recursively applied to the nested loops. Once the outermost loop is split, the innermost one is divided too, but only inside the safe region of the outermost loop. Hence the transformation of $L(i, l_i, u_i, L'(j, l_j, u_j, B(i,j)))$ is:

$$\begin{array}{c} L_1(i, l_i, l_i^s - 1, L'(j, l_j, u_j, B(check(i), check(j))) \\ L_2(i, l_i^s, u_i^s, \left| \begin{array}{l} L'_1(j, l_j, l_j^s - 1, B(nocheck(i), check(j))) \\ L'_2(j, l_j^s, u_j^s, B(nocheck(i), nocheck(j))) \\ L'_3(j, u_j^s + 1, u_j, B(nocheck(i), check(j))) \end{array} \right|) \\ L_3(i, u_i^s + 1, u_i, L'(j, l_j, u_j, B(check(i), check(j))) \end{array}$$

Also in this case, some redundant checks may be still present; however there is a safe region in which the checks are not needed.

4.6.5 The implementation

After the basic ideas of the algorithm have been explained, in this section it will be discussed how it has been implemented it. The *Bounds Checks Optimizer* (*BCO*) takes as input the *AST* generated by the *C* code transformation and returns a new tree which represents the optimized version of the original code. The algorithm is divided in the following phases:

- Loop research.
- Tree backup.
- Loop invariance verification.
- Collection of data.
- Identification of the subscript function form.
- New code generation.

Before starting to analyze each phase, it is important to make some considerations on the structure of the code that is going to be processed. It is important to focus on loops with array references, in particular on *for* loops. This because in a *for* loop the initial value of the iteration variable, in many cases, can be easily found (in case the initial value of the iteration is assigned before the loop then this one is not optimized).

Listing 4.62: *Pseudo C* code for an array access inside a loop

```
1 for(int i = 0; i < A.length; i++)
2 {
3   {
4     int tmp1 = i;
5     if((ulong) tmp1 >= A.length)
6       RaiseException();
7     A[tmp1] = i;
8   }
9 }
```

The other loops (e.g. *while* or *do-while*) are not treated in this work, even if they are not impossible to handle. It is known that each array reference has its bound check just before. It is important to know precisely their structure. Listing 4.62 shows an example in *pseudo C* code of an array access inside a

loop with its bound check, as it is translated by the *EveC* compiler into *C* code.

The instruction on line 4 is useful for that cases in which the access is made with a method call (i.e. `A[funcall(i)]`); in this example that instruction avoids an unuseful multiple call to that method. Note that there is an inner block inside the block of the loop. This because every instruction in *EveC* is translated in one or more instructions in *C* surrounded by the curly brackets (in this way it is easier for the compiler to modify the code; see Chapter 3). Hence a loop with many instructions inside its body assumes the form of code shown in listing 4.63.

Listing 4.63: *Pseudo C* code for a loop with many instructions

```
for(int i = 0; i < A.length; i++)
{
  {
    int tmp1 = i;
    if((ulong) tmp1 >= A.length)
      RaiseException();
    A[tmp1] = i;
  }
  {
    otherInstruction();
  }
  {
    otherInstruction2();
  }
}
```

The only instruction that is not surrounded by brackets is a variable declaration, because otherwise the variable could not be used outside. Hence, if in the loop there is a declaration with an array access, the code is something like the one shown in listing 4.64.

Listing 4.64: *Pseudo C* code for a loop with a variable declaration

```
for(int i = 0; i < A.length; i++)
{
  int tmp1 = i;
  if((ulong) tmp1 >= A.length)
    RaiseException();
  int k = A[tmp1];
  {
    otherInstruction();
  }
  {
    otherInstruction2();
  }
}
```

Hence, the *AST* can assume two different forms. As for the code of listing 4.63 the tree structure is shown in figure 4.14(a), while for the code of listing 4.64 the corresponding tree structure is the one in figure 4.14(b).

The next sections are focused on the implementation and motivations of each phase.

Loop Research

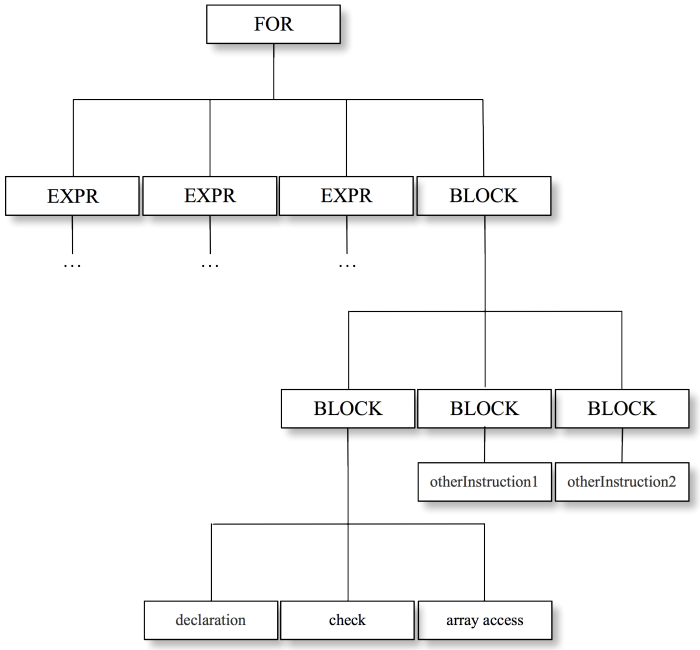
The very first stage of the process is trivially the research of a loop. On the *AST* a pre-order visit is performed. When the keyword *FOR* is found, the whole subtree is passed as an argument to the method that will perform the optimization.

Tree Backup

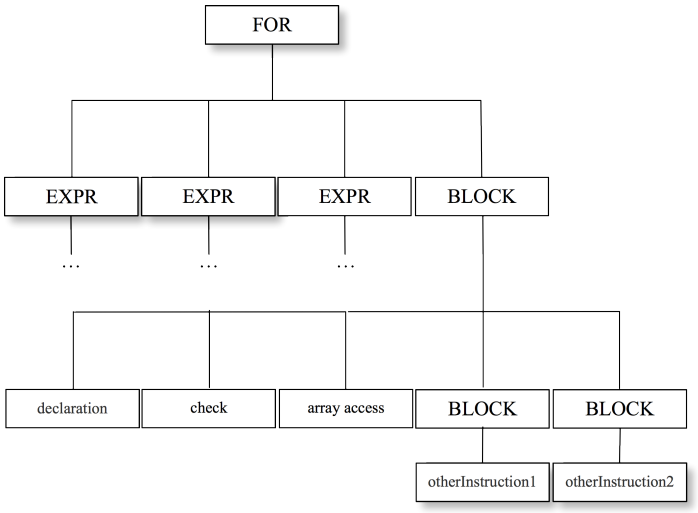
Before the optimization is performed, a deep copy of the tree is created. This is necessary because, during the algorithm execution, the tree is modified many times. In case of errors or exceptions it is crucial to rollback to the original tree in order to generate the correct code. The algorithm in fact intercepts every exception and returns the tree backup in that case.

Loop Invariance Verification

For the algorithm to be valid, there is an important constraint that has to be satisfied: the loop must be invariant. This means that inside the loop there must not be any reassignment of the iteration variable or of the accessed

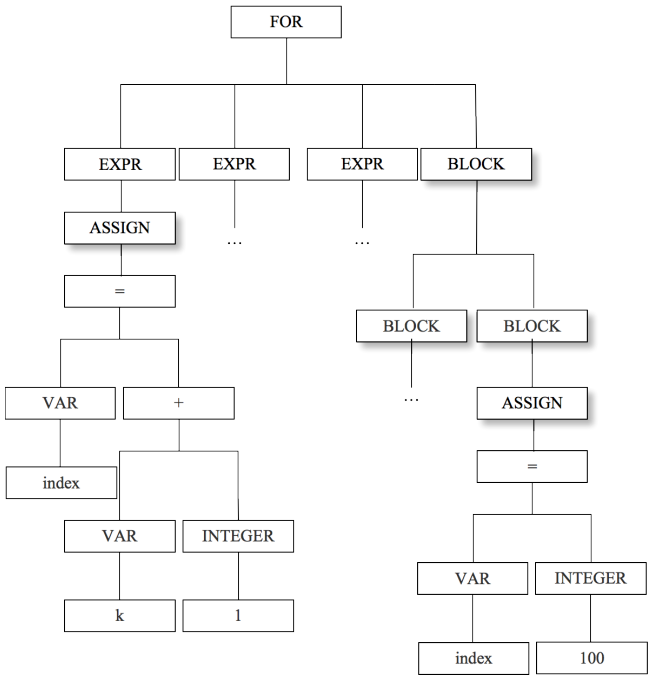


(a)

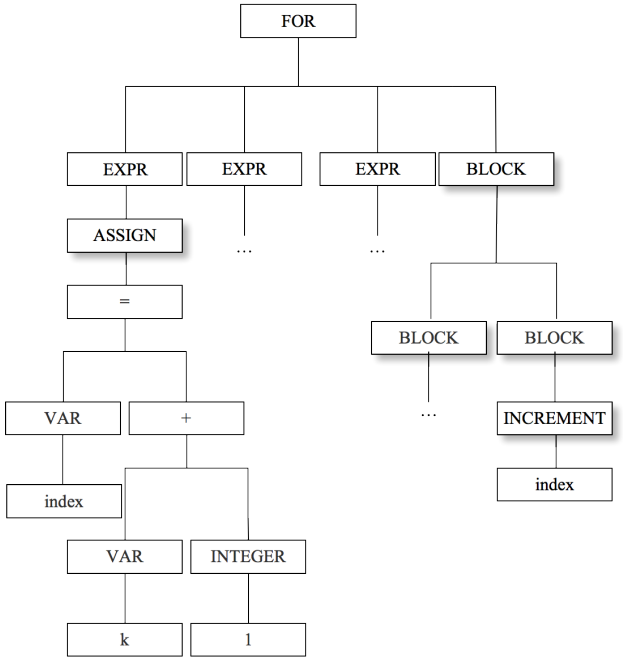


(b)

Figure 4.14: The tree structures of the code shown in listings 4.63 and 4.64.



(a)



(b)

Figure 4.15: When a re-assignment of the iteration index is found, the loop cannot be optimized.

arrays. This because the constraints of the loop generated by the *BCO* are calculated taking into account two values:

- The sizes (calculated before the loop) of the arrays accessed inside the loop.
- The iteration variable.

Hence, if an array is reassigned inside the loop, the algorithm will create wrong regions, thus threatening the safety and the correctness of the language. The code of listing 4.65 shows a loop with a completely safe array access.

Listing 4.65: A completely safe array access inside a loop

```
int [] A = new int [3];
for(int i = 0; i < 3; i++)
{
  if(check(A,i))
    ThrowException();
  A[i];
}
```

The *BCO* will create three loops as shown in listing 4.66.

Listing 4.66: The transformed loop starting by listing 4.65

```
int [] A = new int [3];
for(int i = 0; i <= -1; i++)
{
  if(check(A,i))
    ThrowException();
  A[i]
}
for(int i = 0; i <= 2; i++)
{
  A[i]
}
for(int i = 2; i <= 1; i++)
{
  if(check(A,i))
    ThrowException();;
  A[i]
}
```

Listing 4.67: An unsafe array access inside a loop

```
int [] A = new int [3];
for(int i = 0; i <= 2; i++)
{
  A = new int [2];
  if(check(A,i))
    ThrowException();
  A[i];
}
```

If in the loop body there is a reassignment of `A`, as shown in listing 4.67, then the only nonempty region will be again the safe one and the array will be accessed without any check for all the iterations.

This time, however, the access is not always safe. As we can notice in listing 4.68, now the array A has length equals to 2, hence the last valid index is 1, but it is "safely" accessed with index 2.

Listing 4.68: The transformed loop starting by listing 4.67

```
int [] A = new int [3];
for(int i = 0; i <= -1; i++)
{
  A = new int [2];
  if(check(A,i))
    ThrowException();
  A[i]
}
for(int i = 0; i <= 2; i++)
{
  A = new int [2];
  A[i]
}
for(int i = 2; i <= 1; i++)
{
  A = new int [2];
  if(check(A,i))
    ThrowException();
  A[i]
}
```

For the same reason it is important to verify the invariance of the iteration variable too. If the index is incremented (or decremented) inside the loop body, then wrong regions will be generated again. If the invariance is not guaranteed, then the loop is not optimized. To verify whether the array or the iteration variable are not reassigned inside the loop body, they have to be searched inside the loop body through the *AST* and, if found, they are verified not to be the left child of an *ASSIGN* node (figure 4.15(a)) or a child of an increment or decrement instruction (figure 4.15(b)).

Collection of data

Once verified the loop invariance, it is necessary to collect all the data needed by the algorithm. These data are:

- The loop bounds.
- The array accesses and their relative checks.

This operation, always performed through the *AST*, requires a particular form of the tree, i.e. it needs the loop bounds to be expressed as a function of the same variable (the iteration variable) as shown in figure 4.16; otherwise, it is not possible to perform the optimization and the algorithm returns the tree backup. It is important to notice that the collected data are not values, but rather expressions whose values are not known at *compile-time*. They are assigned before the loop to variables that are used instead of them in the algorithm. In this way it is possible to optimize loops whose bound values are not known at *compile-time*, but only at *run-time*. The same consideration can be made on the subscript function, which for example can be in the form: $ai + b$ (where i is the iteration variable). Also in this case it is not important to know the values of a and b at *compile-time*. When an array reference is found inside a loop, its relative check (or checks, if it is a Matrix or Volume) is removed and stored in a particular structure. It will be reinserted only in the loop body version of the unsafe regions, during the new code generation phase.

Identification of the subscript function form

As already said, in the previous section, the algorithm works with different subscript function forms. This work has focused on the linear form ($ai + b$, where i is the iteration variable, a and b are constants with respect to the loop). However, the structure of this part of the algorithm is easily expandable for other function forms. The algorithm, in fact, finds the subscript

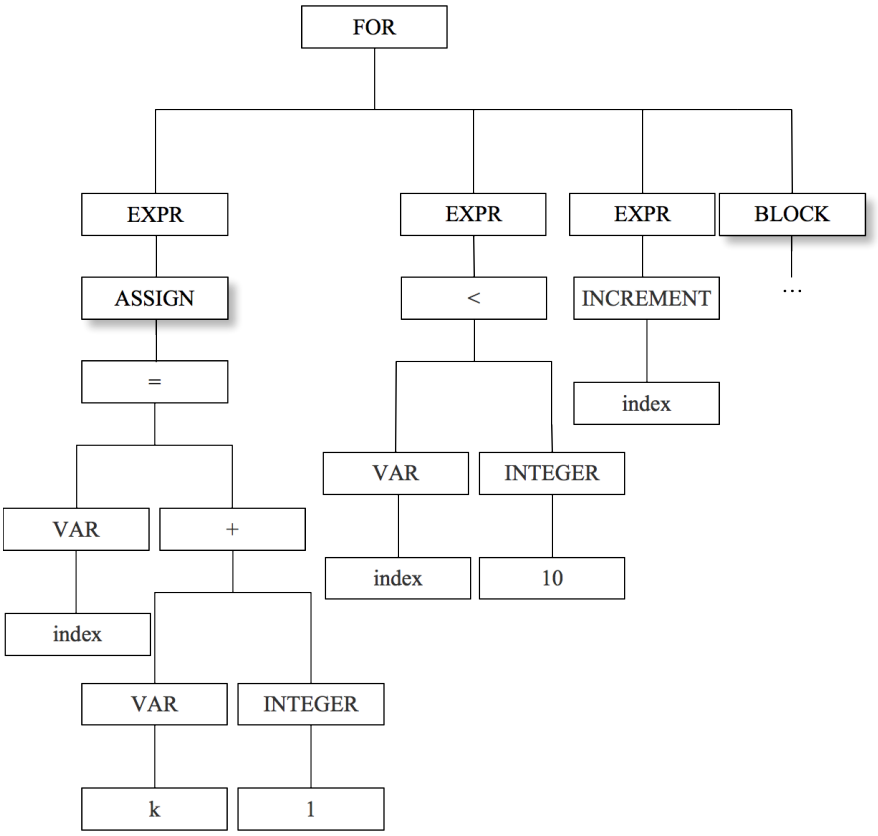


Figure 4.16: An example of tree structure required for the loop to be optimized.

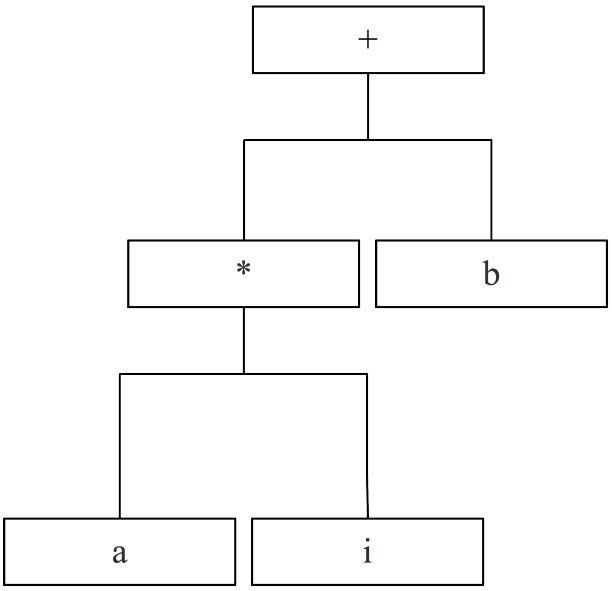


Figure 4.17: The tree structure of a linear function $ai + b$.

function (expressed as a tree) and transforms it in such a way that the independent (w.r.t. the iteration variable) parts of it are substituted by a predefined symbol (the chosen one is "?") while the dependent parts are substituted with i. In such a way, the function $ai + b$ (represented in figure 4.17, and in a written way as $(+(*ai)b)$) is transformed in $(+(*?i)?)$. Then this string is compared to the strings stored in a *Hash Map* and, if found, the values of a and b are retrieved. Once these values are known, the algorithm can compute the bounds of the array.

New code generation

Once the bounds are known, the algorithm can proceed and create the needed loops. The loops are created in a string which represents a portion of C code. The first lines are the bounds of the loop and of the regions of the arrays, stored in variables, named in a pre-defined way. The subsequent lines are the loops (3^ρ, where ρ is the number of unique references, with the exact method): obviously, the loop body is not always the same as the checks are added only when needed. However, there is one last consideration to take into account: if the original loop iteration space is empty, hence the lower bound (l) is greater than the upper bound (u), considering a positive increment of the iteration variable, then the algorithm won't work. To avoid this issue an `if` statement that checks whether this condition is verified is added, surrounding all the generated loops. Once the loops are created, the

whole generated code is transformed in a tree and added to the original *AST*, removing the previous subtree representing the original loop.

4.6.6 The BCO cost

The algorithm just described cannot guarantee a better performance in every case. In fact it adds an overhead to the original program in order to calculate the necessary bounds as described in the previous section. In order to understand in which cases the BCO would not provide better performances, the added overhead can be compared to the removed overhead (given by the elimination of the range checks). Given the just described implementation of the algorithm, the added overhead consists in (in the worst case): 2 initial assignments, 1 initial check, 6ρ checks (where ρ is the number of references in the loop body), 6ρ assignments (these costs are given by the variables initialization and the computation of the bounds of the *regions*) and 3^ρ assignments plus 3^ρ checks (given by the effective splitting of the original loop into 3^ρ loops). This consideration is valid for a loop that has a depth equals to 1 and can be represented, considering A the cost of a single assignment and C the cost of a single check, by the expression:

$$2A + 1C + 6\rho C + 6\rho A + 3^\rho A + 3^\rho C \tag{4.41}$$

This expression can be reduced to:

$$(6\rho + 3^\rho + 2)A + (6\rho + 3^\rho + 1)C \tag{4.42}$$

This expression, as said, must be compared to the overhead removed by the algorithm, which is the number of checks that are not executed thanks to the BCO. In order to calculate this number, an assumption has to be made: it can be assumed that the method removes all the checks for an array reference. This is not a strong assumption: in fact, considering the exact method, the algorithm would remove all the checks if there are no unsafe accesses (in case there are unsafe accesses the removed overhead is lower). In this case, the overhead removed by the optimization is the number of checks multiplied by the size of the iteration space (s). This because for each iteration of the loop, ρ array accesses' checks are not executed.

Hence, fixed ρ, the removed overhead grows with the growth of the size of the iteration space, while the added overhead is constant. The expected behaviour is that for a low value of s the added overhead is greater than the removed one, while for s greater than a certain threshold, the added overhead is lower than the removed one; only in the second case the optimization leads to better performances.

In order to calculate the threshold, it can be reduced the equation:

$$(6\rho + 3^\rho + 2)A + (6\rho + 3^\rho + 1)C = \rho s \tag{4.43}$$

Once reduced, the threshold s trivially is:

$$s = \frac{(6\rho + 3^\rho + 2)A + (6\rho + 3^\rho + 1)C}{\rho} \tag{4.44}$$

To test whether this consideration was true or not, a simple program has been used. The program executes an array initialization for many times, as can be seen in the code shown in listing 4.69.

Listing 4.69: The program used for testing the cost of BCO

```
while(i < steps)
    {
      for(int j = 0; j < length; j++)
      {
        array[j] = j;
      }
      i++;
    }
```

The program has been tested with a value of `steps` big enough to have significant results (100.000.000) and an increasing value of `length`. The performance is calculated with the execution time expressed in milliseconds, hence the expected behaviour is that, for low values of `length`, the execution time of the program without the optimization would be lower than the one of the program with the optimization, and vice versa for higher values of `length`. This is exactly what happens as can be seen in the graphic shown in figure 4.18 (the test has been executed on the machine used for all the tests; see chapter 5).

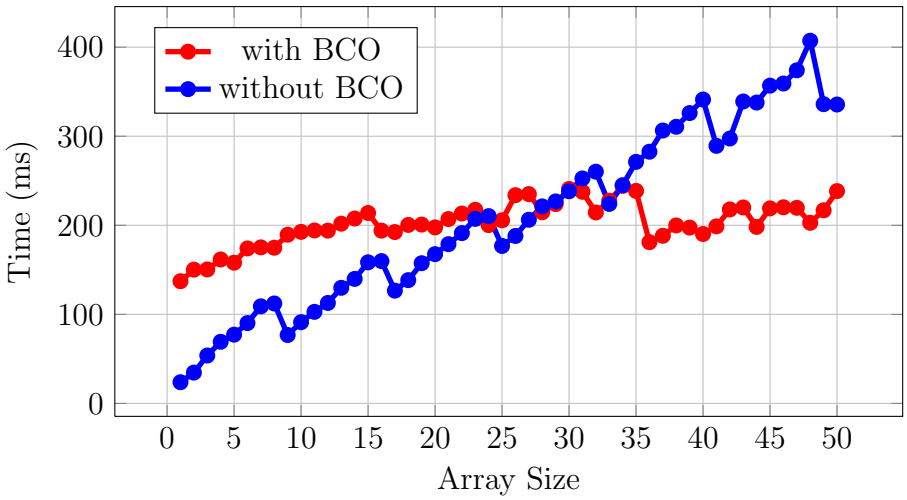


Figure 4.18: The time of execution of the program shown in listing 4.69 with and without the optimization with different array sizes.

Chapter 5

Results

5.1 Introduction

The estimation of the performance of an entire programming language is not an easy task. For this reason, a large number of benchmarks are used. The scope of this thesis is to develop a new programming language, *EveC*, in order to obtain a more efficient language than *Java*. Therefore, the results report the comparison of performances for *EveC* and *Java*, both in terms of memory usage and computational speed. Details on the version of software and specification of the different architectures used to evaluate the *EveC* programming language are described in details in section 5.2.

In the first part of this chapter, an overview on the global performance of *EveC* is provided, while in the second, starting from section 5.6, results focused on each single optimization are described.

The benchmarks used in the first part can be divided into two groups:

- Micro-benchmarks, that try to evaluate the performance of a particular construct, such the addition of two integer number, or the cost of a method call.
- Macro-benchmarks, that are bigger application that should represent parts of real world applications.

Micro-benchmarks are important in order to evaluate single constructs of the programming language, and can also be useful to find bottlenecks that can be target of further optimizations. On the other hand, micro-benchmarks are very small applications that do not represent real world ones, formed by thousands lines of code. For this reason, results on macro-benchmarks are also reported. Marco-benchmarks represent bigger applications, closer

to real world applications than micro-benchmarks. At this stage of development, a huge real world application cannot be ported or written using the *EveC* programming language because not all the functionalities of a complete library are implemented yet. For example, it is not possible to read or write a file; threads are not available yet; part of the *Collections Framework* is not ported yet. On the other hand, the ability to produce correct code for big and different benchmarks and for an important part of the library system demonstrates that the actual implementation of the *EveC* compiler and *Runtime System* is quite stable and it can be used in future to compile production level applications.

Some of the used benchmarks are freely available on the web and are also well known in literature. Other benchmarks have been written in order to explore some other situations and to validate the performance gain of some particular optimizations. For instance, different well-known sorting algorithms are used in order to validate the *Bound Check Optimization*. For all of these benchmarks the source code is included.

5.2 Architecture used

To perform all the benchmarks a machine with an Intel i5 processor has been used. The hardware specification of the machine is presented in table 5.1. The Ubuntu 12.10 operating system, equipped with Linux kernel version 3.5.0-24-generic, has been used. The *JVM* chosen is the *OpenJDK 64-Bit Server VM 23.7-b01*, that supports *Java 1.7.0_13*. In order to compile the *C* code produced by the *EveC* compiler, the *GCC 4.7.2* is exploited, with the flag `-O3` enabled, so as to produce optimized code.

Table 5.1: The hardware specification of the machine used to test all the bechmarks.

Component	Values
CPU	i5-2430M
Number of Core	2
Number of Thread	4
Clock Speed	2.4 GHz
Max Turbo Frequency	2.9 GHz
RAM	6 GB
Memory Type	DDR3-800/1066

5.3 Benchmarks

5.3.1 Java Grande Benchmarks

Java Grande Benchmarks is a suite of benchmarks relevant for testing the performance of *Java* for scientific computations. It is described in details in [51]. It can be divided into three different sections: low-level benchmarks that test general language features and operations (e.g. method invocation); scientific and numerical application kernel; full-scale science and engineering application. In the following all the benchmarks used in this work are reported and briefly described. The described benchmarks are all successfully ported in the *EveC* programming language, and the results of the different computations are verified to be correct. In order to get more stable results, all the benchmarks that are part of the *Java Grande Benchmarks* are executed five times, and only the arithmetic mean is reported for each one. Each benchmark reports a score, that represents the number of the basis operations executed by that benchmark per seconds.

Low-level benchmarks

The low-level benchmarks are a collection of micro-benchmarks that test some general language constructs and operations. They consist in:

- *Arith*: it executes basic arithmetic operations.
- *Assign*: it does variable assignments.
- *Cast*: it does cast between *primitive ypes*.
- *Create*: it generates different kinds of objects:
  - the simplest *Java* object, `java.lang.Object`;
  - a simple user defined object with no contents and no constructors;
  - a simple user defined object with no contents but with a constructor;
  - a more complex object with primitive variables;
  - an object which instantiates other objects;
  - arrays of different types and sizes.
- *Math*: it executes all the mathematical functions present in the class `java.lang.Math`.

- *Method*: it performs method calls of different types.
- *Loop*: it executes *while* loops and different *for* loops that iterate forward and backward, also with a synchronized block.

All of the low-level benchmarks run the test within a *for* loop, with each loop performing 16 operations, so that the loop overhead is not significant.

Scientific and numerical application kernel

The scientific and numerical computational benchmarks test well-known computational benchmarks included in an huge number of real world application. For this reason and also because they are bigger and more complex than the low-level benchmarks, scientific and numerical computational benchmarks are included into the macro-benchmarks section. They are composed by:

- Fourier: it computes fourier coefficients for the function $f(x) = (x+1)^2$ on the interval $[0, 2]$.
- Heap Sort: it sorts an array of integers using the heap sort algorithm, and tests generic integer performances.
- Crypt: it performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes.
- FFT: it performs a one-dimensional forward transformation of N complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.
- SOR: it performs 100 iterations of successive over-relaxation on a NxN grid. The performance reported is in iterations per second.
- Sparse Matrix Multiplication: it uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references. A NxN sparse matrix is used for 200 iterations.

All the benchmarks have been executed with two different sizes (the value of N): A and C as defined in [51].

5.3.2 jBYTE

The *jBYTE* benchmark is a collection of well-known algorithms proposed by the *BYTE Magazine*. This benchmark is used in different works such as [6]. The ported benchmarks are:

- Numeric Sort: it sorts an array of 32-bit integer using the heap sort algorithm.
- Bitfield: it executes a variety of bit manipulation functions.
- Emulated floating-point: it performs floating-point operation via software.
- Fourier Coefficients: it executes a numerical analysis routine for calculating series approximations of waveforms.
- Huffman compression: a well-known text and graphics compression algorithm.
- IDEA encryption: a block cipher algorithm.
- Neural Net: a small but functional back-propagation network simulator.
- LU Decomposition: a robust algorithm for solving linear equations.

An important built-in feature of the *jBYTE* is a set of statistical-analysis routines. The *jBYTE* keeps score as follows: each test is run five times. These five scores are averaged, the standard deviation is determined, and a 95% confidence half-interval for the mean is calculated. This tells us that the true average lies (with a 95% probability) within plus or minus the confidence half-interval of the calculated average. If this half-interval is within 5% of the calculated average, the benchmarking stops. Otherwise, a new test is run and the calculations are repeated. The upshot is that, for each benchmark test, the true average is (with a 95% level of confidence) within 5% of the average reported. All the benchmarks within *jBYTE* are macro-benchmarks and, for each benchmark, the *indexed score* is reported [52].

5.3.3 Simplex

The *Simplex* benchmark is a simple implementation of the well-known optimization algorithm, which finds an optimal solution in a minimization problem (or maximization) of a linear function with linear constraints. This benchmark is considered as a macro-benchmark. The algorithm uses an implicit enumeration technique to explore all the possible solutions terminating when the optimal solution is found. The implementation of the algorithm is taken from [53], and a simplified version is used. The complete *EveC* source code is presented in listing 5.1: the implementation relies on the standard

random generator defined for *Java* to initialize the data processed by the algorithm. The *Simplex* benchmark is executed with a different number of variables and constraints. The reported results are the arithmetical means on five run, in order to achieve better results stability (i.e. results with higher confidence due to multiple runs).

Listing 5.1: *EveC* implementation of the class `Simplex`.

```
/************************************************
 *  Given an M-by-N matrix A, an M-length vector b,
    and an
 *  N-length vector c, solve the  LP { max cx : Ax
    <= b, x >= 0 }.
 *  Assumes that b >= 0 so that x = 0 is a basic
    feasible solution.
 *
 *  Creates an (M+1)-by-(N+M+1) simplex tableaux
    with the
 *  RHS in column M+N, the objective function in row
     M, and
 *  slack variables in columns M through M+N-1.
 *
 ************************************************/
import |lang.Random;
import |lang.Integer;
public class Simplex {
    private static final double EPSILON = 1.0E-10;
    private double[,] a;   // tableaux
    private int M;          // number of constraints
    private int N;          // number of original
       variables
    private int[] basis;    // basis[i] = basic
       variable corresponding to row i
                            // only needed to print
                               out solution, not
                               book
    // sets up the simplex tableaux
    public Simplex(double[,] A, double[] b, double[]
        c) {
        M = (int)b.size;
        N = (int)c.size;
        a = new double[M+1,N+M+1];
```

```
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                a[i,j] = A[i,j];
        for (int i = 0; i < M; i++) a[i,N+i] = 1.0;
        for (int j = 0; j < N; j++) a[M,j]   = c[j];
        for (int i = 0; i < M; i++) a[i,M+N] = b[i];
        basis = new int[M];
        for (int i = 0; i < M; i++) basis[i] = N + i
           ;
        solve();
        // check optimality conditions
      if (!check(A, b, c)) {
      System.out.println("Error: Not optimal
         solution");
    }
    }
    // run simplex algorithm starting from initial
       BFS
    private void solve() {
        while (true) {
            // find entering column q
            int q = bland();
            if (q == -1) break;  // optimal
            // find leaving row p
            int p = minRatioRule(q);
            if (p == -1) throw new RuntimeException
               ("Linear program is unbounded");
            // pivot
            pivot(p, q);
            // update basis
            basis[p] = q;
        }
    }
    // lowest index of a non-basic column with a
       positive cost
    private int bland() {
        for (int j = 0; j < M + N; j++)
            if (a[M,j] > 0) return j;
        return -1;  // optimal
    }
   // index of a non-basic column with most positive
```

```
   cost
private int dantzig() {
    int q = 0;
    for (int j = 1; j < M + N; j++)
        if (a[M,j] > a[M,q]) q = j;

    if (a[M,q] <= 0) return -1;  // optimal
    else return q;
}
// find row p using min ratio rule (-1 if no
   such row)
private int minRatioRule(int q) {
    int p = -1;
    for (int i = 0; i < M; i++) {
        if (a[i,q] <= 0) continue;
        else if (p == -1) p = i;
        else if ((a[i,M+N] / a[i,q]) < (a[p,M+N]
            / a[p,q])) p = i;
    }
    return p;
}
// pivot on entry (p, q) using Gauss-Jordan
   elimination
private void pivot(int p, int q) {
    // everything but row p and column q
    for (int i = 0; i <= M; i++)
        for (int j = 0; j <= M + N; j++)
            if (i != p && j != q) a[i,j] -= a[p,
               j] * a[i,q] / a[p,q];
    // zero out column q
    for (int i = 0; i <= M; i++)
        if (i != p) a[i,q] = 0.0;
    // scale row p
    for (int j = 0; j <= M + N; j++)
        if (j != q) a[p,j] /= a[p,q];
    a[p,q] = 1.0;
}
// return optimal objective value
public double value() {
    return -a[M,M+N];
}
```

```
    // return primal solution vector
    public double[] primal() {
        double[] x = new double[N];
        for (int i = 0; i < M; i++)
            if (basis[i] < N) x[basis[i]] = a[i,M+N
               ];
        return x;
    }
    // return dual solution vector
    public double[] dual() {
        double[] y = new double[M];
        for (int i = 0; i < M; i++)
            y[i] = -a[M,N+i];
        return y;
    }
    // is the solution primal feasible?
    private boolean isPrimalFeasible(double[,] A,
       double[] b) {
        double[] x = primal();

        // check that x >= 0
        for (int j = 0; j < (int)x.size; j++) {
            if (x[j] < 0.0) {
                System.out.println("x[" + j + "] = "
                    + x[j] + " is negative");
                return false;
            }
        }
        // check that Ax <= b
        for (int i = 0; i < M; i++) {
            double sum = 0.0;
            for (int j = 0; j < N; j++) {
                sum += A[i,j] * x[j];
            }
            if (sum > b[i] + EPSILON) {
                System.out.println("not primal
                   feasible");
                System.out.println("b[" + i + "] = "
                    + b[i] + ", sum = " + sum);
                return false;
            }
```

```
        }
        return true;
    }
    // is the solution dual feasible?
    private boolean isDualFeasible(double[,] A,
       double[] c) {
        double[] y = dual();
        // check that y >= 0
        for (int i = 0; i < (int)y.size; i++) {
            if (y[i] < 0.0) {
                System.out.println("y[" + i + "] = "
                    + y[i] + " is negative");
                return false;
            }
        }
        // check that yA >= c
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int i = 0; i < M; i++) {
                sum += A[i,j] * y[i];
            }
            if (sum < c[j] - EPSILON) {
                System.out.println("not dual
                   feasible");
                System.out.println("c[" + j + "] = "
                    + c[j] + ", sum = " + sum);
                return false;
            }
        }
        return true;
    }
    // check that optimal value = cx = yb
    private boolean isOptimal(double[] b, double[] c
       ) {
        double[] x = primal();
        double[] y = dual();
        double value = value();
        // check that value = cx = yb
        double value1 = 0.0;
        for (int j = 0; j < (int)x.size; j++)
            value1 += c[j] * x[j];
```

```
        double value2 = 0.0;
        for (int i = 0; i < (int)y.size; i++)
            value2 += y[i] * b[i];
        if (Math.abs(value - value1) > EPSILON ||
           Math.abs(value - value2) > EPSILON) {
            System.out.println("value = " + value +
               ", cx = " + value1 + ", yb = " +
               value2);
            return false;
        }
        return true;
    }
    private boolean check(double[,]A, double[] b,
       double[] c) {
        return (isPrimalFeasible(A, b) &&
           isDualFeasible(A, c) && isOptimal(b, c));
    }
    // test client
    public static void main(String[] args) {
    Random r = new Random(101010);
        int M = Integer.parseInt(args[0]);
        int N = Integer.parseInt(args[1]);
        double[] c = new double[N];
        double[] b = new double[M];
        double[,] A = new double[M,N];
        for (int j = 0; j < N; j++)
            c[j] = r.nextInt(1000);
        for (int i = 0; i < M; i++)
            b[i] = r.nextInt(1000);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                A[i,j] = r.nextInt(100);
        long start = System.currentTimeMillis();
        Simplex lp = new Simplex(A, b, c);
    long end = System.currentTimeMillis();
        long time = end - start;
        System.out.println(time);
        System.out.println(lp.value());
    }
}
```

5.3.4 Prime Sieve

The *Prime Sieve* benchmark computes prime numbers using the Sieve of Eratosthenes algorithm [54]. The *EveC* version of the benchmark is presented in listing 5.2. The maximum number below which prime numbers are searched is two billion; the results presented are the mean of ten run to achieve better results stability.

Listing 5.2: *EveC* implementation of the *Prime Sieve* benchmark.

```
/*******************************************
 *  Computes the number of primes less than or equal
     to N using
 *  the Sieve of Eratosthenes.
 *******************************************/
public class PrimeSieve {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        long start = System.currentTimeMillis();
        // initially assume all integers are prime
        boolean[] isPrime = new boolean[N + 1];
        for (int i = 2; i <= N; i++) {
            isPrime[i] = true;
        }
        // mark non-primes <= N using Sieve of
           Eratosthenes
        for (int i = 2; i*i <= N; i++) {
            // if i is prime, then mark multiples of
                i as nonprime
            // suffices to consider mutiples i, i+1,
                ..., N/i
            if (isPrime[i]) {
                for (int j = i; i*j <= N; j++) {
                    isPrime[i*j] = false;
                }
            }
        }
        // count primes
        int primes = 0;
        for (int i = 2; i <= N; i++) {
            if (isPrime[i]) primes++;
        }
        long end = System.currentTimeMillis()-start;
        System.out.println("The number of primes <=
           " + N + " is " + primes);
        System.out.println(end);
    }
}
```

5.4 Preliminary Results

In this section, some preliminary results are presented. The performance of *Java* and *EveC* without any optimizations are considered and compared. From table D.1 to table D.9 the results for low-level and for scientific and numerical computational benchmarks, part of the *Java Grande Benchmark*, are presented. In all of these tables, the first column indicates the performance of the *Java* version of the examined benchmark, the second column indicates the performance of the *EveC* version of the benchmark without any optimization activated, and the last column presents the ratio between the second column and the first column, indicating the gain or the loss of performance of the *EveC* programming language. In all of these tables, high values mean better results.

In the indicated tables some values are missing. This happens because the execution time of the respective benchmarks is extremely short. More precisely, as the execution time is calculated in milliseconds, it happens that the related benchmark takes less than one millisecond; hence dividing the number of operations by the, very short, elapsed time, the outcome generated is a large number that is approximated to infinite. The *Arith* benchmark shows results that highlight a performance gain of *EveC* with respect to *Java*. As can be noted in the table D.1, the *EveC* performances are similar to *Java*, with an important gain in the integer and long division. The *Assign* benchmark shows even better results, with a performance gain of at least 1100%. The *Cast* results are not analysable as the *EveC* outcomes are all infinite.

In the *Create* benchmark, the results show that *Java* is faster than *EveC*, even more when `Object` are allocated. This because the *JVM* performs *Escape Analysis* optimization, that allows to allocate all the objects created in this benchmark on the stack memory rather than on the heap memory. Moreover, the memory management system and the garbage collector are core parts of the *JVM*, and they have been highly optimized in the last decade. As for the *Loop* benchmark, the same consideration made for the *Cast* benchmark can be asserted. In the *Math* benchmark, different results emerge: *EveC* has a maximum performance gain of 590% with respect to *Java*, and a maximum loss of 97%. It is important to notice that in the actual implementation the *EveC* version of the `lang.Math` class is only a wrapper to the mathematical library of the *C* standard library. Some functions, like the calculation of the maximum or minimum between two numbers, can be

easily implemented with only one instruction, and the cost to call them is greater to the cost of their execution. This justifies some loss of performance in some cases in this benchmark.

In the *Method* benchmark, *Java* has better performances in comparison to *EveC*. An important consideration is that the *EveC* compiler does not perform any optimization for synchronized method call or statement, while in *Java* this constructs are deeply studied and optimized. Moreover, as previously explained, the *JVM* performs *Devirtualization* and *Method Inlining* in order to reduce the cost of method calls.

Table D.9 presents the results for scientific and numerical computational benchmarks. In all of these benchmarks, with the exclusion of the *SOR* and *Heap Sort* benchmark, *EveC* is faster then *Java*, with a peak of performance gain of 227% in the *Series* benchmark.

In table D.10, the results for all the *jBYTE* benchmarks are reported. It can be noted that *EveC* has a performance gain with respect to *Java* in three of the eight benchmarks, with a peak gain of 68% in the *Bitfield Operations* benchmark. The worst performance of *EveC* is in the *FP Emulation* benchmark. The remaining benchmarks show a slightly better performance of *Java*.

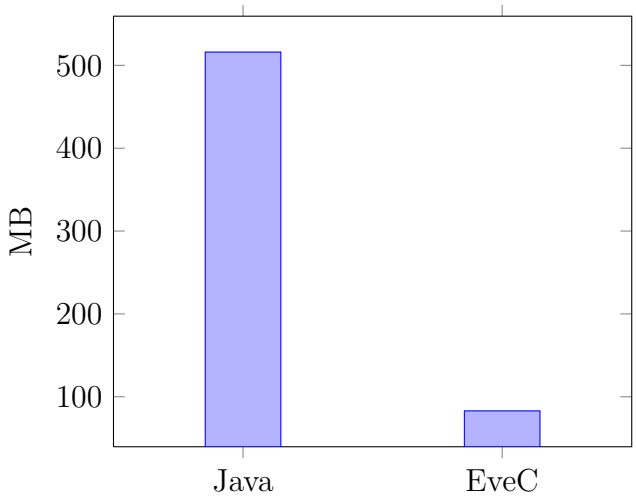
The results for the *Simplex* benchmark are presented in table D.12. Different dimensions for the input data are used and defined in table D.11. The results report the time (in milliseconds) necessary to find the optimal solution, that is checked to be the same for the *EveC* version and for the *Java* version, allowing the verification of the correctness of the code produced by the *EveC* compiler. Thus, in table D.12, lowest values mean better performances.

The last column of table D.11 indicates the ratio *Java*/*EveC*, and thus a higher value in this column means a higher performance gain. It can be noted that *EveC* is always faster than *Java* for the *Simplex* benchmark. For small data, the difference is also bigger. Small data means lower execution time, and this means that the *JIT* of the *JVM* does not perform aggressive optimizations, in contrast to what happen for longer run. Despite this, also for huge data of run G (10000 constraints and 10000 variables correspond to more than 2 Gigabytes of allocated memory) *EveC* is 4% faster than *Java*.

As regards the *Prime Sieve* benchmark, *Java* employs 35470 milliseconds, while *EveC* employs 35892. Hence, the performance of unoptimized *EveC* is very close to *Java*.

In the following, an analysis of the peak memory used for all the benchmarks both for *EveC* and *Java* is presented. All the values are reported in Megabytes, so lower values means a lower memory consumption. To get this results, the memory usage is sampled every 0.1 seconds during benchmark

Figure 5.1: Memory usage peak in the *Java Grande Benchmark* for *Java* and *EveC*.



executions. It is obtained using the standard *Unix* command `ps`.

Figure 5.1 and figure 5.2 reports the peak memory used respectively for the *Java Grande Benchmark* and for *jBYTE*. For the *Java Grande Benchmark*, the peak memory usage is 516 MB for *Java*, and only 83 MB for *EveC*. For *jBYTE*, the peak memory usage is 45 MB for *Java*, and 14 MB for *EveC*. As regards *Prime Sieve*, the peak memory usage is respectively 1931 MB and 1913 MB for *Java* and *EveC*.

In figure 5.3, the peak of memory used in function of the data dimension for the *Simplex* benchmark is presented. For a better comparison between the memory used by *Java* and *EveC*, both memory usage are plotted in the graph. In all of the different runs, the memory consumption of *EveC* is lower than the one of *Java*.

5.5 Final Results

In this section, the results of the same benchmarks executed by *EveC* with all the optimizations enabled are presented. The aim is to show the boost the optimizations give to *EveC* and the final gain obtained with respect to *Java*.

From table D.13 to D.21, the results of the *Java Grande Benchmarks* are presented. The first column of each table indicates the performances of *EveC* compiled without activating any optimization, while the second column reports the performances of *EveC* compiled with all the optimizations

Figure 5.2: Memory usage peak in the *jBYTE* benchmark for *Java* and *EveC*.

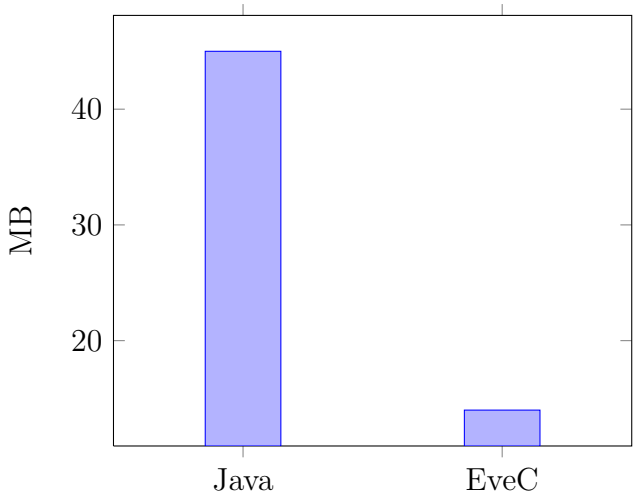
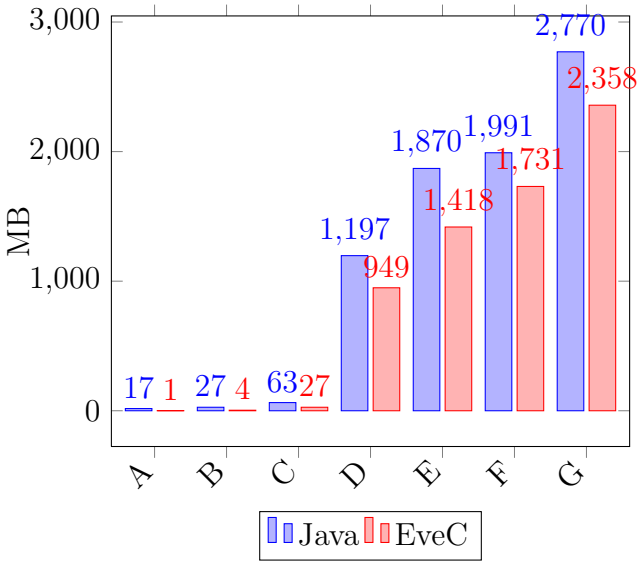


Figure 5.3: Memory usage peak in the *Simplex* benchmark for *Java* and *EveC*, using the different data dimension defined in table D.11.



enabled. The third and the fourth columns indicate respectively the ratio between the performances of *EveC* optimized and *EveC* not optimized, and the ratio between the performances of *EveC* optimized and *Java*. For all of these results, higher values mean better results. With the exception of some particular benchmarks, the optimizations do not allow to obtain significant better performances. This is an expected result, because a great number of these benchmarks are micro-benchmarks, which test the performance of only a particular construct, that is not affected by the optimizations. It is important to notice the performance gain in the *Method* benchmark, with a peak of more than 400% with respect to *EveC* without optimizations. In this case, the *Devirtualization* and the *Method Inlining* work well, allowing such a performance gain. These optimizations allow also reducing the gap from *Java*, that still remains consistent. The gap between *Java* and *EveC* is caused by the overhead introduced for any method call by the implementation of the *Stack Trace*. Table D.23 shows the results of the *Method* benchmark with the *Stack Trace* disabled. It can be noticed that, with the *Stack Trace* disabled, the *EveC* version of the benchmark is extremely faster than *Java*.

In the *Create* benchmark there is an significant performance gain with respect to *EveC* without optimizations due to the *Escape Analysis*. In fact, all of the object created can be trivially allocated on the stack memory instead that on the heap memory.

As said before, the implementation of the `lang.Math` class is only a wrapper to the mathematical function of the standard *C* library. Since the library is compiled into native code, it is not possible to perform optimizations on it. In table D.22 the results of the *Math* benchmark of the *Java Grande Benchmarks*, compiled using the source code of the `lang.Math` class instead that the pre-compiled library, are reported. It can be noted that using the source code there is a general performance gain with a peak of 488% with respect to *EveC* using the compiled library.

In table D.24, the results of the *jBYTE* benchmark executed by *EveC* with all optimizations activated are presented. It can be noted that the optimizations allow a performance gain with respect to the not optimized *EveC* version. As can be noted, there are some better results and some worse outcomes with respect to *Java*. Table D.25 shows the gain *EveC* obtains by eliminating bounds checks, thus corrupting the safeness of the language. This result is not significant per-se, but it represents an upper bound for the performance gain achievable through the optimization of the bound-check code.

In table D.26, the results of the *Simplex* benchmark executed by *EveC* with all the optimizations activated, and with the same data size defined in table D.11, are presented. As for table D.12, the results show the time in

milliseconds necessary to find the optimal solution and, thus, a lower value means a better result. The last two columns indicates respectively the ratio *EveC/EveC Opt* and *Java/EveC Opt*, that represent respectively the performance gain of optimized *EveC* with respect to non-optimized *EveC*, and optimized *EveC* with respect to *Java*. It can be noted that the optimizations have some effects, allowing an average performance gain of 2% with respect to non-optimized *EveC*. This improvement affects also the performance gain with respect to *Java*, that is 6% for the larger dataset tested.

The optimization allows also a performance gain in the *Prime Sieve* benchmark. The optimized *EveC* version of the benchmark is executed in 34418 ms, that is more than one second less than the unoptimized version (35892 ms). The performance gain is also enough to reach a better performance with respect to *Java* (35470 ms).

The peak memory usage for the optimized version of *EveC* is substantial the same than the non-optimized version of *EveC*. Thus, the results presented in section 5.4 are also correct for the optimized version of *EveC* and are not reported for the sake of brevity.

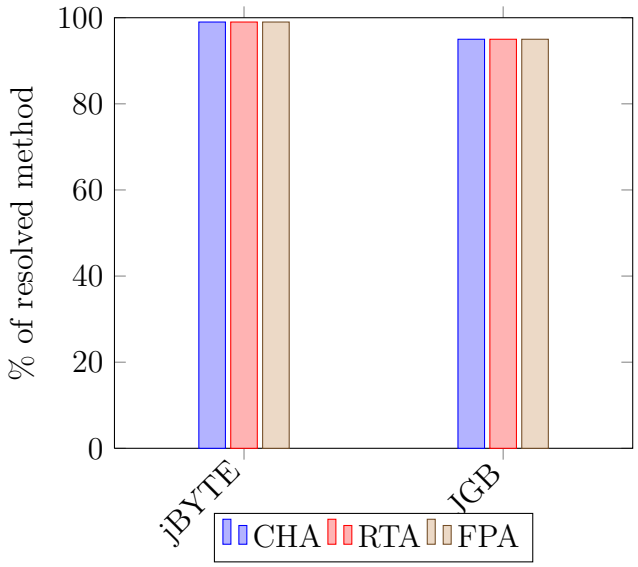
5.6 Devirtualization

As mentioned before, the *Devirtualization* optimization alone does not provide any performance gain, because the method call cost in not as expensive as in other programming languages. However, the *Devirtualization* is essential because it allows further optimizations, like *Method Inlining*. For this reason, in figure 5.4 only the percentage number of methods that has been resolved at compile time is presented. More precisely, the percentage number of resolved methods are presented for the different optimization, that are *CHA*, *RTA* and *FPA*. It can be noted that *CHA* manages to resolve all the resolvable methods for both the tested benchmarks. This optimization has been tested with only these two benchmarks because it was not useful to test it on the other benchmarks presented in this thesis. This is because these benchmarks do not use any complex class hierarchy; hence the optimization was not needed.

5.7 Array Explosion

Since each time an array is accessed different operations on it are performed, the presence of indirection accessing it can cause a loss in execution performances. In order to reduce the affect of this issue, *Array Explosion* has been

Figure 5.4: Method resolved for the *CHA*, *RTA* and *FPA* optimization.



implemented, as described in section 4.5. The results of this optimization is presented in table D.27. This table presents the results of the *Simplex* method, comparing *EveC* with no optimization enabled and *EveC* with only the *Array Explosion* optimization enabled. It can be noted that the *Array Explosion* optimization allow a performance gain in all of the different run of the benchmark.

5.8 Method Inlining

Before testing the effectiveness of *Method Inlining*, a consideration has to be made: as already underlined in section 4.3, this optimization is implemented basically in order to improve the score of the other optimizations such as *Escape Analysis*, which results are shown in the next section. This because, *Method Inlining* is implemented in most of *ANSI-C* compilers, including the *GCC*. In fact, in order to test the effectiveness of this optimization implemented in the *EveC* compiler, the corresponding optimization performed by the *GCC* have to be turned off. In this way, the real effectiveness of the *EveC* implementation of *Method Inlining* can be estimated, and also compared with the *GCC* implementation. The benchmark chosen for this task is the *Method* benchmark of the *Java Grande Benchmarks*, which determines the cost of a method call (see section 5.3.1). This benchmark consists in a series of loops in which different types of method are called: static, non-static, final, ab-

stract and synchronized methods. Table D.28 shows the average results of ten runs of the presented benchmark. The first column indicates the average number of operations performed per second concerning the runs without *Method Inlining*; the second one illustrates the same result for the runs with the *EveC* implementation of *Method Inlining*, while the third shows the ratio between these values.

The results are comparable to the ones obtained with the *Method Inlining* algorithm implemented by the *GCC* (see table D.20): this means that the *Method Inlining* algorithm implemented in the *EveC* compiler is, at least for this benchmark, as effective as the one implemented in the *GCC*, with a maximum gain of performance amounting at 433%. The only kind of method calls that are not affected by this optimization are the synchronized methods, either static or non-static. A first idea about reason for this results is that the lock and unlock operations performed for each synchronized method call causes a stall in the operations executed by the hardware processor. However, this is a non-verified hypothesis which has to be more deeply analysed and studied when the multi-threading feature will be implemented in *EveC* system.

5.9 Escape Analysis

The effectiveness of such an optimization can be measured calculating two different parameters:

- The percentage of object allocated on stack with respect to the total number of objects.
- The effective gain on the total run time.

As for the first parameter, it has been tested only on the *Java Grande Benchmarks*, because it is the only tested benchmark that utilized a huge number of objects. The *Escape Analysis* has been able to allocate on the stack only the 3% of the objects found without the help of the *Method Inlining* optimization; on the contrary, with the help of such an optimization, *Escape Analysis* has been able to avoid heap allocation for the 35% of the objects. This result highlights the importance of *Method Inlining* optimization in order to restrain the life-time of the object into the creating function. However this performance is negatively affected by the *Method Inlining* limits. These limits influence *Escape Analysis* in the sense that a less-restricted *Method Inlining* algorithm would result in the possibility to allocate a greater number of objects on the stack. In particular, a considerable negative influence come

from the unavailability of the source code of methods and constructors of all library classes, that makes these types of objects escape the creating function when such methods are called. In order to estimate the real effectiveness of the allocation of an object on the heap, *Escape Analysis* has been tested on the *Create* benchmark of the *Java Grande Benchmarks*, that measures the cost of allocating different type of object (see section 5.3.1). The table that summarizes the results of the benchmark is table D.17 in section 5.5, which shows that for all types of objects created, *Escape Analysis* optimization determines a gain of the performance amounting roughly at 300%. The only kind of object for which the optimization does not improve the performance is the most complex object, whose member is another object. Since the low level benchmark considered consists only of an allocation statement within a loop, *Bound Check Optimization* and *Method Inlining* do not affect performance. This means that the performance gain registered is completely imputable to *Escape Analysis*.

5.10 BCO results

5.10.1 Benchmarks

In order to evaluate the performance of the *Bound Check Optimization* the right benchmarks have to be chosen. These benchmarks must trivially contain array accesses inside a loop to make possible the optimization. A very common problem with this characteristic in literature is the sorting of an array [55]. The chosen benchmarks are algorithms that solve these kind of issues. In order to prove the usefulness of the optimization, *Bubble Sort* and *Insertion Sort* have been chosen, because they can be fully optimized. On the contrary, an algorithm such as *Quick Sort* cannot be optimized as described further on. In the next sections the algorithms chosen will be shortly introduced.

Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements “bubble” to the top of the list. It performs the sorting of an array with a quadratic performance with respect to the size of the array. The pseudo-code of the algorithm is presented in listing 5.3.

Listing 5.3: *Bubble Sort* algorithm's pseudo-code.

```
public static int[] bubbleSort(int [] intArray)
{
  long n = intArray.length;
    int temp = 0;
    int i = 0;
  while(i < n){
        for(int j=1; j < (n-i); j++){
                if(intArray[j-1] > intArray[j]){
                  temp = intArray[j-1];
                    intArray[j-1] = intArray[j];
                    intArray[j] = temp;
                }
        }
        i++;
    }
    return intArray;
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. *Insertion sort* iterates, consuming one input element each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position. The pseudo-code of the algorithm is presented in listing 5.4.

Listing 5.4: *Insertion Sort* algorithm's pseudo-code.

```
public static void insertionSort(int [] intArray)
{
  int i, j, index;
  for(i = 1; i < intArray.length; i++)
  {
    index = intArray[i];
    int a = i;
    for(j = 0; j < i; j++)
    {
      a = i;
      if(intArray[a-j-1] > index)
        break;
      intArray[a-j] = intArray[a-j -1];
    }
    intArray[a-j] = index;
  }
}
```

Quick Sort

Quick Sort is a "divide and conquer" sorting algorithm. In fact it recursively divides the array into two parts by selecting a *pivot* element. Then it searches in each array an element that is in wrong position with respect to the *pivot* (hence for the first part of the array, it searches an element that is greater then the pivot if considering ascending sorting; vice versa for the second part of the array). Once found the two elements it swaps them and repeats the procedure until all the element are in the right position with respect to the pivot element. Then the algorithm is recursively applied to the two parts of the array. The pseudo-code of the algorithm is reported in listing 5.5. *Quick Sort* algorithm cannot be optimized by the current implementation of the BCO because the loops are all *while* loops. It has been added to the benchmarks to highlight the importance of enlarge the cases that can be optimized.

Listing 5.5: *Quick Sort* algorithm's pseudo-code.

```
public void sort(int[] values) {
  if (values ==null || values.length==0){
    return;
  }
  this.numbers = values;
  number = values.length;
  quicksort(0, number - 1);
}


private void quicksort(int low, int high) {
  int i = low, j = high;
  int pivot = numbers[low + (high-low)/2];
  while (i <= j) {
    while (numbers[i] < pivot) {
          i++;
      }
    while (numbers[j] > pivot) {
          j--;
      }
        if (i <= j) {
          swap(i, j);
          i++;
          j--;
      }
  }
  if (low < j)
    quicksort(low, j);
  if (i < high)
    quicksort(i, high);
}
```

5.10.2 Results

The results show the time, in milliseconds, that the program took to execute and are calculated on the average of ten run of each program. *Bubble Sort* and *Insertion Sort* have been run with an array size of $100,000$, while for the *Quick Sort* algorithm a size of $1,000,000$ has been used in order to obtain

more significant results (the *Quick Sort* algorithm is faster than the other two; hence it has been possible to test it with a greater array size).

Table D.29 shows the comparison of the results of the non-optimized *EveC* version of the algorithms and the *Java* version. As can be noted, *Java* has a little advantage on all the algorithms (7.5% for the first, 9.9% for the second and 3.8% for the third.

The table D.30 shows the comparison between *EveC* and *EveC* with the *Bound Check Optimization*. In *Bubble Sort* the boost is 39.7% and in *Insertion Sort* is 30.4%. On the contrary, as for *Quick Sort*, practically there is no gain (actually there is a little gain of 7% because in the calculation of execution time the initialization of the array is included and hence optimized).

Finally, the table D.31 shows the comparisons between optimized *EveC* and *Java*. Particularly significant are the results in *Bubble Sort* and *Insertion Sort*, with a gain of respectively the 29.2% and 17.6%.

The last table (D.32) shows the results obtained by *EveC* without bound checks. This represents the hard upper limit for any *Bound Check Optimization*.

Chapter 6

Conclusions and future work

In the first part of this chapter, possible improvements applicable to each *EveC* optimization are discussed, while in the last one final considerations are reported.

6.1 Method Inlining improvements

In order to improve *Method Inlining*, there are two possible ways:

- Enhancing and validating the inlining policy.
- Removing, as much as possible, the limits of the algorithm.

As for the policy, the aim is to make it more sophisticated and more selective, in order to improve effectiveness. For now, it is affected only by two parameters: an estimation of the code size, and the number of call sites found in the program. Moreover the decision on which functions have to be inlined is made statically: once decided the thresholds for both the parameters, the algorithm will inline all those functions that do not exceed the thresholds, without exceptions. A step in this way could be done by introducing some features. Firstly, a context-aware analysis could be implemented in order to find spots in which the call sites are performed a certain number of times. The aim is to consider call sites differently, according to the context in which they are found: a function call within a loop will be generally performed more times than one located into a conditional branch. The result should be a more selective policy according to which a particular procedure may be inlined at a certain call site (for instance within loops), and not at others. Moreover, since other optimizations heavily rely on *Method Inlining*, as already introduced in section 4.3, the policy should be addressed also to them.

For instance, it should be able to recognize spots in which inlining a function ensues new opportunities for allocating objects on the stack and, in this case, to inline also functions whose size exceeds the default threshold. However, some limits derives from the way in which *Method Inlining* is implemented: as analysed in section 4.3, in the current implementation there is no possibility to inline functions which present a return statement within a loop. In order to eliminate this issue, there is the possibility to handle in a different way the return of a macro: using labels and `goto` statements, instead of the `break` statement, could permit to inline also this kind of functions. This feature has not been implemented so far because it produces a further issue. The idea is to replace the `break` statement, which substitutes the return statement in the standard function, with a `goto` statement, that would lead to a label declared at the end of the macro, in order to preserve the original behaviour of the code. Nevertheless, if a macro built in this way is called more than once, the *C* preprocessor replaces, at each site, the call to the macro with its body, with the result that the label declared in it is duplicated. This would result in a compiling error generated by the *GCC*, because it is not possible in the *C* language to declare the same label more than once. However there is a solution to this issue too: in order for the label to have, at each call to the macro, a different name, a further parameter could be added to the macro, that would take the place of the label name. Moreover, in the current implementation, all the parameters passed by copy to a function, have to be manually passed by copy to the corresponding macro too, in order to obtain the same behaviour (see paragraph 4.3.3). This could introduce an overhead due to the definitions inserted and should be avoided whenever possible: an improvement in that sense can be implemented by analysing, for each parameter passed by copy, if it is somehow edited in the macro body or not.

6.2 Escape Analysis improvements

The improvements proposed for the *Escape Analysis* optimization concern the enhancement of the algorithm that decide whether or not an object can be allocated on the stack as well as the integration of this optimization with other features that rely on it. As for the first type of improvements proposed, the aim is to allocate on stack as much objects as possible, without altering the behaviour of the program. To reach this result, the first sophistication proposed is to "follow" the object, even when it is passed to a function as an argument, analysing the operation performed by that function on it. As listing 6.1 shows, in the current implementation of the *Es-*

cape Analysis optimization, the object `o` is labelled as escaping `function`.

Listing 6.1: Object passed to a non-inlined function.

```
void function()
{
  struct Object * o = SuperMalloc(sizeof(Object));
  otherFunction(o);
}
```

This is a conservative assumption that can be a limit to the effectiveness of the algorithm and can be avoided by analysing the operations performed by `otherFunction` on `o`, whenever possible (i.e. when the source code of `otherFunction` is available). As well as enhancing the effectiveness of the algorithm itself, *Escape Analysis* can be improved by upgrading it with some other features like *Scalar Replacement* [5]. When is verified that a certain object does not escape the creating function, besides allocating it on the stack, its fields can be replaced by scalar variables. In this way, the compiler can eliminate both the allocation of memory on the heap and the initialization of the object. Moreover, the access to a scalar variable is cheaper than the access of a field, because it does not require any dereferencing. The last upgrade proposed for such an optimization is *Synchronization Removal*. Besides analysing if an object does escape a function, can also be verified if an object does escape the thread: if an object is accessed only by a thread, synchronization operations on it can be safely omitted because it will never be locked by another thread.

6.3 BCO improvements

In order to improve the *Bound Check Optimization* algorithm, the set loops that can be optimized has to be extended. To obtain this, three methods are possible:

- Normalizing loops.
- Handling more subscript function forms.
- Handling *while* and *do-while* loops.

These methods, described in the next sections, have not been implemented yet.

Loops normalization

The current implementation of the BCO algorithm is able to optimize only loops with a particular structure:

```
for(i = l; i ≤ u; i++){
    B(i);
}
```

The only normalization currently implemented is the case in which the upper bound is defined with $<$ and not with $\leq$. There are two other possible normalizations. The first concerns the increment of the iteration variable. In case that the increment is not a single unit one (e.g. `i++`) there exists a normalization that transforms the original loop into a new one that has the same behaviour than the previous one but with a single unit increment of the iteration variable. The transformation is the following:

$$\left|\begin{array}{l} for(i = l; i \leq u; i = i + s)\{ \\ \quad B(i) \\ \} \end{array}\right| \Longrightarrow \left|\begin{array}{l} for(i = 0; i \leq \lfloor \frac{u-l}{s} \rfloor; i++)\{ \\ \quad B(l + is) \\ \} \end{array}\right|$$

On the other hand, if the variation of the iteration variable is negative there is another transformation that generates a new loop with a positive stride with the same behaviour with respect to the original:

$$\left|\begin{array}{l} for(i = u; i \geq l; i = i - s)\{ \\ \quad B(i) \\ \} \end{array}\right| \Longrightarrow \left|\begin{array}{l} for(i = l; i \leq u; i = i + s)\{ \\ \quad B(u + l - i)) \\ \} \end{array}\right|$$

Handling non linear subscript functions

In the current implementation of the algorithm, as already said, only linear subscript are handled. The are methods to handle different subscript functions, such as modulo functions. The procedure to handle these cases is described in [50].

Handling *while/do-while* loops

The current version of the BCO handles only *for* loops. This is because, as already said, in a *for* loop it is easy to compute the loop bounds and the increment step of the iteration variable, and it is also easy to find the iteration variable. The difficulty that resides in *while* or *do-while* loops is the scattering

of the information about them. In order to proceed with the optimization of this kind of loops, the first step to execute is to find the iteration variable: usually it can be found inside the condition for the loop termination (but there can be more than one condition and variable; in this case the loop cannot be optimized). Once the iteration variable and the upper loop bound are found, it must be recovered the lower bound loop. This operation is the most difficult to execute, as the definition of the lower bound loop is before the loop itself (and may be far from the loop). In addition the increment step of the iteration variable must be recovered inside the body loop. It is also important its position with respect to the array accesses. All of these issues make this operation much more complicated than the optimization of a *for* loop, even if it is theoretically possible.

6.4 General considerations

The presented work, consisting of the development of a new programming language, *EveC*, and its corresponding compiler, have been fully implemented, and have reached a satisfying level of stability, with an adequate set of high-level optimizations, developed as well as the compiler in this project. Besides performing other tests in order to validate and stabilize the system even more, it needs no adjustments to be utilized for the development of real applications. Moreover, despite the fact that the *EveC* programming language does not support all the features implemented in the huge *Java* standard library system, the system developed is ready to be installed on all machines that support an *ANSI-C* compiler and a *posix* environment. The obtained results are encouraging: most of the *EveC* implementations of the tested benchmarks proved to have at least similar, when not better, computational performances than the corresponding *Java* implementation. However, not all the scores are better than the ones that came out from the *Java* implementations. This result is expected at least for three reasons: first, at present not all the features implemented, e.g. the synchronized methods, have been optimized; in addition, the *EveC* compiler relies on a off-the-shelf garbage collector that is not always well-suited for the characteristics of the language; moreover, as proved in chapter 5, the current implementation of the *Stack Trace* causes a significant overhead on the execution of certain applications.

In addition, *EveC* proved to use constantly less memory than *Java*, both for macro and micro-benchmarks. This result is particularly significant in those systems where it is important to monitor the amount of consumed memory, or the available memory is limited. The overall results achieved demonstrate the validity of the different approach to object-oriented lan-

guages pursued in this project, and that a static compilation technique can lead to similar or better results than the traditional *JIT* compilation technique at the state of the art. Despite the fact that the system built in this project is already usable as it is, there are lots of improvements to be implemented yet, in order to guarantee complete stability and achieve better performances. Since the development of a programming language and its corresponding compiler is quite a huge work for a thesis, there is still abundant space for future improvements. For sake of brevity, in this section are presented only the enhancements that, according to experience and experiments performed, would be most effective both in terms of performance and in terms of usability:

- Implementation of a garbage collector tailored to the *EveC* programming language.
- Improvement of the current library system and porting of more libraries from *Java* standard library.
- Extension of the high-level optimization set.
- Porting of the system on *non-posix* environments, e.g. *MS Windows*.

As already introduced, using a pre-made garbage collector entails an important drawback: since it is not tailored to an object-oriented language, the collector does not consider the possible availability of information about objects to be allocated (i.e. it is conservative). Therefore, a first improvement is to modify the garbage collector in order to feed it with the object type information necessary to perform a precise garbage track. Another option is to develop a dedicated garbage collector from scratch, conceived from the start for the *EveC* programming language, in order to exploit all the available informations and avoiding unnecessary operations. A high priority must be given to the improvement of the current library system. For now, since libraries are compiled into native code, their source code is not available for any *EveC* application. Hence, both *EveC* compiler's and *GCC* optimizations are limited, because they cannot optimize library code. A solution in order to reduce this limit can be the replacement of the libraries pre-compiled into native code with their *C* source code. Moreover, this solution brings another advantage: the usage of the source code for the libraries make them architecture-independent and, as a consequence, easier to distribute. However, as already said, the libraries currently available for *EveC* are just a sub-set of the *Java* standard library. This means that the *EveC* programming language does not support yet all the features of *Java* like the thread

sub-system or networking facilities. Hence, also the porting of as many missing libraries as possible is a high priority task to be fulfilled. Moreover, the introduction of new features entails the necessity of optimizations concerning them. As instance, multi-threading introduces the need of synchronization operations to be performed over objects, that, however, are not necessary if an object is not shared between two or more threads. This check can be performed by upgrading the *Escape Analysis* optimization (see section 6.2). Hence, future work must include the extension of the existing optimizations in order to make them suite also to newly introduced features. Nevertheless, the set of high-level optimizations implemented so far can be extended in order to produce more optimized *C* code: *Automatic Object Inlining* could reduce the access costs of fields and arrays by improving the cache behaviour and by eliminating unnecessary field loads [5]; in addition *Automatic Object Colocation* could change at *run-time* the order of objects on the heap so that objects accessed together are consecutively placed in memory.
As already described, the *EveC* system can work on a system that support an *ANSI-C* compiler and a *posix* environment. In order to make it available for different operating systems, it is important to port *EveC* also to *non-posix* environments, like the widespread *Windows* system.
Moreover, through all the possible machines that could exploit the *EveC* programming language, also another important family of devices must be taken into account: since the memory used by *EveC* is considerably lower than the one used by *Java*, a natural application of the *EveC* programming language could be the entire ecosystems of embedded devices, where the memory usage is a priority. In this ecosystem, *EveC* could fit particularly well, because it can bring a simpler, faster and safer way to build an application than a traditional unsafe language like *C*, performing efficient computation both in terms of memory consumption and execution time.

Appendix A

Programming Language Specifications

A.1 The programming language

In this section the definition of the language created is reported, and the basic constructs

This section reports the definition of the created language, highlighting the constructs that is possible to define in the *EveC* programming language and providing examples of their usage when necessary.

A.1.1 Comments

There are two different types of comment. They are defined as in *C/C++*: text starting with the `/*` character is treated as a comment and ignored. The comment ends at the first `*/` character. Moreover, a single line comment is also defined: text starting with the `//` character is treated as a comment and ignored until the end of the line.

A.1.2 Class

EveC is an object oriented language, therefore everything is an object, with the only exception of some primitive types. An object is an instance of a class. A class is defined as follows:

```
public class ClassName {

//all the code class is here
```

```
}
```

This code shows the definition of a class named `ClassName`. A class can be declared as *public*, *protected* or *private*. This represents the visibility of the class outside the package in which it is declared.

Moreover, a class can be also defined as *final* or *abstract*, introducing the corresponding keyword between `public/protected/private` and `class`. A class defined as *final* cannot be extended by a subclass, while an `abstract` class cannot be instantiated. A class can contain attributes and methods, that are described later.

An *EveC* file is defined with the *.ec* extension and can contain only a single class. Moreover, the class must have the same name as the belonging file.

A.1.3 Packages

The *EveC* programming language allows the definition of *Package*s. A *Package* is a collection of classes, interfaces and other *Package*s. The *Package* in which a class or an interface is included must be defined before the class or the interface definition and before any *import* statement and it must be unique. The *Package* must be defined as follows:

```
package PackageName;
```

where `package` is a keyword, and `PackageName` specifies the *Package* in which the class or the interface belong. `PackageName` must start with a letter, followed by letters, numbers or the `_` character.

Thanks to the *Package*s, classes and interfaces can be referenced with a complete name, as described as follows:

```
PackageName.ClassName
```

where `PackageName` is the name of the *Package* in which the class or the interface is defined, and `ClassName` is the name of the referenced class or interface. For example, if the class `SomeClass` is defined in the `SomePackage` *Package*, the complete name is `SomeClass.SomePackage`.

It is necessary that each class and interface have different complete name. Thus, it is possible to define two classes or interfaces with the same name if they belong to different packages: the correct class or interface can be referenced using the complete name or, if not specify, the last defined class or interface is referenced. For instance, consider the case of the existence of two different classes, whose complete names are, respectively, `PackageA.SomeClass`

and `OtherPackage.SomeClass`; `PackageA.SomeClass`, in this case, is defined before `OtherPackage.SomeClass`:

```
...
SomeClass obj;
...
PackageA.SomeClass other;
```

The object `obj` is defined as `OtherPackage.SomeClass`, because the complete name is not specified. On the contrary, `other` is defined with the complete name `PackageA.SomeClass`, and thus it is an instance of the class `PackageA.SomeClass`. A *Package* must correspond to a physical folder or directory. For instance, if the class `SomeClass` is defined in the package `SomePackage`, than a folder named *SomePackage* must exist and the file containing the implementation of the class `SomeClass`, which must be named as *SomeClass.ec*, must exist in that folder.

Sub-packages

As defined before, a *Package* can contains also other *Package*s, which are called *Sub-package*s. A class or interface can belong to a *Sub-package* if the definition of its *Package* is the following:

```
package OuterPackage.InnerPackage;
```

where `OuterPackage` is the name of the external *Package* and `InnerPackage` is the name of the *Sub-package*. There is no limitation in the number of *Sub-package*s allowed.

A *Sub-package* must correspond to a physical folder or directory. For example, if `SomePackage` has a *Sub-package* named `Subpkt`, then a folder named *SomePackage* must exist and a folder named *Subpkt* must be placed inside the *SomePackage* folder.

A.1.4 Import statement

The import statement allows to import classes and interfaces defined both in the same package and in different packages. It is possible to import a class or interface as follows:

```
import localpath.folder.subfolder.classA;
import localpath.differentfolder.*;
import |lang.*;
```

The *import* statement must be introduced before the class or the interface definition. It is possible to import either only a single class file or all the classes in a folder. In the first case, the file name must be specified, while in the second, it is used the `*` character at the end of the import statement in order to import all the classes in the specified folder. The `.` character is necessary in order to specify the path of the file to import. For instance, if it is necessary to import the class file "src/classes/classA.ec", the import statement is "src.classes.classA". Moreover, it is possible to import either non-compiled or library classes; in addition, introducing the | character at the start of the *import* statement, the compiler will search in the *classpath* for the class (or the interface) to be imported.

A.1.5 Primitive Types

The primitive types are the only exception of the object-oriented model. A primitive type can be a variable, an attribute, an argument of a method or a returned type of a method. The primitive types are the following:

- `boolean`, assuming only the *true* or *false* value.
- `byte`, a signed integer number of 1 byte, representing numbers from -2^7 to $2^7 - 1$.
- `short`, a signed integer number of 2 byte, representing numbers from -2^{15} to $2^{15} - 1$.
- `int`, a signed integer number of 4 byte, representing numbers from -2^{31} to $2^{31} - 1$.
- `long`, a signed integer number of 8 byte, representing numbers from -2^{63} to $2^{63} - 1$.
- `ubyte`, an unsigned integer number of 1 byte, representing numbers from 0 to $2^8 - 1$.
- `ushort`, an unsigned integer number of 2 byte, representing numbers from 0 to $2^{16} - 1$.
- `uint`, an unsigned integer number of 4 byte, representing numbers from 0 to $2^{32} - 1$.
- `ulong`, an unsigned integer number of 8 byte, representing numbers from 0 to $2^{64} - 1$.

- `float`, a single precision 4 byte floating point number.
- `double`, a double precision 8 byte floating point number.
- `char`, a single UNICODE text character.

A.1.6 Attributes

An attribute is an instance or a global variable of the class and is defined as follows:

```
AccessLevel Modifier Type AttributeName;
```

where `AccessLevel` can be one among the following keyword:

- `public`, the attribute is accessible from the outside of the class.
- `private`, the attribute is not accessible from the outside of the class.
- `protected`, the attribute is accessible only from the inside of the class and from the inside of its derived classes.

and `Modifier` can be one, both or none of the following keyword:

- `final`, the attribute can be initialized only in the construction methods or when declared.
- `static`, the attribute belongs to the class rather than to one of its instances.

`Type` is the type of the attribute, which can be a primitive type or a class; `AttributeName` is the name of the attribute, which must start with a letter, and can contain either upper- or lower-case letters, numbers and the `_` character.

Moreover, an attribute can be also initialized when declared in the class, as follows:

```
public class ClassName {

public int Number = 28;
private SomeObject obj = new SomeObject();

}
```

A.1.7 Methods

A method is a subroutine or a function associated to a class, and can access and edit all the members of the class itself. It can be declared either static or non-static. In the first case, it is called by an instance of the class, while in the second by the class itself, and has no access to non-static member or methods.

A method is defined as follows:

```
AccessLevel Modifier RetType MethodName(Type arg0, Type arg1, ...)  {
...
}
```

where `AccessLevel` can be one among the following keywords:

- `public`: the method will be accessible from the outside of the class;
- `private`: the method will not be accessible from the outside of the class;
- `protected`: the method will be accessible only from the inside of the class and from the inside of its derived classes.

and `Modifier` can be one among the following keywords:

- `final`: the method cannot be overloaded by a subclass.
- `static`: the method belongs to the class instead of an object.
- `abstract`: the method body is not defined and its a responsibility of subclasses to implement it; a class with at least an abstract method must be declared as abstract.

Moreover a method can also be defined as synchronized, introducing the corresponding keyword (`synchronized`) between `AccessLevel` and `RetType`. If a method is defined with this option, only one thread per time can access this method through a shared object.

`RetType` is the type of the object (or the *primitive* type) that the method returns at the end of its execution. If `RetType` is equivalent to `void`, nothing is returned by the method.

`MethodName` is the name of the method and must start with a letter.

In the round brackets, the list of arguments is defined. The arguments are separated by the `,` character, `Type` is the type of the argument, which is followed by the argument name. An argument can be defined as `final`, so it cannot be instantiated or assigned in the method.

Method Overloading

In *EveC* it is possible to define more than one method with the same name but with different arguments list. The compiler will choose the correct method to invoke.

In the actual implementation, the overloading has some limitations:

- A non-static method cannot be overloaded by a static method and vice versa.
- A method cannot be overloaded by a method with different access rights.

Constructors

Constructors are special methods that are called when a new instance of a class is created. The method name must be equal to the class name. It is not allowed to explicitly call a constructor method. Constructors differ from the normal method definition because it is not necessary to define a return type and modifier keywords are not allowed. Furthermore, it is possible to overload constructors just like it happens for methods.

The static constructor

To initialize a static attribute it is possible to define a static constructor as follows:

```
static  {
//initialization of static attributes here.
}
```

that is very useful if a more complex initialization of static attributes is needed.

The compiler ensures that the call of the static constructor is performed before the first access to any static field, the first call of a static methods or the first call of any construct of the class.

A.1.8 Programming Constructs

In this parts, *EveC* programming language constructs are described. They can be used into any method body.

Block

It is possible to create a new code block with a new scope as follows:

```
{
...
}
```

If a variable is declared in the new block, it is not accessible from outside the block.

Variable declaration

It is possible to declare a variable as follows:

```
(final) Type VariableName;
```

where `final`, when present, means that the variable cannot be modified once initialized; `Type` is the variable type that can be a *primitive type* or a class and `VariableName` is the name of the variable.

Moreover, in *EveC*, it is also possible to define more than one variable as follows:

```
Modifier Type VariableName1, VariableName2, ...;
```

Every variable assignment can be used to initialize the variable declared.

Variable assignment

A variable can be assigned as follows:

```
VariableName = assign;
```

where `assign` can be a mathematical expression, or another object.

Only an expression that has a compatible type with the variable type can be assigned to a variable. For instance, it is possible to assign a `byte` variable to a `long` variable. The opposite is not allowed. Formally, the following operations are allowed:

- assign a variable of a primitive type to a variable of another primitive type that has a bigger or equal dimension.
- assign an integer variable of any dimension to a *float* or *double* variable;

- assign an object of a class `A` to a variable of a class `B` only if `A` is a derived class of `B`;
- assign an object of a class `A` to a variable that is an interface only if class `A` implements that interface.

It is possible to supplant these strong rules with an explicit type cast. The type cast can be used with both *primitive* types and classes. In the first case, the type cast is checked at compile type. In the second case, the type cast is checked at *run-time*. If the type cast is not allowed, a proper exception is raised.

The type cast is defined as follows:

```
(SecondType)variable;
```

where `SecondType` is the type in which the variable is converted.

Besides the `=` operator, the other assignment operators of *EveC* are: The assignment operator `=` can be combined with others operator with the following syntax:

```
variable1 (op)= variable 2;
```

with the same meaning as:

```
variable1 = variable1 (op) variable2;
```

where `(op)` can be one of the mathematical operators.

Mathematical expression

A mathematical expression is an expression formed by one or more mathematical operators. Operators are special symbols that perform specific operations on one or two operands. The following list enumerates all the operators:

- `+`, performing the sum of the variables.
- `-`, performing the subtraction of the variables.
- `&`, performing a bitwise *AND* between the variables.
- |, performing a bitwise *OR* between the variables.
- $\sim$, performing a bitwise *NOT* between the variables.
- `^` that makes a bitwise *XOR* between the variables.

- `*`, performing the multiplication of the variables.
- `/`, performing the division of the variables.
- `%`, performing the remainder of the division of the variables.
- <<, performing a left shift operation between the variables.
- >>, performing a right shift operation between the variables.
- `expr++`, `expr--`, that are, respectively, the increment operator (increments a value by 1) and the decrement operator (decrements a value by 1) performed after the evaluation of *expr*.
- `++expr`, `--expr`, that are, respectively, the increment operator (increments a value by 1) and the decrement operator (decrements a value by 1) performed before the evaluation of *expr*.

The following construct must be used to assign a value to an attribute of the actual object: The proper syntax in order to assign a value to an attribute of an object is the following:

```
object.AttributeName (op)= assign;
```

Inside the definition of a method belonging to a certain class, the object that will call that method can be referenced through the *this* keyword, that represents a pointer to the object itself. As a consequence, a member of that object can be assigned, in one of the class methods, with the following syntax:

```
this.AttributeName = assign;
```

If in the actual scope there is no definition of variable with the same name as `AttributeName`, the keyword `this` can be omitted.

On the contrary, in order to assign a new value to a static attribute of a class, the statement is defined as follows:

```
ClassName.attribute = assign;
```

The new statement

The new statement allows to create a new instance of a class and the proper syntax is the following:

```
new ClassName(arg0, arg1, ...);
```

It can be assigned to a variable, passed as an argument when calling a method, returned by a method or found as an instruction itself. When an instance of the class `A`, that extends class `B`, is created using the new statement, if the constructor of the class `B` with no arguments is defined, it is implicitly called by the constructor of the `A` class, in order to initialize the attributes of `B`.

There is no need to explicitly deallocate the memory used for instantiate a new object. The *Runtime System* provides a *Garbage Collection* mechanism that reclaims unreachable objects.

If-then and if-then-else statement

Also the *if-then* and *if-then-else* statements are available with the typical *C* syntax:

```
//if-then
if (expression) {
...
}

//if-then-else
if (expression) {
...
} else {
...
}
```

where `expression` is a boolean expression. The defined boolean operators are the same defined in the *C* language:

- < check if the first expression is less then the second.
- > check if the first expression is greater then the second.
- <= check if the first expression is less or equal to the second.
- >= check if the first expression is greater or equal to the second.

- == check if the first expression is equal to the second.
- != check if the first expression is not equal to the second.

Moreover, two boolean expressions can be combined with the following operators:

- `&&`, the condition is satisfied only if both the expressions are true.
- ||, the condition is satisfied if at least one of the expressions is true.

It is possible to invert the value of expression using the `!` operator before the expression.

The language also define an *inline-if*, that can be used with expression. It is defined as follows:

```
(expression) ? value1 : value2;
```

If `expression` is true, the result is `value1`, otherwise it is `value2`.

The switch-case statement

EveC also presents the *switch-case* statement, as *C*, with the following syntax:

```
switch (IntegerVariable) {
case 0: { ... } break;
....
default: { ... } break;
}
```

In the current specification of the *EveC* language, it can be used only with an integer variable.

The for loop statement

A *Java-like for* loop is defined as follows:

```
for (varDefinition;conditional expression; operation expression) {
....
}
```

where `varDefinition` is a definition of a variable used inside the loop block or an assign to a previously defined variable. This field can be empty, but the `;` character must be specified anyway. `conditional expression` is as the one defined in the *if-then* statement or it can be empty. As before, the `;` is compulsory. `operation expression` is a mathematical expression.

The while-do loop statement

The *while-do* loop is defined as follows:

```
while (expression) {
....
}
```

where `expression` is defined as in the *if-then* statement.

The do-while loop statement

The do-while loop is defined as follows:

```
do  {
...
}
while (expression);
```

where `expression` is defined as in the if-then statement. The difference with respect to the *while-do* loop is that, in the *do-while* loop, the condition is checked after each cycle, rather than before.

Break and continue statements

The language defines, just like *Java*, a *break* and a *continue* statements, that can be found only within a loop. As for the *break* statement, it terminates the execution of the nearest enclosing loop in which it appears. Control passes to the statement that follows the terminated loop. On the contrary, the *continue* statement forces transfer of control to the controlling expression of the smallest enclosing loop. Any remaining statement in the current iteration is not executed. The next iteration of the loop is determined as follows:

- In a do or while loop, the next iteration starts by re-evaluating the controlling expression of the do or while statement.
- In a for loop it causes first `operation` expression to be executed. Then `conditional expression` is re-evaluated and, depending on the result, the loop either terminates or another iteration occurs.

They are defined as follows:

```
//break statement
break;

//continue statement
continue;
```

Method call

It is possible to call a method as follows:

```
ObjectName.MethodName(arg0, arg1, ...);
```

where `ObjectName` is an instance of a class, `MethodName` is a public method defined in that class and `arg0`, `arg1` are the arguments of the method call.

Moreover, a static method can be called as follows:

```
ClassName.StaticMethod(arg0, arg1, ...);
```

where `ClassName` is the class and `StaticMethod` is a public static method defined in `ClassName`.

Moreover, in a constructor method, it is possible to use some special methods, as follows:

```
super(arg0, arg1, ...);
this(arg0, arg1, ...);
```

Calling the `super` function, means calling the constructor of the father class. Calling the `this` function, means calling the constructor of the current class with the correct augments list.

The return statement

The return statement is used to end the execution of a method and it is defined as follows:

```
return expression;
```

where the `expression` must be the same type as defined in the method signature. If the method is defined to return a `void` type, expression must be empty. `expression` can be a mathematical expression, a new statement or a *primitive* type value (that is a number, a char or a boolean).

Synchronized statement

In a multi-threading environment, it is very important to ensure that only one thread per time can access an object. In *EveC*, this can be obtained declaring a method as *synchronized*. In this case, all the instructions on the method body are parts of the critical section. Otherwise, it is possible to define a smaller critical section. This can be done using the synchronized statement as follow:

```
synchronized(object) {
...
}
```

where `object` is a shared object used as a lock. Only one thread per time can acquire the lock and enter the critical section. The lock can be the object that is modified in the critical section, or a different shared object. It is recommended to declare the lock object as final, to ensure that it is not reassigned.

The instanceof statement

The *instanceof* statement can be used to check if an object belongs to a certain class. It can be used as an expression in an *if-then* statement or in a loop. It is defined as follows:

```
if (objectA instanceof ClassName)  {
...
}
```

In the depicted case it is used in an *if-then* statement. `objectA` is the object to be checked and `ClassName` is the name of the class. If `objectA` is a `ClassName` object, than the expression is true; otherwise, it is false. It is important to say that if the `ClassName` is a superclass of the `objectA` the expression will return true. Moreover, the expression will return true also if `ClassName` is an interface and `objectA` implements it.

A.1.9 Inheritance

In *EveC* it is possible to create a new class using an existing one as base. The new class will inherit methods and attributes that are declared as public or protected in the extended one. Thus, the new class extends the old one, and the old class is named the *superclass* of the new class.

A class can extends only one single class, but can be extended by none, one or more classes.

Assuming the existence of a class named `BaseClass`, a new class can extend the `BaseClass` as follows:

```
public class NewClass extends BaseClass {
...
}
```

In *EveC*, all classes extend a class, with the only exception of the `Object` class, that is the ancestor class. Therefore, if the father class is not explicitly defined, the class will extend the `Object` class.

Method Overriding

It is possible for a class to declare its own implementation for public or protected method of the father class. To do this, a method with the same name, arguments, access level and modifier of the father class' method must be declared and implemented. This feature is called method overriding.

When a new instance of the new class is created and the overridden method is called, the method implemented in the new class is called instead of the implementation of the father class.

Extending an abstract class

When an abstract class is extended, it is necessary to implement all the method declared as abstract. If this is not performed, also partially, the new class must be declared as abstract and so it cannot be instantiated.

Polymorphism

EveC implements a standard polymorphism mechanism as the one implemented in *Java*. Consider the case that two classes exist, `NewClass` and `BaseClass`, and `NewClass` extends `BaseClass`. `NewClass` overrides the public method of the class `BaseClass` named `Method`. Then, the following code is allowed:

```
...
BaseClass obj = new NewClass();
obj.Method();
...
```

a new object of type `NewClass` is created and it is assigned to a variable of type `BaseClass`. Then, the method `Method` is called. Since `obj` is defined as `NewClass` and `NewClass` overrides the method of `BaseClass`, the implementation of the `NewClass` for the method `Method` is called, rather than the implementation of the `BaseClass`.

A.1.10 Interfaces

The *EveC* language also allows to define interfaces. An interface is a contract, that forces a class to a pre-determined behaviour. When a class implements

an interface, it signs a contract in which it is committed to implement all the methods defined in the interface. An interface is defined as follows:

```
AccessLevel interface InterfaceName {
//method signatures here
}
```

where `AccessLevel` can be one among `public`, `protected` and `private`, and specify the visibility property for the interface. `InterfaceName` is the name of the defined interface.

A method in an interface must be defined as follows:

```
RetType MethodName(Type0 arg0, Type1 arg1, ...);
```

where `RetType` is the return type of the method, and can be a class, an interface or a primitive type. `MethodName` is the name of the method, `Type0` is the type of the first argument, `arg0` is the name of the first arguments, and so on. It is important to notice that the `AccessLevel` element is not present. In fact, all the methods defined in the interface must be public. An interface can also contains static methods and attributes as a class. For both static attributes and static methods, the same rules as the ones of the classes are applied.

An interface can be extended by another interface, as for class. It must be done as follows:

```
public interface NewInterface extends BaseInterface {

...

}
```

where `NewInterface` is the name of the new interface and `BaseInterface` is the interface that is extended. When a class implements an interface that extends another interface, it must implement not only the methods defined in the first interface, but also the ones in the extended interface. The difference with respect to inheritance of classes is that an interface can extend also more than one single interface. The multiple extension is defined as follows:

```
public interface NewInterface extends BaseInterface1,
BaseInterface2, ... , BaseInterfaceN {

...

}
```

Using interface

An interface can be used everywhere a class can be used. The only difference is that an interface cannot be instantiated. For example, the following code is not allowed:

```
public class SomeClass {

public void SomeMethod() {
NewInterface Interface = new NewInterface();
}

}
```

Rather than creating a `NewInterface` object, it is allowed to create a new instance of a class that implements the `NewInterface` interface. Thus, assuming that a class named `ClassA` implements `NewInterface`, the following code is allowed:

```
...
NewInterface Interface = new ClassA();
...
```

It is also possible to call an interface method implemented by an instance of an object. Assuming that `ClassA` implements `NewInterface`, and that `NewInterface` defines a method named `Method`, the following code is allowed:

```
...
NewInterface Interface = new ClassA();
Interface.Method();
...
```

A.1.11 Nasted Classes and Inner Classes

A *nested class* is a class defined inside another class or interface, with the following syntax:

```
class OuterClass{
...
public static class NestedClass {
...
}
...
}
```

The *nested class* has the same property of any other class. The difference is that a *nested class* inherits the *Package* of the outer class. More precisely, if the *package* of the outer class is `Pack`, and the outer class is called `Outer`, then the *package* of the *nested class* is `Pack.Outer`.

*inner class*es differ to standard and *nested class*es since they are associated with the instance of the class in which they are created. Thanks to this property, *inner class*es can access the attributes and the methods of the outer class in which are defined, always referring to an instance of the outer class. An *inner class* is defined as follows:

```
class OuterClass{
...
public class InnerClass {
...
}
...
}
```

Since *inner class*es must be associated with an instance of the outer class, they can be instantiated only inside the outer class in which are defined using the standard new statement, or outside the outer class using a particular new statement. For example, let `InnerClass` be defined as an *inner class* of `OuterClass`. Outside the context of `OuterClass`, it is possible to instantiate an object of type `InnerClass` in the following way:

```
OuterClass foo;
...
InnerClass obj = foo.new InnerClass(...);
...
```

A.1.12 Error Handling and Exception

EveC defines an exception system that can be used for error handling. Instead of using, for instance, the return value of a method to check if it was executed correctly, it is possible to isolate the code that can report an error or that can have an abnormal execution in a try block. If something goes wrong during the execution of the try block, the execution path is transferred to the catch block where the error or the abnormal execution can be handled. It is also possible to execute a finally block, in order, for example, to make a clean up. The finally block is executed either in the case that the try block is executed correctly, or that the catch block is executed.

The definition of the *try/catch/finally* statements is the following:

```
try {
...
} catch (Exception e) {
...
} finally {
...
}
```

It is important to say that the finally block is optional. Moreover, it is possible to define more than one catch block for one try block, in order to differently handle different types of exception. To transfer the execution from the try block to the catch block, a new exception must be throw. An exception is defined as an instance of class that extends at least the `lang.Throwable` class. The throw statement must be followed by a new statement that create a new instance of a class. For example, assuming that the class named `IOException` extends the `lang.Exception` class which, in turn, extends the `lang.Throwable` class. The throw statement is the follow:

```
throw new IOException();
```

A throw statement can be used directly in the try block or not. In the second case, a method that can throw an exception must have an additional part in its definition:

```
AccessLevel Modifier RetType MethodName(Type arg0,
Type arg1, ...) throws ExceptionType0, ...  {
...
}
```

that is, the method must explicitly define the types of the exceptions that can throw. In *EveC*, only two types of exceptions are directly derived from `lang.Throwable`. The first one is `lang.Exception`, while the second is `Error`. The other types of existing exceptions are specialized exceptions that are derived from these two classes. The main difference between `Exception` and `Error` is that an `Error` cannot be never caught and it causes an abnormal exit of the program execution. Another important exception type is the `RuntimeException`, that extends the `Exception` class. Exceptions that cannot be caught are called unchecked exceptions, and they all extend the `Error`, All the other exceptions are named checked exceptions. Only checked exception must be declared when the method is defined if they are not caught in the method body.

In *EveC*, it is also allowed to define new exceptions, that extend an existing exception.

As introduced before, different catch blocks check different exceptions: the *Runtime System* examines the catch blocks in the order in witch they are written; in addition, the *EveC* language also gives the possibility to use nested try/catch blocks. Finally, for the sake of completeness, it is also possible to disable the exception system, specifying it explicitly to the compiler.

A.1.13 Array

EveC defines a linear memory space that can contain a fixed number of elements of the same type, that is an *array*. The type of an array can be a *primitive type*, a class or an interface. It is possible to define at *run-time* the dimension of the *Array* and, as a consequence, the number of element that it can contain. The syntax used to create a new *array* is the following:

```
Type[] array = new Type[Size];
```

where `Type` is the type of the element and `Size` is the dimension. Just like *Java*, *EveC* provide a way to assign and to read an element of the *array*:

```
array[i] = new Type(); //assigning
Type obj = array[i]; //reading
```

In the *EveC* programming language, an attribute, called *size* is always associated with an *array* and represents its dimension. Since the `size` attribute is a `long` value, it is possible to create array with up to $2^{63} - 1$ elements.

When the type of the elements of an *array* is not a *primitive type*, each element has to be allocated through the `new` statement. In fact, the creation of a new *Array* implies only the allocation of a chunk of memory big enough to contain the pointers to the objects. Thus, the constructor must be called for each element. It can be done, for example, with a loop:

```
Type[] array = new Type[Size];
for (long i=0; i<array.size; i++) {
array[i] = new Type();
}
```

If the type of the elements of the *Array* is a *primitive type*, it is also possible to create a new array as follows:

```
char[] digits = {
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
    'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
    'u', 'v', 'w', 'x', 'y', 'z'
  };
```

In this example, a new array of char is created and initialized with the elements between brackets. This initialization can be used only for the attributes of a class, but not for a local variable initialization.

The *EveC* programming language ensures that every access to an *array* element is safe, in the sense that it is possible to access only elements from 0 to $N-1$, where N is the length of the *array*. If an element outside the *array* is tried to be accessed, an `ArrayIndexOutOfBoundsException` is thrown.

Besides *arrays*, *EveC* also provides the possibility to define a bi-dimensional memory space that contains elements as in a *matrix*, and a three-dimensional memory space that contains elements as in a *volume*.

They can be created as follows:

```
//a new matrix of size Sizex * Sizey
Type[,] matrix = new Type[Sizex, Sizey];

//a new volume of size Sizex * Sizey * Sizez
Type[,,] volume = new Type[Sizex, Sizey, Sizez];
```

where `Type` is the type of elements and `Sizex` is the number of rows, `Sizey` is the number of colons and `Sizez` is the number of matrices. The proper syntax to assign and to read an element of a matrix or a volume is the following:

```
matrix[i,j] = new Type(); //assigning in a matrix
Type obj = matrix[i,j]; //reading in a matrix

volume[i,j,k] = new Type(); //assigning in a volume
Type obj = volume[i,j,k]; //reading in a volume
```

Since `sizex`, `sizey` and `sizez` are `long` value, it is possible to create *matrices* with up to $(2^{63}-1)*(2^{63}-1)$ elements and volumes with up to $(2^{63}-1)*(2^{63}-1)*(2^{63}-1)$ elements..

When the type of the element of the *matrix* (or a *volume*) is different to a *primitive type*, each element need to be initialized, as it happens for unidimensional *arrays*. Moreover, the *EveC* programming language ensures that every access to a *matrix* or to a *volume*, just like for *arrays*, is safe. If an element outside the range of a *matrix* or an *array* is tried to be accessed, an *ArrayIndexOutOfBoundsException* is thrown.

A.1.14 Generics

The *EveC* programming language allows to define special classes and interfaces, named respectively *Generics Classes* and *Generics Interfaces*. With *generics*, it is allowed to not specify a type of an object for attributes, arguments of methods and return type of methods. Instead of them, it is possible to use place-holders. When an object of a *Generic Class* or a *Generic Interface* is defined or instantiated, the type used for the place-holders of the class or interface must also be defined.

Generics are extremely useful in container classes. For example, consider the following `Box` class, that can contains one object:

```
public class Box {

private Object element;

public Box(Object element) {
this.element = element;
}

public Object get() {
return this.element;
}

}
```

Since the `Object` class is the ancestor class, every object can be contained in an instance of the `Box` class. For example, the following code is allowed:

```
Box box = new Box(new String("This is a string"));
```

However, to assign the element contained by the `Box` class to a `String` object a type cast is needed:

```
String element = (String)box.get();
```

because the method `get()` returns an `Object`, not a `String`. Thus, for the compiler, also the following code is correct:

```
DifferentType element = (DifferentType)box.get();
```

but the typecast will generate a `lang.ClassCastException` exception at runtime, because the element contained by the box is a `String`, not a `DifferentType` object.

The `Box` class can be rewritten using generics as follows:

```
public class Box<E>  {

private E element;

public Box(E element) {
this.element = element;
}

public E get() {
return this.element;
}

}
```

where `E` is the place-holder used instead of the `Object` class. It is possible to define an instance of the `Box` class for `String` element as follows:

```
Box box<String> = new Box<String>(new String("This is a string"));
```

In this case, the compiler check that the element passed in the constructor is a `String` element. Thus, the following code will generate a compile time error:

```
Box box<String> = new Box<String>(new DifferentType());
```

Moreover, in this way it is not necessary to cast the object returned by the *get()* method:

```
String element = box.get();
```

If, instead of `String`, element is an object of a different type, a compile time error raises.

For this reason, generics are a very powerful construct to verify programming error at compile time instead of runtime.

Place-holders in a Generic Class

A *Generic Class* (*Interface*) must have one or more place-holders. The place-holders must be defined after the class (interface) name within the angle brackets. Place-holders must be a single capital letter, and so the maximum number of place-holders which can be defined in a single *Generic Class* (*Interface*) is 26.

For convention, the place-holder E is used for an Element of the class, K for a Key and V for a Value.

Finally, it is important to say that when an instance of a *Generic Class* or an instance of a *Generic Interface* is created, the place-holder must be substituted only by class type, not primitive type.

Implementing a Generic Interface in a Class

A class can implement a *Generic Interface* replacing each place-holder of the *Generic Interface* with an object type:

```
public interface Comparable<E> {
...
}
```

a class can thus implements the interface `Comparable<String>`, but not the interface `Comparable<E>` or `Comparable`.

Implementing a Generic Interface in a Generic Class

A *Generic Class* can implement a *Generic Interface* defining the type of object for each place-holder of the *Generic Interface*. Moreover, also the place-holder of the *Generic Class* can be used to define the place-holder of the *Generic Interface*:

```
public interface Map<K,V> {
...
}
```

a *Generic Class*, for example `HashMap<K,V>`, can implement the interface `Map<K,V>` or, for example, the interface `Map<String,V>` or the interface `Map<K,String>` and so on.

Assignment of object in a generic class

Let the class `Animal` be extended by the class `Cat`:

```
Box<Animal> box = new Box<Father>(new Cat());

Father element = box.get();
```

This is allowed because an instance of `Cat` is also an instance of `Animal`, so the *is-a* relationship is not violated. Of course, the returned element of method `get()` is a `Father` class, not a `Cat` class, according to the definition of the `Box` class. In *Java*, a similar behaviour is reached using the keyword "extends" when defining the class used in the place-holder. Moreover, a *Generic Class* or a *Generic Interface* can be used to specify the place-holder of a *Generic Class* or a *Generic Interface*. For example, the following code is allowed:

```
Box<String> obj;
...
Box<Box<String>> box = new Box<Box<String>>(obj);
...
Box<String> element = box.get();
```

In this case, `box` is an object of type `Box<Box<String>>`, and thus the constructor accept as argument an object of type `Box<String>`, and the method `get()` returns an object of type `Box<String>`.

Generics and Inheritance

A *Generic Class* (*Interface*) can extend either a *Generic Class* (*Interface*) or a standard class (interface) and can be extended by either a *Generic Class* (*Interface*) or a standard class (interface), with the only assumption that the new class (interface) specify the place-holder of the extended class (interface).

For example, a class `HashMap<K,V>` can be extended by a class `SomeClass` as follows:

```
class SomeClass extends HashMap<String,OtherType> {
...
}
```

In this case, the `HashMap` class is extended using respectively types `String` and `OtherType` for the place-holders `K` and `V`.

Moreover, the class `HashMap<K,V` can be extended by a *Generic Class* `SomeOtherClass` as follows:

```
class SomeOtherClass<K> extends HashMap<String,K> {
...
}
```

In this second example, the class `SomeOtherClass` specify a place-holder that is used to extend the `HashMap` class. More precisely, the `HashMap` class is extended using respectively the type `String` and the place-holder defined in `SomeOtherClass` `K` for its place-holders `K` and `V`.

This examples are valid also for the interfaces, since the same rules apply.

A.1.15 The ClassPath and the libraries system

The *ClassPath* is a collection of classes and interfaces that provides some basic functionality needed by the complete environment. The *ClassPath* must include the package `lang`, which is formed by classes like `Object`, `String`, and so on.

EveC defines a standard format for libraries. A library consists of a single class or a single interface compiled into native code; it must be a compressed zip file, with the extension `.lib`. This archive will contain two or more files. The first one is contains the meta-data of the class or the interface, so it must contain its definition, methods and attributes. The second file is a platform dependent static library which can be used at *linking-time* when a program is compiled into native code. Moreover, other files can be included into the libraries and can be used by compilers to store additional information about the library.

A.1.16 EveC and C

The *EveC* programming language provides the possibility to add *C* code inside an *EveC* file, class or method. Since *C* is an unsafe language, if *C* code is inserted into *EveC* code, the whole program can be potentially unsafe. However, using *C* code is very useful in some situation, for instance when it is necessary to communicate directly with the operating system or the hardware. Some *ClassPath* classes include *C* code in order to interact with the operating system or the hardware.

Anyway, by default, it is not allowed to add *C* code inside an *EveC* file. In order to have this permission, it is necessary to explicitly inform the compiler that the file to be compiled contains unsafe code, through the correct flag.

Moreover, in order to let the compiler know where the unsafe code is placed, each *C* code line written must be inserted into a particular statement:

```
@unsafeCodeStart@
...
@unsafeCodeEnd@
```

Inside this statement, it is possible to write a normal *C* code, for example import a header file, define a function, write a function or a global variable.

Moreover, it is possible to interact with the *EveC* code in the following way:

- in a method body it is possible to access to all the variable defined: the *primitive type* variables are defined as normal variables, all the instance of classes are defined as pointer. They are all defined with the same name used in the *EveC* code. Moreover, it is also possible to access to all the attribute of a class, also the private ones.
- in a method body, it is possible to call a method as follows:

  ```
  obj-$>$MethodTable-$>$MethodNamei(\_evec\_env, obj, arg0, arg1, ...);
  ```

  as can be seen, it is necessary to add two arguments to the called function, the `_evec_env`, and the instance of the class. Moreover, the `i` at the end of the method name is a number generated by the compiler. If the method is not overloaded, `i` is equal to 0. Since it is not predictable to know, for the overloaded methods, the correspondence with the number `i`, it is recommended to call only the not overloaded methods.

On the contrary, it is never allowed to directly access to ones of the variable passed as argument to a method; it is never allowed to access or call a method using the `this` pointer; it is never allowed to access static attributes of classes

However, in the unsafe region it is not allowed to directly access to variables passed as argument to a method, nor to access or call a method using the `this` pointer. In order to perform this actions, an intermediate step is necessary:

```
public class SomeClass {
static int cont;
public void SomeMethod(int arg) {

int a = arg;
SomeClass c = this;
@unsafeCodeStart@
a++;
c->OtherMethod0(_evec_env, c);
@unsafeCodeEnd@
arg = a;
}
public void otherMethod() {
...
}

}
```

On the contrary, other actions are not allowed in the unsafe region: it is not possible to access a static attribute of the class nor to call static methods.

Appendix B

The EveC Development Plug-in

The *Plug-in Development Environment* (PDE) [56] provides tools to create, develop, test, debug, build and deploy *Eclipse* [11] plug-ins. Using this tool the *EveC Development Plug-in* (EDP) has been created. In the next section a description of how it works is provided.

B.1 The interface

The *EveC Development Plug-in* is an add-on integrated in the *Eclipse* environment and provides a handy user interface that allows to avoid the need of command line to compile and run an *EveC* project.

It provides many useful features, such as:

- automatic compiling
- automatic running
- automatic creation and editing projects and files (classes and interfaces)
- automatic indentation
- keyword highlighting
- addition of custom run and compile settings.

The interface, as already said, is fully integrated in the *Eclipse* environment as shown in figure B.1.

In order to start programming the *EveC* language it is sufficient to specify the path of the runnable *EveC* compiler, the classpath (i.e. the path of the standard library used by the programming language) and the *C* compiler

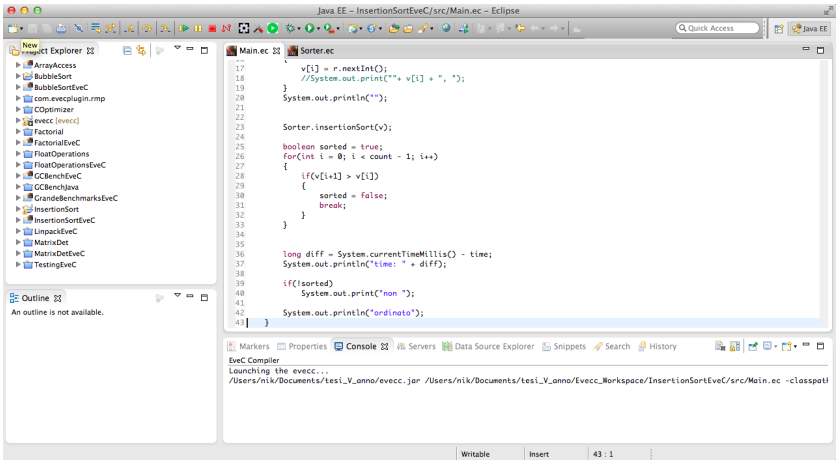


Figure B.1: The EDP interface.

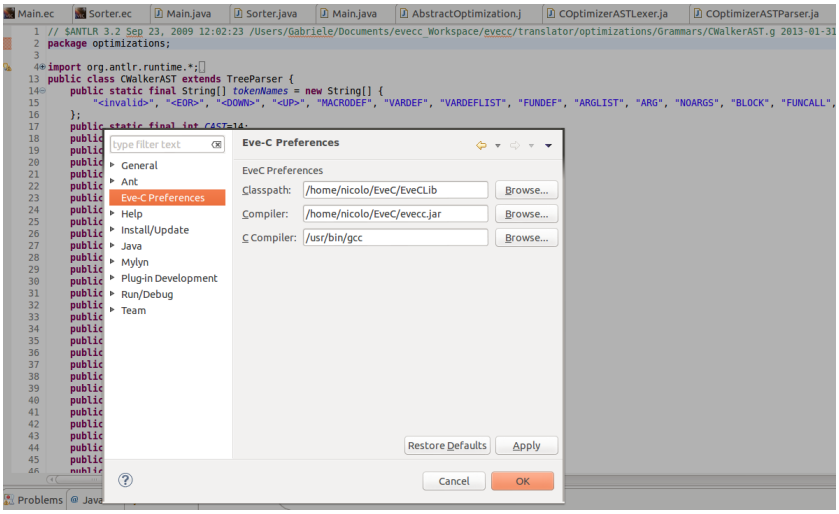


Figure B.2: The preference panel.

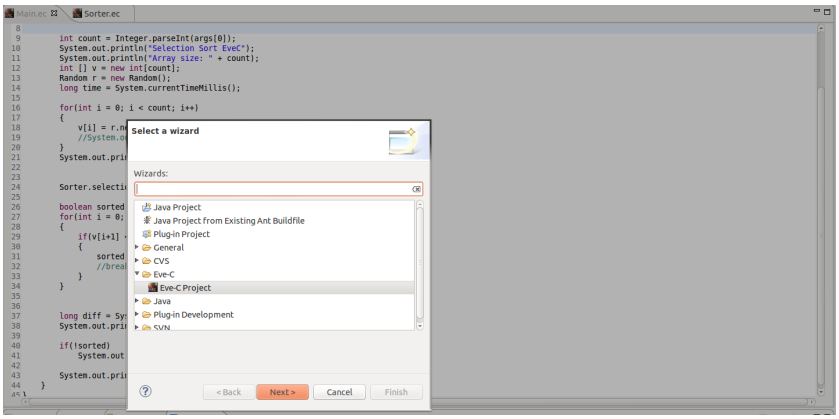


Figure B.3: The *New Project* wizard.

chosen. This can be done in the *EveC* preference panel inside the *Eclipse* environment, as shown in figure B.2.

Once specified such information, it is possible to start programming. To create a new project it is sufficient to click the *New EveC Project* button under the *File* menu (figure B.3).

Once created the new project and added some *EveC* files, which have the ".ec" extension, all the available actions are represented by the three buttons highlighted by a red rectangle in figure B.4.

The first button is used to set the properties of the project. It opens a window in which arguments can be passed to the compiler or to the running program; furthermore, *C* compiler optimizations and many other features can be disabled (figure B.5).

The other two buttons are used to compile and run the program. The console of a run of the program shown in listing B.1 is presented in figure B.6. The program simply print the numbers from 0 to 19.

Listing B.1: A simple *EveC* program that prints numbers from 0 to 19.

```
public class Main {

  public static void main(String[] args) {
  for(int i = 0; i < 20; i++){
      System.err.println(i);
    }
  }
}
```

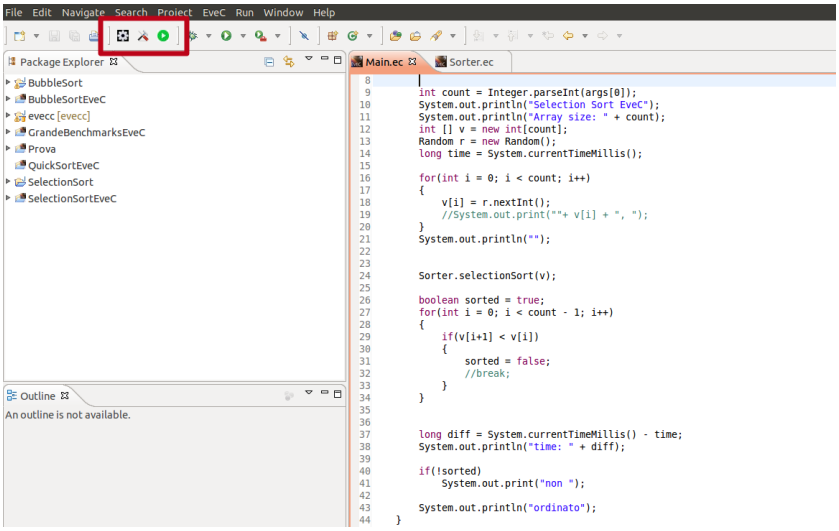


Figure B.4: The *EveC* buttons.

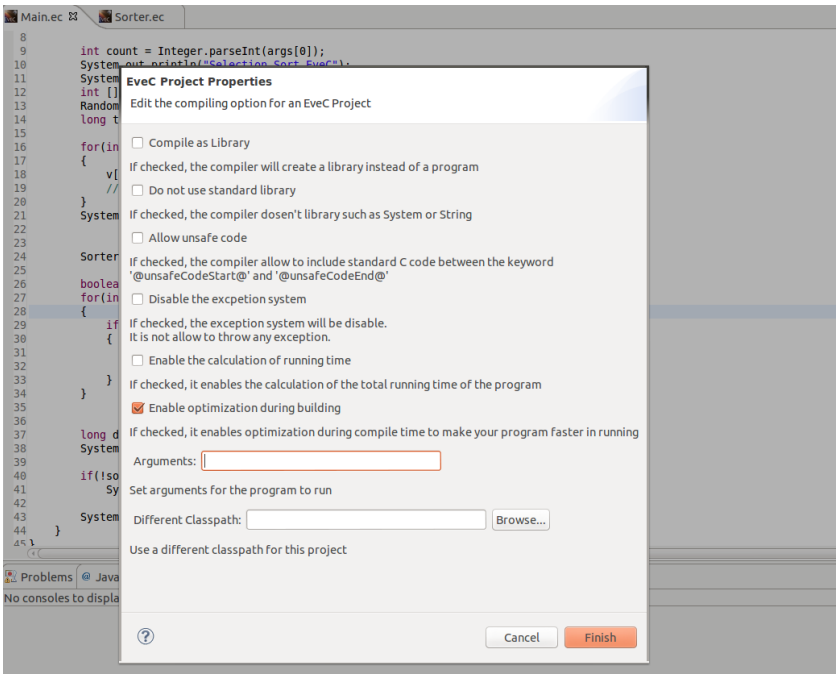


Figure B.5: The project properties panel.

Figure B.6: The console for a run of the program in listing B.1.

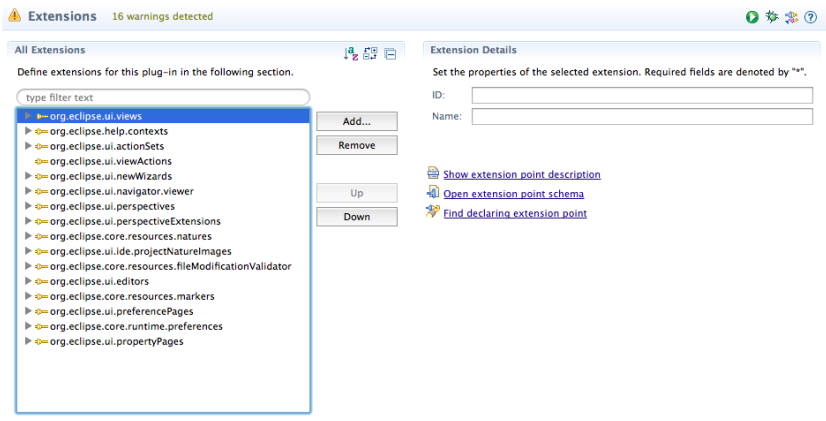


Figure B.7: The Extension Manager.

B.2 Developing an Eclipse Plug-in

With the *Plug-in Development Environment*, provided by the *Eclipse* community, it is easy to create a plug-in for a custom programming language. In fact, an easy-to-use tool, the *Extension Manager*, is provided in order to enrich the plug-in with new features (figure B.7). Once added a new characteristic, it is possible to customize it by modifying its behaviour via *Java* code. As an example, to add the project preferences' page, one just needs to add the corresponding extension; once that this has been done, the *Eclipse* environment creates a new set of *Java* files that provides the chosen characteristic: this is fully customizable, by modifying the user interface and the back end behaviour.

Once the plug-in is ready, it can be deployed using the *Export* button under the *File* menu. At the end of this operation it is sufficient to move the deployed plug-in file (with the *.jar* extension) into the *Eclipse* plug-in folder in order to start using it.

Appendix C

ANTLR

C.1 The ANTLR tool

ANTLR is a parser generator that automates the construction of language recognizers: in other words it generates a program that determines whether sentences conform to a language.Language recognition is much easier if it is broken it into two similar but distinct tasks or phases, mirroring the way a person reads English text. One does not read a sentence character by character; instead, the sentence is perceived as a stream of words. The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing grammatical structure. Thus the first mentioned task is the translation phase, called lexical analysis, operatating on the incoming character stream. The second phase is called parsing and operates on a stream of vocabulary symbols, called tokens, originating from the lexical analyzer. ANTLR automatically generates the lexical analyzer and parser by analyzing the provided *grammar*. A grammar is a sequence of rules that determine how the language to be recognized is structured. Performing a translation often means just embedding actions (code) within the grammar. ANTLR executes an action according to its position within the grammar. In this way, different code is executed for different phrases (sentence fragments). Some translations should be broken down into even more phases. Often the translation requires multiple passes: rather than reparse the input characters for each phase, it is more convenient to construct an intermediate form to be passed between phases. This intermediate form is usually a tree data structure, called abstract syntax tree (AST), and is a highly processed, condensed version of the input. The task of the following phases is to walke this tree in order to perform some optimizations of collect some information. A final phase, called the emitter, ultimately produces

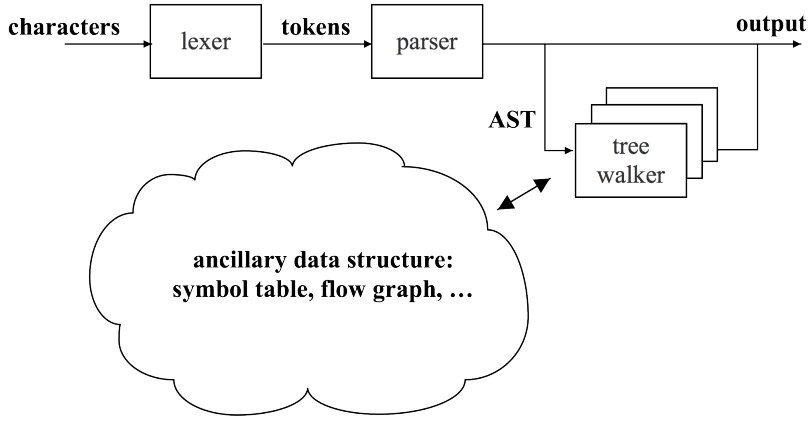


Figure C.1: Overall translation data flow; edges represent data structure flow, and squares represent translation phases.

the output using all the data structures and computations from previous phases. Figure C.1 illustrates the basic data flow of a translator that accepts characters and emits output. The lexical analyzer, or lexer, breaks up the input stream into tokens. The parser takes as input this token stream and tries to recognize the sentence structure. The simplest translators execute actions that immediately emit output, bypassing any further phases. More complicated translators use the parser only to construct ASTs. Multiple tree parsers (depth-first tree walkers) then walk the *AST*s, computing other data structures and information needed by future phases. Although it is not shown in this figure, the final emitter phase can use templates to generate structured text output. A template is just a text document with holes inside it that an emitter can fill with values. These holes can also be expressions that operate on the incoming data values. ANTLR formally integrates the *StringTemplate* engine to make it easier to build emitters. *StringTemplate* is a domain-specific language able to generate structured text from internal data structures.

C.2 Writing grammars with ANTLR

The best way to learn about how ANTLR works is to walk through a simple but useful example: the implementation of a simple grammar able to recognize mathematical expressions involving the sum, subtraction, multiplication and division operators with the usual precedence. The implementation of the translator is conceptually divided into two phases:

- Building the expression grammar.

- Adding actions to the grammar in order to evaluate expressions and emit results.

Only once implemented the translator, the parser grammar is modified to build an intermediate-form tree data structure, instead of immediately computing the result. Furthermore, a tree grammar is built, in order to walk along this tree, with the addition of actions to evaluate and print the result. The first phase of the implementation consists of building a sequence of rules that determine whose sentences conform to the language. The rules constituting the on-going example are shown in listing C.1.

Listing C.1: ANTLR grammar sample

```
prog :  stat+ ;

stat   :  (expr';');

expr :  multExpr (('+'|'-') multExpr)*;

multExpr :  atom (('*'|'/') atom)* ;

atom :  INT
   |  ID
     |  '(' expr ')' ;

ID :    ('a'..'z' |'A'..'Z' )+ ;
INT:    '0'..'9'+ ;
WS :    (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

The first rule indicates the overall structure that a program must have in order to conform to the language. This program has to be a sequence of `stat` rules. Each `stat` is an expression followed by the ";" token. The rule called `expr` represents a complete expression; it will match operators with the weakest precedence (sum and subtraction) and will refer to a rule, `multExpr`, that matches sub-expressions for operators with the immediately highest precedence, multiplication and division. The `atom` rule defines the entities to which the operators can be applied: such entities are integers, identifiers or mathematical expressions themselves, surrounded by brackets. The possibility to choose between one or another entity is implemented by the *OR* (`|`) operator. Turning to the lexical level, these vocabulary symbols (tokens) have to be defined. Any other white space is ignored. All lexical rules begin with an uppercase letter in ANTLR and typically refer to char-

acter and string literals, rather than to tokens, as the parser rules do. The operators `+` and `*` (that are different from the characters meaning the operations, which are surrounded by the `'` character) state that the rules on which they are applied (as instance the `stat` in the first rule, or the second part of the `expr` rule) can occur more than once. In particular, the former indicates that the rules can occur from once to infinite times, while the latter states that they can also not occur. After this explanation, it is clearly visible that the `INT` literal is formed by a sequence of digits that must occur at least once. Finally, rule `WS` (*whitespace*) is the only one associated to an action (skip();) telling ANTLR to throw out what it just matched and to look for another token. In this way all the spaces, tabulations and new line characters found in the input text are ignored. From the grammar, ANTLR generates a program recognizing valid expressions and automatically issuing errors for invalid expressions. To move from a recognizer to a translator or interpreter, some actions have to be added to the grammar:

- Define a *Hash Map* to store a variable-to-value map.
- Upon expression, print the result of evaluating it.
- Upon INT, return its integer value as a result.
- Upon ID, return the value stored in the memory corresponding to the variable represented by that ID. If the variable has not been defined, emit an error message.
- Upon parenthesized expressions, return the evaluation of the nested expression as a result.
- Upon multiplication of two atoms, return the multiplication of the two atoms' results.
- Upon addition of two multiplication (or division) sub-expressions, return the addition of the two sub-expressions' results.
- Upon subtraction of two multiplication (or division) sub-expressions, return the subtraction of the two sub-expressions' results.

These actions are implemented by embedding some Java instructions in the grammar, as shown by listing C.2.

Listing C.2: ANTLR grammar sample

```
@header {
import java.util.HashMap; }
@members {
      /** Map variable name to Integer object
         holding value */
      HashMap memory = new HashMap();
      }

prog: stat+ ;
stat:  // evaluate expr and emit result
       /* $expr.value is return attribute 'value'
          from expr call */
      (expr';') {System.out.println($expr.value);};

expr returns [int value]:
     e=multExpr {$value = $e.value;}
    ( '+' e=multExpr {$value += $e.value;} | '-' e=
       multExpr {$value -= $e.value;} )*;

multExpr returns [int value]
: e=atom {$value = $e.value;} ('*' e=atom {$value *=
    $e.value;} | '/' e=atom {$value /= $e.value;})*
   ;

atom returns [int value]:   /* value of an INT is
   the int computed from char sequence */
        INT {$value = Integer.parseInt($INT.text);}
    |   ID // variable reference
            {    // look up value of variable
           Integer v = (Integer)memory.get($ID.text)
              ;
                 /* if found, set return value else
                    error */
           if ( v!=null ) $value = v.intValue();
           else System.err.println("undefined
              variable "+$ID.text);
            }
  /* value of parenthesized expression is just the
     expr value */
  | '(' expr ')' {$value = $expr.value;} ;
ID :    ('a'..'z' |'A'..'Z' )+ ;
INT:    '0'..'9'+ ;
WS :    (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

For the rules involved in the expressions evaluation, it is really convenient to have them return the value of the subexpression they match. So, each rule matches and evaluates a piece of the expression, returning the result just like a method return value. As for the simplest sub-expression rule, the result of an `INT` atom is just the integer value of the `INT` token's text. An `INT` token with text 91 results in the value 91; on the contrary, the integer result of the `ID` rule, is the value of the corresponding variable stored in the *Hash Map*. If the variable is not found in the map, than it has not been defined and the program must return an error. Rule `atom`'s third alternative recursively invokes rule `expr`. This alternative introduces the possibility to modify the standard precendence of the mathematical operators through the brackets. Since, in thi case, there is nothing to compute, the result of this `atom` evaluation is just the result of the evaluation of `expr`. The `multExpr` rule matches an `atom` optionally followed by a sequence of `*` or `/` operators and `atom` operands. If there are no operators following the first `atom`, then the result of `multExpr` is just the `atom`'s result. For any multiplications or divisions following the first `atom`, the `multExpr` result, $*value*, is kept updated. Every time there are an operator and an `atom`, the `multExpr` result is multiplied or divided by the atom result. The actions for the `expr` rule, the outermost expression one, mirror the actions in `multExpr` except the operators.

C.3 Generating the Abstract Sintax Tree

The implemented program uses a grammar to match mathematical expressions and uses embedded actions to evaluate those expressions. However, for more complicated translations, there might be the need to build an intermediate representation of the input, which could be walked through and edited more easily, as in the case of the optimizations of this thesis. The parser grammar to be used is the same as before, described by listing C.1, with the replacement of the embedded actions with tree construction rules. Moreover, once built the tree, a tree parser is exploited to walk through it and to execute embedded actions. ANTLR automatically generates a tree parser starting from a tree grammar. The parser grammar converts a token stream into a tree to be parsed and evaluated by the tree grammar. Although the previously described approach is more straightforward, it does not scale well to a full programming language. Adding constructs such as function calls or while loops to the language means that the interpreter must execute the same bits of code more than once. Every time the input program invokes a method, the interpreter will have to re-parse that method. Such an approach

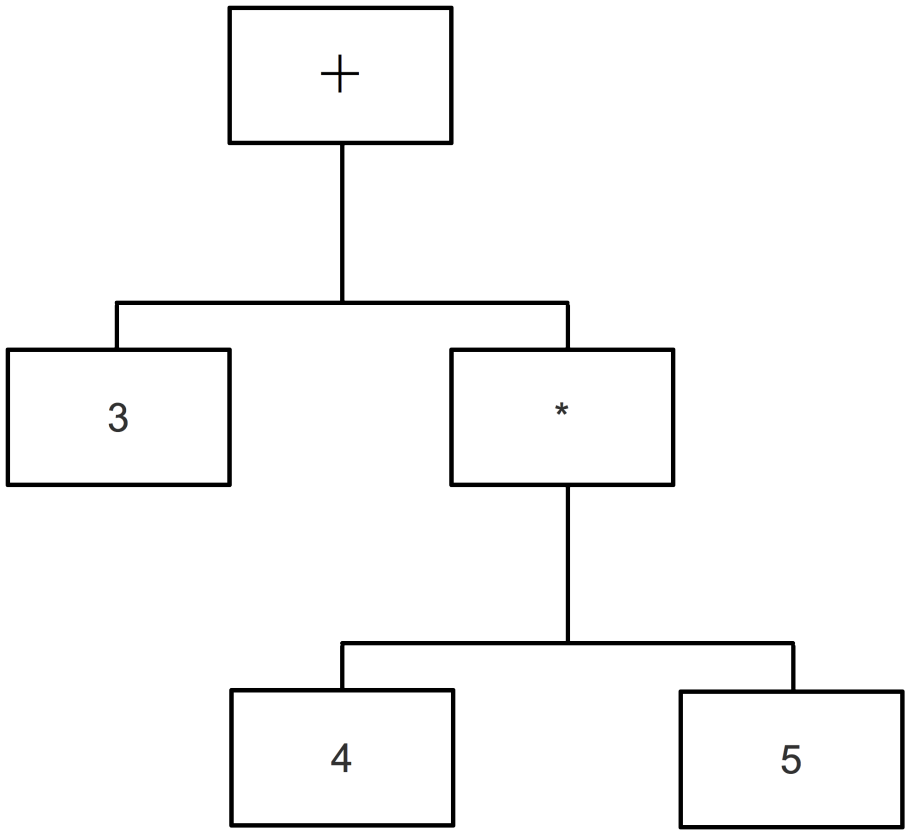


Figure C.2: Tree structure for the input sequence `3+4*5`

works but it is not as flexible as building an intermediate representation (e.g. an *AST*) and then walking that data structure to interpret expressions and assignments. Repeatedly walking an intermediate-form tree is much faster than re-parsing an input program. An intermediate representation is usually a tree and it records not only the input symbols but also the relationships among those symbols, as dictated by the grammatical structure. As depicted in figure C.2, the tree structure implicitly encodes the precedence of the operators: the multiplication must be done first because the addition operation needs the multiplication result as its right operand.

Therefore the properties that an intermediate *AST* representation should have are:

- Recording the meaningful, and only the meaningful, tokens.
- Encoding, in the two dimensional structure of the tree, the grammatical structure used by the parser to match the associated tokens but not the superfluous rule names themselves.

- Being easily navigated and recognized by a walker program.

In this way, an *AST* summarizes the essential information conveyed by the input program discarding the unnecessary one, thus allowing the walker program to recognize more easily the structure of the input program. For instance, consider the *C*-like statement in listing C.3: building the intermediate representation entails that the walker does not have to walk the whole sub-tree to be aware of what kind of statement is parsing. In fact, as shown in figure C.3, the walker just need to analyze the root node of the sub-tree in order to comprehend the nature of the statement. Without the intermediate representation a walker program has to walk the entire expression in order to comprehend if the statement is a function call rather than an assignment or another kind of statement. This example shows how an *AST* representation can make the interpretation of the code more efficient: walkers with different tasks can walk through it, not considering large parts of the tree according to their task.

Listing C.3: An indirect function call

```
a->b(5);
```

Building an *AST* with ANTLR is straightforward: it is sufficient to add *AST* construction rules to the parser grammar in order to indicate the chosen tree shape. Listing C.5 shows how to build it starting from the already implemented grammar. The beginning of the construction rule is always determined by the `->` operator and usually follows each grammar rule. However, by default, the parser adds nodes for all tokens to the tree as siblings of the current rule's root: all the `stat` tokens in the `prog` rule are siblings. Therefore in some cases it is not even necessary to write a construction rule, as for the `stat` rule. The only operator necessary in such a rule is the `!` one: it states that no node has to be created for the semicolon following the `expr` rule. The `^` operator has the task to build sub-trees, with a determined root and a series of children: in order to understand how it works, a new rule has been added to the sample grammar considered so far. In the grammar in listing C.5, there is the possibility to assign an expression to a variable. The construction rule of the second alternative for the `stat` rule states that the root of the sub-tree of an assignment statement must be the `=` token, while the `ID` and the `expr` tokens are, respectively, the first and the second child. This kind of structure for such a simple grammar is sufficient in order for the tree walker to distinguish the assignment statement from all the others, but in a more complex grammar it cannot be sufficient. For instance,

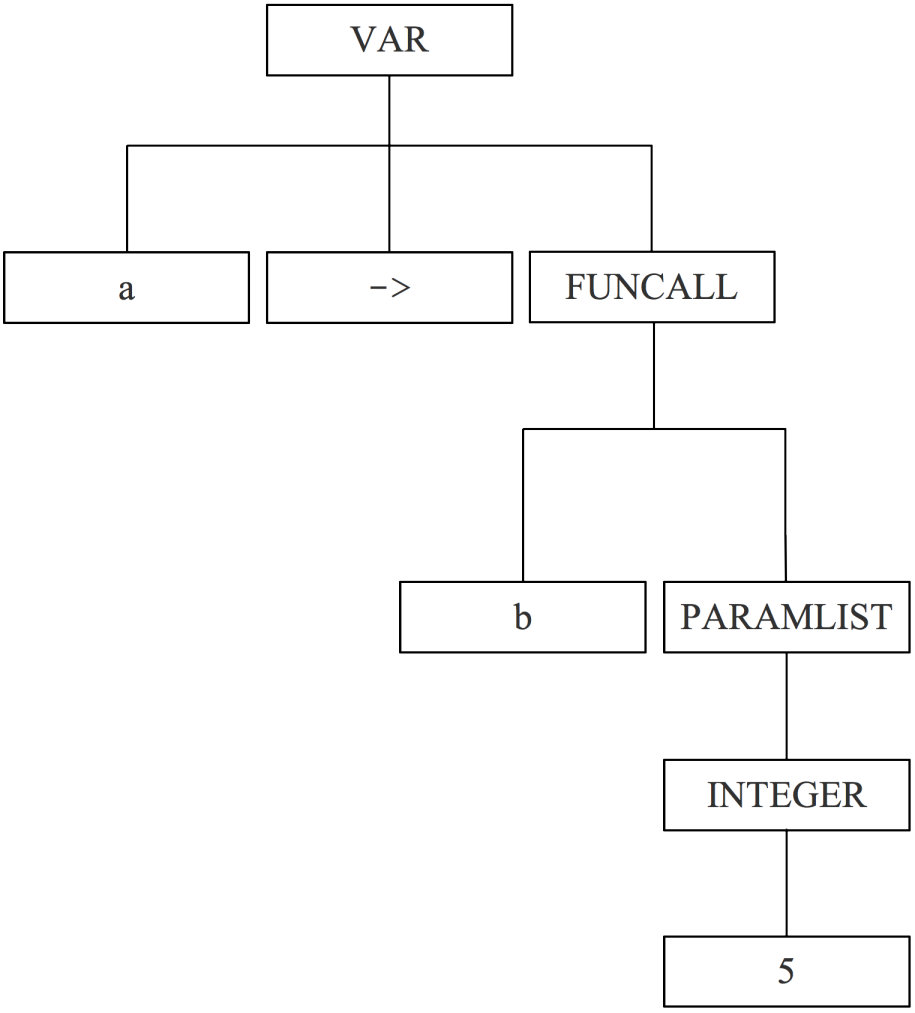


Figure C.3: Tree structure for the input in listing C.3.

if the grammar also permits to call a function (see listing C.4), using the *C* standard syntax, the `ID` token is not sufficient in order to disambiguate this kind of statement with respect to the assignment statement. In order to solve this issue, ANTLR allows to create imaginary nodes, nodes containing a token type for which there is no corresponding input symbol. When such a statement is encountered, the corresponding sub-tree root node will be either the *ASSIGN* or the *FUNCALL* node. In this way the program that will walk through the tree will distinguish between the statements by just visiting their root node.

Listing C.4: augmented grammar

```
...
expression :    ID '=' expr  -> ^(ASSIGN ^('=' ID
   expr))
           |    ID '('params ')' -> ^(FUNCALL ID
              params)
...
```

However, the real power of *AST* construction operators is revealed by expression rules. Consider the `expr` rule: it states that the `multExpr` nodes are always the children and that the mathematical operator is always the sub-tree root. The tree becomes one level higher for each iteration of the `(( '+' ^ | '-' ^ ) multExpr)*` sub-rule, giving an exact order to the performed operations.

Listing C.5: ANTLR tree parser grammar.

```
prog :  stat+ ;

stat   :  (expr !';');
   |  ID '=' expr ';' -> ^('=' ID expr)

expr :  multExpr (('+' ^|'-' ^) multExpr)*

multExpr :  atom (('*' ^|'/' ^) atom)* ;

atom :  INT
   |  ID
     |  '(' expr ')' ;  -> expr

ID :    ('a'..'z' |'A'..'Z' )+ ;
INT:    '0'..'9'+ ;
WS :    (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

Once built the *AST*, various walkers can be implemented, from the one that simply reconstructs or interprets the input code, to walkers that perform some optimizations on it or collect information. However all the possible walkers that can be implemented have to know the structure of the tree and to verify that it has not only the proper nodes but also the proper two-dimensional structure. This is implemented by simply pasting the tree parser grammar and removing recognition grammar elements at the left of the `->` operator, leaving the *AST* rewrite fragments, as shown in listing C.6.

Listing C.6: ANTLR tree walker grammar.

```
prog :  stat+

stat   :  expr
   |  ^('=' ID expr)

expr :  ^('+' expr expr)
   |  ^('-' expr expr)
   |  ^('*' expr expr)
   |  ^('/' expr expr)
   |  ID
   |  INT
```

The rules are quite the same as in the tree parser grammar. The only exception is the simplification of the `expr` rule: it represents all the possibilities of mathematical expressions that can be matched. The simplification is due to the fact that the walker already knows what is the correct order of the mathematical operations just by walking through the tree, so the order does not have to be made explicit.

C.4 *StringTemplate* engine

The last step to complete the implementation of a tree walker is to embed into the grammar the actions that it has to perform. In order to build an interpreter of the input code, there is no more work to do. It is sufficient to add the same actions embedded in the parser grammar (see listing C.2). However, the tree walker implemented for this thesis has a different purpose: after building the tree and permitting to the optimization modules to edit it, the ANTLR tree parser grammar is used to reconstruct the optimized code, starting from the tree. The reason for this, as explained in chapter 1, is that the goal of this thesis is to translate a *Java*-like language into *C* code that can be compiled by an *ANSI-C* compiler, instead of interpreting it. The optimized *C* code to be compiled is reconstructed from the tree using ANTLR and the *StringTemplate* template engine; actually it could be reconstructed also using embedded actions that print the statement corresponding to each sub-tree. Consider the two rules from a Java tree grammar illustrated in listing C.7; they contain embedded actions to spit Java code back out based upon the input symbols where `emit( )` is essentially a print statement that emits output sensitive to some indentation level:

Listing C.7: Reconstructing code with embedded actions.

```
methodHead
    :   IDENT {emit(" "+$IDENT.text+"(");}
        ^( PARAMETERS
            (   p=parameterDef
        {if (there-is-another-parameter) emit(",");}
      )*
        )
        {emit(") ");}
        throwsClause?
;

throwsClause
  : ^( "throws" {emit("throws ");}
      ( identifier
        {if (there-is-another-id) emit(", ");}
      )*
    )
;
```

To figure out what these rules generate, you must imagine what the output looks like by "executing" the arbitrary embedded actions in your mind. Embedding all those actions in a grammar is tedious and often makes it hard to read the grammar. On the contrary, *emitters* (i.e. the programs reconstructing code) built with templates are easier to write and read because within them the output structure is specified. Moreover, translating the input matched by a rule without a template engine is painful and error prone: actions must be embedded in the rule to translate and then emitted or buffered up each rule element. ANTLR integrates *StringTemplate* by providing template construction rules. These rules permit to specify the text to emit for any given grammar rule in a manner that parallels the *AST* construction rules. Listing C.8 shows the template construction rules embedded in a tree walker, like the one illustrated in the previous section.

Listing C.8: Reconstructing code with template construction rules.

```
prog :  s = stat+   ->template(arg={$s})"<arg;
   separator=\"\n\">"

stat   :  expr  ->template(arg ={$expr.st}) "<arg>"
   |  ^('=' ID expr) ->template(arg1 = {$ID.text},
      arg2 ={$expr.st}) "<arg1> = <arg2>"

expr :  ^('+' e1=expr e2=expr) ->template(a1 = {$e1
   },a2 = {$e2})"(<e1>)+(<e2>)"
   |  ^('-' e1=expr e2=expr) ->template(a1 = {$e1},
      a2 = {$e2})"(<e1>)-(<e2>)"
   |  ^('*' e1=expr e2=expr) ->template(a1 = {$e1},
      a2 = {$e2})"(<e1>)*(<e2>)"
   |  ^('/' e1=expr e2=expr) ->template(a1 = {$e1},
      a2 = {$e2})"(<e1>)/(<e2>)"
   |  ID   ->template(arg ={$ID.text}) "<arg>"
   |  INT  ->template(arg ={$INT.text}) "<arg>"
```

For each sub-tree there is a corresponding template construction rule that emits the specified output for that particular sub-tree. The `->` operator introduces the rule, while the `template` keyword indicate a so-called anonymous template, i.e. an in-line specified template. This means that the output for that template is directly specified in the line in which the template construction rule is described; in particular the output is defined within double quotes. There is also the possibility to pass some arguments to the rule: for instance, in the template of the first choice for the `stat` rule, a single argument has been specified, `expr.st`, which carries the string template rule of `expr`; on the contrary, for the second choice of the rule, the template rule has two arguments and the emitted output is just like an assignment in a *C*-like language. However, for more complex grammars and target languages, anonymous in-line templates are not the best solution: in order to separate the output specification from the parser there is the possibility to recall rules defined in a separate file, a *StringTemplateGroup* file (see [26]). Moreover this solution allows also to change the target language without changing a single line of the implemented grammars. Listings C.9 and C.10 show respectively the in-line declaration and the definition of a template construction rule: the `assign` rule is declared in-line in the tree walker, but the actual structure of the output to emit is defined in a separate file. It is enough to modify that file in order to change the target language.

Listing C.9: Declaration of a template construction rule.

```
...
stat    : ...

    |   ^('=' ID expr) ->assign(arg1 = {$ID.text},arg2
        ={$expr.st})

...
```

Listing C.10: Definition, in another file, of a template construction rule.

```
...
assign(arg1,arg2)::=<<
<arg1> = <arg2>

>>
...
```

Moreover, *StringTemplate* engine has a powerful management of indefinitely repeated tokens. Consider the `prog` rule in listing C.8: if more than one `stat` token is encountered, the template rule is able to recognize each different token and to emit it separately from the others by a specified token. The token chosen in the example is the *NEWLINE* (\n) token.

Appendix D

Result Tables

Table D.1: *Java Grande Benchmarks* performance for the *Arith* benchmark. The *EveC vs Java* column indicates the ratio *EveC/Java*.

Benchmark	Java	EveC	EveC vs Java
Add:Int(adds/s)	2086194024	2894251359	1.3873
Add:Long(adds/s)	2007307911	–	–
Add:Float(adds/s)	854295452	927826598	1.0861
Add:Double(adds/s)	839346726	910402868	1.0847
Mult:Int(multiplies/s)	797268037	–	–
Mult:Long(multiplies/s)	794963976	–	–
Mult:Float(multiplies/s)	500878531	542243420	1.0826
Mult:Double(multiplies/s)	500292706	538075505	1.0755
Div:Int(divides/s)	114122744	328900541	2.8820
Div:Long(divides/s)	55347750	327271449	5.9130
Div:Float(divides/s)	261738729	253937831	0.9702
Div:Double(divides/s)	257014667	250020673	0.9728

Table D.2: *Java Grande Benchmarks* performance for the *Assign* benchmark. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Same:Scalar:Local(assignments/s)	3659103434	137625600000	37.6118
Same:Scalar:Instance(assignments/s)	9549371035	116834615901	12.2348
Same:Scalar:Class(assignments/s)	9886342609	117554297439	11.8906
Same:Array:Local(assignments/s)	2291187485	111550638298	48.6868
Same:Array:Instance(assignments/s)	1252880324	112510213681	89.8012
Same:Array:Class(assignments/s)	2449784277	111550638298	45.5349
Other:Scalar:Instance(assignments/s)	9808516602	111790532144	11.3973
Other:Scalar:Class(assignments/s)	9550033119	111790532144	11.7058
Other:Array:Instance(assignments/s)	2506644655	112270319835	44.7891
Other:Array:Class(assignments/s)	2413952912	111550638298	46.2108

Table D.3: *Java Grande Benchmarks* performance for the *Cast* benchmark. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
IntFloat(casts/s)	9140070543	–	–
IntDouble(casts/s)	8778016809	–	–
LongFloat(casts/s)	7275981032	–	–
LongDouble(casts/s)	7261976790	–	–

Table D.4: *Java Grande Benchmarks* performance for the *Create* benchmark on arrays. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Array:Int:1(arrays/s)	66135904	8742313	0.1322
Array:Int:2(arrays/s)	67108760	8772911	0.1307
Array:Int:4(arrays/s)	65399458	8105342	0.1239
Array:Int:8(arrays/s)	58428150	7496647	0.1283
Array:Int:16(arrays/s)	50255471	6732259	0.1340
Array:Int:32(arrays/s)	32089635	5454069	0.1700
Array:Int:64(arrays/s)	17463261	4060142	0.2325
Array:Int:128(arrays/s)	8955963	2157515	0.2409
Array:Long:1(arrays/s)	66258928	8882436	0.1341
Array:Long:2(arrays/s)	65489865	8215045	0.1254
Array:Long:4(arrays/s)	58088271	7593861	0.1307
Array:Long:8(arrays/s)	50818121	6830367	0.1344
Array:Long:16(arrays/s)	32070704	5552977	0.1731
Array:Long:32(arrays/s)	17425800	4069022	0.2335
Array:Long:64(arrays/s)	8969365	2134842	0.2380
Array:Long:128(arrays/s)	4602176	1258877	0.2735
Array:Float:1(arrays/s)	65509035	8816753	0.1346
Array:Float:2(arrays/s)	65856576	8822404	0.1340
Array:Float:4(arrays/s)	65565189	8222422	0.1254
Array:Float:8(arrays/s)	59257656	7592844	0.1281
Array:Float:16(arrays/s)	50713057	6814838	0.1344
Array:Float:32(arrays/s)	31973230	5546352	0.1735
Array:Float:64(arrays/s)	17453629	4093513	0.2345
Array:Float:128(arrays/s)	8953026	2160899	0.2414
Array:Object:1(arrays/s)	62330598	8803731	0.1412
Array:Object:2(arrays/s)	62912055	8235270	0.1309
Array:Object:4(arrays/s)	62386777	7654478	0.1227
Array:Object:8(arrays/s)	56513822	6867594	0.1215
Array:Object:16(arrays/s)	49548131	5527164	0.1116
Array:Object:32(arrays/s)	31929087	4109272	0.1287
Array:Object:64(arrays/s)	17339514	2158700	0.1245
Array:Object:128(arrays/s)	8882118	1245392	0.1402

Table D.5: *Java Grande Benchmarks* performance for the *Create* benchmark on objects. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Object:Base(objects/s)	39616010025	8847200	0.0002
Object:Simple(objects/s)	39908201076	8553111	0.0002
Object:Simple:Constructor(objects/s)	39745572079	8606388	0.0002
Object:Simple:1Field(objects/s)	39746939065	8663965	0.0002
Object:Simple:2Field(objects/s)	38241364937	8540799	0.0002
Object:Simple:4Field(objects/s)	38186303513	9476642	0.0002
Object:Simple:4fField(objects/s)	38052066884	9413659	0.0002
Object:Simple:4LField(objects/s)	38269197080	8505826	0.0002
Object:Subclass(objects/s)	37914863551	8858629	0.0002
Object:Complex(objects/s)	38269197080	4101962	0.0001
Object:Complex:Constructor(objects/s)	37243198535	4392222	0.0001

Table D.6: *Java Grande Benchmarks* performance for the *Loop* benchmark. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Loop:For(iterations/s)	2390085309	–	–
Loop:ReverseFor(iterations/s)	–	–	–
Loop:While(iterations/s)	792708451	–	–

Table D.7: *Java Grande Benchmarks* performance for the *Math* benchmark. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
AbsInt(operations/s)	2041440480	421863917	0.2067
AbsLong(operations/s)	2290065388	406103192	0.1773
AbsFloat(operations/s)	2021515781	304287675	0.1505
AbsDouble(operations/s)	2023785161	319357802	0.1578
MaxInt(operations/s)	1128921938	409571160	0.3628
MaxLong(operations/s)	2271434164	374513286	0.1649
MaxFloat(operations/s)	267642553	267736442	1.0004
MaxDouble(operations/s)	271364632	281683680	1.0380
MinInt(operations/s)	1052906559	423101232	0.4018
MinLong(operations/s)	2287388014	423121289	0.1850
MinFloat(operations/s)	250692623	266863687	1.0645
MinDouble(operations/s)	272124654	282087694	1.0366
SinDouble(operations/s)	27452711	76210777	2.7761
CosDouble(operations/s)	20161295	58857914	2.9194
TanDouble(operations/s)	17803527	59317707	3.3318
AsinDouble(operations/s)	48536511	54990407	1.1330
AcosDouble(operations/s)	6309923	43639662	6.9160
AtanDouble(operations/s)	67597153	88033518	1.3023
Atan2Double(operations/s)	35570505	59247191	1.6656
FloorDouble(operations/s)	246031323	290013830	1.1788
CeilDouble(operations/s)	244209429	305564690	1.2512
SqrtDouble(operations/s)	270061642	168232660	0.6229
ExpDouble(operations/s)	23524605	37596873	1.5982
LogDouble(operations/s)	36428491	20932237	0.5746
PowDouble(operations/s)	4147051	160204	0.0386
RintDouble(operations/s)	255088352	296379411	1.1619
Random(operations/s)	29935088	10380530	0.3468
RoundFloat(operations/s)	93731413	239628446	2.5565
RoundDouble(operations/s)	102013714	272025209	2.6666
IEEERemainderDouble(operations/s)	42806797	148483487	3.4687

Table D.8: *Java Grande Benchmarks* performance for the *Method* benchmark. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Same:Instance(calls/s)	19058575799	368425105	0.0193
Same:SynchronizedInstance(calls/s)	4852261562	30358743	0.0063
Same:FinalInstance(calls/s)	18982996812	325721764	0.0172
Same:Class(calls/s)	18977039618	1655186032	0.0872
Same:SynchronizedClass(calls/s)	4529427594	31832778	0.0070
Other:Instance(calls/s)	18997819237	350374468	0.0184
Other:InstanceOfAbstract(calls/s)	18977111654	308266573	0.0162
Other:Class(calls/s)	19120657120	1629164913	0.0852

Table D.9: *Java Grande Benchmarks* performance for the Section 2 for Size A (small) and Size B (big). The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Series:Kernel:SizeA(coefficients/s)	3069	6196	2.0188
HeapSort:Kernel:SizeA(items/s)	3850764	4627750	1.2018
Crypt:Kernel:SizeA(Kbyte/s)	31744	46876	1.4767
FFT:Kernel:SizeA(Samples/s)	1371924	1352219	0.9856
SOR:Kernel:SizeA(Iterations/s)	160	112	0.7003
SparseMatmult:Kernel:SizeA(Iterations/s)	600	585	0.9742
Series:Kernel:SizeC(coefficients/s)	1911	6267	3.2792
HeapSort:Kernel:SizeC(items/s)	2223232	1449871	0.6521
Crypt:Kernel:SizeC(Kbyte/s)	35817	48534	1.3551
FFT:Kernel:SizeC(Samples/s)	1123328	1114954	0.9925
SOR:Kernel:SizeC(Iterations/s)	37	28	0.7555
SparseMatmult:Kernel:SizeC(Iterations/s)	17	17	0.9838

Table D.10: *jBYTE* performance for all of its benchmark. For each benchmark, a computational score is reported. The *EveC vs Java* column indicates the ratio $EveC/Java$.

Benchmark	Java	EveC	EveC vs Java
Numeric Sort	950.33	1368.67	1.4402
Bitfield Operations	1209.17	2034.33	1.6824
FP Emulation	2014.00	608.00	0.3019
Fourier	52.00	82.00	1.5769
IDEA Encryption	1238.67	1198.00	0.9672
Huffman Compression	1103.00	928.00	0.8413
Neural Net	1816.67	1204.00	0.6628
LU Decomposition	2211.50	1822.33	0.8240

Table D.11: Different numbers of constraints and different numbers of variables for each run of the *Simplex* benchmark.

Name	Number of Constrains	Number of Variables
A	100	100
B	1000	100
C	1000	1000
D	10000	1000
E	10000	4000
F	10000	6000
G	10000	10000

Table D.12: Results of the *Simplex* benchmark for *Java* and *EveC*. The first column indicates the dimensions of data used as specified in table D.11. The columns *Java* and *EveC* report results in milliseconds.

Run	Java	EveC	EveC vs Java
A	25.6	7	3.6571
B	87.8	47.8	1.8368
C	227.2	182.4	1.2456
D	11749	11647.8	1.0087
E	20324.4	18954.2	1.0723
F	35447.2	33657.4	1.0532
G	61552.4	59139	1.0408

Table D.13: *Java Grande Benchmarks* performance for the *Arith* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Add:Int(adds/s)	2894251359	2964006992	1.0241	1.4208
Add:Long(adds/s)	–	–	–	–
Add:Float(adds/s)	927826598	947747177	1.0215	1.1094
Add:Double(adds/s)	910402868	906517212	0.9957	1.0801
Mult:Int(multiplies/s)	–	–	–	–
Mult:Long(multiplies/s)	–	–	–	–
Mult:Float(multiplies/s)	542243420	550507009	1.0152	1.0991
Mult:Double(multiplies/s)	538075505	541255850	1.0059	1.0819
Div:Int(divides/s)	328900541	342862070	1.0424	3.0043
Div:Long(divides/s)	327271449	347642445	1.0622	6.2811
Div:Float(divides/s)	253937831	270791075	1.0664	1.0346
Div:Double(divides/s)	250020673	268441504	1.0737	1.0445

Table D.14: *Java Grande Benchmarks* performance for the *Assign* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Same:Scalar:Local(assignments/s)	137625600000	149724553846	1.0879	40.9184
Same:Scalar:Instance(assignments/s)	116834615901	127770492512	1.0936	13.3800
Same:Scalar:Class(assignments/s)	117554297439	128138413916	1.0900	12.9612
Same:Array:Local(assignments/s)	111550638298	117029820277	1.0491	51.0782
Same:Array:Instance(assignments/s)	112510213681	127770492512	1.1356	101.9814
Same:Array:Class(assignments/s)	111550638298	128378307761	1.1509	52.4039
Other:Scalar:Instance(assignments/s)	111790532144	127044206105	1.1364	12.9524
Other:Scalar:Class(assignments/s)	111790532144	125751411639	1.1249	13.1676
Other:Array:Instance(assignments/s)	112270319835	122486514909	1.0910	48.8647
Other:Array:Class(assignments/s)	111550638298	122486514909	1.0980	50.7411

Table D.15: *Java Grande Benchmarks* performance for the *Cast* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
IntFloat(casts/s)	–	–	–	–
IntDouble(casts/s)	–	–	–	–
LongFloat(casts/s)	–	–	–	–
LongDouble(casts/s)	–	–	–	–

Table D.16: *Java Grande Benchmarks* performance for the *Create* benchmark on arrays. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the rati $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Array:Int:1(arrays/s)	8742313	8719094	0.9973	0.1318
Array:Int:2(arrays/s)	8772911	8687286	0.9902	0.1295
Array:Int:4(arrays/s)	8105342	8124448	1.0024	0.1242
Array:Int:8(arrays/s)	7496647	7409780	0.9884	0.1268
Array:Int:16(arrays/s)	6732259	6666798	0.9903	0.1326
Array:Int:32(arrays/s)	5454069	5398203	0.9898	0.1682
Array:Int:64(arrays/s)	4060142	3985827	0.9817	0.2282
Array:Int:128(arrays/s)	2157515	2120779	0.9830	0.2368
Array:Long:1(arrays/s)	8882436	8667031	0.9757	0.1308
Array:Long:2(arrays/s)	8215045	8125308	0.9891	0.1241
Array:Long:4(arrays/s)	7593861	7532861	0.9920	0.1297
Array:Long:8(arrays/s)	6830367	6675527	0.9773	0.1314
Array:Long:16(arrays/s)	5552977	5454812	0.9823	0.1701
Array:Long:32(arrays/s)	4069022	4085105	1.0040	0.2344
Array:Long:64(arrays/s)	2134842	2155741	1.0098	0.2403
Array:Long:128(arrays/s)	1258877	1259821	1.0007	0.2737
Array:Float:1(arrays/s)	8816753	8749240	0.9923	0.1336
Array:Float:2(arrays/s)	8822404	8709661	0.9872	0.1323
Array:Float:4(arrays/s)	8222422	8164867	0.9930	0.1245
Array:Float:8(arrays/s)	7592844	7582470	0.9986	0.1280
Array:Float:16(arrays/s)	6814838	6715694	0.9855	0.1324
Array:Float:32(arrays/s)	5546352	5485128	0.9890	0.1716
Array:Float:64(arrays/s)	4093513	4052331	0.9899	0.2322
Array:Float:128(arrays/s)	2160899	2125886	0.9838	0.2374
Array:Object:1(arrays/s)	8803731	8456841	0.9606	0.1357
Array:Object:2(arrays/s)	8235270	8038719	0.9761	0.1278
Array:Object:4(arrays/s)	7654478	7522584	0.9828	0.1206
Array:Object:8(arrays/s)	6867594	6700407	0.9757	0.1186
Array:Object:16(arrays/s)	5527164	5496226	0.9944	0.1109
Array:Object:32(arrays/s)	4109272	4086629	0.9945	0.1280
Array:Object:64(arrays/s)	2158700	2155755	0.9986	0.1243
Array:Object:128(arrays/s)	1245392	1243902	0.9988	0.1400

Table D.17: *Java Grande Benchmarks* performance for the *Create* benchmark on objects. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Object:Base(objects/s)	8847200	47638293	5.3846	0.0012
Object:Simple(objects/s)	8553111	38397000	4.4892	0.0010
Object:Simple:Constructor(objects/s)	8606386	38657465	4.4917	0.0010
Object:Simple:1Field(objects/s)	8663965	40564496	4.6820	0.0010
Object:Simple:2Field(objects/s)	8540799	38643788	4.5246	0.0010
Object:Simple:4Field(objects/s)	9476642	38852265	4.0998	0.0010
Object:Simple:4fField(objects/s)	9413659	38643788	4.1051	0.0010
Object:Simple:4LField(objects/s)	8505826	35396165	4.1614	0.0009
Object:Subclass(objects/s)	8858629	29254531	3.3024	0.0008
Object:Complex(objects/s)	4101962	6580448	1.6042	0.0002
Object:Complex:Constructor(objects/s)	4392222	3978244	0.9057	0.0001

Table D.18: *Java Grande Benchmarks* performance for the *Loop* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
For(iterations/s)	–	–	–	–
ReverseFor(iterations/s)	–	–	–	–
While(iterations/s)	–	–	–	–

Table D.19: *Java Grande Benchmarks* performance for the *Math* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
AbsInt(operations/s)	421863917	400085168	0.9484	0.1960
AbsLong(operations/s)	406103192	420555934	1.0356	0.1836
AbsFloat(operations/s)	304287675	305212458	1.0030	0.1510
AbsDouble(operations/s)	319357802	280389819	0.8780	0.1385
MaxInt(operations/s)	409571160	417538343	1.0195	0.3699
MaxLong(operations/s)	374513286	348486311	0.9305	0.1534
MaxFloat(operations/s)	267736442	268689449	1.0036	1.0039
MaxDouble(operations/s)	281683680	265877097	0.9439	0.9798
MinInt(operations/s)	423101232	424246808	1.0027	0.4029
MinLong(operations/s)	423121289	419767992	0.9921	0.1835
MinFloat(operations/s)	266863687	266507706	0.9987	1.0631
MinDouble(operations/s)	282087694	298576791	1.0585	1.0972
SinDouble(operations/s)	76210777	74206893	0.9737	2.7031
CosDouble(operations/s)	58857914	59614005	1.0128	2.9569
TanDouble(operations/s)	59317707	61119557	1.0304	3.4330
AsinDouble(operations/s)	54990407	57255900	1.0412	1.1796
AcosDouble(operations/s)	43639662	44557205	1.0210	7.0614
AtanDouble(operations/s)	88033518	94816847	1.0771	1.4026
Atan2Double(operations/s)	59247191	63155083	1.0660	1.7755
FloorDouble(operations/s)	290013830	368247622	1.2698	1.4968
CeilDouble(operations/s)	305564690	324347380	1.0615	1.3282
SqrtDouble(operations/s)	168232660	169545198	1.0078	0.6278
ExpDouble(operations/s)	37596873	39118976	1.0405	1.6629
LogDouble(operations/s)	20932237	21825557	1.0427	0.5991
PowDouble(operations/s)	160204	160616	1.0026	0.0387
RintDouble(operations/s)	296379411	415383194	1.4015	1.6284
Random(operations/s)	10380530	10656883	1.0266	0.3560
RoundFloat(operations/s)	239628446	240164466	1.0022	2.5623
RoundDouble(operations/s)	272025209	275581493	1.0131	2.7014
IEEERemainderDouble(operations/s)	148483487	148624651	1.0010	3.4720

Table D.20: *Java Grande Benchmarks* performance for the *Method* benchmark. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Same:Instance(calls/s)	368425105	1754177251	4.7613	0.0920
Same:SynchronizedInstance(calls/s)	30358743	31622293	1.0416	0.0065
Same:FinalInstance(calls/s)	325721764	1645238841	5.0511	0.0867
Same:Class(calls/s)	1655186032	1602644673	0.9683	0.0845
Same:SynchronizedClass(calls/s)	31832778	31677055	0.9951	0.0070
Other:Instance(calls/s)	350374468	1648501726	4.7050	0.0868
Other:InstanceOfAbstract(calls/s)	308266573	1645860113	5.3391	0.0867
Other:Class(calls/s)	1629164913	1600440682	0.9824	0.0837

Table D.21: *Java Grande Benchmarks* performance for the *Section 2* benchmark for Size A (small) and Size B (big). The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$. and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Series:Kernel:SizeA(coefficients/s)	6196	6238	1.0067	2.0323
HeapSort:Kernel:SizeA(items/s)	4627750	4422904	0.9557	1.1486
Crypt:Kernel:SizeA(Kbyte/s)	46876	47244	1.0078	1.4883
FFT:Kernel:SizeA(Samples/s)	1352219	1503665	1.1120	1.0960
SOR:Kernel:SizeA(Iterations/s)	112	118	1.0506	0.7357
SparseMatmult:Kernel:SizeA(Iterations/s)	585	620	1.0595	1.0322
Series:Kernel:SizeC(coefficients/s)	6267	6243	0.9962	3.2792
HeapSort:Kernel:SizeC(items/s)	1449871	1430504	0.9866	0.6521
Crypt:Kernel:SizeC(Kbyte/s)	48534	46690	0.9620	1.3551
FFT:Kernel:SizeC(Samples/s)	1114954	1116801	1.0017	0.9925
SOR:Kernel:SizeC(Iterations/s)	28	28	1.0001	0.7555
SparseMatmult:Kernel:SizeC(Iterations/s)	17.1306118	18.0790554	1.0554	0.9838

Table D.22: *Java Grande Benchmarks* performance for the *Math* benchmark. The *EveC Opt SRC vs EveC Opt* column indicates the ratio $EveCOptSRC/EveC$, and the *EveC Opt SRC vs Java* indicates the ratio $EveCOptSRC/Java$.

Benchmark	EveC Opt	EveC Opt SRC	EveC Opt SRC vs EveC Opt	EveC Opt SRC vs Java
AbsInt(operations/s)	400085168	2165676380	5.4130	1.0609
AbsLong(operations/s)	420555934	2064144072	4.9081	0.9013
AbsFloat(operations/s)	305212458	978675590	3.2065	0.4841
AbsDouble(operations/s)	280389819	709115922	2.5290	0.3504
MaxInt(operations/s)	417538343	2072469638	4.9635	1.8358
MaxLong(operations/s)	348486311	2050338678	5.8836	0.9027
MaxFloat(operations/s)	268689449	394045671	1.4665	1.4723
MaxDouble(operations/s)	265877097	389960934	1.4667	1.4370
MinInt(operations/s)	424246808	2065194091	4.8679	1.9614
MinLong(operations/s)	419767992	2030339366	4.8368	0.8876
MinFloat(operations/s)	266507706	393987143	1.4783	1.5716
MinDouble(operations/s)	298576791	398276886	1.3339	1.4636
SinDouble(operations/s)	74206893	74886021	1.0092	2.7278
CosDouble(operations/s)	59614005	56080784	0.9407	2.7816
TanDouble(operations/s)	61119557	69093332	1.1305	3.8809
AsinDouble(operations/s)	57255900	71955117	1.2567	1.4825
AcosDouble(operations/s)	44557205	46161840	1.0360	7.3158
AtanDouble(operations/s)	94816847	169328000	1.7858	2.5050
Atan2Double(operations/s)	63155083	64416898	1.0200	1.8110
FloorDouble(operations/s)	368247622	1661470910	4.5118	6.7531
CeilDouble(operations/s)	324347380	1659475584	5.1164	6.7953
SqrtDouble(operations/s)	169545198	231192640	1.3636	0.8561
ExpDouble(operations/s)	39118976	47343614	1.2102	2.0125
LogDouble(operations/s)	21825557	21908154	1.0038	0.6014
PowDouble(operations/s)	160616	160495	0.9993	0.0387
RintDouble(operations/s)	415383194	1669872410	4.0201	6.5463
Random(operations/s)	10656883	11023121	1.0344	0.3682
RoundFloat(operations/s)	240164466	278823211	1.1610	2.9747
RoundDouble(operations/s)	275581493	317235037	1.1511	3.1097
IEEERemainderDouble(operations/s)	148624651	161692729	1.0879	3.7773

Table D.23: *Java Grande Benchmarks* performance for the *Method* benchmark disabling the *Stack Trace*. The *EveC Opt no ST vs EveC Opt* column indicates the ratio $EveCOptnoST/EveCOpt$ and the *EveC Opt no ST vs Java* column indicates the ratio $EveCOptnoST/Java$.

Benchmark	EveC Opt	EveC Opt no ST	EveC Opt ST vs EveC Opt	EveC Opt ST vs Java
Same:Instance (calls/s)	1754177251	388361481481	221.3924	20.4815
Same:SynchronizedInstance (calls/s)	31622293	31730415	1.0034	0.0065
Same:FinalInstance (calls/s)	1645238841	299593142857	182.0971	15.8000
Same:Class (calls/s)	1602644673	299593142857	186.9367	15.6571
Same:SynchronizedClass (calls/s)	31677055	31488012	0.9940	0.0069
Other:Instance (calls/s)	1648501725	299593142857	181.7366	15.6571
Other:InstanceOfAbstract (calls/s)	1645860113	308404705882	187.3821	16.1471
Other:Class (calls/s)	1600440682	308404705882	192.6999	16.1176

Table D.24: *jBYTE* performance for all of its benchmark. For each benchmark, a computational score is reported. The *EveC Opt vs EveC* column indicates the ratio $EveCOpt/EveC$, and the *EveC Opt vs Java* indicates the ratio $EveCOpt/Java$.

Benchmark	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
Numeric Sort	1368.67	1427.00	1.0426	1.5016
Bitfield Operations	2034.33	2199.00	1.0809	1.8186
FP Emulation	608.00	620.00	1.0197	0.3078
Fourier	82.00	81.00	0.9878	1.5577
IDEA Encryption	1198.00	1201.33	1.0028	0.9699
Huffman Compression	928.00	1130.00	1.2177	1.0245
Neural Net	1204.00	1310.00	1.0880	0.7211
LU Decomposition	1822.33	1896.33	1.0406	0.8575

Table D.25: *jBYTE* performance for all of its benchmark without performing bounds check. For each benchmark, a computational score is reported. The *EveC No BC vs EveC Opt* column indicates the ratio $EveCNoBC/EveCOpt$.

Benchmark	EveC Opt	EveC No BC	EveC No BC vs EveC Opt
Numeric Sort	1427	1472	1.0315
Bitfield Operations	2199	2740	1.2460
FP Emulation	620	841	1.3565
Fourier	81	106	1.3086
IDEA Encryption	1201	1257	1.0463
Huffman Compression	1130	1820	1.6106
Neural Net	1310	2765	2.1107
LU Decomposition	1896	2114	1.1148

Table D.26: Results of the *Simplex* benchmark for *Java* and *EveC*. The first column indicates the dimensions of data used as specified in table D.11. The columns *EveC* and *EveC Opt* report results in milliseconds.

Run	EveC	EveC Opt	EveC Opt vs EveC	EveC Opt vs Java
A	7	9	0.7778	2.8444
B	47.8	47.2	1.0127	1.8602
C	182.4	180.2	1.0122	1.2608
D	11647.8	11233	1.0369	1.0459
E	18954.2	18352.4	1.0328	1.1075
F	33657.4	32869.8	1.0240	1.0784
G	59139	57726.6	1.0245	1.0663

Table D.27: Results of the *Simplex* benchmark for *EveC* and *EveC* with the only *Array Explosion* optimization enabled. The first column indicates the dimensions of data used as specified in table D.11. The columns *EveC* and *EveC Opt* report results in milliseconds.

Run	EveC	EveC AE	EveC AE vs EveC
A	7	5.8	1.2069
B	47.8	41.2	1.1602
C	182.4	148.2	1.2308
D	11647.8	11412.6	1.0206
E	18954.2	18709.2	1.0131
F	33657.4	33346	1.0093
G	59139	58418.8	1.0123

Table D.28: *Java Grande Benchmarks* performance for the *Method* benchmark. The *EveC inlining vs No inlining* column indicates the ratio $EveCinlining/Noinlining$.

Benchmark	No inlining	EveC inlining	EveC inlining vs No inlining
Same:Instance (calls/s)	392695678	2093383908	5.3308
Same:SynchronizedInstance (calls/s)	30820165	32106603	1.0417
Same:FinalInstance (calls/s)	371782725	1647927078	4.4325
Same:Class (calls/s)	372601805	1620423427	4.3489
Same:SynchronizedClass (calls/s)	30217631	31897206	1.0556
Other:Instance (calls/s)	372205026	1633550397	4.3888
Other:InstanceOfAbstract (calls/s)	370731155	1671570221	4.5088
Other:Class (calls/s)	372231451	1652081298	4.4383

Table D.29: Performances for the Sorting algorithms for the non-optimized *EveC* implementation w.r.t. *Java*. The *EveC vs Java* column indicates the ratio $Java/EveC$.

Benchmark	EveC	Java	EveC vs Java
Bubble Sort	24631.8	22786.7	0.925
Insertion Sort	4737.6	4273.1	0.901
Quick Sort	214	206	0.962

Table D.30: Performances for the Sorting algorithms for the non-optimized *EveC* implementation w.r.t. *EveC* with BCO. The *EveC vs EveC BCO* column indicates the ratio $EveC/EveCBCO$.

Benchmark	EveC	EveC BCO	EveC vs EveC BCO
Bubble Sort	24631.8	17625	1.397
Insertion Sort	4737.6	3632.7	1.304
Quick Sort	214	200	1.07

Table D.31: Performances for the Sorting algorithms for the optimized *EveC* implementation w.r.t. *Java*. The *EveC BCO vs Java* column indicates the ratio $Java/EveCBCO$.

Benchmark	Java	EveC BCO	EveC BCO vs Java
Bubble Sort	22786.7	17625	1.292
Insertion Sort	4273.1	3632.7	1.176
Quick Sort	206	200	1.03

Table D.32: Performances for the Sorting algorithms for *EveC* implementation without bound checks w.r.t. *Java* and *EveC* BCO. The *BCO vs NBC* column indicates the ratio $EveCBCO/EveCNBC$ and the *NBC vs Java* column shows the ratio $Java/EveCNBC$.

Benchmark	Java	EveC BCO	EveC NBC	BCO vs NBC	NBC vs Java
Bubble Sort	22786.7	17625	14000	1.259	1.627
Insertion Sort	4273.1	3632.7	1795	2.023	2.380
Quick Sort	206	200	181	1.104	1.138

Bibliography

[1] "The java programming language." `http://www.java.com/`.

[2] "Java white paper." `http://www.oracle.com/technetwork/java/langenv-140151.html`.

[3] K. Ogata, D. Mikurube, K. Kawachiya, S. Trent, and T. Onodera, "A study of java's non-java memory," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, (New York, NY, USA), pp. 191–204, ACM, 2010.

[4] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, pp. 97–113, 2003.

[5] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot client compiler for java 6," *ACM Trans. Archit. Code Optim.*, vol. 5, pp. 7:1–7:32, May 2008.

[6] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An optimizing compiler for java," tech. rep., 1998.

[7] "Excelsior jet." `http://www.excelsior-usa.com/jet.html`.

[8] "Ibm high performance compiler for java." `http://www.research.ibm.com/topics/popups/innovate/java/html/hpcj.html`.

[9] A. Varma and S. Bhattacharyya, "Java-through-c compilation: an enabling technology for java in embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 3, pp. 161 – 166 Vol.3, feb. 2004.

[10] Y. Chiba, "Translating java to c without inserting class initialization tests," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pp. 117 –123, 2002.

[11] "The eclipse foundation open source community website." http://www.eclipse.org/.

[12] "The gnu c compiler." `http://gcc.gnu.org/`.

[13] "clang: a c language family frontend for llvm." `http://clang.llvm.org/`.

[14] "A garbage collector for c and c++." `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[15] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pp. 13–, IBM Press, 1999.

[16] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[17] "Gnu classpath." `http://www.gnu.org/software/classpath/`.

[18] "Sqlite database." `http://www.sqlite.org/`.

[19] A. Aho and J. Ullman, *Principles of compiler design*. Addison-Wesley series in computer science and information processing, Addison-Wesley Pub. Co., 1977.

[20] "Malloc function specification." `http://pubs.opengroup.org/onlinepubs/009695399/functions/malloc.html`.

[21] "The mozilla project." `http://www.mozilla.org/`.

[22] "The mono project." `http://www.go-mono.com/`.

[23] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1996.

[24] M. Paleczny, C. Vick, and C. Click, "The java hotspot(tm) server compiler," in *In USENIX Java Virtual Machine Research and Technology Symposium*, pp. 1–12, 2001.

[25] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 428–455, 1994.

[26] T. Parr, *The definitive ANTLR reference: building domain-specific languages*. 2007.

[27] B. Calder and D. Grunwald, "Reducing indirect function call overhead in c++ programs," in *In POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 397–408, ACM, 1994.

[28] "The modula 3 programming language." `http://www.modula3.org/`.

[29] "The cecil programming language." `http://www.cs.washington.edu/research/projects/cecil/`.

[30] "The self programming language." `http://selflanguage.org/`.

[31] O. Agesen and U. Hölzle, "Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages," in *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, (New York, NY, USA), pp. 91–107, ACM, 1995.

[32] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, (New York, NY, USA), pp. 324–341, ACM, 1996.

[33] G. Aigner and U. Hölzle, "Eliminating virtual function calls in c++ programs," pp. 142–166, Springer-Verlag, 1996.

[34] A. Diwan, J. Eliot, B. Moss, and K. S. M. Kinley, "Simple and effective analysis of statically-typed object-oriented programs," pp. 292–305, 1996.

[35] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," pp. 77–101, Springer-Verlag, 1995.

[36] L. O. Andersen, "Program analysis and specialization for the c programming language," tech. rep., 1994.

[37] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using bdds," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, (New York, NY, USA), pp. 103–114, ACM, 2003.

[38] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, (New York, NY, USA), pp. 59–76, ACM, 2005.

[39] O. Waddell and R. Dybvig, "Fast and effective procedure inlining," in *Static Analysis* (P. Hentenryck, ed.), vol. 1302 of *Lecture Notes in Computer Science*, pp. 35–52, Springer Berlin Heidelberg, 1997.

[40] S. Jagannathan and A. Wright, "Flow-directed inlining," in *In Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 193–205, 1996.

[41] "The c preprocessor of gcc." `http://www.gcc.gnu.org/onlinedoc/cpp/`.

[42] T. Suganuma, T. Yasue, and T. Nakatani, "An empirical study of method inlining for a java just-in-time compiler," in *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pp. 91–104, 2002.

[43] H. van den Brink, "The current and future optimizations performed by the java hotspot compiler,"

[44] "Hotspot internals: Server compiler inlining messages." `https://wikis.oracle.com/display/HotSpotInternals/Server+Compiler+Inlining+Messages`.

[45] M. Paleczny, C. Vick, and C. Click, "The java hotspot tm server compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pp. 1–1, USENIX Association, 2001.

[46] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff, "Escape analysis for java," *ACM SIGPLAN Notices*, vol. 34, no. 10, pp. 1–19, 1999.

[47] R. Bodik, R. Gupta, and V. Sarkar, "Abcd: Eliminating array bounds checks on demand," in *In ACM Conference on Programming Language Design and Implementation*, pp. 321–333, 2000.

[48] F. Qian, L. J. Hendren, and C. Verbrugge, "A comprehensive approach to array bounds check elimination for java," in *Proceedings of the 11th*

International Conference on Compiler Construction, CC '02, (London, UK, UK), pp. 325–342, Springer-Verlag, 2002.

[49] A. Gampe, J. von Ronne, D. Niedzielski, J. Vasek, and K. Psarris, "Safe, multiphase bounds check elimination in java," *Softw. Pract. Exper.*, vol. 41, pp. 753–788, June 2011.

[50] J. E. Moreira, S. P. Midkiff, S. P. Midkiff, J. E. Moreira, M. Snir, and M. Snir, "Optimizing array reference checking in java programs," 1998.

[51] J. A. Mathew, P. D. Coddington, and K. A. Hawick, "Analysis and development of java grande benchmarks," in *In Proc. of the ACM 1999 Java Grande Conference*, pp. 72–80, 1999.

[52] "jbyte benchmarks." `http://www.tux.org/~mayer/linux/byte/bdoc.pdf`.

[53] "The simplex algorithm implemented in java." `http://algs4.cs.princeton.edu/65reductions/Simplex.java.html`.

[54] "Sieve of eratosthenes." `http://www.algolist.net/Algorithms/Number_theoretic/Sieve_of_Eratosthenes`.

[55] R. Sedgewick, *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Algorithms in C++, Pearson Education, 1998.

[56] "The plug-in development environment." `http://www.eclipse.org/pde/`.

Ringraziamenti

Desideriamo ringraziare:

- Il Chiar.mo Prof. Mauro Migliardi, per l'esperienza e la competenza che ci ha messo a disposizione durante lo sviluppo del progetto.
- Il Prof. Alessio Merlo, per essere stato il primo a credere nel progetto.
- Le nostre famiglie, per il costante sostegno.
- Enrica per l'aiuto nella stesura di questa tesi.
- La segreteria del DIBRIS, in particolare Daniela, che ci ha aiutato nella parte piú difficile di questo lavoro, ovvero quella burocratica.

Io, Nicoló, desidero ringraziare:

- Giorgia, perché e' sempre stata al mio fianco e mi ha sempre appoggiato e sopportato, donandomi un contributo fondamentale per il raggiungimento di questo traguardo.
- La mia famiglia adottiva milanese, in particolare Patrizia, Francesco, Valentina e il mio coinquilino Barsi.
- Tutti gli amici che in questi anni mi hanno regalato svaghi e riempito il mio tempo libero, aiutandomi a proseguire nel mio percorso.

Io, Luca, desidero ringraziare:

- Silvia, per essermi sempre stata accanto e sostenuto.
- Mia madre, per aver corretto il mio inglese.
- Piú in generale, vorrei anche ringraziare tutte quelle persone che hanno sempre creduto in me e che mi hanno sempre sostenuto nei momenti difficili.

Io, Gabriele, desidero ringraziare:

- Ambra, per essermi sempre stata accanto e avermi sostenuto in questo lungo percorso.
- Tutti i miei amici, che mi hanno aiutato a sostenere il peso di questo progetto, allietando le mie giornate.