IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Domain Driven Program Evolution

PhD Program in Computer Science and Engineering

XX Cycle

By

Diego Colombo

2009

The dissertation of Diego Colombo is approved.

Program Coordinator: Prof. Ugo Montanari, University of Pisa

Supervisor: Dr. Antonio Cisternino, University of Pisa

The dissertation of Diego Colombo has been reviewed by:

Dr. James S. Miller, Microsoft Corporation

Dr. Walter Cazzola, University of Milan

IMT Institute for Advanced Studies, Lucca

2009

❝ Alchemy is the science of breaking down the matter to rebuild it again, everything can be broken down in its alchemic components and then recomposed to create new things. But to gain something an equal value must be lost : that is the equivalent exchange rule. ❞

To my wife Alessia, my family, Ugo, Jacques, Carmilla, Mushra, Holly, Fofo, Liliana, Pina, Niccolino, Gargia. Thanks to the Duncan, Bert, Sam, Helena, Karim, Robbie, MikeD and Russell. Thanks to all the people that give me the chance and the support to get to the end of my PhD. A special thank you to Bruno Quarta and Marco Combetto, most of my work is your fault. Finally I want to acknowledge people that supported me and provided valuable discussions to refine my ideas : Maurizio De Pascale, Artur —a lot of names and last names— Moreira, Denis Samoylov, Randy Henne, Kane and Lynch.

To the Great Architect of the Universe

Alba gu bra

Contents

List of Figures

List of Tables

Listings

Acknowledgements

The experiments in Chapter 4 have been realised and developed in collaboration with Realtime Worlds ltd. I want to acknowledge and thank Duncan L. Harrison, Russell W. Kay and Mike J. Dailly for the contribution and the support in developing, testing, and benchmarking the scenarios.

We want also to acknowledge and thank Aneglo Pesce (Electronic Arts), Marco "better than Payne" Marconi, Marco Geddo, Paolo Milani and Christian Orlandi from Milestone srl for the chance to participate in game engine development sharing their knowledge on the domain and providing the scenario for this thesis.

A special thanks to Piotr Puszkiewicz, Steve Herndon, Jim Miller, Claudio Caldato, Chris To, Rob Unoki and Scott Holden from **Microsoft**; their team offered me the opportunity to participate at the first release of XNA discussing the perspective of .NET based games in the industry.

Vita

February 12, 1977	Born, Portoferraio (Livorno), Italy
2002	Bachelor Degree in Computer Science Final mark: 104/110 University of Pisa, Pisa, Italy
2004	Master Degree in Computer Science Final mark: 105/110 University of Pisa, Pisa, Italy
2005	Graduate Fellow University of Pisa, Pisa, Italy
March 2006	Internship Microsoft Robotics Studio Microsoft Research, Redmond(WA), US
June 2006	Internship Common Language Runtime Microsoft Corporation, Redmond(WA), US
November 2006	Research and Development Milestone, Milan, Italy
2007	Research Fellow VGC Lab ISTI-CNR, Pisa, Italy
November 2007	Senior Software Engineer Realtime Worlds, Dundee, UK
December 2008 - Present	Software Developer Engineer Microsoft Ireland Research, Dublin, Ireland

Publications

1. Cazzola W., Colombo D., Harrison D.L. "Aspect-Oriented Procedural Content Engineering for Game Design", ACM SAC'09, PSC track, Honolulu
2. Brugali, D., Broten, G.S.; Cisternino, A., Colombo, D., Fritsch, J., Gerkey, B., Kraetzschmar, G, Vaughan, R., Utz, H., Trends in Robotic Software Development,"Software Engineering for Experimental Robotics", Volume 30/2007, pp. 259-266, ISBN 978-3-540-68949-2, Springer
3. Cisternino, A., Colombo, D., Ambriola, V., Combetto, M., Increasing Decoupling in the Robotics4.NET Framework, "Software Engineering for Experimental Robotics", Volume 30/2007, pp. 307-324, ISBN 978-3-540-68949-2, Springer
4. A. Cisternino, D. Colombo, V. Ambriola, G. Ennas, "Increasing decoupling in a framework for programming robots", In proceedings of Principles and practices of software development in robotics (SDIR2005), ICRA Workshop, Barcelona
5. A. Cisternino, D. Colombo, G. Ennas, D. Picciaia, "Robotics4.NET: Software Body for controlling robots", IEE Proceedings Software, Vol. 152, No. 5, pp. 215-222, October 2005
6. Cisternino, A., Cazzola, W., Colombo, D., "Metadata-driven library design", Proceedings of Library Centric Software Development Workshop, October 16, San Diego, 2005
7. Cazzola, W., Cisternino, A., Colombo, D., "Freely Annotated C#", Journal of Object Technology, 4(10):31-48, December 2005
8. Cazzola W., Cisternino A., Colombo D., "[a]C#: C# with a Customisable Code Annotation Mechanism", ACM SAC'05, OOPS track, Santa Fe
9. G. Attardi, A. Cisternino, D. Colombo, "CIL + Metadata > Executable", In proceedings of .NET: The Programmer's Perspective, ECOOP 2003 workshop, Darmstad
10. G. Attardi, A. Cisternino, D. Colombo, "CIL + Metadata > Executable", Journal for Object

Presentations

1. Video Game Code Design and Roles, XNAFest, University of Ulster, 2009
2. Aspect-Oriented Procedural Content Engineering for Game Design, ACM SAC 09, 2009
3. Dependency Injection and Software Engineering practices, Microsoft Ireland Research, 2009
4. Partial Evaluation in Video Game Code, Microsoft Ireland Research, 2009
5. Runtime optimisation through domain knowledge, Microsoft Ireland Research, 2008
6. Runtime AOP with declarative XAML, Realtime Worlds, 2007
7. Program evolution for texture synthesis and computer Vision, Dundee University 2007
8. Video game software development, Microsoft Corporation, 2006
9. Annotating code blocks, an overview on [a]C#, Microsoft Corporation, 2006
10. Software Development and Integration in Robotics SDIR 2007 Understanding Robot Software Architectures, ICRA 2007 Roma, program Committee, 2006
11. International Workshop on Principles and Practices of Software Development in Robotics ICRA 2005 Barcelona, 2005
12. Designing Robot applications for everyday use international workshop, 2005
13. Speaker at the Microsoft Embedded RFP Workshop on Embodied Agents and Robotics: R2D2 project, 2004
14. Speaker at ECOOP 2003 at .NET: The Programmer's Perspective workshop, 2003
15. Invited speaker at the Microsoft Crash course on VisualStorms Project, 2003

Abstract

Software configuration and adaptation are becoming key aspects of Computer Science; programs are executed in stages in very complicated lifetime cycles starting from development to their execution. Various forms of Meta-programming have been developed to support program evolution over time, during development and execution. Virtual machines have greatly encouraged this trend since programs are annotated with metadata that can be easily analysed by meta-programs. In this thesis we investigate a particular class of programs capable of evolving their own structure over time in order to adapt to particular execution conditions and to the user. We investigate this class of programs in the context of computer games, programs with great need for adaptivity both in terms of program specialisation for optimisation and content generation. In any case the program transformation is performed at runtime since it depends upon data available only while executing. Our work is based on programs written for Common Language Runtime and available in the form of intermediate language.

Chapter 1

Introduction

Once programs were written to solve problems, either numerical or data manipulation, and their lifecycle was simple as write→debug→run. Computers have changed the role they occupy in the society, and programs have changed their structure, keeping data-management and algorithm at their core, but also including support for configuration and adaptation to a world made of heterogeneous architectures capable of parallel and distributed execution of programs. The *cloud computing* term has been adopted to indicate part of this infrastructure, in which something that is perceived as a program is in fact the result of complex interactions among a large number of interacting software entities.

Program infrastructures, while preserving their traditional structure oriented to functionality (as widely discussed in the aspect oriented approach), have developed to address the problems posed by the new software systems. In the last fifteen years programming systems based on Virtual Machines (VM) such as the Java Virtual Machine or the .NET Common Language run-time, have literally exploded and are now the basis for an ever growing amount of programs. It is not by chance that Microsoft has built its high performance computing platform using these technologies instead than relaying on more traditional programming languages such as C or C++.

In such a scenario software has continued to grow in size and its man-

agement has become a relevant problem on a large scale. The term *class library* term has been replaced by the *Framework* term in order to acknowledge the change in nature of reusable code. A Framework is needed when objects and patterns are required to develop a software since the base class library (we will refer it as BCL) and the software itself are so big that some facility must be provided to interact with it. Famous technologies are STL, C5, Cocoa, Java, .NET.

A major contribution of wide adoption of the Java run-time system has been the redefinition of what a program is. Instead of being a number of segments of machine code a Java program is a set of classes loaded dynamically and whose methods are defined in terms of the bytecode of a virtual machine. A program is not anymore a monolithic piece of machine language, instead is made of types, annotated so that the virtual execution system can not only execute but also inspect and manipulate the program itself. Reflection is one of the building blocks of these execution environments, and one of the key aspects of the notion of metaprogramming. With .NET also the run-time code generation has been officially supported by the BCL, featuring an environment in which it is natural to write programs that manage, create and execute other programs. In this way Web Services interfaces can be synthesised from their XML description, and code fragments such as regular expressions can be optimised by compiling them on the fly instead of resorting to interpretation. Program manipulation is sometimes iterated, in the sense that a program generates another program which in turn generates another one.

Someone may think that many if not all these considerations can be applied to the LISP system, which is true if we consider only expressivity aspects of the system. But these virtual machines have been optimised to generate on the fly machine language code targeting specific architectures. The virtual machine bytecode is compiled at run-time into machine language thanks to *Just in time* compilation. In the .NET framework the component responsible for the compilation of the bytecode into native code is called **JIT** (Ayc03; SNS03); the JIT is also able to perform optimisation specific to the target machine so that the same code pattern

can be transformed in different ways and with different optimisation strategies. Virtual machines provide also memory management strategies through garbage collectors; those are specific to target machines as we can see with the .NET framework and all its flavours (Compact, Micro, Full, XBox360).

Traditionally program specialisation has always been rooted in the *s-m-n* theorem proved by Kleene and foundation of the idea that we can optimise a program if some of its inputs become known before run-time. Several techniques have been developed in the area of partial evaluation and program specialisation; however, the focus has always been the specialisation before run-time. There are several reasons for this: first of all program manipulation at run-time costs, and only recently hardware has become so powerful to make this approach viable in many contexts; moreover, program manipulation in binary form has always been not so easy, making difficult to express even the simplest transformation.

Recent works in the area of *aspect oriented programming* (see Section 2.3) and, more generally, meta-programming have introduced general purpose tools capable of performing program transformations according to well defined metaphors[1] hiding many of the details involved in the program manipulation. The CodeBricks (ACK03) library has shown that it is possible to define a general purpose operation for expressing code generation at run-time by mixing fragments of pre compiled code and expressing multi stage and homogeneous program transformations. A particular class of meta programs is the class of *self-evolving programs* (HLL06; SE06), which are programs that change their own definition by rewriting themselves during execution. This class includes the set of programs capable to specialise themselves depending on the user input.

In this thesis we are interested in studying the potential impact that this class of programs may have on the nature of programs. We are also investigating the models and tools needed to support this evolutionary process. It is important to notice that we are not restricting ourselves to

[1] Every approach in meta-programming is designed starting from a transformation model that is characteristic. Intentional Programming, Aspect Oriented Programming, Template Meta-programming use different strategies and metaphors.

the problem of program optimisation, but we are willing also to explore other forms of program evolution.

Our studies will be discussed in the context of computer games, an applicative domain which is challenging because of the execution environment and the huge amount of data processed in real time. This domain is also resorting to virtual machines to govern the complexity, and several frameworks for programming computer games are now based on Virtual Machines (a relevant example is the XNA framework (Gou08) that Microsoft has developed to program games also for the XBox360 game system).

The contribution of this work is to study program analysis and transformations in program expressed in intermediate language (i.e. bytecode); in particular we will focus our studies on three aspects of the game application:

- optimisation of general purpose mathematical library
- procedural content engineering
- game logic generation by means of software compositions

1.1 Optimisation of General Purpose Mathematical Library

One of the most critical components of a game engine is the library which deals with math and linear algebra: graphics, simulation and content rely heavily on math related data structures and algorithms (see Appendix B). Due to the pervasion of this code both space and time complexity are extremely relevant however, on the other hand, generality and maintainability are very strong requirements. Approaches like Boost (boo) and Blitz++ (bli) take advantage of the compilation process to perform advanced optimisations based on the expression tree evaluation to specialise the code.

In scenarios involving VMs such as the CLR the impact on the execution is a very complex problem as the garbage collector will deal with

memory allocations, object creations and destruction. Execution and allocation patterns will alter the GC behaviour (see Appendix A) and therefore the overall run-time performance of the application. The JIT however will try to perform optimisation but with a restricted amount of time (a significantly lower time budget than the one available to a C++ compiler) preventing any major optimisations of the code. Another issue complicating this scenario is related to the fact that general purpose optimisations are not the optimal solution since the domain (rendering, simulation, AI, etc.) can be used to adopt more efficient and specific code transformations.

1.2 Procedural Content Engineering

In a game application most of the content is composed of 3D models and other geometric objects used to populate the screen. In a disc based scenario the game data is usually compiled into a very concise binary format, this approach when applied to consoles means that we can have a single continuous block of data on the storage. Such a data representation affords the ability to load a large sequential chunk of bytes from the storage medium and have all of the information ready to be used by the application code (GMC+06a). Making a concise representation of the data[2] is crucial for this scenario so that the application can start presenting coarse representations of content while waiting for the fine grain data to load and be prepared for presentation (different level of detail in different moments).

To use this strategy coarser representations of the game content are built providing a faster start up and affording memory usage optimisation at run-time. Using simplified data to boost the performance of the application has a cost: the cost of extra space on the storage to save simplified data along with the high quality content, and a smart lookup strategy.

Bandwidth is a factor that is quite important when a next generation game is loading live from a media like DVD. Smart caching techniques

[2]Coarser representation of a data are called Level of Detail or LOD.

can be used to address this problem but are applicable only on specific hardware, another way to reduce bandwidth is to generate the data on-the-fly and to store/transmit only the function and the inputs to produce them. Instead of storing a cube in an explicit manner, usually 3D model data are stored as a set of faces and vertices, we store the geometric primitive as a position and set of parameters that can be used to generate the unique instance. This provides a potentially significant saving over the amount of storage space required for the explicit alternative.

Computational cycles are far less expensive than the bandwidth to deliver the content, a procedural generation from a very concise representation can lead to a drastic requirement relaxation for streaming based applications. Procedural generation of content (and we will refer to this technique as *procedural content* generation) is a set of input values and the code to build the destination data. Given $\mathcal{G}$ the set of *generator functions* the procedural content is defined as:

$$\bigcup_{g \in \mathcal{G}} \{g\,(s) \mid \forall s \in \text{Domain}(g)\}$$

in a specific point the procedural content is represented by the couple (g, s) where $g \in \mathcal{G}$ and $s \in \text{Domain}(g)$.

In our case, the chosen family of generators must output data displayable on screen. So the co-domain of the generators is the set of the, so called, *renderable mesh*) defined as:

$$RM = \{(m, g) : m \text{ is a graphics material, } g \text{ is the geometry}\}$$

and the generator, called *inflator*, is represented by

$$\text{inflator}_i : \text{Domain}(\text{inflator}_i) \rightarrow RM$$

Designing and engineering procedural content is quite important because the amount of data can increase (especially from a semantic perspective) whilst keeping the occupied space constant or at least bounded. A lot of *digital content creation tools* (we will refer them as DCC) store user files in a way that the content affords easy manipulation instead of an

optimised raw data set. Quite often data are kept as extensive as possible in the content production pipeline and get finalised once built for the final application. This finalisation process usually entails the generation of coarser granularity *levels of detail* and the computation of final layout on storage. This can be thought of as storing data as a program with an input set until ready for the final build where the target product will use the program's output with the given input set. Most of the strategies for content simplification deal with the final content and not with the content generation procedure where the static output is needed to obtain coarser version of the data. Changing the perspective from data to program can lead to a different paradigm for content engineering, more related to the data synthesis than to the data representation itself.

The video game Crackdown[3] from Realtime Worlds[4] uses procedural built cities and road, this means that, on the disc, data are represented by the code to generate the content and the input data for the procedure and the rendering step.

1.3 Game logic Generation

Most games are composed of a sequence of levels the player traverses, once the completion condition for a level is met a transition is triggered to a new level (or to the final screen if the game is beaten). Sometimes the word level is used also to refer missions (in games like Grand Theft Auto [5]) or quests (as in World of Warcraft [6]); more generally a level can be seen as a particular node inside a finite abstract state machine (AST) modelling the game story (a game can contain forests of AST) and the player must satisfy at least one of the node conditions to execute a transition toward another state[7].

[3] Crackdown at http://crackdownoncrime.com.

[4] Realtime Worlds at http://www.realtimeworlds.com.

[5] GTA at http://en.wikipedia.org/wiki/Grand_Theft_Auto_(series)

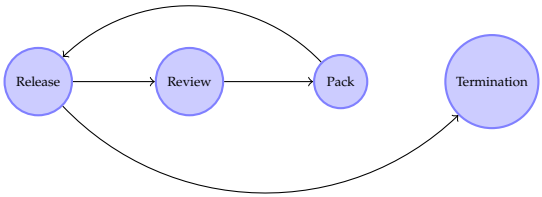
[6] WoW at http://en.wikipedia.org/wiki/World_of_Warcraft

[7] AST can be used to model both the game status or the player status; some games do not have a "story" so cannot have AST associated with the game logic, examples of such a games are typically MMORPGs like of World of Warcraft, APB, etc.

The life span for a game is determined by the *engagement*[8] of the player which is very important as it drives the revenue of the company. One of the most common approach for extending the life is the provisioning of additional content (usually called expansion pack, mission pack) to revive the interest of the players.

Pack creation has a cost for the company since designers have to work on new content and new logic; as the pack creation effort is meant to maintain the player base and attract new consumers, statistics and feedback are used to drive the new content creation. The release cycle for video games can be formalised as shown in Figure 1.

Figure 1: Life Cycle for games



In video game development levels are usually created by formalising the set of features that can be used by level Designers to assemble the level. Game Developers are responsible for creating the code behind the formal abstraction so that the game can load the instance variables created by Game Designers and then use them to initialise the level instance. Using reflection and meta-programming techniques it is possible to make the Review step automatic, in such a scenario the game can use the metrics to generate new levels and reuse them at run-time.

[8]Generally the *engagement* is built on either emotional connection or knowledge of the rules of the game world; game designers uses those two elements to maximise the time players spend with the game and their satisfaction.

1.4 Organisation and Reading Plans

The goal of the thesis is to provide capabilities to program (and thus programmers) to allow software to evolve and adapt at run time. The domain knowledge can be used to perform transformations that can improve different sections of the program with different approaches. Real world applications are generally created with different domains co existing inside the code base, games for example are composed of:

- mathematic domain
- algebraic domain
- geometric domain
- physic domain
- graph theory domain
- AI domain
- rendering domain

Multiple domains inside the application code base can benefit from different programming patterns, abstractions and optimisations.

In Chapter 2 we present the state of the art and the technologies used to design our approach. The Chapter introduces the meta-programming field focusing on Aspect Oriented Programming, Virtual Machines and Strongly Typed Execution Environment are described as well. The VM adoption in software is increasing as the byte code retains information that common compilers lose during the translation from source code to executable. Aspect Oriented Programming is interesting because its model is based on code injection at specific points. Most of the AOP implementations are source to source transformations which are infeasible in real world scenarios where source is usually unavailable. Focusing on byte code manipulation is more interesting since it allows targeting of all the languages available for the VM.

Chapter 3 will be describing the run-time code generation approach we propose and the code query technique. The VM dynamic loading allows code transformations to be performed at load time, this will avoid the need to process the entire code base and to focus on loaded modules only. As described in the chapter byte code can be inspected obtaining also run time behaviour information (like real local variable usage, code access security changes), along with type and execution aspect of the software transformations can be performed safely from both type and execution perspective.

In Chapter 4 we present the experimental result we obtained applying our technique to the problem introduced in the Introduction. The chapter shows how meta-programming with run time code generation support can impact the performance, the design, and the development cycle of video game software. The experiments described will provide a quick glimpse about using domain knowledge to create an application capable to adapt and evolve at run time.

Appendix A will provide information about the garbage collection strategy in the .NET framework. This is relevant to understand the problematic that a GC can introduce in realtime applications.

Appendix B describes the software architecture behind video games and the core software components involved in development.

The details about garbage collection and game software architecture are provided so that the reader can see how game engines are actually a specific targeted implementation of VM as JVM and CLR.

Appendix C is focused on realtime rendering and a meta-programming based approach to improve it, this is one of the most relevant issues for video games and some of the argumentations in Chapter 2 will be referring to it.

Finally Appendix D contains code snippet with details about Chapter 4 experiments.

Chapter 2

State of the Art and Technologies

This chapter has the goal to introduce the state of the art and the technologies relevant to our meta-programming model. The Virtual Machine scenario is introduced giving an over view of its features. Particular attention is paid to reflection and code generation capability of Virtual Machines, their type systems and execution model.

Since our approach is based on **Microsoft**.NET more details are provided on features like function objects and byte code structure. We will be introducing the .NET delegate type, this is quite relevant to our work since is the way .NET lifts functions to types (even if this is not the same concept of functional language as F# (SGC07)).

Meta-programming is introduced as well with a section giving details on Aspect Oriented Programming and some of the most interesting implementations with respect to our goal. The section about meta-programming and AOP is supported with code examples from the approaches we present.

2.1 Virtual Machines

2.1.1 Strongly Typed Execution Environments

In the last few years the number of languages based on virtual machines has substantially increased. There are several reasons for this trend: hardware is becoming ever faster and we can afford to pay some overhead for more reusability, security and robustness from our programs; programming has become a harder task requiring an ever increasing number of services.

Garbage collection (see Appendix A for details), libraries of pre built functionality (base class libraries, we will refer them as *BCL*) are often considered a requisite for a programming language. Programming languages based on virtual machines allow programs to be run across different platforms at only the cost of porting the execution environment rather than having to recompile every program. Virtual machines also offer the opportunity of achieving better security: the execution engine mediates all accesses to resources made by programs verifying that the system can not be compromised by running applications.

Java is a successful programming language based on a virtual machine that has been considered closer to compiled languages such as C++ rather than to interpreted languages such as Perl. In the past other programming languages with the same architecture, essentially p-code (PD83), have been proposed (see for instance the introduction of (Kra98)) but Java was the first to have a huge impact on the programming mainstream. Nowadays Microsoft is pushing virtual-machine programming languages based on the Common Language Infrastructure (CLI) (standardised by ECMA (ECMb) and ISO (ISOb)). The core of CLI is the virtual execution system also known as Common Language run-time (CLR).

Both JVM (bTLY99) and CLR (ECMb) implement a multi-threaded stack-based virtual machine, that offers many services such as dynamic loading, garbage collection, Just In Time (JIT) compilation, and stack inspection. When a virtual machine is stack-based the operations read values from a stack of operands and push the result on the stack.

Many other languages adopt a stack based virtual machine: OCaml (OCV), Python (pvm), TEA (tvm), XSLTVM (xvm), are but few examples. An alternative to stack-based virtual machines are register-based virtual machines, these offer the abstraction of registers instead of a stack of operands to pass the values to the instructions; an example of register based virtual machine is the Perl virtual machine implementation called Parrot (par). Although both stack-based and register-based machines are Turing equivalent there is a fervent debate among the implementers of which model can offer better performance. In (DP), for instance, is proposed an alternative interpreter for Python which is register-based.

In this thesis we focus our attention on stack-based virtual machines. In particular we are interested in virtual machines which are type oriented. JVM and CLR are examples of such machines whereas CVM, the virtual machine of OCaml, it is not. CVM does not provide the ability of defining types: it provides simple operations and the only types other than strings and numbers are closures and word blocks.

Our interest relies in those execution environments that contain information about the program types, and their structure. In particular we require that the environment is able to reflect types and their methods to running programs. The JVM and CLR are good examples of these environments. We do not strictly require support for inheritance; we rather should be able to close a value to a function.

A Strongly Typed Execution Environment (we will refer it as *STEE*) is an execution environment which implements a virtual machine that provides an extensible type-system, reflection capabilities, and the execution model which guarantees that type of values can always be established and values are accessed only using the operators defined on them. The results presented in this thesis can be extended to stack-based STEEs like JVM and CLR. Most of the details and the model are built around CLI because it is a standard. Nonetheless JVM and CLR are quite similar and the results can be expressed in the same way.

Recently VMs are taking the abstraction level higher, trying to hide the concepts of thread, process, and even machine to the programmer. CLR does this through the BCL thanks to the task oriented programming

and the service model of .NET Remoting and WCF. That HW abstraction is anyway not enough to cover some of the techniques that today are prevalent in High Performance Computing (*HPC*) such as the usage of GPU (Fer04; RF05; Ngu07) (or other specialised processing units).

One problem with CPU-GPU interaction is that the GPU capabilities are defined by the accelerator model and brand, therefore the code must deal with a set of possible scenarios (different shading language version, number of instructions, set of instructions). Cuda (NVI), OpenCL (ocl), and RapidMind (RPM) offer to C++ a compile time support and a run-time dispatcher to deal with such scenario. Even if tools can provide a configuration facility the programmer must provide all the implementation for the target platforms potentially involved in the computation (RapidMind provide a BCL to handle GPU, multi core and CELL scenarios).

Compiler extensions and compile time approaches make scalability and performance boosting quite complex in heterogeneous hardware and deployment scenarios, programming abstractions are not enough to address this problem while a joint action of language, VM and run-time transformations (including JIT) could make this programming paradigm more accessible and easier to design for and engineer.

2.1.2 Structure of a STEE

A STEE is a type driven execution system. Its input is a program expressed in a language which is called intermediate language. The CLR provides the Common Intermediate Language (CIL or IL), the JVM executes bytecode. The intermediate language is shipped in binary form, though usually is platform independent. In Figure 1.2 is shown a typical state model for an EE such as the JVM or CLR.

The intermediate language expresses a program executed by a thread: the execution begins from a method which may invoke other methods. Each method invocation corresponds to the addition of a stack frame which contains the local variables and the input arguments. Both JVM and CLR share essentially the same structure of the stack frame from the

IL standpoint.

A STEE can interpret the IL language or compile it using a Just In Time compiler into machine language. The JVM is an interpreter that could rely on a JIT compiler to improve execution speed; CLR assumes that the IL is always compiled before execution. In this thesis we assume this level of detail of the execution system which is the model of the virtual machine. This is a convenient model to work with: it hides many details such as registers, stack and heap implementation. This is the reason for many researchers decision to work at the level of IL/bytecode although this may introduce some overhead (CTHL01; TSDNP02; MY01; Ses).

2.1.3 Delegates in CLR

The CLR provides a special mechanism for having function objects: delegates. Although a delegate is treated ad-hoc inside the execution engine, from an IL standpoint this is nothing more than a class that derives from `System.MulticastDelegate`. Each delegate class is characterised by an Invoke method whose signature is the same of the class of methods it describes. Let us consider for instance the C# declaration:

```
1 delegate int F(int a, int b);
```

This declaration corresponds to the following class:

```
1  sealed class F : System.Delegate {
2    public F(object o, IntPtr m) {}
3    public virtual
4      System.IAsyncResult BeginInvoke(int32 a, int32 b,
5        System.AsyncCallback callback, object o) {}
6    public virtual
7      int32 EndInvoke(class System.IAsyncResult r) {}
8    public virtual
9      int32 Invoke(int32 a, int32 b) {}
10 }
```

As explained in section 13.6 of Partition II of ECMA standard 335 (ECMb), a delegate class should define a constructor and three methods, namely `BeginInvoke`, `EndInvoke` and `Invoke`. These methods do not have a body because of their special handling. When a delegate is created the

constructor requires two arguments: an object *o* and a pointer *m* to the code of a method that should satisfy the following requirements:

- *o* cannot be null unless *m* is a pointer to a static method
- if *o* is an instance of class *c* then *m* should be a pointer to a method of the same class *c*
- the signature of *m* should be the same of `Invoke`

When the method `Invoke` of a delegate is called the execution environment invokes the method referred by m on the object o (if m is an instance method). Delegates can be represented in Java as interfaces with single methods. Although the run-time is unaware of the notion of delegate so the implementation would be less efficient. Peter Sestoft in (Ses) describes a possible implementation of delegates in Java. A hand-written schema for having anonymous functions is used by DynJava (OMY01) to run code generated at run-time.

2.1.4 Reflection in STEE

Reflection is often considered an essential feature of a modern programming language. Component systems are easier to use as long as it is possible to query the structure of a component at run-time instead of at compile time: it is possible to write generic components capable of dealing with other components by inspecting their structure at run-time. The evolution of both COM (Rog97) and CORBA (Lew98) has been towards the support of metadata associated with components and accessible at run-time.

A system is reflective if it is able to access its own internal execution state and possibly manipulate it. Reflection can be implemented at different levels of complexity (KCC00):

- **Introspection:** the program can access a representation of its own internal state. This support may range from knowing the type of values at run-time to having access to a representation of the whole source program.

- **Intercession:** the representation of the state of the program can be changed at run-time. This may include the set of types used, values and the source code.

Both introspection and intercession require a support, called reification, to expose the execution state of a program as data. Despite its great expressivity, support for reflection may have a significant impact on the performance of program execution. Traditionally compiled languages offer little reflection support: the source program and its abstractions are no longer available at run-time. C++, for instance, supports run-time type identification (RTTI) that allows a program to have exact information about type of objects at run-time. Stroustrup (Str94) strongly encourage the use of compile-time type checking relying on RTTI only in absolutely necessary cases. Interpreted languages tend to offer rich reflection support because the execution system holds information that compiled languages throw away during compilation[1]. Interpreters have exact information about type of values and may also interpret code generated at run-time. ECMA Script (ecmc), which is the standard for Javascript, supports introspection of both data types and source code.

Reification exposes an abstraction of some elements of the execution environment. These elements may include programming abstractions such as types or source code; they may also include other elements, like the evaluation stack (as in 3-LISP (SdR84)), that are not modelled by the language. Each element is exposed with a set of abstract operations to manipulate it. For compiled languages it could be harder to reflect elements of the source language: the object program runs on a machine that usually is far from the abstract machine of the source language. Enabling RTTI in C++, for instance, requires that the run-time support contains additional code to keep track of types at run-time. Besides, the programmer would expect abstractions compatible with the structure of the programming languages abstract machine (unless he is interested in manipulating the state of the machine which is target of the compilation).

[1] In .NET and Java is it also possible to extend type information adding custom element metadata to a type or assembly, this mechanism is grained on member level such as fields, properties, methods.

STEEs require information on types in order to enforce type safety at run-time[2]. It is easy to expose them to running programs as objects and it comes for free.

The CLR exposes its reflection system through the classes contained in the `System.Reflection` namespace. Classes are described by instances of `System.Type` class and accessed through the static method `Type.GetType()` or the instance method `GetType()` inherited by `System.Object`.The JVM adopts the same structure with different names. Using this support it is possible to inspect the types of a program as well as changing fields of objects and invoking methods.

Languages that generate code for a STEE tend to expose types of the execution environment as types within the language. This assists the programmer in reusing their libraries as well as the base class libraries that are part of the system. An example of this is SML.NET (sml), an implementation for .NET of SML: the language has been extended to allow manipulating CLR types. Thus from the SML.NET programmer standpoint the reflection is limited only to the types of the execution engine, other types are not easily accessible (perhaps because they are mapped to non trivial CLR types). Though method bodies are part of the type definition usually they are not available through the reflection interface. There are several reasons for this choice:

- Execution engine knows only their compiled version and the source code is no longer available. The only form available is the list of the abstract machines instructions, which are unknown at the source language level

- It would be possible to support only introspection of code: allowing changes to the running code would have impact on systems performance. Moreover the execution environment enforces properties such as type safety and protection mechanisms that would be harder to guarantee if intercession on method bodies was allowed

[2]Type safety can be enforced relying only on type-checking performed at compile time (as it happens in ML), though type safety depends on the compiler. STEE enforce type safety at run-time through run-time types.

- Program source could be made available within metadata, but often companies or programmers do not like to distribute their source code

In synthesis an appropriate abstraction for code is available such that programs can use it in a profitable way. In this dissertation we introduce an appropriate abstraction for representing code executed by a STEE. We introduce the notion of code value: a value which represents a well defined piece of code. Method bodies can be represented by such values, though this is a weak form of reflection: it is possible to manipulate method bodies without being able to access their structure. This extension does not introduce any additional overhead into the execution engine and it is able to support meta-programming and multi-stage languages.

2.1.5 Bytecode analysis

Virtual machines use an intermediate language for several reasons:

- **Security and verification:** the loading model based on dynamic loading of types requires metadata to be available at run-time and bytecode allows for verification of programs
- **Adaptation:** continual hardware evolution asks for a better separation between programs and computing architectures in order to better exploit the capabilities of new components
- **Rich set of services:** programs are instrumented with a number of services including memory management, security, network communications, graphics, interoperability, and many others. These services are exposed either as core services or through the base class library.

Intermediate language analysis allows us to extract relevant information about program structure that can be used to analyse and transform programs in binary form. The opportunity to transform programs in

their binary form allows several varieties of program specialisation relevant to software adaptation. In particular it is possible to mimic metaprogramming patterns typical of generative programming and based on C++ template metaprogramming.

Describing a type-system is a complex task. In Partition II of Common Language Infrastructure standard (ECMb; ISOb) the metadata format used by .NET assemblies is defined. A single assembly describes a set of types and may contain references to other assemblies. Thirty-six tables are used to describe all the types contained in the assembly and their relation with types contained in other assemblies.

Metadata contains all of the information related to types and their structure (with additional custom information provided by the programmer). Both in JVM and CLR binary formats it is possible to include extra information to metadata[3]. Method bodies are defined using the intermediate language, though they are not accessible through reflection abilities. Nevertheless several libraries have been developed to address this issue; in our system based on .NET we used CLIFileRW library (Cis).

The choice to represent types and code in intermediate language form, rather than machine code, is somewhat constrained because of design goals. Without information on types it is almost impossible to have general support for dynamic loading of modules (one weakness of COM (Rog97) was the incomplete type system), reducing reusability of the software. The ability to verify that types are used correctly helps to avoid memory corruption through misuse of programming abstractions: this contributes to reduce the corruption of the execution state and implement security checks.

Types are good for software reuse because they are one of the foundations of modern programming languages. Traditionally run-times, like the C run-time or even the ML (OCV) run-time, share a little amount of

[3]In .NET the ability to annotate using metadata is exposed in programming languages such C# (ECMa; ISOa): all programming elements exposed through reflection (types, methods, assemblies and so on) can be annotated by means of custom attributes (NV02). Java bytecode (LiY99) is also able to contain custom information (class file attributes) though this feature is not exposed to the language: it was conceived to support programming tools like debuggers.

programming abstraction with the programming language: in C numerical values are the same as used by the processor, in ML there is slightly more similarity. When types become a shared abstraction between the execution environment and the programming language a larger amount of information is made available about the program to the run-time and to all the other programs interested in code analysis. Partial evaluators and programs alike can even execute these binaries with different semantics from the one of the execution environment. The simplicity of manipulating intermediate language and metadata makes code analysis possible that would be hard to perform in other contexts. In (MY01; TSDNP02; Cis03) are reported examples of analysis and manipulation of binaries for CLR and JVM.

Besides ordinary code analysis, the fact that types are shared between the execution environment and the programming language implies that it is possible to provide libraries without implementation. The programmer makes use of such libraries and its invocation to library methods and types are used as placeholders into the binary format for further processing. After compilation programs may manipulate the output looking for special patterns inside the intermediate language, types and metadata. The post processing may be done for several reasons: in (Cis03) it is done for run-time code generation; a post processor would optimise patterns deriving from the use of domain specific operators; some aspect could be woven into the code; the executable is translated into an executable for a different platform.

2.1.6 CLR components

The CLR virtual Machine has to deal with several management aspects during a programs execution. Those aspects are important components in program transformation where metaprogramming behaviours are moved inside the run-time itself. The CLR Host management components are:

- **Assembly loading management**: Enables the host to customise the locations from which assemblies are loaded, the way versions are managed, and the formats from which assemblies can be loaded.

For example, assemblies could be loaded from a database instead of from files on the hard disk

- **Policy management**: Enables the host to specify the way program failures are handled, to support different reliability requirements
- **Memory management**: Enables the host to control memory allocation by providing replacements for the operating system functions that the CLR uses to allocate memory
- **Garbage collection management**: Enables the host to implement methods to receive notification of the beginning and end of garbage collection. Enables the host to initiate collections, to collect statistics, and to specify some characteristics of collection
- **Debug management**: Enables the host to discover whether a debugger is attached, to provide additional debugging information, and to customise debugging tasks
- **CLR event management**: Enables a host to register for notification of the events like domain unloading, stack overflows
- **Task management**: Enables the host to be notified whenever a thread makes a transition between managed and unmanaged code. Allows the host to control thread affinity, when tasks are started and stopped, and how they are scheduled
- **Thread pool management**: Enables the host to implement its own thread pool for the run-time to use
- **Synchronisation management**: Enables the host to implement its own synchronisation primitives for the run-time to use. The host can provide events, critical sections, and semaphores
- **I/O completion management**: Enables the host to implement its own implementation of asynchronous input/output

Thanks to the set of services exposed by the CLR it is possible to analyse and react to the behaviour of the application. A common tool which can

be created using these features is a profiler. Other common programs realised using the hosting apis are the sandbox environments, they are very useful in security and policy enforcement scenarios (like code execution in browsers, etc.).

2.2 Meta-programming

Meta-programming is a general idea driving those programs (called *meta-programs*) which manipulate other programs (called *object-programs*). Meta-programming has been studied in various forms and aspects, ranging from compilers, to partial evaluation and specialisation. Recently meta-programming techniques have focused mainly on software engineering, in particular aspect-oriented programming, a research area where programs have been considered as slices of code that are rearranged by the so-called aspect-weaver into the final program. Another relevant area in which metaprogramming techniques have been adopted is generative programming, in particular for domain specific languages, where programs are optimised depending on how domain operators are used.

As reported in (She01) an object-program is any sentence in a formal language. Meta-programs include things like compilers, interpreters, type checkers, theorem provers, program generators, transformation systems, and program analysers. In each of these a program (the meta-program) manipulates a data-object representing a sentence in a formal language (the object-program).

In the following section of this chapter meta-programming techniques will be exposed with more details and particular focus on techniques that uses VM and byte code level manipulation.

2.2.1 Reflection, Meta-Programming and Run-Time Code Generation

The current programming model is based on an execution paradigm which does not seem to scale well. As pointed out in (CE00) programmers are forced to reimplement functionality daily to provide more a op-

timal solution than existing implementations. We are incapable of writing perfectly reusable software though many reuse mechanisms have been prevalent for a long period of time.

The reuse mechanisms we can rely upon are ultimately based on the notion of procedure: a fragment of code that can be reused many times during the execution of a program. We have built many programming abstractions but the primitive notion is still the function, or method, or procedure or any other of the many names used for it.

If programs were made only by functions we would have probably achieved better reuse. Unfortunately programs need to represent data structures and manipulate those through functions. The notion of reuse implies that data types and functions provide interfaces that are general enough to be reused. Within this need for generic interfaces lies the main problem with code reuse models based on reusing prewritten functionality.

If we assume that we want to reuse a function we should make design choices with the smaller number of constraints in its usage. Although in this way we reduce the number of assumptions on which we rely upon affecting dramatically the performance of a program. In many application domains standard libraries are avoided in favour of highly efficient, hand-written, non generic libraries. An evident example of this trend is three dimensional graphics (Ang06): the rendering pipeline is based on matrix operations. All 3D libraries provide their own matrix implementation rather than reusing a general purpose, full fledged matrix library. Why? Because a general purpose library for numeric computation cannot assume fixed size matrices with few operations. Thus a matrix library for 3D graphics will be faster than a generic one.

The problem is that whenever we offer an option to the programmer we almost surely add at least a test in our code. The goal of generative programming (CE00) is to provide this reusable set of functionality in a way that specialise libraries and components when the options are specified by the programmer. To achieve real code optimisation we need to generate programs whenever some information becomes available that allows us to employ better algorithms or reducing the number

of tests needed by the program. A number of programming techniques are discussed in the research community to achieve the same ultimate goal. run-time code generation (LL94; MY01; LL96b; BDB00; OMY01; GMP+00; CN96; PA02; Eng96; LL96a), Meta-programming (She01; SJ02; Vel98; TS97), Partial evaluation (JGS93; JG02; She01). It is interesting to notice that the goal of specialising components requires some abilities:

- Analysis of components usage
- Generation of the code representing the optimised version of some portion of a program
- Iterated program specialisation in order to make use of information when it becomes available during its life cycle.

Every aspect of the problem influences programming systems at all levels, we need programming constructs for controlling the generation process and the specialisation of programs; the code analysis and generation support should be included in run-time libraries so that programs can rely upon.

2.2.2 What kind of meta-programs are there?

Meta-programs fall into two categories: program generators and program analysers. A program generator (a meta-program) is often used to address a whole class of related problems, with a family of similar solutions, for each instance of the class. It does this by constructing another program (an object-program) that solves a particular instance. Usually the generated (object) program is specialised for a particular problem instance and uses less resources than a general purpose, non-generated solution.

A program analysis (a meta-program) observes the structure and environment of an object-program and computes some value as a result. Results can be data- or control- flow graphs, or even another object-program with properties based on the properties of the source object-program. Examples of these kind of meta-systems are: program transformers, optimisers, and partial evaluation systems. In addition to this

view of meta-programs as generators or analysers (or a mixture of both), there are several other important distinctions.

- **Static vs. run-time.** Program generators come in two flavours: static generators, which generate code which is then written to disk and processed by normal compilers etc. and run-time code generators which are programs that write or construct other programs and then immediately execute the programs they have generated. If we take this idea to the extreme, letting the generated code also be a run-time code generator, we have multi-stage programming. Examples of run-time program generators are the multi-stage programming language MetaML (OS99; TeaBS98), run-time code generation systems as the Synthesis Kernel (Mas92; CPI88), 'C (PHEK99), and Fabius (LL). An example of a static program generator is Yacc (Joh75).

- **Manually vs. automatically annotated.** The body of a program generator is partitioned into static and dynamic code fragments. The static code comprises the meta-program, and the dynamic code comprises the object-program being produced. Staging annotations are used to separate the pieces of the program. We call a meta-programming system where the programmer places the staging annotations directly a manually staged system. If the staging annotations are place by an automatic process, then the meta-programming system is an automatically staged system. Historically, the area of partial evaluation pioneered both the technique and terminology of placing the staging annotations in an automatic way without the intervention of the programmer. Write a normal program, declare some assumptions about the static or dynamic nature of the programs inputs, and let the system place the staging annotations. Later, it became clear that manually placing the annotations was also a viable alternative.

- **Homogeneous vs. heterogeneous.** There are two distinct kinds of metaprogramming systems: homogeneous systems where the

meta-language and the object-language are the same, and heterogeneous systems where the meta-language is different from the object-language. Both kinds of systems are useful for representing programs for automated program analysis and manipulation. But there are important advantages to homogeneous systems. Only homogeneous systems can be n-level (for unbounded n), where an n-level object-program can itself be a meta-program that manipulates n+1-level object-programs. Only in a homogeneous meta-system can a single type system be used to type both the meta-language and the object-language. Only homogeneous meta-systems can support reflection, where there is an operator (*run* or *eval*) which translates representations of programs, into the values they represent in a uniform way. This is what makes run-time code generation possible. Homogeneous systems also have the important pedagogical and usability property that the user need only learn a single language.

2.3 Aspect Oriented Programming

Amongst all the approaches for metaprogramming we found Aspect Oriented Programming a very interesting flavour to be investigated further. For a complete overview of the meta-programming techniques we suggest to see (CE00) and (She01). The choice of AOP as starting point for our work is motivated by the transformation model that AOP uses to realise meta-programming. In AOP piece of code called *aspects* are injected in specific points (pointcuts) of other programs matching an expression (join point).

Aspect Oriented Programming (AOP) is a direction in programming originally proposed by researches from Xerox Palo Alto Research Center (asp). The main observation that lies behind AOP is that, though the separation of concerns is a fundamental engineering principle, programming languages do not provide full support for it. Programming languages tend to stress the functional aspect of the program, where the behaviour of a program depends on the execution of a lot of procedures.

Since first year courses it is said that a problem should be decomposed in smaller problems and solved separately and the outcome recombined in the final solution. Kiczales et. al. noticed that this approach to problem decomposition focuses only on separating its functional aspects.

It is well known that there are programming aspects which are orthogonal to program functionality but are essential. Thus, with the exception of toy programs, we cannot really separate all the concerns of a program. The goal of AOP is to provide methods and techniques for decomposing problems into a number of functional components as well as a number of aspects that crosscut functional components, and then compose these components and aspects to obtain system implementations. The activity of adding an aspect to a component is called weaving.

An aspect oriented system provides a means for identifying *join points*, which are the points in a program that can be used to refer points in the program where effects can be performed by an appropriate mechanism. Aspect weaving is performed by an aspect weaver that is responsible for following *advices* at given join point matches (*pointcuts*). The way join points are defined depends on the join point model adopted which may allow defining syntactic patterns as it happens in systems like AspectJ (asp), or characterise in a more semantic join points manner. Advices indicate the actions to be performed at specified join points and usually allow inserting before or after the join point code. Again the weaving process can be performed at source code level or on the executable. Although the second approach has been used the most effective has been the former since advices need to refer to objects where the advice code must be inserted, and their specification is more understandable if performed on the source code.

2.3.1 AspectJ

AspectJ (asp) is the first of the AOP systems and it is often considered as a reference in the field. The approach of the tool is rather syntactic since it is implemented mostly as a source-to-source code transformation system based on the Java programming language. AspectJ is also capable

of weaving aspects when classes are loaded, though the expressiveness of the join point model used for defining join points at load time is a strict subset of the model available when weaving is performed at source code level.

As discussed in (DFSJ08; EL03) AspectJ can use the polymorphism to extend the flexibility and the expressivity of aspects. The AspectJ compiler is also integrated in the Eclipse IDE to support the programmer providing different views of the program as a result of the crosscutting of different concerns.

Listing 2.1: AspectJ sample Program

```
1  public class BankAccount {
2        private String name;
3        private float deposit;
4        public BankAccount(String name, float deposit) {
5              super();
6              this.name = name;
7              this.deposit = deposit; }
8  public boolean withdraw(float amount) {
9        if (deposit > amount) { deposit -= amount; return true;}
10       else { return false; }
11       }
12  public String getName() { return name; }
13  public void setName(String name) { this.name = name; }
14  public float getDeposit() { return this.deposit; }
15 }
16                                                          Aspect
17  public aspect AccountNameCheck {
18       private boolean CheckAccountName(String name) {
19             return (name == "Danisio");}
20
21       pointcut WithDraw(BankAccount account, float amount):
22             target(account) && args(amount) &&
23             execution(boolean withdraw(float));
24                                                          Pointcut
25
26       boolean around(BankAccount account, float amount):
27             WithDraw(account, amount) {
28             if (CheckAccountName(account.getName()) ) {
29                   return proceed(account, amount); }
30             else { return false; }
31       }
32                                                          Advice
33  }
34
```

The code in Listing 2.1 defines a class `BankAccount` with a method which can perform withdrawal if the deposit is positive; with the aspect we want to add a check to test if the Name of that account is permitted to perform withdrawals. The aspect matches the `BankAccount` type and its subclasses surrounding the execution of the join point with some extra code (the original code is executed when the *proceed* instruction is fired). Marching the execution instead of the call makes a difference in the advice step, in this example the classs method is the one woven, while in a call pointcut all the calls to the method are the woven. AspectJ pointcuts are quite expressive and we can also express some more detailed aspect. For example we can formulate the aspect to be matched when the deposit field is changed inside the code of the `BankAccount` method.

This is very helpful when dealing with types and their subtypes (in the case of a virtual implementation of the withdrawal method) as can occur in the case where the method has been overridden. The `withincode()` pointcut uses the lexical structure of the program and so the infrastructure needs to be able to weave the source code.

AspectJ can perform the weaving process at load time for dynamic aspects and so can involve the evaluation of expression with values at run-time. The problem comes with introductions (in AspectJ introductions are called inter type declarations): they are static aspects of a type in AspectJ so the weaving is performed at compile time therefore we must be careful with the aspects definition if we want to be executable at run-time.

A *pointcut* is a program element that picks out join points and exposes data from the execution context of those join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other *pointcuts*. The primitive pointcuts and combinators provided by the language are:

- **call(*MethodPattern*) :** Picks out each method call join point whose signature matches *MethodPattern*.

- **execution(*MethodPattern*) :** Picks out each method execution join point whose signature matches *MethodPattern*.

- **get(*FieldPattern*) :** Picks out each field reference join point whose signature matches *FieldPattern*[4].

- **set(*FieldPattern*) :** Picks out each field set join point whose signature matches *FieldPattern*[5].

- **call(*ConstructorPattern*) :** Picks out each constructor call join point whose signature matches *ConstructorPattern*.

- **execution(*ConstructorPattern*) :** Picks out each constructor execution join point whose signature matches ConstructorPattern.

- **initialization(*ConstructorPattern*) :** Picks out each object initialisation join point whose signature matches ConstructorPattern.

- **preinitialization(*ConstructorPattern*) :** Picks out each object preinitialisation join point whose signature matches ConstructorPattern.

- **staticinitialization(*TypePattern*) :** Picks out each static initialiser execution join point whose signature matches TypePattern.

- **handler(*TypePattern*) :** Picks out each exception handler join point whose signature matches *TypePattern*.

- **adviceexecution() :** Picks out all advice execution join points.

- **within(*TypePattern*) :** Picks out each join point where the executing code is defined in a type matched by *TypePattern*.

- **withincode(*MethodPattern*) :** Picks out each join point where the executing code is defined in a method whose signature matches *MethodPattern*.

[4]Note that references to constant fields (static final fields bound to a constant string object or primitive value) are not join points, since Java requires them to be inlined.

[5]Note that the initialisations of constant fields (static final fields where the initialiser is a constant string object or primitive value) are not join points, since Java requires their references to be inlined.

- **withincode(*ConstructorPattern*) :** Picks out each join point where the executing code is defined in a constructor whose signature matches *ConstructorPattern*.
- **cflow(*Pointcut*) :** Picks out each join point in the control flow of any join point P picked out by Pointcut, including P itself.
- **cflowbelow(*Pointcut*) :** Picks out each join point in the control flow of any join point P picked out by Pointcut, but not P itself.
- **this(*Type or Id*) :** Picks out each join point where the currently executing object (the object bound to this) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any join points from static contexts.
- **target(*Type or Id*) :** Picks out each join point where the target object (the object on which a call or field operation is applied to) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any calls, gets, or sets of static members.
- **args(*Type or Id, ...*) :** Picks out each join point where the arguments are instances of the appropriate type (or type of the identifier if using that form). A null argument is matched iff the static type of the argument (declared parameter type or field type) is the same as, or a subtype of, the specified args type.
- **PointcutId(*TypePattern or Id, ...*) :** Picks out each join point that is picked out by the user-defined pointcut designator named by PointcutId.
- **if(*BooleanExpression*) :** Picks out each join point where the boolean expression evaluates to true. The boolean expression used can only access static members, parameters exposed by the enclosing pointcut or advice, and *thisJoinPoint* forms. In particular, it cannot call non-static methods on the aspect or use return values or exceptions exposed by after advice.

- **! *Pointcut* :** Picks out each join point that is not picked out by Pointcut.
- ***Pointcut0* && *Pointcut1* :** Picks out each join points that is picked out by both *Pointcut0* and *Pointcut1*.
- ***Pointcut0* ||*Pointcut1* :** Picks out each join point that is picked out by either *Pointcut0* or *Pointcut1*.
- **(*Pointcut*) :** Picks out each join points picked out by *Pointcut*.

AspectJ supports the following advices:

- **before**
- **after**
- **around**

The *after* advice has three variations:

- **after**
- **after returning**
- **after throwing**

While before advice is relatively unproblematic, there can be three interpretations of after advice: After the execution of a join point completes normally, after it throws an exception, or after it performs either. AspectJ allows after advice for any of these situations, see Listing 2.2.

Listing 2.2: After advice

```
1   aspect A {
2       pointcut publicCall(): call(public Object *(..));
3       after() returning (Object o): publicCall() {
4          System.out.println("Returned normally with " + o);
5       }
6       after() throwing (Exception e): publicCall() {
7          System.out.println("Threw an exception: " + e);
8       }
9       after(): publicCall(){
10         System.out.println("Returned or threw an Exception");
11      }
12  }
```

We now proceed to show the application and limitations in the rendering scenario described in Appendix C. Let us assume that the application needs to build PVS from the SceneGraph before rendering the visible objects. We showed that the basic structure for SceneGraph and PVSs is the collection. But we would be able to specify which collection implementation we would like to use in the software we are designing, thus we need to advice that our PVS is built using an internal container with a specific implementation.

Before defining the internal container we are able to describe what happens when we add an element to the PVS. We can use an abstract aspect to introduce the fact that we want to weave the PVS class inserting a field that is the internal collection implementation (lets say we are using a sorted list actually) and then we append to the constructor the code to initialise the collection. Introductions in AspectJ are implemented in a way that allows the definition of fields related to the aspect or to the target, thus means that we can add fields and methods (including constructors) to the target object. The following example defines the collection type but because we need to specify the body of the Add method we need to give the code inside the aspect definition.

Listing 2.3: Sorted List Collection

```
1 aspect SortedListCollection {
2    private interface ICollection {}
3    declare parents: (PVS || SceneGraph) implements ICollection;
4
5    private SortedList ISortedListCollection.items;
6    public void ISortedListCollection.Add(element obj)
7       { items.Add(obj); }
8  }
```

Because we are performing an introduction this aspect needs to be woven before the compiler execution. There is no way to modify a type structure at run-time and thus is clear that we cannot use run time values to drive the weaving process. If we introduced an abstract method in the previous aspect we could then extend the aspect with one which carries the final implementation of the Add method with the right comparison and sorting strategy.

Another issue arises if we define what to be done with the Add method of the sorted list when it is contained as field of PVS, in this case we need to be sure of the order used to weave aspects in the target types. For example we would like to use the around advice for the pointcut which will exists after the `SortedListCollection` aspect has been woven on the PVS type.

What is still missing in the approach is that we always assume that the original type system is made of almost empty classes and that we have a basic set of advices defined to implement the base implementation of the code. In such a scenario the final user will have to deal with the source code of the aspect library to remove what is no longer needed before starting the introduction of both new aspect and specialised versions for already existing classes.

Another way to achieve our goal could be to combine the AOP technique with a skeleton design pattern (bEGHJV94) where the purpose of a lot of new methods introduced is to behave like some kind of barrier or code block annotation. In the example of the `ComputePVS` we expect to be presented with code like the following:

Listing 2.4: ComputePVS

```
1  Pair<PVS, PVS> ComputePVS(camera c) {
2                PVS transparent = new PVS(), solid = new PVS();
3                foreach (mesh in CurrentScene)
4      if (IsVisible(mesh,c))
5  if (mesh.Alpha)
6                 FillTheTransparentPVS(transparent, mesh);
7  else
8                 FillTheSolidPVS(solid, mesh);
9               return new Pair<PVS, PVS>(solid, transparent);
10 }
```

We can target the execution of `FillTheTransparentPVS` and `FillTheSolidPVS` methods with an aspect describing the technique we wish to use. We cannot reach the full goal we discussed in the introduction, even if the `FillTheTransparentPVS` is an empty method we are performing the if test for every object in the `SceneGraph`. A more accurate version of the skeleton would lead us to a code like the following:

Listing 2.5: ComputePVS variant

```
1 Pair<PVS, PVS> ComputePVS(camera c) {
2              PVS transparent = new PVS(), solid = new PVS();
3              foreach (mesh in CurrentScene)
4             if (IsVisible(mesh,c))
5             FillThePVSs(solid, transparent, mesh);
6     return new Pair<PVS, PVS>(solid, transparent);
7 }
```

We can describe the `FillThePVSs` methods with an aspect which introduce the right skeleton and then other aspects to deal with the execution of `FillTheTransparentPVS` and `FillTheSolidPVS` within the code of the `FillThePVSs` methods. Taking this path will lead to very complex and potentially unreadable code, class design will be misleading as we are now introducing methods to describe the algorithm of other methods or a sub section of another algorithm. This can also transform crosscut aspects in no more cross cutting characteristic of a program and thus deceive the AOP definitions and goals.

2.3.2 Aspect .NET

In Aspect.NET the reflection support provided by .NET is used extensively: custom attributes (annotations that are available at run-time) are used to implement an AOP system targeting the .NET platform. A language called *Aspect.NET ML* is used to describe aspects, and it is subsequently transformed into a C# class by a preprocessor, then the C# code is compiled and used for the weaving process. The class generated by the preprocessor extends the Aspect class. Let us consider the following example:

```
1 \%instead \%call *BankAccount.withdraw(float) && args(..)
2 \%action
3 public static float WithdrawWrapper(float amount)
4  {
5       BankAccount acc = (BankAccount)TargetObject;
6       if (isLicenceValid(TargetMemberInfo.Name))
7       {
8             return acc.withdraw(amount);
9       }
10      Console.WriteLine("Withdraw operation is not allowed");
11      return acc.Balance;
12 }
```

In this example we are defining a join point by pattern match over the intermediate language. Every call to the withdraw method in the bytecode is replaced by the static method `WithdrawWrapper`. The join point model of Aspect.NET allows matching only on the semantic elements available in the bytecode.

Modules are translated as static public methods of the Aspect while the rules are implemented using a custom attribute AspectAction so that they can be manipulated by the reflection and metadata facilities of the .NET run-time. The process of aspect weaving consists of two phases:

- scanning (finding join points within the target application)
- weaving the calls of the aspect actions into the join points found.

The weaving step in Aspect.NET is performed at .NET assembly (MSIL code + metadata) level. The resulting assembly can be further processed by standard .NET tool such as ildasm, or debugger.

Since Aspect.NET weaving is performed on the bytecode it is possible to constrain a join point pattern to be restricted within boundaries like a method implementation, in this way Aspect.NET.ML can control the scan phase. Due to the interactive nature of the process the approach followed by Aspect.NET is is integrated in **Microsoft** Visual Studio and the developer interacts with the GUI to control the aspect weaving. The current implementation works through static methods of the Aspect derived type, there is some restriction when using the *%instead* rule should be targeting a static method of the target object. Another point is the fact that the aspect can only access public fields of the targets, because the weaving is performed after compilation even access to internal members will be prevented.

It is important to notice that in the bank account example that the Aspect project needs to know about the `BanckAccount` class to perform the cast and proceed with the invocation of the withdraw method and access the Balance field. We can solve the invocation using the reflection object `TargetMemberInfo` but how can we be sure that a class matching the pointcut has a Balance field. Using an introduction pointcut will not solve this issue because it will introduce a static field shared by all the

woven classes. We can solve this using reflection again but in that case we will complicate the Aspect modelling too much for the programmer.

The approach of weaving bytecode can be performed at load time, this would allow us to incrementally generate new specialised types and use them at run-time. The current implementation of the system, however, does not allow such a form of weaving, it is not a conceptual limit but a technological one. The advice model provided by Aspect.NET allows us either to replace method calls or insert before (or after) a method call the invocation of the woven aspect.

2.3.3 Spring.NET

Spring is a framework for Java (Spra) and .NET (sprb) enterprise application development. The framework provides the facilities needed for AOP oriented development. There is no custom language for aspect and point-pointcutcuts definition, the framework uses interfaces and classes to implement AOP concepts. The weaving phase is performed at run time. Using a technique similar to object activation with the remoting framework proxies are emitted to intercept calls and weave advices in the appropriate pointcut. The interaction between the proxy and the target object is modelled within the containment relationship they are (proxy can be thought implementing a classic Decorator pattern). Spring.NET's pointcut model enables pointcut reuse independent of advice types. It is possible to target different advice using the same pointcut.

The `Spring.Aop.IPointcut` interface is the central interface, used to target advices to particular types and methods. Basic building blocks are:

```
1  public interface IPointcut
2  {
3   ITypeFilter TypeFilter { get; }
4   IMethodMatcher MethodMatcher { get; }
5  }
6
7  public interface ITypeFilter
8  {
9   bool Matches(Type type);
10 }
```

```
11
12 public interface IMethodMatcher
13 {
14  bool Isrun-time { get; }
15  bool Matches(MethodInfo method, Type targetType);
16  bool Matches(MethodInfo method, Type targetType, object[] args);
17 }
```

pointcuts can be specified by matching an attribute type that is associated with a method. Advice associated with this pointcut can then read the metadata associated with the attribute to configure itself. The class `AttributeMatchMethodPointcut` provides this functionality. The following sample will match all methods that have the attribute `CacheAttribute`.

```
1 <object
2  id="cachePointcut"
3  type="Spring.Aop.Support.AttributeMatchMethodPointcut,
4  Spring.Aop">
5  <property
6   name="Attribute"
7   value="Spring.Attributes.CacheAttribute, Spring.Core"/>
8 </object>
```

This can be used with a `DefaultPointcutAdvisor` as shown below

```
1 <object
2  id="cacheAspect"
3  type="Spring.Aop.Support.DefaultPointcutAdvisor,
4  Spring.Aop">
5  <property name="Pointcut">
6   <object
7     type="Spring.Aop.Support.AttributeMatchMethodPointcut,
8     Spring.Aop">
9    <property
10    name="Attribute"
11    value="Spring.Attributes.CacheAttribute, Spring.Core"/>
12  </object>
13  </property>
14  <property name="Advice" ref="aspNetCacheAdvice"/>
15 </object>
```

where `aspNetCacheAdvice` is an implementation of an `IMethodInterceptor` that caches method return values. As a convenience the

class `AttributeMatchMethodPointcutAdvisor` is provided to define an attribute based Advisor as a somewhat shorter alternative to using the generic `DefaultPointcutAdvisor`. An example is shown below.

```
<object id="AspNetCacheAdvice"
 type="Spring.Aop.Support.AttributeMatchMethodPointcutAdvisor,
 Spring.Aop">
 <property name="advice">
  <object type="Aspect.AspNetCacheAdvice, Aspect"/>
 </property>
 <property name="attribute"
   value="Framework.AspNetCacheAttribute, Framework" />
</object>
```

Spring.NET advices can be shared across all advised objects, or can be unique to each advised object. This corresponds to *"per class"* or *"per instance"* advice. Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments. Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object. It is possible to use a mix of shared and per-instance advice in the same AOP proxy. The most fundamental advice type in Spring.NET is interception around advice. Spring.NET is compliant with the AOP Alliance interface for around advice using method interception. Around advice is implemented using the following interface:

```
public interface IMethodInterceptor : IInterceptor{
object Invoke(IMethodInvocation invocation);
}
```

The `IMethodInvocation` argument to the `Invoke` method exposes the method being invoked, the target join point, the AOP proxy, and the arguments to the method. The `Invoke` method should return the invocation's result: the return value of the join point. A simple `IMethodInterceptor` implementation looks as follows:

```
public class DebugInterceptor : IMethodInterceptor {
 public object Invoke(IMethodInvocation invocation) {
  Console.WriteLine("Before: invocation=[{0}]", invocation);
```

```
4   object rval = invocation.Proceed();
5   Console.WriteLine("Invocation returned");
6   return rval;
7  }
8 }
```

Note the call to the `IMethodInvocation`'s `Proceed` method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, an `IMethodInterceptor`, like any around advice, can return a different value or throw an exception rather than invoke the `Proceed` method.

Thanks to the per-instance advice, introductions, and the `Proceed` method the Spring.NET framework would partially solve the issues in the `BankAccount` case shown in the Section 2.3.2. To solve it completely we need to know that the target of the interceptor is the same of the introduction for the Balance field (or more probably a property), we need some advanced form of the C++ forward declaration. Supported advices are:

- **Before advice:** Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- **After returning advice:** Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
- **After throwing advice:** Advice to be executed if a method exits by throwing an exception.
- **After (finally) advice:** Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
- **Around advice:** Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behaviour before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Spring.NET allows you to add new methods and properties to an advised class. This would typically be done when the functionality you wish to add is a crosscutting concern and want to introduce this functionality as a change to the static structure of the class hierarchy. For example, you may want to cast objects to the introduction interface in your code. Introductions are also a means to emulate multiple inheritance. Introduction advice is defined by using a normal interface declaration that implements the tag interface `IAdvice`. Introduction advice is not associated with a pointcut, since it applies at the class and not the method level. As such, introductions use their own subclass of the interface `IAdvisor`, namely `IIntroductionAdvisor`, to specify the types that the introduction can be applied to.

```
1 public interface IIntroductionAdvisor : IAdvisor
2 {
3 ITypeFilter TypeFilter { get; }
4 Type[] Interfaces { get; }
5 void ValidateInterfaces();
6 }
```

The `TypeFilter` property returns the filter that determines which target classes this introduction should apply to. The `Interfaces` property returns the interfaces introduced by this advisor. The `ValidateInterfaces()` method is used internally to see if the introduced interfaces can be implemented by the introduction advice.

Spring.NET provides a default implementation of this interface (the `DefaultIntroductionAdvisor` class) that should be sufficient for the majority of situations when you need to use introductions. The most simple implementation of an introduction advisor is a subclass that simply passes a new instance to the base constructor. Passing a new instance is important since we want a new instance of the mixin classed used for each advised object.

```
1 public class AuditableAdvisor : DefaultIntroductionAdvisor
2 {
3 public AuditableAdvisor() : base(new AuditableMixin()) { }
4 }
```

Other constructors let you explicitly specify the interfaces of the class

that will be introduced. We can apply this advisor programmatically, using the `IAdvised.AddIntroduction()`, method, or within XML configuration files using the `IntroductionNames` property on the type `ProxyFactoryObject`, which will be discussed later.

Unlike the AOP implementation in the Spring Framework for Java, introduction advice in Spring.NET is not implemented as a specialised type of interception advice. The advantage of this approach is that introductions are not kept in the interceptor chain, which allows some significant performance optimisations. When a method is called that has no interceptors, a direct call is used instead of reflection regardless of whether target method is on the target object itself or one of the introductions. This means that introduced methods perform the same as target object methods, which could be useful for adding introductions to fine grained objects.

2.3.4 Blueprint

The Blueprint (CP07b; CP07a; Pin07) aspect-oriented language permits the selection of the join points of interest by describing their supposed location in the application through a UML activity diagram representing patterns on the application behaviour, called *join point blueprint*. These join point blueprints are not subsets of the application design information. They do not describe the application behaviour, rather they describe the desired properties and behaviours we are looking for in the application. The Blueprint framework foresees a matching and unification phase that permits to perform queries such as ”print the value of a variable used in a loop test condition and modified in the loop body”. This kind of query is expressed describing the context we would like to get and the position where we would like to raise effects.

To carry out this kind of query the description must be compared with the source code of the base-program during the weaving process. There is no need to use position qualifiers such as *before* and *after* advice to indicate where to insert the concern inside the base code. Because we describe the context we can either locate the join points exactly where we

want to insert the new code or to highlight the portion of behaviour we want to replace.

The Blueprint pointcut can address even a single statement, which is already a significant change in the joint point model (JPM) if compared to most of the AOP implementations. More important is that the blueprint name comes from the ability to express a join point as a template to be matched in the application code. A blueprint (which is the name of a blueprint pointcut) is modelled as UML Activity Diagram, this can then model a joint point as a behaviour we want to find inside an application rather than a mere method or constructor call.

The Blueprint weaving process consists of the following phases:

- **Pre-weaving phase:** the abstraction level of the join point blueprint and of the Java bytecode is equalised
- **Matching phase:** the matching is performed by traversing the model/graph of the blueprint and the model/graph of the program in parallel
- **Advice weaving phase:** the advice code is inserted at the captured join points

Pre-Weaving Phase. The base program and the join point blueprints are at different levels of abstraction. To fill this gap and allow the weaving, it is necessary to build a common representation for the base program and the join point blueprints. The abstract syntax tree (AST) perfectly fits the problem; both source code (through its control flow graph) and join point blueprints can be represented by AST-like descriptions.

Matching Phase. After obtaining the same level of abstraction for source code and blueprints, the next step is to find all matches among each Blueprint_Graphs, generated in the previous phase and the application AST.

Advice Weaving Phase. This is the last step of the weaving phase. During this step the advice code is inserted into the application. This final step starts only when the previous step obtains a sure matching for

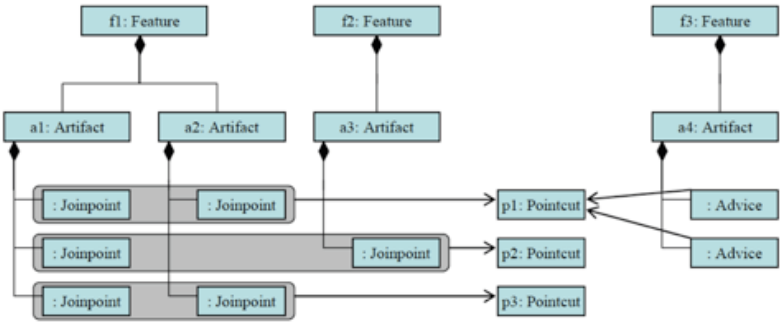
the considered blueprint. To develop this step, the framework uses all information about join points and unifications stored during the matching phase and the advice source code.

2.3.5 AOX approach

In the AOX programming environment, a software system is modelled as a collection of features. A feature is an entity that encapsulates a specific concern of the system. It is implemented by one or more elements called artefacts. The artefacts are organised in a tree structure via an element called a group. When features are woven, a group and an artefact are mapped to a file and a directory in a file system, respectively.

A feature can modularise concerns that crosscut the file boundaries. This is done by the well-known advice-pointcut join point mechanism used in conventional AOP languages. In the model presented in (PH09), an artefact consists of a text, join points and pieces of advice. Here, a join point represents a specific location (offset) in the text and a piece of advice represents a text fragment that can be copied to the join points. A pointcut is defined as a named group of join points. These three elements are associated as shown in Figure 2.

Figure 2: Object diagram shows relationships among join points, pointcuts and pieces of advice



In Figure 2, six join points scattered in three artefacts and two features

are captured by three pointcuts. Among them, one pointcut is advised by two pieces of advice defined in the third feature. Features interact with each other only via crosscutting interfaces, each of which consists of related pointcuts. This idea is inspired by the idea of XPI (GSS+06). A feature can implement a crosscutting interface by declaring join points and associating them with the pointcuts in the interface. Similarly, the interface can be used by creating pieces of advice that advise some pointcuts in the interface.

The features, artefacts and interfaces are all represented in XML files stored in a hierarchy of directories in a file system that we call a repository. Specifically, inside an XML file for an artefact, a join point is represented by a `<joinpoint>` tag. The location of the join point is implicitly determined to be the point where the tag appears in the text. Similarly, `<advice></advice>` tag denotes a piece of advice. Any text other than the XML tags is considered source code and is not interpreted by AOX. Their approach has the following design goals:

- **Language independence:** An operating system commonly consists of various types of artefacts written in different programming languages. Core functionality are usually written in C or C++, while machine-level functionality, such as context switching, initialisation and interrupt handling are written in assembly languages. In addition, makefiles are used to define software build processes, and linker scripts are employed to specify the memory layout of the target hardware. In order to modularise concerns that are scattered on such artefacts, we need to make the aspect model independent of languages used.

- **Fine-grained join points:** Due to performance constraints, operating systems often require highly optimised code. As a result, functions and data structures are deeply intertwined beyond module boundaries. For example, a function that creates a new process usually contains code fragments for seemingly unrelated features, such as context management, synchronisation, signal, file and memory management. An interrupt handler array may contain elements for

different devices. In order to enable such optimisation, our aspect model needs to support fine-grained join points.

- **Advice ordering and replacement:** Programmers have to be able to decide the advising order of pieces of advice if a single pointcut is advised by multiple pieces of advice (NBA05). This is particularly important since the execution order of program statements, the placement order of fields in a structure and the order of elements in an array have important meanings in operating systems design. Along with the advice ordering, developers have to be able to replace existing pieces of advice with new ones. This is needed when developers want to design default pieces of advice that can be overridden by other pieces of advice. This effectively reduces the size of an operating system since replaced pieces of advice are removed.

Given these design requirements, their approach proposes the following solutions.

- **XML annotations:** The AOP language, an artefact is annotated by predefined XML tags. As the text that surrounds the tags is not parsed or interpreted, our model can be applied to artefacts written in any programming language. Also, this model achieves fine grained join points since the tag for a join point can be declared anywhere in an artefact without restriction. We are well aware that our model may be unsafe since it does not prevent programmers from declaring join points in inappropriate places or writing pieces of advice with illegal instructions. To overcome this drawback, the programming programming environment automatically performs weaving and compilation as a background task and immediately notifies programmers of errors caused by such unsafe join points and pieces of advice.

- **Timestamp-based advice arrangement:** In order to support the advice ordering and replacement, they devise the time stamp based

advice arrangement mechanism, in which ordering and replacement relationships among pieces of advice are represented in a two-level list structure. We store this structure in the most recently modified piece of advice to ensure that the ordering and replacement information is always up-to-date. To determine the most recently modified one, we associate each piece of advice with a time stamp value that is updated when its ordering or replacement relationship has been changed.

Using this technique they implemented an embedded operative system , the approach and toolset they developed requires access to source code representation of every artefact used in the composition. The use of XML to implement the AOP language and the choice to be cross language while acting on source code level makes the system non homogeneous and not so easy to integrate in off the shelf pipelines. Similar approach is used in **Microsoft** Windows Embedded (mse) where the features weaving process was driven by a macro based technique.

2.3.6 AOP at Virtual Machine Level

The article (SHH09) shows an approach for AOP based on extensions to VM. The machine model used to implement several programming language approaches to modularising crosscutting concerns has been introduced in (aop07). Core features of the model pertain to the representation of application entities, and that of join points. The latter are regarded as *loci of late binding*, and hence of virtual functionality dispatch, where dispatch is organised along multiple dimensions. Each dimension is one possible way to choose a particular binding of a piece of functionality to a join point, e. g., the current object, the target of a method call, the invoked method, the current thread, etc.

Objects are, using a prototype-based object-oriented environment, represented as "*seas of fragments*" (Piu05) : each object is visible to others only in the form of a proxy. Messages sent to an object are received by its proxy and delegated to the actual object. Classes are represented likewise: each class is a pair of a proxy and an object representing the actual

class. Each object references its class by delegating to the class's proxy.

The granularity of the supported join point model is that of message receptions. Member field access is also mapped to messages. A join point's nature as a locus of late binding is realised by inserting additional proxy objects in between the proxy and the actual object, or in between the class object's proxy and the actual class-representing object. That way, a message passed on along the delegation chain can be interpreted differently by various proxies understanding it, establishing late binding of said message to functionality.

Weaving—both static and dynamic—is realised by allowing for the insertion and removal of proxy objects into and from delegation chains. The model is able to represent control flow dependent advice and dynamic introductions in very natural and simple ways (aop07). Along with the VM and JIT, a set of programming languages has been modelled to be supported, we are not describing them here but more details can be seen in (SHH09).

2.4 Conclusion on Metaprogramming

While the AOP metaphor is very interesting to us we found that in general it is prone to be implemented in a way that prevents general multi staging. In general a *weaver* takes aspect definitions and programs as input and produces a new program as output, the new output can be used again in the weaving process as input to achieve multi staging but it is quite to include aspects as output as well. Most of the AOP implementations are non homogeneous and this complicates the idea of bringing aspect generation to the same level as the program modification through the weaver. The biggest limitation we found is that even if the AOP implementation allows run-time weaving the set of aspects must be available, there is no way to "extract" aspects from existing code and use them in the weaving process, this makes most of the AOP techniques unfit in multistage scenarios.

Most of the frameworks we have introduced in this chapter uses a different language to express either aspects or joint point (as 2.3.6, 2.3.5,

2.3.2, and 2.3.1), this choice prevents us from attaining an unbound multistage approach even where multistage can be achieved. The most constraining characteristic with respect to the flexibility of the approach is that almost all the models to express join points are ”punctual” (from there the *pointcut* term for the match) and limited to method calls. Such a limitation in the model will force programmers to work in a stressed skeleton design pattern in order to facilitate the application of advice in almost any point of their code. The majority of methods will therefore be short and made mainly of calls to other small methods, this makes the code harder to maintain and the compile time optimisation will extremely limited. Another possible effect of an exaggerated skeleton pattern usage is related to parameter passing when they are ”values” and thus copies of them will be involved.

Implementations such as 2.3.3, even if homogeneous and with run time weaving, can bring some serious performance problem as they typically rely on proxy objects. In .NET proxy objects (and the `dynamic` type) perform methods invocation through reflection which is the most expensive call which can be performed at run-time.

Run-time approaches are far more interesting than those requiring the availability of source code. In VM scenarios working at bytecode level means that we have been able to load the elements before transforming them thus leverage the verification step of VM so that a lot of possible bugs due to errors in the new generated code can be detected and avoided.

The limitation of AOP join point model being ”punctual” and able to intercept calls to methods is defiled by the Blue Print approach; it is different as it allows selection of trees of calls, but still it is limited to static manipulations.

2.5 The CodeBricks Approach

CodeBricks (ACK03) is a framework for metaprogramming that provides a different perspective from AOP while maintaining some of the basic concepts.

The framework relies on the fact that programs compiled for an intermediate language, such as the JVM bytecode or the .NET Common Intermediate Language (CIL), retain enough information about high-level types that a correspondence between high-level and low-level manipulations can be retained. Consequently type checking and verification can be performed, guaranteeing that the code generated by the framework is well typed (Cis03). When an intermediate language common to several languages is used, as in our current implementation based on CIL, code fragments even from different languages can be combined.

Differently from other approaches to metaprogramming which operate at the source level and require a language processor to be available at run time to run the resulting program, our approach operates directly on pre-compiled executable versions of programs. A similar technique of combining pre-compiled binary fragments is used in Tempo templates (CE00), which are filled in at run time with constant values.

Code manipulation primitives are provided through a library rather than as a language extension, allowing cross language code generation as well as a more robust and maintainable approach than language extensions. Programs can be specialised more than once, for instance as more information becomes available to the program. Since no language processor is involved, specialisation can happen at different stages through the lifetime of a program, even beyond the program build: for instance at installation time, at load time, or on the client side in a multi-tier application. If we can realistically assume that each stage is capable of running code for the same VM, the framework can be used as support for real multistage programming.

The framework has been implemented as a class library, called **CodeBricks**, on the **Microsoft**.NET platform (.ne). Code generation is fast since no compiler invocation is involved and produces efficient code. The benefits of the approach can be summarised as follows:

- the generated code is expressed and manipulated in a high-level language
- efficient binary code is produced and code generation overhead is

minimal

- no high-level language processor is required to run the generated code
- the solution is not tied to a programming language
- generated code can be involved in further code generation and can itself generate code, providing full multi stage capabilities

CodeBricks extends the type system introducing the `Brick` type which represents a set of type safe and stack safe il instructions. This type can be extracted from method objects and can be used to create new methods. The metaphor of code manipulation by means of brick assembling is extremely powerful as it can be combined with the reflection and code generation capabilities of the .NET platform to implement other metaprogramming techniques (such partial evaluation, Aspect Oriented Programming, Feature Oriented Programming, etc.) keeping homogeneity of languages and adding features like unbound multi staging and run time execution support. We found the CodeBricks approach the key support to realise our framework presented in Chapter 3

Chapter 3

Run-time Code Manipulation

In our work we will study how programs can change their own structure, possibly at run-time, to evolve their behaviour for performance or adaptation reasons. Both aspects are crucial to the gaming application domain where these techniques may lead to significant improvements to the state of the art. In this chapter we focus on the meta-programming tools we will use in the rest of our work. Part of these have been developed, both formally and technically, in the context of this work and are to be considered as one of the results. In particular the definition of a query model for IL programs based on the same ideas of the *CodeBricks* work allows to have a complete algebra for manipulating programs expressed in intermediate form, a tool that can be implemented efficiently and it is more expressive than AOP tools (in particular because it is possible to match code regions rather than just join points).

According to the taxonomy presented in (She01) the transformation model we propose in this thesis is homogeneous, indeed the main design trait of our proposal is to have the *object-program* and the *meta-program* sharing the same language. A design requirement behind our work has been to avoid introducing a new programming language: although useful for theoretical research, new programming languages, or custom vari-

ants of existing ones, are rarely adopted in the real world and it is difficult to have a measure of results in application domains such as gaming. We already made a bet that virtual-machines based programming environment will arise as the dominant paradigm in a domain actually dominated by C++. Several indicators (for instance **Microsoft** XNA[1] (Gou08; KH09) for programming XBox, PC and Zune platforms) let us believe that this will be the natural consequence of the current trends, thus more abstract systems or languages would have been harder to validate. We followed a number of design principles in defining and realising the code manipulation support:

- homogeneous metaprogramming approach (transformation of the program in its intermediate form)
- no assumption about programming languages other than the intermediate language
- type sound and stack safe transformation only
- code generation only through composition of already existing code fragments
- program transformations can be performed at run-time
- transformations must be available at program level
- time and space efficiency
- possibility to express multistage computations
- no instrumentation is needed to use the transformation model

We borrowed several concepts from Aspect Oriented Programming, though the actual systems are less expressive than ours, in particular because AOP weaving is often performed with the program in its source form (there are also bytecode based implementations), and however the

[1]Microsoft XNA Game Studio available at http://msdn.microsoft.com/en-us/xna/default.aspx

program manipulation operations are point wise. Nonetheless, the AOP weaving process has been a major source of inspiration for our rewrite system. We felt the need for a new rewriting system because even systems such as (sprb) feature homogeneous implementation of AOP, they relay heavily reflection comes to take the execution cost at its maximum. As discussed in (ACK03) reflection is the most expensive call methodology inside .NET framework and thus largely avoided in any time sensitive application, so we decided that the execution cost of our approach should avoid to bring any heavy weight to target applications. Another idea we took from AOP is the *pointcut* and *jointpoint* but we want those to be used in both extraction and injection, in our idea only valid code snippet can be used for injection and so need to be able to select portions of code (adequately expanded), those needs lead us to think of approaches like (CP07a; CP07b).

3.1 Code Fragment Selection

Selection of byte code snippet for later manipulation is a task that must be executed carefully in order to produce a transformation that will lead to a valid result. Intercepting and replacing calls is a very basic operation and to be properly performed it is important to expand the selection to the minimum portion of bytecode that will lead to a safe stack status. Starting with a method call as target will be a good scenario to understand and to show that any arbitrary portion of byte code can be the target of code manipulation as long as we can compute the signature of the snippet.

3.1.1 Calls as Placeholder

Using methods calls as a placeholder is the starting point for our technique. Choosing method calls instead of mere instructions has been driven by the fact that methods in a framework models a DSL and thus are far more meaningful than an add operation. In (ACC04) we used that technique to generate code for Lego™Mindstorm VM using a .NET as-

sembly as a source. Intercepting a method call inside the byte code can be performed by search for a set of specific instructions provided by the CLR as:

- `Call` used to invoke static method calls, instance method calls, and virtual method calls with a resolution scope limited to the type used for the invocation
- `Calli` used to perform native calls
- `Callvirt` used to perform instance method calls, late-bound method calls

In (ACC04) we used types to model class of Lego™hardware and methods were used to express hardware specific functions like turning motors on or off, reading from specific type of sensors, configuring sensors and concurrent tasks. All the basic operations of the Lego™VM can be mapped 1:1 with operators of any VM; operations like add, sub, div and similar were of no particular interest.

Intercepting a call does not mean to be able to perform any valid operation with that, we must be able to capture all the context needed by the call such as parameters and return values[2].

The first important difference between static and instance method call is the first argument to be loaded on the stack, in instance calls it is required to pass the instance's reference to be used for the invocation and then all the arguments from 1 to N. When the call is executed on the *this* reference the IL block generated will always be in a pattern like the one shown in 3.1.

Listing 3.1: IL Pattern for an instance method call

```
1       ...
2       ldarg.o
3       ...
4       ...
5       callvirt ...
6       // something could have been pushed on the stack
7       ...
```

[2]In the CLR a method execution will be responsible to push at most one element on the stack.

In [a]C# (CCC05) static method calls with a void (void) signature have been used to annotate code fragment. Using method calls as placeholders is really safe since the C# compiler will not be removing static calls to methods defined in another assembly and also the calls will not be reordered since they can cause side effects unknown to the compiler. On such basis we have been able to design and implement [a]C# using calls to mimic the usage of custom attributes to mark portion of code inside a method. In [a]C# a code snippet like:

```
1  public void SomeMethod()
2  {
3        [AnnotationA]
4        {
5              ...
6        }
7        [AnnotationB]
8        {
9              ...
10       }
11 }
```

would have been realised by the following C# block:

```
1  [AnnotationA(1), AnnotationB(2)]
2  public void SomeMethod()
3  {
4        Annotation.Begin(1);
5              ...
6        Annotation.End(1);
7        Annotation.Begin(2);
8              ...
9        Annotation.End(2);
10 }
```

The method calls used in [a]C# are very easy to be detected. The problem we are going to face now is about what kind of transformation we want to operate over the selection we are marking in the IL code since, unless we are constrained to work only with void (void) signatures, we have to take care of the status of the execution stack on the caller side.

3.1.2 Stack Abstract Interpretation

As mentioned in 3.1.1 any operation we perform over the IL body of a method must preserve the execution stack health. To be able to hold the stack health we have to perform an abstract interpretation of the stack. The .NET assembly does not contain such information even if the metadata portion of the assembly is very rich. Anyway the assembly format and its metadata are what we need to interpret the execution stack of any method body defined in the assembly.

Abstract Interpretation (Cou97; CC77; CC79) is a theory for formally constructing conservative approximations of the semantics of programming languages. In program analysis an abstract domain is defined, and each program instruction affects the abstract state. The abstract interpreter is capable of performing the execution of the program in terms of the abstract state which is computable statically allowing for before-runtime property checking.

Abstract interpretation has been successfully employed in the context of virtual machines for type-checking a program in its intermediate form. The virtual machine features a stack-based machine in which operations manipulate the stack of operands by pushing and popping values. The Java bytecode and a subset of CIL is verifiable (ECMb) in the sense that the intermediate language program can be interpreted statically to simulate the behaviour of the operand's stack ensuring that operations and type of values are compatible with respect to the type system. For each intermediate language instruction a stack behaviour in terms of the types of the values used as arguments and the type of the value returned. The behaviour of loops in the abstract domain of types does not depend on the iterations, thus the program can be analysed statically and the verification becomes decidable.

It is worth noting that the ultimate goal of the analysis is security and there are additional properties checked during the verification phase (when the program gets loaded by the virtual machine), for instance an additional constraint imposed by CLI standard is that the height (in addition to the types of its content) of operands' stack should be the same

whatever path is followed by the control flow. This imposes, for instance, that the behaviour of code of a conditional branch should be the same independently by the value of the condition; or the body of a `while` loop should not change the structure of the operands' stack since it may be skipped altogether.

Abstract stack interpretation has been used to check many program properties (RRW05) and in the context of CLI programs there are several examples of interpreters such as the shared source implementation provided by Microsoft (rot), the Mono implementation (cec), the CLIFileRW developed at University of Pisa (Cis), the RAIL project (CMS05). The first two libraries are designed as part of the virtual machine implementation, the others are research projects developed to study binary program analysis.

An example of the abstract interpretation process is given by the following `Factorial` method implementation:

```
1  .method public hidebysig static int32 Factorial(int32 n)
2   cil managed
3  {
4    // Code size       16 (0x10)
5    .maxstack 8
6    IL_0000: ldarg.0
7    IL_0001: brtrue.s IL_0005
8    IL_0003: ldc.i4.1
9    IL_0004: ret
10   IL_0005: ldarg.0
11   IL_0006: ldarg.0
12   IL_0007: ldc.i4.1
13   IL_0008: sub
14   IL_0009: call   int32 Example::Factorial(int32)
15   IL_000e: mul
16   IL_000f: ret
17 } // end of method Example:: Factorial
```

First of all we note the `.maxstack` directive which states that the maximum height of the operands' stack will be of 8 values (if during the abstract interpretation this constraint is violated the program is considered not valid). The rest of the program is easy to understand: the first argument of the method (the `n` parameter) is checked for zero and in case the constant `1` is returned, otherwise recursive invocation takes place. Instructions `IL_0005`-`IL_000e` are worth of further discussion.

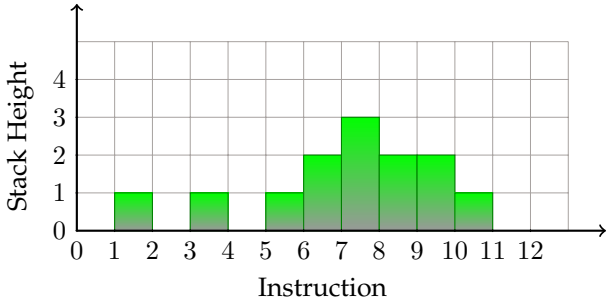
The equivalent C# of the expression `n*Factorial(n-1)` loads arguments in a way that is typical of IL programs: instruction `IL_0005` loads the argument 0th (the `n` argument) on the stack, then the `n - 1` sub-expression gets computed (instructions `IL_0006`-`IL_0008`) and its result left on top of the stack as argument for the subsequent `call` instruction.

The `mul` instruction will find values on the stack loaded by `IL_0005` and `IL_0009` instructions, and pushes the result of the product on top of the stack. If we follows the abstract interpretation of the stack we obtain the following trace:

```
1  Initial stack: []
2  IL_0000: [ (int, IL_0000) ]
3  IL_0001: []
4  IL_0003: [ (int, IL_0003) ]
5  IL_0004: []
6  IL_0005: [ (int, IL_0005) ]
7  IL_0006: [ (int, IL_0005) (int, IL_0006) ]
8  IL_0007: [ (int, IL_0005) (int, IL_0006) (int, IL_0007) ]
9  IL_0008: [ (int, IL_0005) (int, IL_0008) ]
10 IL_0009: [ (int, IL_0005) (int, IL_0009) ]
11 IL_000e: [ (int, IL_000e) ]
12 IL_000f: []
```

We will later use graphs like Figure 3 to show the effect of code modification on the stack.

Figure 3: Stack size evolution over execution



We have reproduced the abstract stack state after the execution of each instruction. The example shows a typical behaviour of stack based computations: intermediate results are accumulated onto the stack in a way that is very regular and suited for automatic pattern recognition of program fragments.

For the purpose of this work we are interesting at finding method invocation and statements. As shown in the previous example a single line of a high level programming language maps down into several intermediate language instructions. Since different programming languages compile down to CIL we must give a reasonable definition of statement that is language independent and that can be found by program analysis.

Definition (statement): A statement is a sequence of CIL instructions such that the height of the abstract stack is zero before the first instruction and zero after the last instruction, which should not be a conditional branch.

In the previous example there is a statement at instructions `IL_0003-IL_0004` (corresponding to a C# statement `return 1`) and another statement at `IL_0005-IL_000f` (corresponding to the statement `return n * Factorial(n-1);`).

Method calls are easier to define though more difficult to match precisely. A method call takes place wherever a `call`, `calli` or `callvirt` instructions occurs (see 3.1.1). The problem is that the actual arguments of the method call are loaded by instructions preceding the call instruction, depending on the method target of the call. At the call site there must be an entry on the stack for each argument, so by means of abstract stack interpretation it is possible to track the instructions that have generated each argument in the same way we did in the example.

We can also recursively perform the analysis and reconstruct the whole tree of a method call, including method calls performed to compute intermediate results. The ability to recognise compositions of method calls is important when implementing explicit support for domain specific languages as discussed in (CE00). In particular we are interested in spotting the utilisation of those primitives that are critical in the performance of a computer game.

Let us consider the following C# (equivalent IL in Listing D.1 and Listing D.2) code snippet:

Listing 3.2: Triangle's normal calculation

```
...
//compute triangle normal
Vector3Df e1 = p1 - p0;
Vector3Df e2 = p2 - p0;
Vector3Df n = Vector3Df.cross(e1,e2);
n.Normalise();
...
```

In this example the method calls can be seen as abstract operations that are recognised and replaced with the following optimised code:

Listing 3.3: Triangle's normal calculation

```
...
//compute triangle normal
float e1x = p1.X - p0.X;
float e1y = p1.Y - p0.Y;
float e1z = p1.Z - p0.Z;

float e2x = p2.X - p0.X;
float e2y = p2.Y - p0.Y;
float e2z = p2.Z - p0.Z;

float nx = e1y*e2z - e1z*e2y;
float ny = e1z*e2x - e1x*e2z;
float nz = e1x*e2y - e1y*e2x;

float m = Math.Max(nx, Math.Max(ny,nz));
nx /= m;
ny /= m;
nz /= m;

m = Math.Sqrt(nx*nx + ny*ny +nz*nz);

nx /= m;
ny /= m;
nz /= m;

Vector3Df n = new Vector3Df(nx,ny,nz);
...
```

We used the abstract stack interpretation support featured by *CLI-FileRW* library (Cis), and contributed to improve it. The library provides a cursor to enumerate CIL instructions of a given method, and if

requested the abstract stack is interpreted as the cursor advances. We contributed to the arguments parsing of a method call so that when such instruction is found information about arguments is already available. The complexity of the algorithm is linear.

3.1.3 Signature of IL fragments

Once we perform a selection over a stack safe block of instruction it is possible to compute the signature of the block. That portion of code can be replaced with an equivalent method call that satisfy the execution stack. The C# code snippet in Listing 3.4 can be replaced with any method call as long as the method executed has a signature `int (void)`, we can preserve the execution stack if we transform Listing 3.4 in the code in Listing 3.5.

The C# examples are easier to read for a reader not familiar with the IL code but, from now on, we will move most of the example of this section to IL since it is easy to understand the code manipulation operations we are going to show. The previous example is then represented by the transformation of Listing 3.6 in Listing 3.7; in the IL snippet is quite easy to see that the execution stack has gone through a safe transformation (anyway there is nothing to guarantee the semantic equivalence). The section we are looking at is starting loading two strings on the stack, then pops them and push back the concatenation, after that the new string is popped to invoke the property accessor which will push an integer on the stack.

Now it quite clear that the instruction block is equivalent to a method call with signature `int (void)` since the code starts pushing values on the stack, then consumes them all and push an `int`, no previous loaded values are used by the snippet.

Listing 3.4: Source Fragment

```
1       ...
2       ...
3       string a = "first string";
4       string b = "second string";
5       int l = (a + b).Length;
6       ...
```

```
7       Console.WriteLine(l);
8       ...
```

Listing 3.5: Fragment Replaced

```
1       ...
2       ...
3       int l = ReturnAnInteger();
4       ...
5       Console.WriteLine(l);
6       ...
```

Listing 3.6: IL Source Fragment

```
1       ...
2       ...
3       IL_0079: ldstr  "first string"
4       IL_007e: stloc.s a
5       IL_0080: ldstr  "second string"
6       IL_0085: stloc.s b
7       IL_0087: ldloc.s a
8       IL_0089: ldloc.s b
9       IL_008b: call   string [mscorlib]
10              System.String::Concat(string, string)
11      IL_0090: callvirt instance int32 [mscorlib]
12              System.String::get_Length()
13      IL_0095: stloc.s l
14      ...
15      IL_00be: ldloc.s l
16      IL_00c0: call   void [mscorlib]
17              System.Console::WriteLine(int32)
18      ...
```

Listing 3.7: IL Fragment Replaced

```
1       ...
2       ...
3       IL_008b: call   int32 [myassmely]
4               Test.Program::ReturnAnInteger()
5       IL_0095: stloc.s l
6       ...
7       IL_00be: ldloc.s l
8       IL_00c0: call   void [mscorlib]
9               System.Console::WriteLine(int32)
10      ...
```

The signature computation is quite relevant since can be used to determine the similitude amongst IL snippet from a stack safety perspective.

Another relevant feature about the signature computation is that we can also compute the minimum set of permission required by the IL portion we are dealing with, in such a way it is possible to perform manipulations that can satisfy the code access security policy though the stack inspection as shown in (BBR+04). To obtain the signature is the key step we have to perform before being able to manipulate the bytecode just to give a quick example we can always remove any void(void) signature block being sure that the removal will be always stack safe (without any guarantee from a semantic point of view, the signature is not enough to foresee side effects of the snippet).

3.1.4 Code Fragments

In this work we are interested in program transformation, possibly at run-time, in order to specialise and evolve program definition over time. We introduce a formal definition of program transformations involved in this process in order to precisely define how we identify and change its fragments even if expressed in intermediate form. We will express program transformations in terms of program fragments that can be searched and manipulated using a set of general purpose transformations.

In our dissertation we will leverage on former work on Code Bricks (ACK03) that introduces a well defined notion of code fragment together with the `Bind` operation allowing for their manipulation. Nevertheless, we should extend the model since we need not only to build code fragments out of code fragments, but also be able to look for particular code fragments in a program.

A *code fragment*, or *brick*, is an element of the set

$$\mathrm{CF} : \mathrm{Sign} \times \mathrm{Env} \times \mathrm{Code}$$

Where *Sign* is the set of signatures, *Env* an environment for values bound to the code, and a body in the form of a sequence of intermediate language instructions.

The body of a code fragment should be consistent with respect to the stack behaviour as indicated by ECMA CLI specification (ECMb),

moreover the behaviour should be compatible with the related signature. We assume that the *bind* operation is defined in the *CF* set as discussed in (Cis03).

Each code fragment can easily know where it belongs, i.e. the location into an executable file, since it is obtained through a query (a mechanism which will generalise the original creation operation based on methods), an operation that will be introduced in the next section. The *FragmentLocation* function allows retrieving the location of a specific code fragment:

$$\text{FragmentLocation} : \text{CF} \rightarrow \text{MethodInfo} \times \text{Loc}$$

where *MethodInfo* is the set of method descriptors featured by the reflection and *Loc* is the set of location within a method body.

3.1.5 Code selection through query

As witnessed by the AOP community an essential element of a program transformation system is the ability to select appropriate locations in the code where some change is desirable. The AOP tradition focuses on the notion of *join point* which is a point in a program where an aspect can be woven.

Several languages have been adopted by various aspect weavers, a great number of them is based on the analysis of the program in its source form (though there are restricted form of join point matching on binary files (asp)).

The original definition of Code Bricks introduced a simple mechanism to lift actual methods into the code fragment values representing them. We are interested in extending the original mechanism provided by the framework into a generalised form that allows us to specify queries against a program. Nevertheless, we are interested to preserve the original approach taken by Code Bricks at providing the feeling that code is manipulated at source level, even though the actual program transformation is performed when the program is in its intermediate form. The query mechanism we are introducing is different from AOP join points since we will be able to capture a region of code and not just

a single point within the program. Another distinctive feature of this approach is that a query in our framework can be expressed at source level language and then compiled into the query that gets executed.

A query is an operation defined on a code fragment:

$$\blacktriangle : \mathrm{CF} \times \mathrm{CF} \rightarrow (\mathrm{CF} \times \mathrm{CF}^n)^m$$

The operation requires a code fragment which where another code fragment should be searched for. Since the query is a code fragment it has a signature which can be considered as the arguments that should be matched within the query fragment. This approach is a form of query by example (Zlo75) a well known approach that is easy to understand for people writing queries.

As we will discuss shortly queries will be expressed in intermediate language (as they are code fragments) but for the sake of exemplification we will use C# to show simple queries.

Queries with parameters allows to express exact match, for instance the following query looks for expressions adding two integer values:

```
1 int QueryAdd(int x, int y) {
2   return x + y;
3 }
```

A code fragment will match the query if has a signature compatible with that of the query and adds two integer values. The two arguments acts as type-safe placeholders and the code fragments representing `x` and `y` are part of the result of the query match.

Consider the following method:

```
1 Matrix MatrixSum(Matrix a, Matrix b) {
2   var res = new Matrix(a.RowCount, a.ColCount);
3   for (int i = 0; i < a.RowCount; i++)
4     for (int j = 0; j < a. ColCount; j++)
5       res[i][j] = a[i][j] + b[i][j];
6 }
```

The result of

$$\blacktriangle(\texttt{MatrixSum}, \texttt{QueryAdd})$$

is the following[3]:

$$(\overline{a[i][j] + b[i][j]}, \overline{a[i][j]}, \overline{b[i][j]})$$

Where the first match is the code fragment matching the query and the second and the third the two values captured by its argument. If an expression can be matched several times the query operation will return all of them, we will introduce derived operations to select the largest match for a particular query.

A parameter less query is used to express exact match:

```
1 void ParameterlessQuery() {
2   Console.WriteLine("********************");
3   Console.WriteLine("* Welcome");
4   Console.WriteLine("********************");
5 }
```

In this case we are looking for a given sequence of instructions without any variant inside. This kind of queries are useful to identify join points, and in particular method calls.

A more interesting type of query are the high order queries, i.e. those queries that allow to match code fragments of variable length. Let us consider the following example:

```
1 delegate void Cmd();
2
3 void MatchLock(object o, Cmd body) {
4   lock (o) {
5     body();
6   }
7 }
```

Here we are looking for all the places in which there is a code fragment with the signature

$$\text{void} \rightarrow \text{void}$$

which is protected by a lock statement. Here the high order argument[4] is used to as a placeholder within the query's body for a sequence of

[3] We will use a line over a code fragment to indicate the code fragment value representing the corresponding code.

[4] C# delegates represent signatures of methods that are invoked without other knowledge other than the invocation behaviour.

instructions with the given behaviour. In this way it is possible not only to match a point in the code, but also its surroundings, imposing extra requirements about the context in which a code fragment is situated.

Now we introduce some useful schema that we will use in the rest of the thesis without having to resort to the base operation definition. We finally discuss how queries can be defined in terms of intermediate language instructions and implemented.

Querying assemblies

The ▲ operation is always defined on code fragment, and methods are code fragments. We extend the operation definition to an assembly a, which is a collection of types containing methods, in the following straightforward way:

$$\blacktriangle(\mathrm{a}, \mathrm{q}) \equiv \bigcup_{\mathrm{m} \in \mathrm{Methods(a)}} \blacktriangle(\mathrm{m}, \mathrm{q})$$

where $\mathrm{Methods}$ is a function returning all the methods defined inside an assembly (which is the content of the *Method* table inside the logical definition of the assembly (ECMa; ECMb; MR03)).

Largest fragment match

The query operation has been designed to return all the possible matches for the given query against a code fragment, this definition allows capturing all possible instances of compatible code fragments. This behaviour, though expressive, sometimes makes complicate to find the largest match which is often the one we are interested into. Let us consider the `Match-Lock` query when executed against the following code fragment that we will refer as `C`:

```
1 lock (connection) {
2   var cursor = connection.Select(dbquery);
3    lock (storage) {
4      foreach (row in cursor)
5        storage.save(row);
6    }
7
```

```
 8    lock (buffer) {
 9      buffer.LastUpdate = DateTime.Now;
10    }
11  }
12  // ...
```

There are three possible matches for the given query since the `lock` statement has been used thrice. Moreover, one match includes the other two, thus the operation

$$\blacktriangle(\texttt{C}, \texttt{MatchLock})$$

will return the three matches

$$[(1..11, (2..10)), (3..6, (4..5)), (8..10, (9..9))]$$

where numbers refer to line numbers of the previous listing. According to the definition of the query operator for each match we find a list of captures identified by the query arguments. Note how easy is to check whether two matches are nested since each code fragment knows the position within the program, so that checking for nesting of code fragments is reduced to checking for integer intervals containment.

Often we are interested at the outermost match so we will use a variant of the query operation which is greedy in the same way the traditional Unix regular expressions are. We will define a variant of the query operation as follows:

$$\overline{\blacktriangle}(\mathrm{CF}, \mathrm{q}) \equiv \text{let } \mathrm{R} = \blacktriangle(\mathrm{CF}, \mathrm{q}) \text{ in}$$

$$\{\mathrm{res} |\ \nexists \mathrm{r} \in \mathrm{R} \text{ s.t. } \mathrm{res} \subset \mathrm{r}\}$$

The result of the query

$$\overline{\blacktriangle}(\texttt{C}, \texttt{MatchLock})$$

will be

$$[(1..11, (2..10))]$$

which is the enclosing `lock` statement.

Operators sequence match

Another situation that is interesting to capture during code analysis is the sequence of operator's application. It is well known (CE00) that libraries often expose non-optimised versions of operators because of their ability to be composed. Matrix addition, for instance, can be optimised by avoiding the allocation of a temporary matrix while adding two matrices; moreover the whole operation can be done with just a nested loop for adding a number of matrices at once.

We introduce a variant for the query operation to match a sequence of matches of the same query, so that it is possible to easily find program regions that can be optimised or rewritten. Let us consider the following query[5]

```
1 Matrix MatrixAdd(Matrix a, Matrix b) {
2   return a + b;
3 }
```

If we consider a sequence of applications of the addition operation

$$a + b + c + d + ...$$

the standard query operation would match all applications of the addition operation. But in order to be able to optimise the addition we must be able to recognise that all the matches intersects pair wise and reconstruct the sequence.

The $\blacktriangle_s$ operation does this for us:

$$\blacktriangle_s(\mathrm{CF}, \mathrm{q}) \equiv \text{let } \mathrm{R} = \overline{\blacktriangle}(\mathrm{CF}, \mathrm{q}) \text{ in}$$

$$\{\mathrm{res} | \forall m, n \in R(m \subset res) \wedge SameBegin(n, res)\}$$

The $SameBegin$ function is assumed to be true if two code fragments share the same beginning in the code. Because of the stack based nature of the intermediate language the sequence of operation application

[5] In this example we explicitly use the `Matrix` type to capture additions between two matrices. An alternative query would have been to define a high order type to indicate an expression returning a `Matrix`.

results in an iterated application of the function starting at the same instruction.

For instance the application of the addition operator would result into a sequence such as:

```
1 ld a
2 ld b
3 add
4 ld c
5 add
6 ld d
7 add
8 ...
```

where the `ld` instruction is assumed to be the appropriate instruction to refer the name `a` after the compilation (i.e. either a field, an argument or a local variable).

The $\blacktriangle_s$ operator allows us to easily find sequence of application of one or more operations.

3.1.6 Definition of the query operator at IL level

In this section we will outline a possible implementation of the query operation to search for code fragments of programs expressed in intermediate language. The matching procedure has been inspired from regular expression theory and in fact can be considered a natural extension. But before delving into details it is important to discuss a little bit about the relation between the code snippets we have shown so far and the their counterparts expressed in intermediate language.

In defining the query operation we followed the same approach used in *CodeBricks* providing an operator functioning at the IL level, but that can be perceived at the level of a programming language. This is because compilers implement a translation function that is consistent and, due to the expressivity of the IL with respect to a traditional assembly language, which maps constructs over the same IL fragments. Moreover compilers, even optimising ones, tend to be consistent during translation, so that a C# `while` loop is always translated in the same way. Moreover method

calls, which are perhaps the most important elements in pattern recognition, are expressed in a well defined and unambiguous way.

Queries, as code fragments, are always compiled into intermediate language and the query operation is defined when programs are in their intermediate form. The query process can be considered as an extension of regular expression matching in the sense that the sequence of IL instructions representing a query are considered as symbols of an alphabet. However, sequence of instructions are different from sequence of symbols since we associate a semantics that corresponds to the program execution. In particular our matching is intended to be type-aware and a match is valid only if the abstract stack interpretation of the query code fragment is compatible with that of the match found.

Suppose that we are interested to match the expression `1 + x` where `x` is an argument of the query, the associated IL will be:

```
1 ldc.i4.1
2 ldarg.0
3 add
```

In our interpretation we are interested in a sequence of IL which pushes the constant `1` on top of the operand stack, then it loads an integer value from somewhere (that will be a local, an argument or a field) and then these two values are added.

The program we are interested to search is the following (we assume `x` is an input argument):

```
1 int y = 0;
2 if (dbisopen)
3   y = 1 + x;
4 else
5   y = -1;
6 // ...
```

The equivalent IL program (as translated by the C# compiler) is the following:

```
1 ldc.i4.0
2 stloc.2 // y = 0
3 ldloc.0
4 ldc.i4.0
5 ceq     // !dbisopen?
```

```
6  brtrue.s ToElse
7    ldc.i4.1
8    ldloc.1
9    add
10   stloc.2
11   br.s EndIf
12 ToElse:
13   ldc.i4.m1
14   stloc.2
15 EndIf:
```

Lines ranging from 7 to 9 included represent the code fragment we are looking for, although we do not really know that variable `x` corresponds to the local variable whose index is `1`. The matching process can proceed as if the code is a sequence of symbols, each for instruction, looking for the instruction `ldc.i4.1` and then check if the sequence is compatible with the one of the query. In this respect this process is analogous to the usual regular expression matching, and this is important because we can adapt algorithms from literature to our approach. We can also borrow performance and complexity results.

If the search process recalls the regular expression matching, we need a broader notion of match for instructions in order to be able to match fragments. In our example the first instruction of the query fully matches with the instruction at line 7 of the input program, but this is not true for the second line. In this case we need to know that the `ldarg.0` instruction in the query should be considered as a placeholder for an instruction which loads an integer value on top of the operands' stack. In practice we are introducing a *binding* between the variables defined within the query (local and arguments) and those used by the code processed by the search algorithm. In our example the first argument of the query will be bound to the local variable `1` of the program, and if the match will be successful the binding will be returned as part of the query results.

Local variables and arguments in queries

What is the interpretation we should give to local variables and arguments used by an IL fragment used as a query? We already said that input arguments of a query should be considered as the matches we are

interested to find within a code snippet having the structure defined by its body. Thus we are authorised to associate a location of the same type, thus if during the match a load instruction is found the allocation class will be associated with the argument (i.e. an argument, a local variable or a field). The only exception is the use of delegates whose role will be discussed in the next section.

Local variables are treated as input arguments during the matching process, the only difference is that they are not returned as part of the result. The following two queries are equivalent, the only difference is that only in the first case a binding found will be returned along with the result of the query:

```
1 int Query1(int x) {
2   return 1 + x
3 }
4
5 int Query2() {
6   int x = 0; // Variable should be initialized otherwise the compiler will complain
7   return 1 + x;
8 }
```

Access to arrays, property accessors and other amenities is not achieved through the direct use of this capture and should be made explicit through appropriate instructions. For instance if the location of a value is an array the type of the input argument should be array of the appropriate type.

Arguments and local variables play another important role too: they consent to express identity of a location. If a variable is set multiple times we must ensure that the binding is preserved in order to avoid invalid matches in which this identity is not preserved. Once a bind is established during the matching process, it is preserved until the current match is refused or accepted.

Delegates as a type-safe star operation

The Kleene star is one of the most important operators of regular expressions, allowing to express an arbitrary sequence of symbols taken from an appropriate subset. In our query language we have a similar opera-

tor which allows to express arbitrary sequences of IL instructions with a well defined behaviour with respect to the operand stack. We use the delegate type as a mean for expressing an arbitrary code fragment within the query. Let us consider the following query:

```
1 delegate void Cmd();
2
3 void Query(Cmd c) {
4   string s1 = "", s2 = "";
5   Console.WriteLine(s1);
6   c();
7   Console.WriteLine(s2);
8 }
```

We are interested in matching all the code sequences delimited by a call to the `Console.WriteLine` method, thus we are looking for a sequence of instructions whose behaviour with respect to the operands stack is the same as that of a delegate call. In this case we are looking for a sequence of instructions leaving the stack height unchanged. Protection blocks (i.e. exception handling) should be consistent between the query and the matched code.

Suppose we are now interested in searching for the query in the following code fragment:

```
1  // ...
2  int attempts = 5;
3
4  Console.WriteLine("Begin reading data...");
5  while (attempts > 0)
6    try {
7      // Read data
8      for (int i = 0; i < 1024; i++)
9        Console.WriteLine("Read byte {0}", Console.ReadKey());
10   } catch(Exception e) {
11     Console.WriteLine("Error while reading...");
12     attempts--;
13     Thread.Sleep(1000);
14   }
15
16 Console.WriteLine("Done");
17 // ...
```

In this case we have a single match between instructions 4 and 16, since the instruction 11 is within a protection block and the stack of

operands does not satisfy the stack consistency constraint imposed before.

Since the delegate within the query is used as a parameter a code fragment corresponding to the match should be returned, as we already discussed. Now, we may be tempted to assume that the signature of the code fragment returned by our match has the same signature of the delegate we used as a placeholder. The code fragment in our example would be:

```
1  while (attempts > 0)
2    try {
3      // Read data
4      for (int i = 0; i < 1024; i++)
5        Console.WriteLine("Read byte {0}", Console.ReadKey());
6    } catch(Exception e) {
7      Console.WriteLine("Error while reading...");
8      attempts--;
9      Thread.Sleep(1000);
10   }
```

This is not a well formed code fragment since it refers to the `attempts` variable which is not available anymore. We could have assumed a semantics similar to lexical closures and assume that the local variable is captured somehow, but our query system is designed to analyse code statically so that there is no way to close a particular activation of a method call. Therefore, the resulting code fragment of the match will have an argument for any variable that is not accessible once the code has been extracted from its context.

In the same code fragment the local variable `i` needs not to be lifted into the code fragment signature since it is local to the whole code fragment matched (i.e. its scope is fully contained inside the code fragment). It is important to notice that in IL scope of local variables is lost since local variables are all lifted as local variables of a method. However, it is possible to obtain an approximation of the scope by looking for read-/write access to the variable (as discussed in (DCD07)).

Lifted arguments in a code fragment are annotated with the original source so it is possible to compare for identity of a location in different match results.

Since we use delegates as a mean to indicate a well defined pattern on the operands stack before and after a sequence of IL instructions, we can also match expressions in the very same way we matched a sequence of statements. Consider, for instance, the following query:

```
1 delegate double Exp();
2
3 void Query(Exp e) {
4   double x = 1.0 + e();
5 }
```

it matches all the double expressions adding left-wise the constant `1.0` to it. Note that in the expression `1.0 + 1.0 + 1.0` we have three possible matches returned by the query operation (consider associativity).

3.1.7 Queries and Regular Expressions

Our query system has been designed after regular expressions, but there are operations that are not directly expressible in a single query. A regular language is built around the Kleene star operation, as well as the alternative operation that allows picking strings from two different sets.

As we already discussed the Kleene star operation is obtained through the use of delegates as a star operation on a stream of IL instructions with the additional constraint that the static analysis of the operands stack should be compatible with the signature of the given delegate.

The alternative is not directly expressible within a query, in principle we could have used code-level annotations to label special if statements to be considered at the meta level rather than something to look for, but this would have made the query difficult to read and we would have drifted away from the *query by example* approach we followed. Besides, the alternative can be obtained by multiple applications of the ▲ operator with the results that are joined together. An efficient implementation may take advantage of this information to perform multiple queries at once, but this is an optimisation of the implementation. Thus the alternative query equivalent to the regular expression $\mathrm{e}_1|\mathrm{e}_2$ is expressed as:

$$\blacktriangle(c, q_1)|\blacktriangle(c, q_2) \equiv \blacktriangle(c, q_1) \cup \blacktriangle(c, q_2)$$

In the rest of this dissertation, when useful, we will use convenient notations that are drawn from regular expressions syntax and can be easily mapped in the query operations.

As it happened for regular expressions after the Perl's implementation, also for our code querying system there are several variants that can be introduced to allow more compact representations of queries, but for the purpose of this thesis the language is expressive enough.

Although we usually express queries at language level (most in C#) it must always been considered that they are a compact representation for a stream-based analysis of code when programs and queries are compiled down to sequences of IL instructions.

3.2 Manipulation of code snippets

We will introduce now the set of operations to perform code transformations, before we explain the operators it is important to clearly state the overall concept we used to design the manipulation model. Our approach follows the philosophy of CodeBricks (ACK03), we can manipulate IL fragments that have been selected inside existing assemblies. The choice we made implies that we can not generate any arbitrary set of instructions as it is possible with `Reflection.Emit`, on the other side the limitation imposed on the model force the system to work only on valid and safe portion of code (thanks to the stack abstract interpretation 3.1.2, the signature computation 3.1.3, and the stack inspection(BBR+04))

3.2.1 Extrude Evaluate Inject

The transformation model we investigated is built on three main operations:

- Extrusion
- Evaluation

- Injection

Those operations are responsible for the code transformation, a joint point model will be provided later to support the Injection. A delete operation is not provided since it can be expressed by the Injection as we will show later.

Extrusion

The Extrusion operator is responsible for the extraction of an IL snippet (we will refer the extruded snippet as *Fragment*) and is defined as

$$\blacktriangleleft : \mathrm{CF} \rightarrow \mathrm{CF}$$

We will refer the method where snippet is located as *Host* while the type owning the method will be addressed as Host Owner. A Fragment is identified by the IL code of the selected snippet, its signature, its local variables set, its security set (if required), and its parameter binding (this is optional and used by the Evaluation pass as described in 3.2.1). A Fragment can be immediately transformed in a delegate ad used directly as shown in (CCC05) where the case study example demonstrate how the extruded Fragment is transformed into a delegate in order to realise parallel computation through the asynchronous execution of a `Delegate`. The IL selection can use values coming outside the boundaries as

- fields of the Host Owner
- external (static members, or members of other references)
- local variables
- parameters of the Host signature

all such references will be promoted to the Fragment signature which is indeed extremely important since it will be used to perform the binding between open variables of the Fragment and values. The extrusion is performed against a selection of IL that has been expanded enough to be stack safe (this guarantees that the delegate creation and the code execution will not corrupt the run-time health). The Extrusion is responsible

to compute the local variable set of the Fragment and some optimisation can be performed to reduce the set. If a local variable is used in a `stloc` instruction and only one `ldloc` is later loading the variable on the stack they can be both removed unless the instruction responsible for pushing on the stack the value used by the `stloc` a `call` (because there can be side effects and thus instruction reordering can alter the behaviour of the snippet). Following that rule the code in Listing 3.8 can be optimised as Listing 3.9 during the extrusion operation without creating a local variable set for the Fragment.

Listing 3.8: IL Snippet

```
1       ...
2       ldstr   "first string"
3       stloc.s a
4       ldstr   "second string"
5       stloc.s b
6       ldloc.s a
7       ldloc.s b
8       call    string [mscorlib]
9             System.String::Concat(string, string)
10      callvirt instance int32 [mscorlib]
11            System.String::get_Length()
12      ...
```

Listing 3.9: IL Extruded Fragment

```
1       ldstr   "first string"
2       ldstr   "second string"
3       call    string [mscorlib]
4             System.String::Concat(string, string)
5       callvirt instance int32 [mscorlib]
6             System.String::get_Length()
7       ret
```

If a local variable is used in more than one `ldloc` then must be added to the local variable set of the Fragment. When a local variable is used only in `ldloc` instructions without any `stloc` inside the snippet then it is promoted to the Fragment signature.

There is a particular scenario when inside the snippet a local variable occurs only in `stloc` instructions since we have two options. If no `call` instruction is involved in the stack loading for the `stloc` then

all the instructions involved can be removed along with the `stloc`, otherwise if `call` is involved we must preserve it and the `stloc` can be replaced by a `pop` operation. In this scenario can be possible to follow another approach because potentially the `stloc` was saving values for later use, then we can promote the `stloc` targets to the Fragment signature with the `out` modifier. When extruding attention must be paid to the treatment of `ret` instructions since they are responsible for leaving the execution of the fragment and loading the return value stored on the stack.

In our approach the extrusion will change `ret` operation with unconditional jumps to the end on the IL code of the fragment, this is important for the Injection since injecting `ret` instructions will alter dramatically the behaviour of the Target. The Extrusion is working in a similar fashion to the approach described in (GG08).

The extrusion operation can be easily defined in terms of the ▲ operation by using a delegate match that will behave as expected by this operation when lifting unclosed variables into the code fragment signature.

Binding

Binding is used to realise partial evaluation over Fragments and to connect them with the environment on the injection point and the available Binding operators are

- Binding_{Local} to bind against a local variable of the Target
- $\text{Binding}_{Parameter}$ to bind against a parameter of the Target
- Binding_{Member} to bind against a member
- $\text{Binding}_{Constructor}$ to bind against a type constructor
- $\text{Binding}_{Constant}$ to bind against a constant value
- Binding_{Call} to bind against a method call
- $\text{Binding}_{Fragment}$ to bind against another Fragment

- Binding$_{Cast}$ to bind against a cast operation
- Binding$_{Array}$ to bind against an array creation and initialisation with values

The Fragment defines a list of bindings called `ParameterBindings` and one binding called `TargetBinding`, they are used by both Evaluation 3.2.1 and Injection 3.2.1 to both perform optimisation (through partial evaluation and inlining) and to connect the Fragment signature with the Target environment. All the binding operator derive from Binding and they implement the `GetCursor` method (Binding is contract defined in Listing 3.10) in order to provide a cursor over the IL code to be used in place of the occurrences of the parameter bound; the contract is not enough and there is the `BindingMode` attribute to be used on Binding sub types so that the usage of the Binding can be constrained to parameters, Target or both.

Listing 3.10: Binding contract definiton

```
1       public abstract class Binding
2       {
3             \\Gets the Binding Mode.
4             public BindingMode {get{...}}
5
6             \\Gets the Fragment to be
7             \\inlined with the bound parameter.
8             public abstract Fragment GetCursor();
9       }
```

The various form of binding are obtained as a variant to the basic *Bind* operation provided by *CodeBricks*. Since we are simply changing the variable sites the programs transformations remain type-safe as in the original model. Moreover this notion is already available in the current implementation of the library.

Evaluation

The Evaluation step is the core component for obtaining optimisation in the snippet before proceeding with the Injection step. The definition of

the Evaluation function is:

$$\blacktriangledown : \mathrm{CF} \rightarrow \mathrm{CF}$$

In 3.2.1 a small optimisation is performed to reduce the local variable set and to remove meaningless `stloc` and `ldloc` operations but the stand alone Fragment cannot be optimised any further without knowledge about the environment surrounding the injection point in the target method (we will refer this as *Target* and the type containing the method as *Target Owner*). The evaluation step is of no use for any Fragment with a signature having no input parameters since there will be no binding with values coming from outside the Fragment. The binding behaves like described in (ACK03) and allows us to perform a binding between parameters of the signature and:

- local variables of the Target
- constant values
- parameters of the Target
- members of the Target Owner
- method calls
- other Fragments

. The description of the binding operation will be discussed in 3.2.1 we will focus now on the effects that some particular binding can have on the evaluation step. The Idea of the evaluation comes from the partial evaluation theory and it is used here to perform something smarter than mere inlining. The bindings relevant to the evaluation are those involving

- constant values
- other Fragments

. Let us assume that we are working with a non trivial code fragment which contains conditional branches, branches can have also additional local variables defined inside theirs scope so they contribute to both code size and working set of the Fragment. If the condition of some branching is depending only on external values coming from the signature of the Fragment (quite possible for statements like `switch`) the binding to a constant can be used to remove all the unnecessary portion of both code and local variables before we proceed with the Injection operation. This extends to also the case of properties of a constant value that can be statically computed; for example let be the binding be against a parameter named `sParam` of type `string` with a constant value as "String Value", we can now compute expressions like Listing 3.11 at evaluation time.

Listing 3.11: IL Extruded Fragment

```
1       ...
2       int wordCount = sParam.Split('0').Lenght;
3       ...
```

The scenario of binding occurring against another code fragment is the most relevant for the Evaluation for it is built using the code manipulation model itself. Before proceeding with the evaluation the Fragment of the binding must be evaluated because if the signature is not without parameter (or closed by other bindings) every open parameter must be included in the signature of the Fragment produced as output of the evaluation step. The code of the Fragment used in the binding will be used to load the stack instead of the `ldarg` instruction related to the parameter that has been bound. If the Fragment used in the binding definition is without parameters (or the signature is completely bound) then the evaluation can perform an optimisation storing the stack loaded by the Fragment in a local variable and reuse the variable in every place where the `ldarg` instruction was used with the parameter being bound. The process will be recursively applied to any code fragment used to bind a code fragment's signature parameter.

Injection

This is the final step of the proposed transformation model and as 3.2.1 has to deal with the stack alteration of the Target. The binding (discussed in 3.2.1) is responsible to prepare the stack before the injection but it is also in charge for leaving the stack after the injection point in the same state as in the Target without any injection. The definition of the Injection is:

$$\blacktriangleright : (\mathrm{CF} \times \mathrm{Mode} \times \mathrm{CF}) \rightarrow \mathrm{CF}$$

where this first F is the source Fragment, Mode is a value amid

- `before`
- `after`
- `replace`

and the last term is the injection point inside the Target defined as an element of the F domain.

The injection point is computed in the same way as the extraction point is and can therefore represent a set of instructions inside the *Target*. If no bindings are defined in the Fragment the Injection must check that the stack before it is matching the Fragment signature, once the check is positively passed a set of binding will be built and an Evaluation step will be performed to optimised further the Fragment IL before it is injected in the Target. A stack compatibility check must be performed also for the IL code immediately after the injection point to ensure that the signature of the Fragment is not compromising the stack. Since the injection point is represented by an IL instruction or a sequence of of it we can use the signature of the injection point to perform the compatibility test with the signature of Fragment to be injected (after evaluation). The signature of the injection point needs some extra care when computed since it could involve selections containing `ret` instructions or being operated over empty methods. If the injection point's IL ends with a `ret` instruction then the return type of the selection is the return type of the Target and the Fragment must respect this contract. It can happen that

Fragment and injection point have mismatching return types and this situation must be resolved in order to make the injection pass the stack and type compatibility check. In the scenario where the injection is returning a `void` type while the Fragment is not the problem can be easily solved appending a `pop` instruction after the last instruction of the Fragment, in the more general situation of return type mismatch the binding technique is used to address the problem and such binding is called *TargetBinding*. Due to the nature of the TargetBinding it is limited to binding towards:

- local variables of the Target
- parameters of the Target (if marked as `out` or `ref`)
- members of the Target Owner
- constructor call
- cast

When injecting in an empty Target or at the very beginning of the Target body the Fragment parameters (after Evaluation) must be either unambiguously matching the Target's ones or be an empty list. Since the Injection will offset instructions in the Target all the branches targets must be fixed to reflect the new instruction set. Branches fixing requires a first scan before injection to track the target of branches in the Target and a scan over the Fragment's branches as well, once the Evaluation and the Injection are performed a new scan over the generated IL is executed so that all the branches can be restored by pointing to new position of targeted instructions. As the final result of Injection is a new Fragment it can be used to obtain a `Dynamic Method` and so executed through a delegate or reflection invocation, more interesting application is the assembly rewriting since the application of the code transformation will be static and thus the use of the new assembly will not be overloaded by any extra run-time support since our approach needs it only when transformation are performed.

It is important to introduce a special Fragment : the *Empty* Fragment or $\mathrm{Fragment}_{null}$, which is a indispensable building block in our framework. Thank to $\mathrm{Fragment}_{null}$ it is possible to define the Delete operation as:

$$\not\triangleleft\ (\mathtt{F}) \equiv\blacktriangleright\ (\mathrm{Fragment}_{null}), replace, \mathtt{F})$$

The delete operation is essential to perform some of the code transformations that the eval step requires.

Chapter 4

Analysis of experiment's results

In this chapter we present the results we obtained applying our method to game technology. We must thank Realtime Worlds ltd (RTW) for their support during our research. Before showing the experiments and their result we need to introduce the main factors that made us go for a .NET based solution instead of using the classic C++ approach and, in this task, the help and support from RTW has been very precious.

Performance in C++ usually sacrifices code readability and programmer productivity on the altar of execution time. The usage of template based programming is generally a big source of productivity issues as they can trigger long compilation phases even for a small change, this is even more dramatic when they are used in core libraries and so a potentially huge code base can be dependant on such source files.

A serious project can take something like 20 to 40 minutes to compile and link (we have been facing this scenario in more than one company with very skilled C++ programmers) and in a typical 8 hour working day during a major debug step something realistically only 3 hours may be spent actually working.

On average the performance loss with C# is around 10%[1] but the per-

[1]The performance is not related to the language used but it is due to the run-time sup-

formance gap between C# and C++ is getting smaller and smaller at every run-time release from **Microsoft** with improved versions of the JIT and GC, making .NET more affordable from a production point of view. As well as the work **Microsoft** is doing on the CLR and its languages, the Mono version of .NET is bringing the standard on other platforms like MacOS, Linux, iPhone, Nintendo Wii, and many others thereby crowning .NET as a cross language and cross platform framework.

As proposed in (Swe06) the introduction of a VM and higher level support in a language can help the development process in a very significant way and can offset the performance loss on the final product. Static and run-time analysis of programs is becoming a key factor in software design and development and game engines are becoming VM (leaving the JIT out of the discussion for licensing and technical restrictions[2]).

port of the VM and the GC.

[2]Usually a console provides a very strict security policy and license agreements preventing developers from deploying VM and JIT compilations, some hardware also enforces memory protection so that code memory is read only after the program has been loaded.

4.1 Domain driven Optimisation for math library

The problem of a general purpose math library inside a video game is extremely relevant for game performance. Being able to optimise this portion of code and its usage patterns will impact both the simulation and the render step (see Appendix B and Appendix C). In CLR VM the debate between using `class` or `struct` to model mathematic domain objects is a very hot topic because the choice will bring to memory allocation savings or garbage collector activity saving (BD05).

Beside the performance problem we had to face code maintainability issues : the math library is usually piece of the foundation block of a game platform, using explicit optimisations would not be an overall benefit. Most of the experiment results reported here are based on a code base we developed as part of experimentation with the .NET framework at Realtime Worlds ltd and we want to thank them for the great support they provided to our research topic.

To design a math library in a STEE is a serious challenge since we have to deal with performance on one hand and with the type semantic on the other. The first thing we struggle with is the type system as in geometry some operations can not have a meaning whilst it is in the mathematical domain, as a quick example of the problem let us mention the tuples and points. It is possible to add tuples and we will obtain a new tuple however in geometry, there is no meaning for adding points together. The only time we see something like that is when the centre of mass is computed for a volume[3]. The difference is quite subtle since in the centre of mass computation we are not adding points at all, we are adding $\text{point} \times \text{mass}$ entities and only after averaging the result we obtain back a point entity. This leads to a design making a difference between the mathematic, physics and geometric domains when defining operations and algorithms.

Another good example of domain specific behaviours is given by the multiplication between matrices and vectors and the multiplication be-

[3]In a realtime application the math is modelled with a discreet approach so volumes and surfaces are manipulated as a finite set of points in the space

tween matrices and points. In the former case the translation component of the matrix is not used whilst in the latter it is, along with the semantic difference there is a performance and implementation alteration.

With domain modelling problems we face design issues when providing implementations for algorithms. If we think for example at the code responsible for intersection between geometric entities we need to face the choice between instance operators and static operators[4].

Choosing the instance operators approach the code will be more easy to write as every entity will expose all the functionality related to it, this will lead to a lot of code replication since we will need to provide methods for sphere intersecting boxes in the sphere object and the same code will appear in the box objects as well[5].

Following the static function approach can lead to a smaller code base and better performance but will put more pressure in development and code architecture design since types will be just data structure with no functionality and developers must have a quick way to access the code they need.

A lot of details about this type of problem can be found in (Ebe06) and since it is not exactly the goal of our experiment we will not pursue this any further unless required further illumination on the research we are reporting here. The domain of an application is not only a semantic aspect of math operations, it is also a redefinition of optimisation goals and techniques as we can drive the improvement on at least three fronts:

- precision
- execution time
- memory consumption

[4]Since we are working with a strongly object oriented language we do not have the ability to define single functions.

[5]In the real world the implementation of instance function uses the code of static operators, if this can solve code maintainability issues we will be facing a penalty at run-time as an instance and a static call must be performed.

4.1.1 Optimising Vector Algebra

The code we presented in Listing 3.2 is generally used when 3D meshes are generated to compute a normal vector for every triangle of the mesh (see 4.2) and, usually, it can be source of performance problems in a VM scenario. While executing the small code snippet three vectors are created for every triangle adding pressure on the managed heap (since our vector is a class and not a structure).

In some of the experiments we ran at Realtime Worlds we observed that the temporary vector creation was adding an overhead of almost 100 MB of memory allocation trashing Gen_0 and thus raising the thread priority of the GC eventually even above the mesh creation thread priority. The optimisation realised in Listing 3.3 has multiple objectives

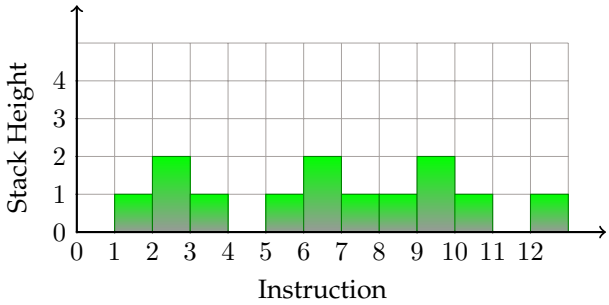
- **Mimic struct behaviour** so that access to the point components is inlined later by the JIT
- **Avoid temporary vector creation** in order to produce no additional stress on the memory manager and GC
- **Avoid external calls** so that the loop execution can be made as fast as possible

As we can see from Figure 4 and Figure 5 graphs we are increasing the `.maxstack` directive since the snippet in Listing 3.2 has a value of 2 and the code in Listing 3.3 increases it to 3 but this resource usage is paid on the method working set and it is constant over the entire loop and not related to the number of triangles of the mesh.

The substitution we performed behaves like a classic inlining technique at first sight but goes further than that: in the C# snippet Listing 3.3 it is evident that the calls to `Cross` and `Normalise` are removed, inlining the IL body of the two invoked ones, but we are performing even more than mere inlining. The domain is known (algebra over vectors and points) so the code optimisation also inlines the components[6] avoiding object creation at the small cost of new local variables of type `float32`.

[6]This behaviour can be observed at JIT time when dealing with `struct` types.

Figure 4: Stack for original code



This technique is not a silver bullet, attention to the stack size must be paid since the size of the stack frame is extremely relevant in recursive calls (when they are not tail calls). Table 1 shows the result we obtained with the code optimisation was used in iterative loops over 1000000 triangles.

In this scenario the query in Listing 4.1 is executed to find where the pattern is used, the code in Listing 4.13 will return a match and the extrusion over it will capture the point local variables loading as the fragment surrounding (signature). The programmer has already provided the optimised fragment as he knows the domain specific goal of the optimisation. Both fragment signatures match so the surrounding of the query result are used to populate the bindings of the optimised snippet before injecting. The transformation is so expressed as

$$\blacktriangleright\ (\blacktriangledown(OptimisedNormal), replace, \blacktriangle(method, NormalQuery))$$

Listing 4.1: Normal Computation Query

```
1 Vector3Df NormalQuery(Point3Df a, Point3Df b, Point3Df c) {
2       Vector3Df e1 = b - a;
3       Vector3Df e2 = c - a;
4       Vector3Df n = Vector3Df.cross(e1,e2);
5       n.Normalise();
6       return n;
7 }
```

Figure 5: Stack for optimised code

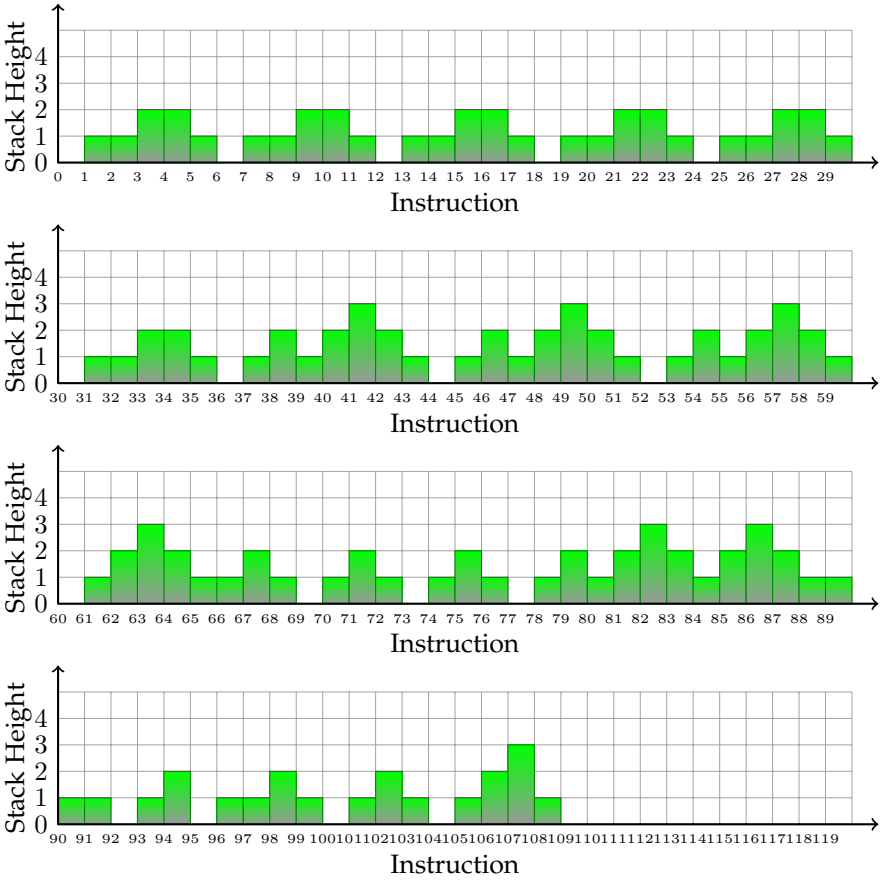


Table 1: Experiment result for triangle normal computation with 3000000 vertices over 1000 test runs

	Original	**Optimised**
Code length	13 IL	109 IL
.maxstack	2	3
Execution time	180 ms	40 ms
Allocation	24436.9921875 MB/s	11884.66796875 MB/s
% Time in GC	27.1% - 5.8%	9.7% - 3.1%

Figure 6: Triangle normal computation

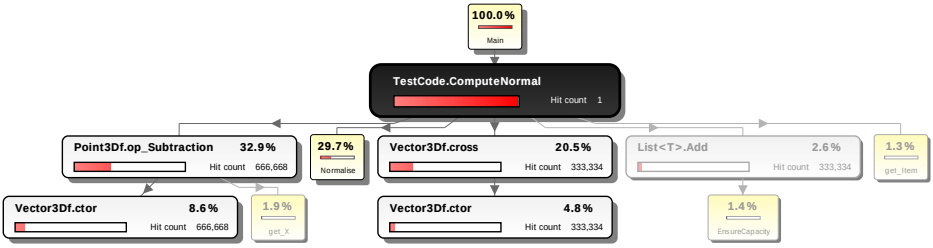


Figure 7: Vector allocation hit in triangle normal computation

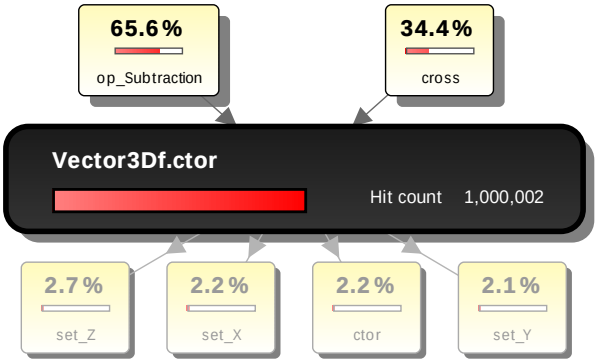
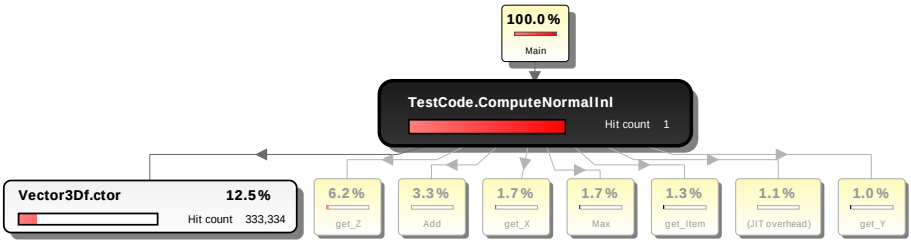


Figure 8: Triangle normal computation with smart inlining



4.1.2 Expression Trees Manipulation

Expression trees can be evaluated and optimised, the next experiment focusses on optimising code for matrix multiplication as to avoid matrix creation obtain a constant memory allocation. The pattern to be identified in this scenario is not simple the occurrence of an operator, we are looking now for a repetition of the operator with a number of occurrences greater than a threshold. We are about to show an experiment we ran on matrix multiplication, multiplication is the most frequent matrix operation in graphics and geometry as it is used to chain transformation along a kinematic chain (or model hierarchy).

Code as Listing D.3 triggers a temporary matrix creation at every multiplication operator occurrence and thus potentially leads to Gen0 thrashing in the GC; such problems can be avoided by transforming Listing D.3 in Listing D.4 obtaining the results shown in Table 2. Despite of the large increase in instructions (optimised version is almost 20 times the size of the original code) we observe a 300% boost in performance. Both codes use the same accessor for extracting and setting the components of every matrix which leads us to the conclusion that the improvements come from the reduced time spent in creating temporary matrices.

The optimisation introduces the cost of creating an array for containing references to all of the matrices involved in the expression and that is the reason why we search for patterns with more than 2 matrices involved. We look for matches in the code where the optimisation leads to a real performance gain; the stack size is changed as well and it increased from two to six (see Figure 13 and Figure 14 to have a better view of the impact of the transformation).

As the results show the original code spends 6.768% of 0.0016 ms (7.2% of the 94% spent in multiplication, as shown Figure 9)in constructor execution and it will increase the stress on GC and allocation, while the optimised code spends only 2.6% of 0.0005 ms (Figure 10) in constructor execution. Of course the optimised version is more demanding at JIT time but the cost is amortised over the number of executions.

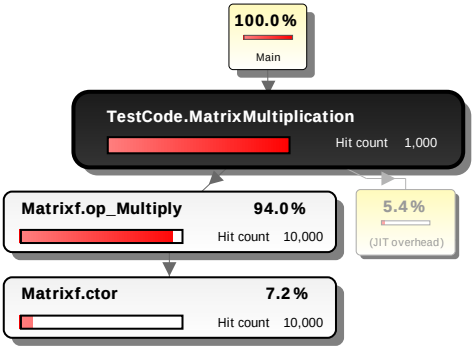
There is a further peculiarity about expression tree matching: most of

the optimisations we were testing can be performed at compile time but the code for both the normal computation and the matrix multiplication can be boosted further by hardware dependant optimisations like SIMD instructions(MY02) or multi core leveraging variants.

Table 2: Experiment result for matrix multiplication with 11 matrices over 1000 test runs

	Original	**Optimised**
Code length	15 IL	247 IL
.maxstack	2	6
Execution time	0.0016ms	0.0005 ms

Figure 9: Matrix Multiplication



As clearly visible from Figure 11 and Figure 12 the 1000 iterations produces 10011 hits on the Matrix constructor code for the original approach and only 1011 for the optimised code, 11 hits are due to the 11 matrices constructor so they are irrelevant to the goal of the experiments. What is worth more attention is that the original approach has a resource usage proportional to the length of the expression while the optimised version is constant in resource consumption.

Figure 10: Matrix Multiplication with smart inlining

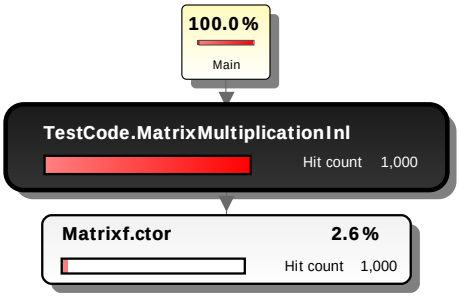


Figure 11: Matrix allocation hit

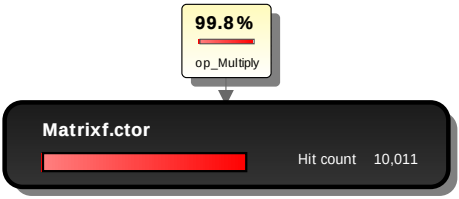


Figure 12: Matrix allocation hit after optimisation

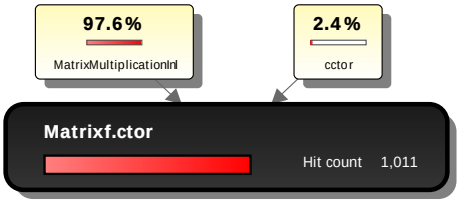
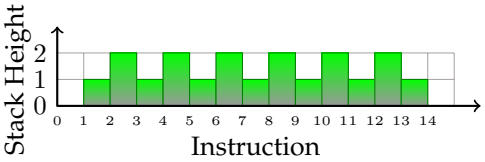


Figure 13: Stack for MatrixMul



The transformation for matrix multiplication is therefore expressed as

$$\blacktriangleright (\text{MulOptFragment}, replace, \blacktriangle_s(m, MatrixMul))$$

where MulOptFragment has the code of Listing D.4 from line 2 to line 43 and the signature `MatrixF (Matrixf[])`, a Binding_{Array} operator is used to bind the input parameter with an array initialised with the matrices involved in the operation surrounding (or the code responsible for their load on the stack since bindings can be composed).

The MulOptFragment is computed from the fragment returned from the

$$\blacktriangle_s(m, MatrixMul)$$

query over the fragment m that represents an entire method body, the query is similar to the one presented in Section 3.1.5.

It is possible to optimise complex expression trees (see Chapter 10 of (CE00) for more details), if we have expressions such as

$$a + b + c + d + (e \times f \times g) + h + a + b + c + ...$$

matching the sum sequence we will obtain a fragment with six elements in the signature (the extrusion operation will promote only unique values to the signature):

- a
- b
- c
- d
- $e \times f \times g$
- h

Due to the expression $a+b+c$ occurring twice it will be pre computed and the final code will be similar to Listing 4.2

Figure 14: Stack for MatrixMulOpt 1 of 3

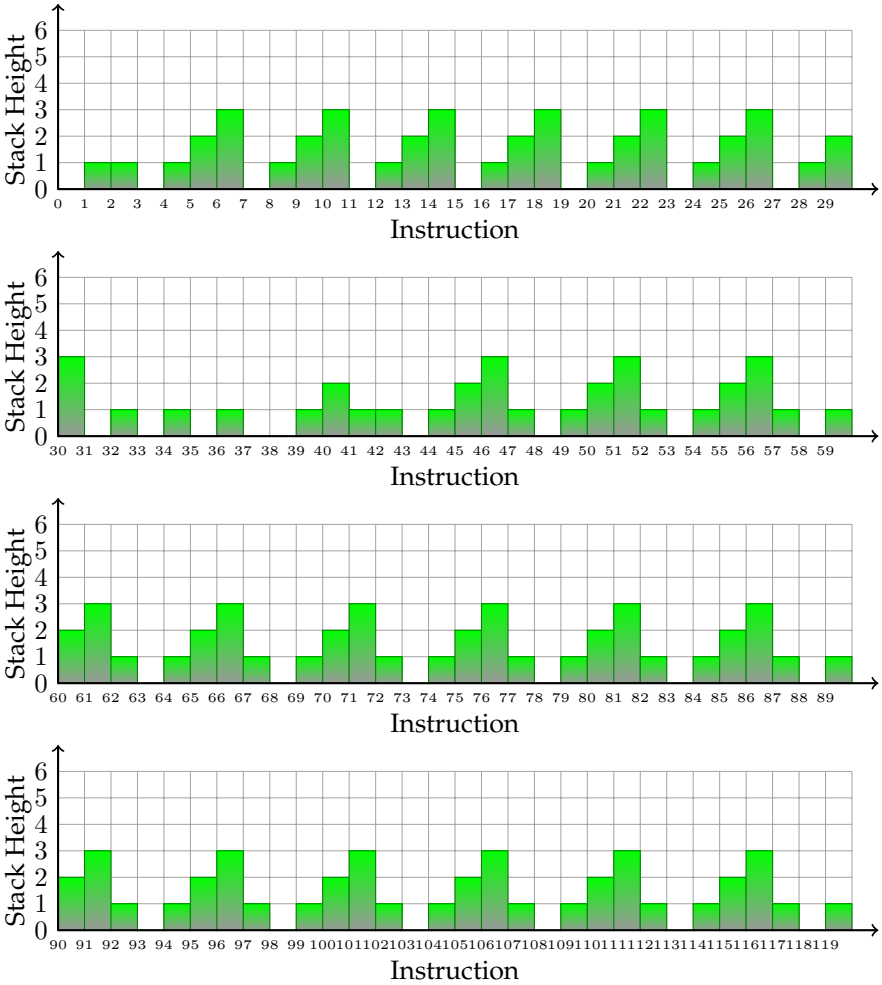


Figure 15: Stack for MatrixMulOpt 2 of 3

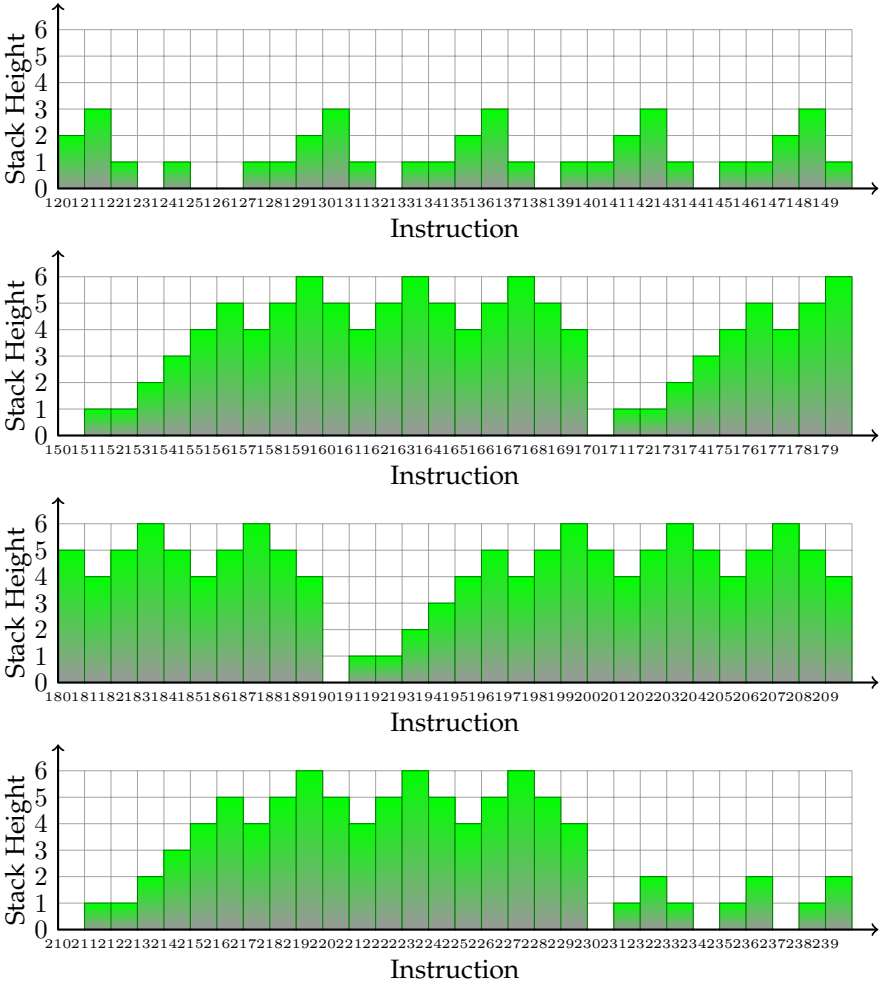
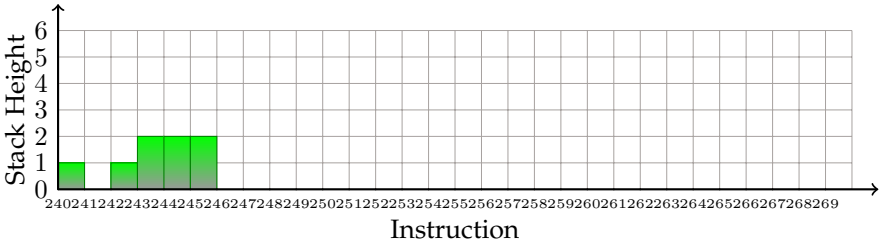


Figure 16: Stack for MatrixMulOpt 3 of 3



Listing 4.2: Expression Optimisation

```
1 Matrx m = a + b + c;
2 Matrix n = e * f * g;
3 Matrix res = m + d + n + h + m;
```

4.1.3 Articulated problem with Object Oriented Bounding Box Computation

The experiments illustrated so far are extremely punctual and while they achieved the goal they are simply building blocks for more articulated portions of code. More interesting and significant examples can be provided by a function computing an Object Oriented Bounding Box (OOBB) for a section of geometry.

An OOB for a 3D mesh is the minimum volume box containing the mesh and aligned with its principal axis, such a structure is important for rendering culling (see Appendix C) and for collision detection in physic simulation.

Listing 4.3 shows the computation of the OOBB using the math library.

Listing 4.3: OOBB Computation

```
1  Vector3Df up; // Y axis
2  Vector3Df dir;// Z axis
3  Vector3Df right;// X axis
4  Point3Df baricentre = Geom.ComputeBaricentre(m.Vertices);
5  Alg.ComputeMajorAxis(m.Vertices, out right, out up, out dir);
6  float w,h,d;
7  w = float.Max;
8  h = float.Max,
9  d = float.Max;
10 float W,H,D;
11 W = float.Min;
12 H = float.Min;
13 D = float.Min;
14 foreach(var vertex in m.Vertices) {
15       Point3Df p = vertex - baricentre;
16       W = Math.Max(W, Math.Abs(p.DotProduct(right));
17       H = Math.Max(H, Math.Abs(p.DotProduct(up)));
18       D = Math.Max(D, Math.Abs(p.DotProduct(dir)));
19       w = Math.Min(w, Math.Abs(p.DotProduct(right)));
20       h = Math.Min(h, Math.Abs(p.DotProduct(up)));
21       d = Math.Min(d, Math.Abs(p.DotProduct(dir)));
22 }
23 OOBBox box = new OOBBox(baricentre,
24       W, H, D,
25       w, h, d,
26       right, up, dir);
```

The loop (line 14 to line 22) presents a pattern not too different from the optimisation in Section 4.1.1. The optimisation we will adopt here as a result is very similar to the one shown before: we will avoid the creation of the point p (line 15) and the dot product computation will be inlined.

The interesting pattern in this example is represented by line 4 and line 5. The code for baricentre calculation is already optimised and it is as in Listing D.5, there is no temporary point creation and the loop is done with a `foreach` statement over a `Point3Df` array so that no boundary check is performed[7]. In the case where the baricentre calculation was not optimised already we would apply a query to find the pattern in Listing 4.4.

Listing 4.4: Baricentre Query

```
1  Point3Df BaricentreQuery(Point3Df[] vertices)
2  {
3        Point3Df b = new Point3Df();
4        foreach(var v in vertices)
5        {
6              b += v;
7        }
8        b /= vertices.Lenght;
9        return b;
10 }
```

The implementation of the `ComputeMajorAxis` method can be found in Listing 4.5. The method call at line 6 is something we wish to investigate further.

Listing 4.5: Major Axis Extraction

```
1  public static void ComputeMajorAxis(Point3Df[] vertices,
2        out Vector3Df right,
3        out Vector3Df up,
4        out Vector3Df direction)
5  {
6        Matrix m = ComputeInertiaTensorMatrix(vertices);
7        Math.EigneenVectors(m,
8              out right,
9              out up,
```

[7]This is another optimisation that can be performed to help the JIT in creating more efficient native code.

```
10              out direction);
11          right.Normalise();
12          up.Normalise();
13          direction.Normalise();
14          Alg.OrderWithWorld(Matrix.Identidy,
15              out right,
16              out up,
17              out direction);
18  }
```

Inertia tensor is a matrix defined as

$$H = \begin{bmatrix} \sum_i m_i(y_i^2 + z_i^2) & -\sum_i m_i x_i y_i & -\sum_i m_i x_i z_i \\ -\sum_i m_i y_i x_i & \sum_i m_i(z_i^2 + x_i^2) & -\sum_i m_i y_i z_i \\ -\sum_i m_i z_i x_i & -\sum_i m_i z_i y_i & \sum_i m_i(x_i^2 + y_i^2) \end{bmatrix}$$

or in a more compact form

$$H = \begin{bmatrix} X & a & b \\ a & Y & c \\ b & c & Z \end{bmatrix}$$

where

$$X = \sum_i m_i(y_i^2 + z_i^2)$$

$$Y = \sum_i m_i(z_i^2 + x_i^2)$$

$$Z = \sum_i m_i(x_i^2 + y_i^2)$$

$$a = -\sum_i m_i x_i y_i$$

$$b = -\sum_i m_i x_i z_i$$

$$c = -\sum_i m_i z_i y_i$$

The code to compute the inertia tensor is therefore as shown in Listing 4.6 where the m_i term is set to 1.

Listing 4.6: Inertia Tensor Computation

```
1  public static Matrix ComputeInertiaTensorMatrix(Point3Df[] vertices)
2  {
3        float X,Y,Z,a,b,c;
4        foreach(var vertex in vertices)
5        {
6              float x = vertex.X;
7              float y = vertex.Y;
8              float z = vertex.Z;
9              X += y*y + z*z;
10             Y += z*z + x*x;
11             Z += x*x + y*y;
12             a += -(x*y);
13             b += -(x*z);
14             c += -(z*y);
15       }
16       return new Matrix(X,a,b,a,Y,c,b,c,Z);
17 }
```

As we can see the inertia tensor and the baricentre computation will be performing a loop over the vertices of the mesh. Since both code perform a *readonly* access on the vertices collection we want them to share one `foreach` loop. Executing $\overline{\blacktriangle}(\texttt{m}, \texttt{Q01})$, where `m` is the fragment containing the code in Listing 4.3 and `Q01` is the query in Listing 4.7, we will obtain a match for

- `m1` at Line 4
- `m2` at Line 5
- `m3` from Line 14 to Line 22

When the matches are extruded the surroundings of `m2` and `m3` will show a *RAW*[8] condition between them, the same behaviour appears also between `m1` and `m3`.

Listing 4.7: Query

```
1  delegate void pIt(ICollection<Point3Df> p);
2
3  delegate void pFunc(ICollection<Point3Df> p);
4
5  void Q01(ICollection<Point3Df> pI,pIt it){
```

[8]Read After Write.

```
6       it(pI);
7 }
8
9 void Q02(ICollection<Point3Df> pI,pFunc pf){
10      foreach(var p in pI){
11            pf(p)
12      }
13 }
```

We are interested in all the matches of `Q01` that do not have dependencies between them, those matches are potentially `forearch` loops of calls to functions which perform loops over the collection of `Point3Df` and we want to recombine them so that we can minimise the number of iterations over the collection.

$\overline{\blacktriangle}(\texttt{m1}, \texttt{Q02})$ and $\overline{\blacktriangle}(\texttt{m2}, \texttt{Q02})$ will not produce any match but $\overline{\blacktriangle}(\texttt{m1}, \texttt{Q01})$ and $\overline{\blacktriangle}(\texttt{m2}, \texttt{Q01})$ will capture the calls to `Geom.ComputeBaricentre` and `Alg.ComputeMajorAxis` in the surrounding of the matches.

The process will be repeated until we find at least two distinct matches for `Q02`, once this condition is met we can proceed with the transformation replacing line 4 and 5 of Listing 4.3 with a new fragment `FELoop`.

`FELoop` is the result of $\blacktriangleleft\ (\blacktriangle(\texttt{Q02}, \texttt{Q02}))$ where the parameter `pF` will by bound with the fragment obtained from the operation `mBaricentre`+ `mInertia`, where the + operator combines by appending the fragments that matched `Q02` over the bodies of

- `Geom.ComputeBaricentre`
- `Alg.ComputeMajorAxis`

The `FELoop` fragment has the code of Listing 4.8 and it will be injected after being combined with the remaining code of the method `Alg.ComputeMajorAxis` replacing the method calls and, finally, obtaining the code reported in Listing 4.9.

Listing 4.8: Fore Each Loop Fragment

```
1 float X,Y,Z,a,b,c,bx,by,bz;
2 float count = vertices.Count;
3 foreach(var vertex in vertices) {
4       // Baricentre
5       bx += x / count;
```

```
6         by += y / count;
7         bz += z / count;
8
9         // InertiaTensor
10        float x = vertex.X;
11        float y = vertex.Y;
12        float z = vertex.Z;
13        X += y*y + z*z;
14        Y += z*z + x*x;
15        Z += x*x + y*y;
16        a += -(x*y);
17        b += -(x*z);
18        c += -(z*y);
19  }
20  Point3Df Baricentre = new Point3Df(bx, by, bz);
21  Matrix InertiaTensor = new Matrix(X,a,b,a,Y,c,b,c,Z);
```

Listing 4.9: Loop usage optimisation

```
1   Vector3Df up; // Y axis
2   Vector3Df dir;// Z axis
3   Vector3Df right;// X axis
4
5   float X,Y,Z,a,b,c,bx,by,bz;
6   float count = vertices.Count;
7   foreach(var vertex in vertices) {
8         // Baricentre
9         bx += x / count;
10        by += y / count;
11        bz += z / count;
12
13        // InertiaTensor
14        float x = vertex.X;
15        float y = vertex.Y;
16        float z = vertex.Z;
17        X += y*y + z*z;
18        Y += z*z + x*x;
19        Z += x*x + y*y;
20        a += -(x*y);
21        b += -(x*z);
22        c += -(z*y);
23  }
24  Point3Df Baricentre = new Point3Df(bx, by, bz);
25  Matrix InertiaTensor = new Matrix(X,a,b,a,Y,c,b,c,Z);
26
27  Alg.EigneenVectors(InertiaTensor,
28        out right,
29        out up,
30        out dir);
31  right.Normalise();
```

```
up.Normalise();
direction.Normalise();
Alg.OrderWithWorld(Matrix.Identidy,
      out right,
      out up,
      out dir);

float w,h,d;
w = float.Max;
h = float.Max,
d = float.Max;
float W,H,D;
W = float.Min;
H = float.Min;
D = float.Min;
foreach(var vertex in m.Vertices)
      Point3Df p = vertex - baricentre;
      W = Math.Max(W, Math.Abs(p.DotProduct(right)));
      H = Math.Max(H, Math.Abs(p.DotProduct(up)));
      D = Math.Max(D, Math.Abs(p.DotProduct(dir)));
      w = Math.Min(w, Math.Abs(p.DotProduct(right)));
      h = Math.Min(h, Math.Abs(p.DotProduct(up)));
      d = Math.Min(d, Math.Abs(p.DotProduct(dir)));
}
OOBBox box = new OOBBox(baricentre,
      W, H, D,
      w, h, d,
      right, up, dir);
```

Now that the code is optimised with respect to memory allocation and loop iterations there is another step that we can perform: if the hardware is a multi core processor we can replace the `foreach` blocks with the extension method `Parallel.Foreach` of `System.Threading` namespace. Before proceeding we need to leverage the code generation capabilities of .NET to correctly address the concurrent access to the variables written in the loops.

We are using increment operators in the first loop while the second uses the `Math.Max` and `Math.Min` to compare the store value and the current, this in not something that we want to resolve using `Interlocked` methods as they will cause a serialisation of every thread. We need to use the local storage of the thread and combine only upon the exit of each thread.

The `Parallel.Foreach` method lets us achieve the goal but allows

for only one object to be returned forcing an aggregation, this means that we need to create new types to contain all the values we need to return. In the first loop we will create the class in Listing 4.10 while the second loop will use the one in Listing 4.11, those classes are created using the surroundings of the fragments that $\overline{\blacktriangle}(\texttt{mOtp}, \texttt{Q02})$ matched against Listing 4.9.

Listing 4.10: State object for Baricentre and InertiaTensor Loop

```
1  public class SharedState01 {
2     public float bX { get; set; }
3     public float bY { get; set; }
4     public float bZ { get; set; }
5     public float X { get; set; }
6     public float Y { get; set; }
7     public float Z { get; set; }
8     public float a { get; set; }
9     public float b { get; set; }
10    public float c { get; set; }
11 }
```

Listing 4.11: State object for extents loop

```
1  private class SharedState02 {
2     public float W { get; set; }
3     public float D { get; set; }
4     public float H { get; set; }
5     public float w { get; set; }
6     public float d { get; set; }
7     public float h { get; set; }
8
9     public SharedState02() {
10       w = float.MaxValue;
11       h = float.MaxValue;
12       d = float.MaxValue;
13       W = float.MinValue;
14       H = float.MinValue;
15       D = float.MinValue;
16    }
17
18 }
```

Once the state objects are generated we need to modify the `foreach` loops with the call to `Parallel.Foreach` methods and the code fragments must be extruded to create the delegates for the body. The code of the body is also used as a template to generate the delegates for the

accumulation step of the loop using a lock statement. Listing 4.12 shows the final stage of the optimisation process where all the fragments are correctly bound to the state objects.

Listing 4.12: Parallel version

```
Vector3Df up; // Y axis
Vector3Df dir;// Z axis
Vector3Df right;// X axis

var acc01 = new SharedState01();
Parallel.ForEach<Point3Df, SharedState01>(points,
   () => new SharedState01(),
   (p, state, it, acc) => {
      float x = p.X;
      float y = p.Y;
      float z = p.Z;

      float count = (float)(points.Length);

      acc.bX += x / count;
      acc.bY += y / count;
      acc.bZ += z / count;

      acc.X += y * y + z * z;
      acc.Y += z * z + x * x;
      acc.Z += x * x + y * y;

      acc.a += -(x * y);
      acc.b += -(x * z);
      acc.c += -(z * y);

      return acc;
   },
   (local) => {
      lock (acc01) {
         acc01.bX += local.bX;
         acc01.bY += local.bY;
         acc01.bZ += local.bZ;

         acc01.X += local.X;
         acc01.Y += local.Y;
         acc01.Z += local.Z;

         acc01.a += local.a;
         acc01.b += local.b;
         acc01.c += local.c;
      }
   });

```

```
Point3Df baricentre = new Point3Df(acc01.bX, acc01.bY, acc01.bZ);
Matrix m = new Matrix(acc01.X,
      acc01.a,
      acc01.b,
      acc01.a,
      acc01.Y,
      acc01.c,
      acc01.b,
      acc01.c,
      acc01.Z);
Alg.EigneenVectors(m, out right, out up, out dir);
right.Normalise();
up.Normalise();
dir.Normalise();
Alg.OrderWithWorld(Matrix.Identity, out right, out up, out dir);
var acc02 = new SharedState02();
Parallel.ForEach<Point3Df, SharedState02>(points,
   () => new SharedState02(), (vertex, state, it, acc) => {
      Vector3Df p = vertex - baricentre;
      acc.W = Math.Max(acc.W, Math.Abs(p.DotProduct(right)));
      acc.H = Math.Max(acc.H, Math.Abs(p.DotProduct(up)));
      acc.D = Math.Max(acc.D, Math.Abs(p.DotProduct(dir)));
      acc.w = Math.Min(acc.w, Math.Abs(p.DotProduct(right)));
      acc.h = Math.Min(acc.h, Math.Abs(p.DotProduct(up)));
      acc.d = Math.Min(acc.d, Math.Abs(p.DotProduct(dir)));
      return acc;
   },
   (local) => {
      lock(acc02) {
         acc02.W = Math.Max(acc02.W, local.W);
         acc02.H = Math.Max(acc02.H,local.H);
         acc02.D = Math.Max(acc02.D, local.D);
         acc02.w = Math.Min(acc02.w, local.w);
         acc02.h = Math.Min(acc02.h, local.h);
         acc02.d = Math.Min(acc02.d, local.d);
       }
    });

OOBBox box = new OOBBox(baricentre,
acc02.W, acc02.H, acc02.D,
acc02.w, acc02.h, acc02.d,
right, up, dir);
```

Table3 reports the results we obtained with a substantial performance boost, the parallel version is executed on a quad core processor and the `System.Threading` namespace is responsible to allocate the correct number of threads over the available cores. It is important to notice that we kept the original algorithmic process which implies that all the per-

formance we gained is due to fewer allocations, loop sharing, and simple multithreaded support.

Table 3: Execution times for 1000000 triangles with 1000 iterations

	Original	**Optimised**	**Optimised parallel**
Execution time (ms)	**394.93**	**233.87**	**179.79**

The final version of the code executes 216.7% faster than the original code with the ability to adapt to the current hardware configuration and computational load of the machine. This is an important result as the debug process would not have been trivial on the parallel variant of the algorithm, even with sophisticated debugging facilities.

4.1.4 Considerations

The optimisation we introduced so far are not meant to just rewrite the math library, they are driven by specific code provided by the programmers with specific knowledge of the application domain. There can be portion of the application that could not be improved by inlining since this could increase the size of the method too much and so be ignored by optimisation rules of the JIT. Since our example are obtained with IL modified by other IL execution rather than being performed by a compiler, metadata can be used to tag code that want to perform optimisations.

Listing 4.13 shows a snippet of C# where the method is tagged with a custom attribute which specifies the intent of the programmer to optimise the code.

Listing 4.13: Method tagged for optimisations

```
1  ...
2  [Optimisation(typeof(ReduceVectorAllocation))]
3  public void CreateMesh()
4  {
5          ...
6          // create  triangles  and  their  normals
7          Vector3Df e1 = p1 - p0;
8          Vector3Df e2 = p2 - p0;
9          Vector3Df n = Vector3Df.cross(e1, e2);
10         n.Normalise();
11         ...
12 }
13 ...
```

The attribute is used to add to the strategy intended for optimisation to metadata of the method. This can be used in conjunction with other optimisations as the type and the assembly can provide other optimisations with a broader scope. This way the method is selected for optimisation and the `ReduceVectorAllocation` strategy will be used first, subsequent stages can apply other optimisation to the new `CreateMesh` method before it is processed by the JIT.

Although these results are compelling we must further consider the outcomes of these experiments. The code without inlining will be compiled quickly by the JIT and the code of the mathematical library will be

compiled only once so the Table 4 shows the number of times a function of the mathematical library must be compiled.

Table 4: JIT executions

	Original	**Optimised**
Usage number (unique methods)	7000	7000
Inlined calls	0	3000
JIT	**1**	**3001**

The Table 4 shows how many times the JIT will go over the code of a function F of the mathematical library per unique method using it at least once in its body, this result only impacts the application bootstrap and thus does not affecting run-time performance. To reduce the overhead in bootstrapping it is possible to cache the native image and avoid compilation in future executions. The experiment concerning the computation of triangle normals shows a speed up of 640% when inlined, the result is even more important if we look at the time spent in executing the `Vector3Df` constructor.

The optimised version spends only 12.5% of its execution time in creating new vectors while the original code uses 28% of the execution time (which is 6.4 times greater already) in constructor execution. This also impacts the GC behaviour as we can see from Table 1 : 25% of the original code execution time is spent in the GC because of the amount of overhead introduced by temporary vector creation.

We referred to our transformation as *smart inlining* as it behaves differently to inlining in C/C++. In C/C++ the inline keyword is used to suggest to the compiler to inline every call to the function while, in our approach, inlining is specified at usage points. To move the inlining/optimisation on the caller means that code bloating can be avoided and that the programmer is aware of the effect of the optimisation. This helps during debugging and code maintenance as the optimisations can be disabled without recompiling the entire application.

In the latest example the transformation is quite articulated but it

is worth mentioning that it is not performed by a generic optimisation strategy. When coding the algorithms we kept them factored for a good maintainability and debugging, this is the most important step in software engineering for companies where the code base is the main asset or product. The programmers, however, knew what patterns are likely to occur and how to improve them by *shuffling* and *translations*.

With our framework they were able to write code to look for specific behaviours and to provide efficient refactoring for them. It would have been of no use to develop using low performance algorithms and then to switch to more sophisticated alternatives as this could alter other aspects of the applications behaviour. The goal was exactly to "improve existing functionality" instead of debugging a prototype and shipping something different.

4.2 Procedural content Design by means of Meta-Programming techniques

Procedural content is content which can be computed from a function and a set of parameters. Digital Content is created through tools called *DCC* (*D*igital *C*ontent *C*reation tool). Procedural content is widely diffused in all the scenarios where community (or user) created content is a component of the game play (APB[9], Spore[10], Little Big Planet[11]) as the immediate benefit gained by having procedural content is a huge saving in storage. Table5 shows a quick sample of storage usage to store a cube and the compression rate achieved with procedural content is quite high. Data compression is realtively important as it will save both

Table 5: Data storage need for a 3D cube

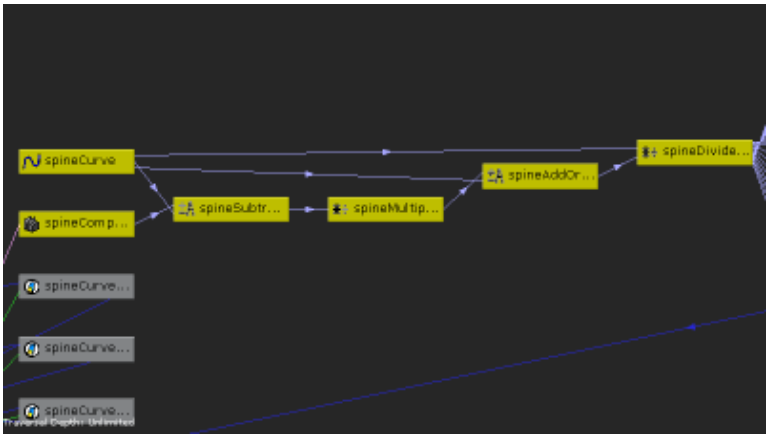
	Procedural	**Static**
Position	3 float	3 float
Size	1 float	0 float
Vertices	0 float	72 float
Normals	0 float	72 float
UVs	0 float	48 float
Total Size	**16 Bytes**	**780 Bytes**

storage and bandwidth (crucial element in network streaming scenarios), saving space has another implication for games, it can affect the performance and the user experience. In (CCH09) a technique based on AOP and annotations (similar to (CCC05)) is used to model procedural content as combination of code and parameters, the technique has been revisited since annotations can be removed as stated in Section 3.1.5. We will revisit the procedural wall example presented in (CCH09) giving more details about he content pipeline and the stages of code generation.

[9] Realtime Worlds APB `www.apb.com`
[10] EA Spore `http://www.spore.com/ftl`
[11] Media Molecule Little Big Planet `http://www.littlebigplanet.com/`

During the content design an artist uses a set of software and tools tailored for the task, both DCC tools and applications share a set of signatures used to express data generation procedures[12]. At the DCC stage the content is created and represented as a data flow where computing and data nodes are object defined in the application type system. Such an approach is quite common in DCC tools and Figure 17 shows the object model behind 3D content in Maya[13].

Figure 17: Maya Hypergraph for content representation



Analysing the procedural wall in Figure 18 we can see that is is generated using a `PolyLine` entity to model the path (Figure4.20(a)), a `PolyLine` object to express the profile (Figure4.20(b)), then a `Extrude` function creates the wall geometry (Figure 4.20(c)). Instead of using the final mesh as an output of the DCC a class will be generated and it is equivalent to Listing 4.15 where the class `ProceduralContent` provides the contract as described n Listing 4.14.

To complete the code in order to create the final style of the wall

[12]Profile extrusion is one of the generation procedures and others are CreateBox, CreateSphere, and so on.

[13]Maya is a DCC tool from Autodesk (`http://usa.autodesk.com/adsk/servlet/index?id=7635018&siteID=123112`), it is largely used in video game content creation and digital movies.

Figure 18: Procedural Wall

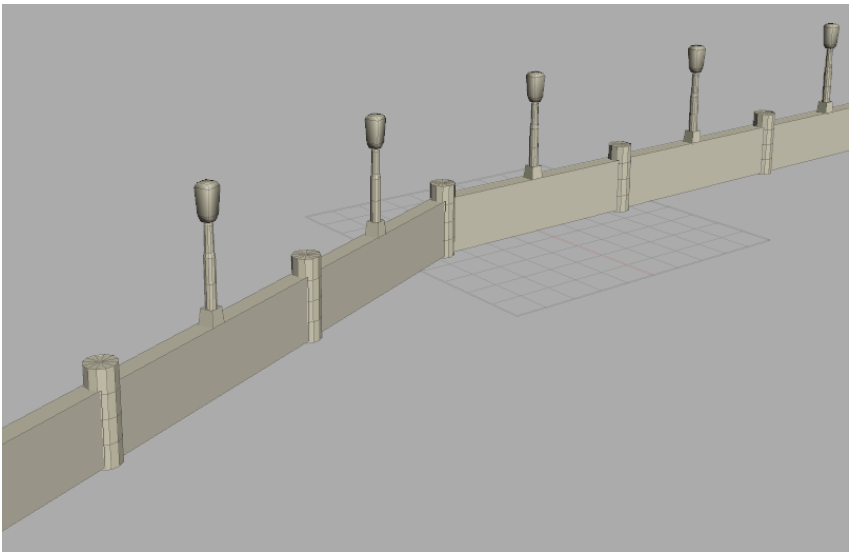
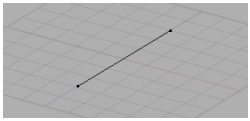
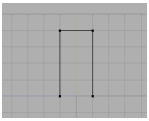


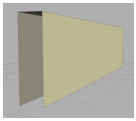
Figure 19: From a polyline to Complete Wall.



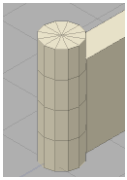
(a) polyline segment



(b) profile to be extruded

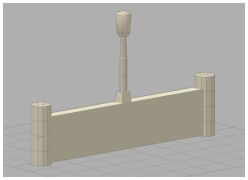


(c) extruded polygon

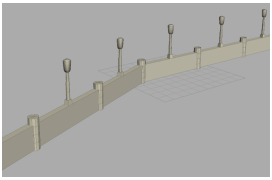


(d) cylindrical pole

(e) lamppost



(f) wall section



(g) full wall

the `Create` method will be as Listing 4.16; the final output of the DCC will be the `ProceduralWall` class and the Path data while all the other members of the class will be initialised in the `initialise` method. The class we have just created is in *Editable* form as the code is not optimised, every call to generator function is explicit and every input is represented by members of the class itself. This is done in order to allow for recreation of the object graph and state inside the DCC tool for further modifications.

When stored in the game data (or game servers in a network streaming scenario) the `ProcedurallWall` class is contained in a stand alone assembly, without any other content generators, and versioning is applied. The application will use only use the latest versions and the `Path` definition is loaded separately. Separation between state and code allows the code to be cached and used against all the Wall entities loaded. The first time a content generator is loaded it is transformed in a new class where the calls to generator functions are inlined and optimisations as described in 4.1 are applied to the new `Create` method.

Class members not marked with the `input` attribute (see line 5 in Listing 4.15 and line 5 in Listing 4.16) are closed in the `Create` method as they are used as constant values in the algorithm and so CLR value types are moved from the managed heap onto the stack. Generator functions are represented in the code library with more than one implementation the choice of which to use is left to the application as it can have hardware or user setting dependent behaviour (GPU code, multicore, etc). Closing values alos has an impact on the `Initialise` method as the body of this method can be reduced once we fix numeric constants (like the float used to computed a point along a segment).

The run-time code generation in the content scenario is quite important since it is not possible to model upfront all of the possible content entities that the user will interact with. This is more evident when content can be edited and created after the application release or deployment. Some video games use dynamic languages and interpreters to deal with the late binding of specific implementations for content generation code howver this leads to execution overhead and potential code injection se-

curity issues. Other benefits of the separation between content data and content generators in network streaming scenarios are

- if only content data is changed content generators do not need to be downloaded and generated again
- if the content generator is updated the content data cached is still valid

Listing 4.14: Procedural Content Contract

```
1 public class abstrc ProceduralContent
2 {
3       public abstract void Initalise();
4
5       public abstract Mesh Create();
6 }
```

Listing 4.15: Procedural Wall Class

```
1  public class ProceduralWall : ProceduralContent
2  {
3        public PolyLine Profile {get;set;}
4        [Input]
5        public Polyline Path {get;set;}
6
7        public virtual void Initalise()
8        {
9              // create  Profile  and Path  objects
10       }
11
12       public virtual Mesh Create()
13       {
14             return Extrude(Profile, Path);
15       }
16 }
```

Listing 4.16: Create method

```
1  public class ProceduralWall : ProceduralContent
2  {
3        public PolyLine Profile {get;set;}
4        [Input]
5        public Polyline Path {get;set;}
6        public float CylinderRadius {get;set;}
7        public float CylinderHeight {get; set;}
```

```
8       public int CylinderSteps {et;set;}
9       public float LightRelPos {get;set;}
10      public Vector3f LightOffset {get;set;}
11
12      public virtual void Initalise()
13      {
14              // create Profile and Path objects
15              // set radius, steps, and height for cylinder
16              // set rel position and offset for light
17      }
18
19      public virtual Mesh Create()
20      {
21            Mesh m = Extrude(Profile, Path);
22
23            foreach (var Vertex in Path)
24            {
25                  Mesh c = Cylinder(vertex, CilinderRadius,
26                        CylinderrHeight, CylinderSteps);
27                  m.Append(c);
28            }
29
30            foreach(var Segment in Path)
31            {
32                  Point3f pos = Segment.GetPoint(LightRelPos)
33                        + LightOffset;
34                  Mesh l = CreateLight(pos);
35                  m.Append(l);
36            }
37
38            return m;
39      }
40  }
```

In (CCH09) we stated that code as representation for content can help in content simplification so to achieve Level Of Detail (LOD) creation (PS97); this is a very sensitive task as automatic data reduction approaches can lead to *unpleasant* results on the screen. Within our framework it is possible to address the problem with two strategies

- custom definition of LOD as new generators
- custom strategies for generator simplifications

While adopting the former strategy we simple reiterate the process for generator creation the latter is more interesting for our experiments.

At the same time the DCC user is creating the wall definition the user can also define simplification methodologies for every aspect of the wall creation. For example the content creator can indicate in the DCC that the action (`Fragment`) in Listing 4.17 can be "simplified" by means of substitution with the action in Listing 4.18.

Listing 4.17: Extrude fragment

```
1 Mesh m = Extrude(Profile, Path);
```

Listing 4.18: Extrude simplification fragment

```
1 Mesh m = new Mesh();
2 float height = MaxY(Profile);
3 foreach (var Segment in Path)
4 {
5       m.Append( Quad(Segment.P0,
6             (Segment.P1 + new Vector(0,height,0)) );
7 }
```

So we will end with having a set of `Simplificator` objects that will perform

$$\blacktriangleright\ (\text{ExtrudeSimplifiedFragment}, \textit{replace}, \blacktriangle(\text{WallAssembly}, \text{ExtrudeFagment}))$$

to simplify extrude operations in WallAssembly rewriting the procedural wall class to obtain an LOD generator. Line 2 in Listing 4.18 will be optimised further as the `Profile` object is not an input of the wall generation and this will be the only line where it is used. The code will be replaced with the value of the computation at evaluation time for the new method body fragment. Procedural Content Generation is particularly interesting as it requires multi staging capabilities and can be schematised as the procedure in Listing 4.19.

Listing 4.19: Multistaging in Procedural Content Generation

```
1 // Create the target assembly
2 var targetAssembly;
3 // create the Lod0
4 var currentLod = Generate(description);
5 targetAssembly.Add(Lod0);
6 for (int i = 1; i < maxLod; i++)
7 {
```

```
8         // Simplify the generator
9         currentLod = Simplify(currentLod);
10        // Add the new genetator to the assembly
11        targetAssembly.Add(currentLod);
12    }
13    // Now perform optimisations
14    Optimise(targetAssembly)
15    // Now finalise the assembly on the disk
16    targetAssembly.Write(path);
```

In 4.2.1 we want to show a peculiar content generation problem, terrain generation. This involves a slightly different approach than the one applied to more general procedural content as shown earlier in this section.

4.2.1 Procedural terrain

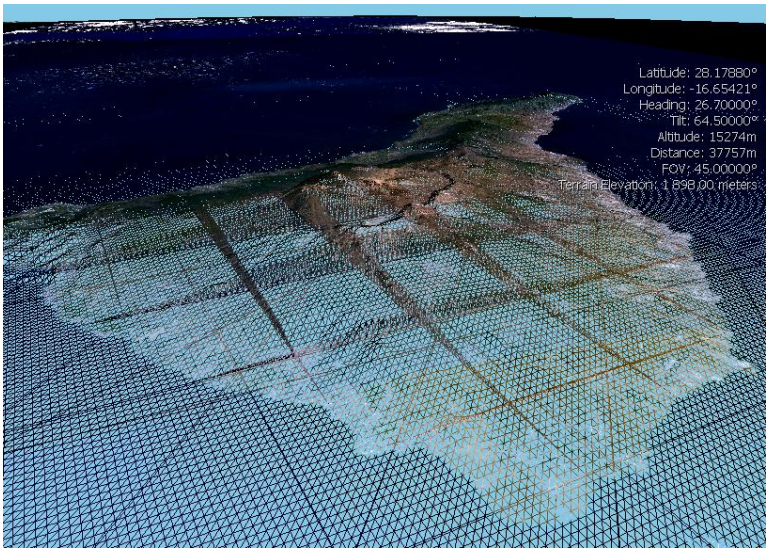
Terrain generation and rendering is a quite interesting scenario due to the fact that it must be always present in the scene and the data needed for high quality images can grow quite large. The basic idea is to use a grid tessellated as triangles, every vertex is raised according to a value read from an height map and finally a texture is used to apply the colour information to the mesh (see Figure20).

The rendering technique in interactive usage of 3D terrain is usually implemented having a fixed grid while the height map and the texture will *slide* as the camera moves around the scene, it is easy to understand that the height map sliding requires the Y component of every point to be updated and this means geometry creation. The geometry creation is avoided using a multi stream technique which is realised with the algorithm in Code 4.20.

Listing 4.20: Algorithm for terrain mesh generation

```
1  create the grid mesh with Y values set to 0
2  while the camera moves
3  {
4   slide the height map
5   create an height stream with Y values only
6  }
```

Figure 20: Wireframe rendering of 3D terrain

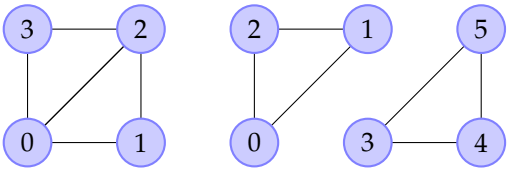


Then the rendering program will be adding the height stream values to the grid mesh Y values in the rendering pipeline. The code to compute the height values is executed in the logic update loop so that code can leverage the cores on a machine by partitioning the height map in slices. This kind of optimisation can be performed only at deploy time to properly compute the number of slices and threads to be used. The same optimisation can be applied to the LOD computation.

LOD for terrain can be computed without the use of extra memory to store them if the grid mesh is treated as an indexed primitive. Indexed primitives are composed by only the unique vertices of the mesh and the triangle tessellation is expressed as an array of indices. The quad in Figure 21 is made of two triangles and that means six vertices if represented as a plain mesh, if represented as an indexed primitive it will be made of four vertices and six indices.

We can then obtain LODs for grid mesh simply dropping indices and sharing the same grid mesh vertices among all the LODs (see Figure 22),

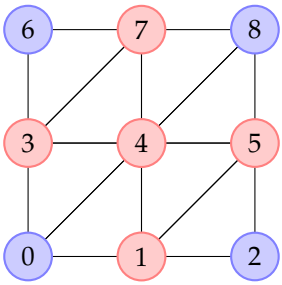
Figure 21: Triangulated Quad



Indexed Quad Plain Quad

that means that when the camera moves a new set of indices must be computed if a LOD definition is dropped. The LOD dropping computation is performed in the logic update step and therefore the same considerations we used for the height stream computation apply here.

Figure 22: Triangulated Quad with two LOD definition



For terrain the information about normal vectors is not stored in the vertex, that is for the grid mesh used to represent the terrain is never transformed (both position and normal are already in world space). We can therefore store the normal data in a normal map texture and access it later at rendering time (this help in reducing the number of unique vertices). The implications of such a choice are quite relevant:

- Normal maps are constant in size and shared amongst all LODs
- Lighting will be consistent while rendering different LODs

As we described since here we have data that can be loaded just once in the application and shared amongst multiple LODs:

- NormalMap
- HeightMap

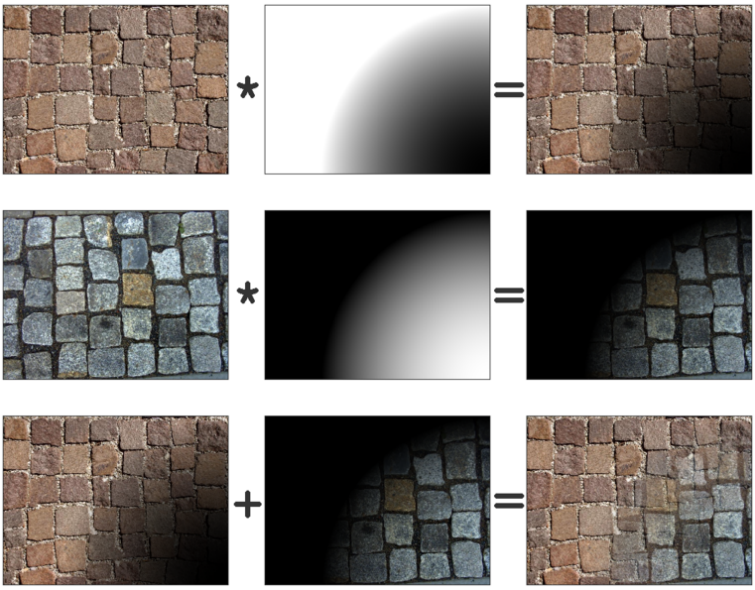
The main problem comes when dealing with the textures used for the colour information since they must have mipmap levels (Wil83) for sampling issues (ali) and this doubles the size of the file which is also further influenced by the coverage we want to achieve mapping image pixels to the *real* size of the terrain. What is more important to notice about terrain textures is that is possible to generate HD images using high quality piece of images to create them, so if we have a good set of grass, soil, stones, sands images we can reproduce with a good approximation any terrain type.

Using a texture splatting technique it is possible to create the land textures using a finite set of high quality samples and a composition description; in Figure 23 a linear combination of samples is used to create a new image using another image as coefficient for the combination. This approach is just to illustrate quickly the basic concept behind the solution but is not what we are aiming for as it requires a large portion of space (storage and bandwidth) to be represented.

The approach in (And07) shows an implementation for texture generation accelerated by the video card, this is beneficial as the newly created texture is already in video memory and thus it does not require any memory transfer and is immediately available for rendering of the terrain.

We target a scenario where content can be manipulated so we represent the final texture as a set of bi dimensional polygons with a set of generators id associated. Every generator id refers to one or more image source to provide variation in the final image. The polygons used to describe the image partitioning have coordinates (x, y) where $x \in [0, 1]$ and $y \in [0, 1]$ as they must be mapped inside the space defined by every

Figure 23: Texture splatting via linear combination



terrain fraction (assuming they are square shaped). The content creator now can change the look of a terrain editing the polygons defining the composition boundaries creating a data representation compacted and significantly smaller than the final texture.

On the program side the texture generation occurs only at terrain loading and we can have different strategies to face its computation:

- using sequential code one texture is computed at time
- using graphics accelerators to minimise the transfer and use k frames to create k terrain textures
- using a multicore approach (since there is no race condition) and build k textures at once and then load them into video memory

the best technique for any given scenario can be only decided at run time as the choice involves both the hardware configuration and user preferences. In this example the choice will impact any of three separate areas of the code generation. Table 6 is a quick recap of terrain generation process.

To decide which is the right combination of implementations is left to the application itself: once the user express the desired behaviour of the application the hardware configuration is used to compute the best implementation combination to achieve the users desire.

This makes the configuration step easier removing the need for the user to provide parameters for a lot of low level details about which features to enable and which to disabled. Another factor to be computed is the background activity of the system as many other services and applications can be running (for example: antivirus, email clients, download managers, etc.).

On a very busy system the application will prefer GPU based implementations[14] to achieve performance goals because such a choice will minimise the interleaved access to resource shared with other processes

[14]This technique is becoming a common practice and a standard even is OS software. Apple provides facilities through the OpenCL framework so that a GPU can help CPU even if task not graphics related.

(as cores and memory). This configuration adaptation is not just a deploy time operation as the software needs to collect enough information to profile the user ”behaviour” regarding the computational pressure on the hardware he uses.

Table 6: Terrain generation

		Implementations		
Steps	**Frequency**	**Sequential**	**Multicore**	**GPU**
Grid mesh generation	Once per application bootstrap	✓	✓	✓
Height stream generation	When camera movement requires terrain sliding	✓	✓	
Index buffer generation	When camera movement requires an LOD swap	✓	✓	
Texture generation	The first time the texture is required	✓	✓	✓

4.3 Extending game life with code generation

Games the such as World of Warcraf[15], Dark age of Camelot[16], Star Wars Galaxies[17] are based on quest completion. A quest is a set of tasks a player has to accomplish in order to gain a reward and progress in the game, every quest has a different complexity level and usually a gamer is able to pick up quests within a range of difficulty that includes the player's character level.

The following is the quest **Raene's Cleansing** from World of Warcraft:

> ❝ Asmexionor, a long time friend of mine is also aiding the Sentinels here in Ashenvale, but he has yet to return. He had leads on finding an item that he thought could slow the furbolg attacks on our people . . . a rod created by a now-dead, evil wizard. Before he left here, he mentioned seeking out a gem for the rod. He mentioned the gem possibly being hidden at the shrine in Lake Falathim at the base of the mountain to the west. The gem was being held there before it was overrun. Find my friend, Asmexionor, please. ❞

The quest is for level 18 characters and requires a player to find and interact with the body of a dead *NPC* (non player character) in Ashenvale (a location in the game world), on completion the player is rewarded with an amount of *XP* (experience points, used to increase character skills) and reputation point against Darnassus faction (reputation points unlocks features). The quest activation can be formalised as the code in Listing 4.21

Listing 4.21: Quest activation

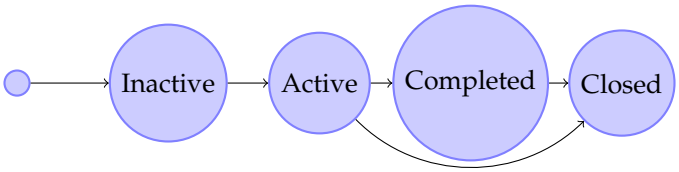
```
1 if (player.Level > 18 and !IsActive) {
2       SpawnCharacter(CharacterState.Dead,
3             new Charater("Teronis", Race.Elf),
4             GetLocation("Ashenvale"));
5       IsActive = true;
6 }
```

[15] http://wow/.com
[16] http://www.darkageofcamelot.com/
[17] http://starwarsgalaxies.station.sony.com/players/index.vm

At every action of the player the action and its context is used to check every active quests (a player can have more than one quest activated at the same time) for state changes due to a quest's goal match. A quest can be modelled as the state machine in Figure 24 where:

- a quest is created (spawned) and its state is set as **Inactive**
- when a player accept the quest it is moved into **Active** state
- the quest remains **Active** until every success condition are met or the player abort it
- if all conditions are met the quest moves into **Completed** state
- if a quest is abandoned then its state is changed in **Closed**
- when a quest state moves from **Completed** to **Closed** the reward is applied

Figure 24: Quest as ASM



Quests are fundamental objects as they are the way the game is expressed and the player makes progress within it. Once a quest is executed is no longer available to the player (but can be for other players) so a set of equivalent complexity quests must be available so that players taste can be met. The problem here is that quests are created by game designers and this has a serious impact on the budget required to create a world class *MMORPG*[18].

Reusing the same quest across all the players makes the world of the game static as actions have no real effect on the game world but only on

[18]MMORPG stands for Massive Multiplayer On line Role Play Game

the character progression. The players perception of a game can affect the life span of the game as players can get bored of always performing the same set of quests. Another aspect of the problem is that not all the quests can meet the player's preferences and are therefore avoided.

The set of actions that the player can perform in the game is a finite set of features in the game code and as the quest completion conditions are defined over the action set this set can be expressed with a finite number of expressions as well. The quest **Raene's Cleansing** action check can be expressed as the code in Listing4.22 and the quest object has a structure as shown in Listing 4.24, the return clause at 27 will be explained later.

Listing 4.22: Quest action check

```
if (!condition) {
 condition = (action.Player.Location == GetLocation("Ashenvale"))&&
 (action.Type == actions.Interact)&&
 action.Target.Id == GetId(Charater.Find("Teronis", Race.Elf));
}
```

Listing 4.23: Quest base class type

```

public class abstract Quest {
 public Status Status = Status.Created;

 public virtual void ActivateQuest(Player player){
  if(Status == Status.Inactive){
   if (MatchRequirements(player)){
    Spawn(player);
    Status = Status.Active;
       player.AddQuest(this);
 }}}

 public bool MatchRequirements(Player player) {...}

 public void CompleteQuest(Player player){
  if (Status == Status.Completed){ Reward(player); }
  Status = Status.Closed;
 }

 public virtual void CheckAction(Action action){
  if(Status != Status.Active) return;
 }

 public void CheckQuest(){
  Status = CheckConditions() ? Status.Completed : Status }
```

```
26
27  public virtual void Spawn(Player player) {}
28
29  public virtual bool CheckConditions() { return false; }
30
31  public virtual void Reward(Player player) {}
32
33 }
```

Listing 4.24: Quest type

```
1  [QuestConstraints(Level = 18)]
2 public class RaeneQuest01 : Quest {
3  public bool condition01 = false;
4
5  public void CheckCondition01(Action action){
6   condition01 = (action.Player.Location == GetLocation("Ashenvale"))
7   &&
8   (action.Type == actions.Interact)
9   &&
10  action.Target.Id == GetId(Charater.Find("Teronis", Race.Elf));
11  }
12
13  public void SpawnCondition01(Player player){
14   SpawnCharacter(CharacterState.Dead,
15    new Charater("Teronis", Race.Elf),
16    GetLocation("Ashenvale"));
17  }
18
19  public override void Spawn(Player player){
20   SpawnCondition01(player);
21  }
22
23  public override void CheckAction(Action action){
24   if(Status != Status.Active) return;
25   if (!condition01){
26    CheckCondition01(action);
27    if (condition01){ return; }
28  }}
29
30  public override bool CheckConditions(){
31   bool expr = condition01;
32   return expr;
33  }
34
35  public override void Reward(Player player){
36   //give the player the reward scaling the XP amount
37   //with the difference between the required level
38   //and the actual level, higher levels will get less XP
39   AddXP(player,1000 * ScaleOnLevel(18, player.Level));
```

```
40   AddFactionPoints(player, GetFaction("Darnassus"), 1000);
41 }
```

While **Raene's Cleansing** is a single objective quest there are many others with more than on goal to achieve, multiple goals can be combined with boolean **AND** and **OR** operators. The code in Listing 4.25 shows a quest with three objectives that must be satisfied all together in order to complete it.

Listing 4.25: Quest type with three objectives

```
1  [QuestConstraints(Level = 20)]
2  public class MultiObjectiveQuest : Quest {
3   public bool condition01 = false;
4   public bool condition02 = false;
5   public bool condition03 = false;
6
7   public void CheckCondition01(Action action){...}
8   public void CheckCondition02(Action action){...}
9   public void CheckCondition03(Action action){...}
10
11  public void SpawnCondition01(Player player){...}
12  public void SpawnCondition02(Player player){...}
13  public void SpawnCondition03(Player player){...}
14
15  public void RewardCondition01(Player player){...}
16  public void RewardCondition02(Player player){...}
17  public void RewardCondition03(Player player){...}
18
19  public override void Spawn(Player player){
20   SpawnCondition01(player);
21   SpawnCondition02(player);
22   SpawnCondition03(player);
23 }
24
25  public override void CheckAction(Action action){
26   if(Status != Status.Active) return;
27
28   if (!condition01){
29    CheckCondition01(action);
30    if (condition01){return;}
31   }
32
33   if (!condition02){
34    CheckCondition02(action);
35    if (condition02){ return; }
36   }
37   if (!condition03){
```

Objective 01

Objective 01

Objective 01

Objective 01

Objective 01

Objective 01

```
38    CheckCondition03(action);
39    if (condition03){ return; }
40   }
41  }
42
43  public override void CheckConditions(){
44   bool expr = condition01 && ●❶ Objective 01
45    condition02 &&
46    condition03;
47   return expr;
48  }
49
50  public override void Reward(Player player){
51   RewardCondition01(player);●❶ Objective 01
52   RewardCondition02(player);
53   RewardCondition03(player);
54  }
55 }
```

In Listing 4.25 we highlighted the portion of the quest being influenced by one of the three objectives, we do so to emphasise the fact that a quest can be formalised as a set of objectives where each objective is defined as spawn, check, and reward functions. With a compositional representation of quests it is possible to generate new quests using pre existent quest fragments as building blocks.

When creating a quest class out of an objective set (we will refer it as `QuestDesciptor` and it is defined as in Listing 4.26) we have to create a new type before progressing with method code manipulation. It is quite clear from Listing 4.25 and Listing 4.23 that for each objective a `boolean` field is introduced, the field is used for two goals:

- stop checking once a condition has been successfully matched
- model the quest state so it can be serialised and de-serialised

New methods are injected inside the quest class and they are created using the spawn, check, and reward functions of every objective. After the injection of the new elements the other virtual methods are rewritten:

- The `Spawn` method is rewritten injecting a call to each spawn method so that the spawn action of every objective is executed

- The `CheckAction` method is rewritten keeping the original code checking for the quest to be active, then for each objective the check function call is injected using a fragment containing such call and bound with a `TargetBinding` (see 3.2.1) to the correspondent boolean field; at line 27 in Listing 4.24 the `return` instruction inside the conditional branch is used so that an action is checked only once a positive match occurs no other condition is checked against the action
- The `Reward` method is rewritten injecting a call to the reward function of every objective
- The `CheckConditions` method is rewritten injecting the boolean expression over all the boolean fields caching each condition state

When the quest has as objective with multiple iterations like "Kill 25 smurfs" the objective is represented by 25 condition "Kill a smurf" where all of them are composed in an boolean expression combining all the the conditions with `AND` operators, we choose such approach for we wanted the state part of the quest object to be easy to generate.

Now that the structure of a quest is clear and we also have a way to create them we need a way to create new quests automatically in order to replace *consumed* quests with *equivalent* ones. It is important to notice in Listing 4.25 that we never performed inlining but kept objective as set of methods and calls to them have been injected. The choice is relevant as it enables *inspection* of the quests so that it is possible to obtain instances of `QuestDescriptor` from quest types.

Listing 4.26: Quest Descriptor

```
1 public class QuestDescriptor{
2  public QuestConstraints Constraints {get;set;}
3  public IEnumerable<QuestObjective> Objectives{get;}
4  public ObjectiveExpression {get;set;}
5 }
```

When a quest is completed it is possible to use the metadata portion of the type to collect other quests with similar constraints. Now for each quest type we obtain the `QuestDescriptor`; from a collection of

descriptors a new one is created by means of combining `QuestObjective` (defined as described in Listing 4.27) objects in a new instance of the `ObjectiveExpression`, with the new descriptor a quest type is synthesised. It is important to notice that every newly created quest will be available later as a new potential generator source for further quests.

Listing 4.27: Quest Objective

```
1 public class QuestObjective{
2  public Fragment Spawn {get;set;}
3  public Fragment Check {get;}
4  public Fragment Reward {get;set;}
5 }
```

The system for automatic quest generation is implemented through the following operators

$$\text{QuestGenerator} : \text{QuestDescriptor} \rightarrow \text{Quest}$$

$$\text{DescriptorExtractor} : \text{Quest} \rightarrow \text{QuestDescriptor}$$

$$\text{ConstratinsExtractor} : \text{Quest} \rightarrow \text{QuestConstraints}$$

$$\text{Recombinator} : \{d : \text{where } d \text{ is QuestDescriptor}\} \rightarrow \text{QuestDescriptor}$$

$$\text{GetSimilarQuests} : \text{QuestConstraints} \rightarrow \{q : \text{where } q \text{ is Quest}\}$$

At this point we need to define some metric to measure the player satisfaction when taking part in quests. For simplicity we will provide the user the ability to refuse quests, this will be an indicator of the players displeasure. When a quest Q_0 is completed we will follow these steps:

- $\text{ConstratinsExtractor}(Q_0)$ to obtain the quest constraints set C_0
- for every quest Q_i returned by $\text{GetSimilarQuests}(C_0)$ we extract the descriptors using $\text{DescriptorExtractor}(Q_i)$ creating a set of descriptors D
- a new quest Q_1 is generated by $\text{Recombinator}(D)$
- Q_0 is removed from the available quests set and Q_1 is inserted

When a quest is rejected by the player we follow the same steps described above but instead of starting with Q_0 we will select a quest Q_x where the intersection between the descriptor sets is lower then the set of Q_0, then will will proceed with a new quest generation step.

This framework which enables software to extend itself using metrics and domain specific generators is not representable with AOP approaches as while aspects can be used to assemble quests there is no way to generate aspects from quest types. In such a scenario quest designers would have to define all the possible objectives as aspects and then generate quests which is not realisable as matter of cost and effort.

Chapter 5

Conclusions

In this thesis we studied the class of programs capable of change their structure during execution. A run-time environment must include a query language, a support for homogeneous code transformations, and a incremental loading model. Strongly typed execution environments provide most of the required features, and the CodeBricks composition model adds the ability to mix existing code fragments. The query system introduced in chapter 3 was the missing tile of the puzzle: it is possible to indicate where code transformations should happen in the compiled program.

Using these tools we have been able to tackle two different problems in the Computer Games application domain: an optimisation problem, and a content generation problem. The former shows how program specialisation can be performed at run-time achieving performance improvements by adapting the program to data that is available only then. The latter focuses about adapting the program behaviour with the goal to the user behaviour by generating content during program lifetime.

At the beginning of this thesis we put a quote on the subject of alchemy which states:

> ❝ Alchemy is the science of breaking down the matter to rebuild it again, everything can be broken down in its alchemic components and then recomposed to create new things.

> But to gain something an equal value must be lost : that is the equivalent exchange rule. ❞

While it is obvious that alchemy is more of a philosophy than a science we wanted to use the alchemic process as a metaphor for our approach[1]. The technique we implemented is able to decompose software in blocks that can then be recombined to generate new features and to change the behaviour of the application. The design choices we took bring some limitations since our approach is not able to synthesise new types and new methods, however this is not a real limit as we can leverage the code generation relying on the performance of compiled code instead of having to deal with interpreter based designs or skeleton patterns or access to source code.

Another advantage of our technique over bytecode is that optimisations can be performed to provide the JIT with code that can be quickly analysed and optimised when translated into native executables since, as we have discussed, the JIT has a very constrained time to perform optimisations and compilation.

The query language can be extended to capture more sophisticated patterns, this is the core of our approach as it models the joint point of it. It could be interesting, for example, to be able to capture calls to methods which contain security checks when they are occurring in loops. Such queries can be used to perform expensive security and policy checks out of the loop and then inlining the method body of the match inside the loop.

The framework provides byte code manipulation facilities but it needs reflection and code emission capabilities to inspect and generate classes. With this support we were able to mimic the AOP approach for metaprogramming. It can however be used as a building block to realise other flavours such as Feature Oriented Programming and so on, bringing unbound multistage support to them.

The homogeneous nature of the method has been crucial in our work as it assists developers while designing the software, they have been able

[1] We use the code name Vitriol for the framework, which is a powerful solvent in alchemy.

to express with the same language and tools the transformations they wish to apply when particular situations are met. This is important to give the program the ability to adapt to a broad set of parameters that can depend on

- execution environment
- user model or feedback
- resource availability (as clouds, network, etc.)

The experiments reported in Chapter 4 cover different scenarios for the code transformation model we just presented in this thesis.

The experiment regarding optimisations and math libraries models a situation where the code manipulation is happening at deploy or load time; in that scenario the input needed for the manipulation is the code structure and the execution environment capabilities.

The procedural content and quest generation experiments require run time transformations since they depend not only on the hardware acceleration features of the execution environment; to perform the specialisations we need feedback and interaction with the user. In the procedural content experiment the content is created and modified at any time by the content creator, so it is unfeasible to precompute all the possible content for a deploy time code manipulation. For the quest generation experiment we use the user's preferences and rating to create new quests, this is a constraint for run time manipulation only.

Beside the video game scenario that we presented in this chapter there is another application for our framework in the domain of "Trust Worthy Experimentation". Microsoft[2], Google[3] and Amazon provide frameworks to run live experiments on web applications (EQvCD08) with the goal to help companies developing large software to achieve the best performance and experience for their users. These frameworks let the developer define variations of the application and metrics (Cli08) to measure the impact of each variation, then the experiment is run live

[2]Microsoft Experimentation Platform http://exp-platform.com/d
[3]Google Web Optimizer http://www.google.com/websiteoptimizer/

against real users. In this way the outcome of application changes are statistically significant and not related to the "HiPPO" (Highest Paid Person Opinion) so that errors can be avoided saving a lot of money for the companies.

The variation granularity goes from an entire page down to a single component or web service logic, multiple experiments can be run in parallel so that interaction between variation can be measured. A lot of Microsoft web sites are actually under constant experimentation using such a system due to the high number of users on a single web site (like MSN, the Microsoft Home page, etc) and the improvements every single team is testing.

The framework is able to assemble the application using the user as an input at the combination step which generally performed server side at the request time. Unfortunately this model is not usable on all web applications as platforms like Microsoft Silverlight and Adobe Flash cannot be assembled as html components. Microsoft Silverlight is based on the .NET CLR and using our framework it would be possible to extend the experimentation support to Silverlight and client applications.

As we mentioned in the introduction, software and applications are growing in size and deployment while unit testing and integration testing are still fundamental practices for software quality and they can not face failure at market. Failing for even a few days can compromise the revenue stream and company asset value even if the software is proven to be state of the art from engineering and technology point of view. The need for different tests to improve *user conversion*[4] is becoming a relevant feature of the development process and to be trusted the values must be realistic. All of the existing frameworks rely on statistical analysis of the data collected from real users.

Just few weeks of execution can produce statistically significant results and in some of the experiments the outcome has helped to avoid potentially large losses (millions of dollars in less then a year) due to unadvisable choices even if perfectly suitable from technological point of

[4]User conversion is the term used to identify the fact that a user has produced real revenue, this conversion depends on the nature of the software and is a per case definition.

view.

Another important scenario where our technique can be adopted is that of *power consumption aware applications* (GM02; CKM+01). In realtime application in scenarios like robotics, smart dust, mobile and embedded systems the impact of an algorithm on the battery life is extremely relevant. The battery level changes over time so the need for code changes at run time is even more important. It is possible to measure the power consumption of different versions of an algorithm and store this information in the metadata of optimisation strategies. Later, at run time, the program can decide to rewrite portions of its code adopting the best choice for battery life improvement.

Within the embedded systems scenarios deployment is as important as power consumption. The .NET framework has a large footprint so a fine grained deployment unit is more valuable than deploying assemblies. Using our approach is possible to refactor assemblies deploying only the used types with all the method bodies empty, then method bodies can be provided *on demand* so that the only code deployed on the board is just the one realising the features needed by the application.

Appendix A

Garbage Collection in the Video Game World

All commercial game engines realise their own memory management, garbage collection, and object pool. Since vide games are close to real time paradigm object creation is a sensitive task to perform since it will affect the frame rate but, on the other side, consoles come with a limited amount of memory compared to actual PCs' equipment. Leaving object pool strategies to game developers GC and allocation is handled by the .NET framework making the interaction between application execution and run-time support crucial. Before diving into GC details memory spaces and their relation to types must be clarified so that implementation choices done in software development can be understood.

A.1 Managed Heap and Stack memory spaces

.NET deals with type differently in memory allocation whether the type is a *reference* type (`class`) or a *value* type (`struct` and primitive types like `int`) using different rules for space and life time. Value Types are created on the stack and their life is related to their scope, such types do not need garbage collection activity and are passed by value when used in method calls. Since value types requires almost no run-time work for

their creation and disposal they are a great choice for performance but their by-value marshalling can be a problem in size for the stack. On the contrary reference types are allocated on the managed heap and their life time is controlled by the GC, when used as parameters their reference is passed without the need of copies (and therefore really stack friendly). When designing object models the choice between reference and value types is the hard core since the choice has implication on the memory usage, GC pressure, and marshalling behaviour when using PInvoke. Value types can be anyway allocated on the managed heap when

- part of a reference type
- boxed because of cast operation
- contained in collections

A.2 Generational Collection

A generational GC divides objects into generations and, on most cycles, will place only the objects of a subset of generations into the initial white (condemned) set. Furthermore, the run-time system maintains knowledge of when references cross generations by observing the creation and overwriting of references. When the garbage collector runs, it may be able to use this knowledge to prove that some objects in the initial white set are unreachable without having to traverse the entire reference tree. If the generational hypothesis holds, this results in much faster collection cycles while still reclaiming most unreachable objects. In order to implement this concept, many generational garbage collectors use separate memory regions for different ages of objects. When a region becomes full, those few objects that are referenced from older memory regions are promoted (copied) up to the next highest region, and the entire region can then be overwritten with fresh objects. This technique permits very fast incremental garbage collection, since the garbage collection of only one region at a time is all that is typically required. Generational garbage collection is a heuristic approach, and some unreachable objects may not

be reclaimed on each cycle. It may therefore occasionally be necessary to perform a full mark and sweep or copying garbage collection to reclaim all available space. In fact, run-time systems for modern programming languages (such as Java and the .NET Framework) usually use some hybrid of the various strategies that have been described thus far; for example, most collection cycles might look only at a few generations, while occasionally a mark-and-sweep is performed, and even more rarely a full copying is performed to combat fragmentation. The terms ”minor cycle” and ”major cycle” are sometimes used to describe these different levels of collector aggression. A generational garbage collector (also known as an ephemeral garbage collector) makes the following assumptions:

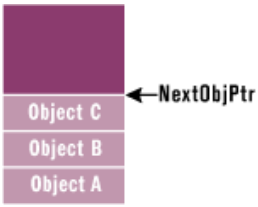
- The newer an object is, the shorter its lifetime will be.
- The older an object is, the longer its lifetime will be.
- Newer objects tend to have strong relationships to each other and are frequently accessed around the same time.
- Compacting a portion of the heap is faster than compacting the whole heap.

A.2.1 Allocation, Collection, and Finalisation in .NET

The **Microsoft**.NET common language run-time requires that all resources be allocated from the managed heap. When a process is initialised, the run-time reserves a contiguous region of address space that initially has no storage allocated for it. This address space region is the managed heap. The heap also maintains a pointer, which is referred as `NextObjPtr`. This pointer indicates where the next object is to be allocated within the heap. Initially, the `NextObjPtr` is set to the base address of the reserved address space region.

An application creates an object using the new operator. This operator first makes sure that the bytes required by the new object fit in the reserved region (committing storage if necessary). If the object fits, then `NextObjPtr` points to the object in the heap, this object's constructor is called, and the new operator returns the address of the object.

Figure 25: Managed Heap



At this point, `NextObjPtr` is incremented past the object so that it points to where the next object will be placed in the heap. Figure 25 shows a managed heap consisting of three objects

- A
- B
- C

The next object to be allocated will be placed where `NextObjPtr` points (immediately after object C).

In a C-run-time heap, allocating memory for an object requires walking though a linked list of data structures. Once a large enough block is found, that block has to be split, and pointers in the linked list nodes must be modified to keep everything intact. For the managed heap, allocating an object simply means adding a value to a pointer this is blazingly fast by comparison. In fact, allocating an object from the managed heap is nearly as fast as allocating memory from a thread's stack.

When an application calls the new operator to create an object, there may not be enough address space left in the region to allocate to the object. The heap detects this by adding the size of the new object to `NextObjPtr`. When it happens that `NextObjPtr` is beyond the end of the address space region, then the heap is full and a collection must be performed.

In reality, a collection occurs when *generation 0* is completely full. Separating objects into generations can allow the garbage collector to collect specific generations instead of collecting all objects in the managed heap.

Collection

The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application. If such objects exist, then the memory used by these objects can be reclaimed. (If no more memory is available for the heap, then the new operator throws an `OutOfMemory` exception.) The garbage collector is able to identify unused object through the *root set*.

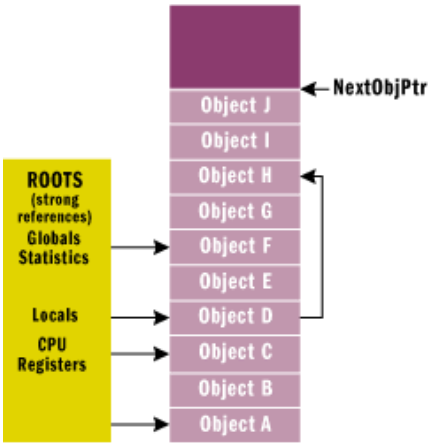
Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null. For example, all the global and static object pointers in an application are considered part of the application's roots. In addition, any local variable/parameter object pointers on a thread's stack are considered part of the application's roots. Finally, any CPU registers containing pointers to objects in the managed heap are also considered part of the application's roots. The list of active roots is maintained by the just-in-time (JIT) compiler and common language run-time, and is made accessible to the garbage collector's algorithm.

When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage (none of the application's roots refer to any objects in the heap). Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots. For example, the garbage collector may locate a global variable that points to an object in the heap.

Figure 26 shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All of these objects become part of the graph. When adding object D, the collector notices that this object refers to object H, and object H is also added to the graph. The collector continues to walk through all reachable objects recursively.

Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. This serves two purposes. First, it helps performance significantly

Figure 26: Allocated Objects in Heap

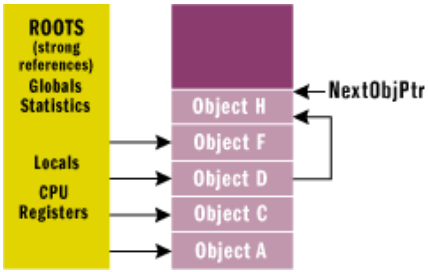


since it does not walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects.

Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space). The garbage collector then shifts the non-garbage objects down in memory (using the standard `memcpy` function), removing all of the gaps in the heap. Of course, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well. Figure 27 shows the managed heap after a collection.

After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been restored,

Figure 27: Managed Heap after Collection



the `NextObjPtr` is positioned just after the last non-garbage object. At this point, the new operation is tried again and the resource requested by the application is successfully created.

As we can see, a GC generates a significant performance hit, and this is the major downside of using a managed heap. However, keep in mind that GCs only occur when the heap is full and, until then, the managed heap is significantly faster than a C-run-time heap. The run-time's garbage collector also offers some optimisations that greatly improve the performance of garbage collection. In the common language run-time, the managed heap always knows the actual type of an object, and the metadata information is used to determine which members of an object refer to other objects.

Finalisation

The garbage collector offers an additional feature to handle resource cleaning: finalisation. Finalisation allows a resource to gracefully clean up after itself when it is being collected. By using finalisation, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory. Here is an oversimplification of what happens: when the garbage collector detects that an object is garbage, the garbage collector calls the object's finalisation method (if it exists) and then the object's memory is reclaimed. The code in Listing A.1 shows a simple type with the finalisa-

tion method implemented:

Listing A.1: Simple finalisation

```
1  public class BaseObj {
2     public BaseObj() {
3     }
4
5     public ~Finalize() {
6        // Perform resource cleanup code here ...
7        // Example: Close file /Close network connection
8        Console.WriteLine("In Finalise.");
9     }
10 }
```

Some time in the program execution, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a finaliser method and will call the method, causing "In Finalise" to appear in the console window and reclaiming the memory block used by this object.

When designing a type it is best to avoid implementing the finalisation and there are several reasons for this:

- Objects get promoted to older generations when finalisable, which increases memory pressure and prevents the object's memory from being collected when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as well
- objects with finalisation code take longer to allocate
- Forcing the garbage collector to execute a finalisation method can significantly impact on the application performance.
- Finalisable objects may refer to other (non-finalisable) objects, prolonging their lifetime unnecessarily.
- There is no control over when the finalisation will execute happen. The object may hold on to resources until the next time the garbage collector runs.

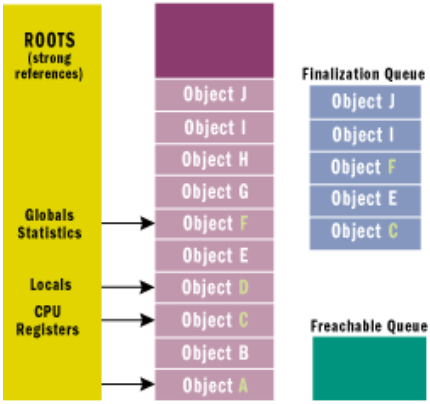
When an application terminates, some objects are still reachable and will not have their finalisation method called. This can happen if background threads are using the objects or if objects are created during application shutdown or `AppDomain` unloading. In addition, by default, finalisation methods are not called for unreachable objects when an application exits so that the application may terminate quickly. All operating system resources will be reclaimed, but any objects in the managed heap are not able to clean up gracefully. The default behaviour can be changed by calling the `System.GC` type's `RequestFinalizeOnShutdown` method. However, this strategy should be used carefully since calling it means that we are controlling a policy for the entire application. The run-time does not make any guarantees as to the order in which finalisation methods are called. For example, let us say there is an object that contains a pointer to an inner object. The garbage collector has detected that both objects are garbage. Furthermore, say that the inner object's finalisation method gets called first. Now, the outer object's finalisation method is allowed to access the inner object and call methods on it, but the inner object has been finalised and the results may be unpredictable. For this reason, it is strongly recommended that finalisation methods not access any inner, member objects.

If a type must implement a finalisation method, then we should have the code executed as quickly as possible. Avoid all actions that would block the finalisation method, including any thread synchronisation operations. Also, if any exception occurs while executing finalisation method without being handled, the system just assumes that the finalisation method returned and continues calling other objects' finalisation methods.

When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a finalisation method, then a pointer to the object is placed on the finalisation queue. The finalisation queue is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its finalisation method called before the object's memory can be reclaimed. Figure 28 shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are

not. When objects C, E, F, I, and J were created, the system detected that these objects had finalisation methods and pointers to these objects were added to the finalisation queue.

Figure 28: Heap with Many Objectsn

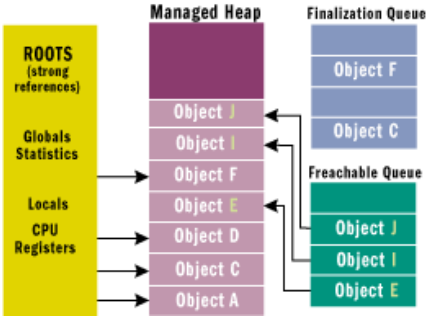


When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalisation queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalisation queue and appended to the *freachable* queue (pronounced "F-reachable"). The freachable queue is another internal data structure controlled by the garbage collector. Each pointer in the freachable queue identifies an object that is ready to have its finalisation method called.

After the collection, the managed heap looks like Figure 29. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a finalisation method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their finalisation method has not been called yet.

There is a special run-time thread dedicated to execute finalisation. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's finalisation method. Be-

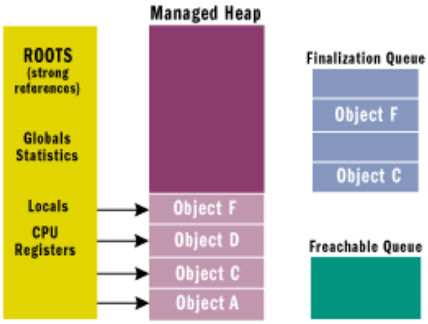
Figure 29: Managed Heap after Garbage Collection



cause of this, you should not execute any code in a finalisation method that makes any assumption about the thread that is executing the code. For example, avoid accessing thread local storage in the Finalize method. The interaction of the finalisation queue and the freachable queue is quite fascinating. First, let me tell you how the freachable queue got its name. The f is obvious and stands for finalisation; every entry in the freachable queue should have its finalisation method called. The "reachable" part of the name means that the objects are reachable. To put it another way, the freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage. In short, when an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalisation queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special run-time thread empties the freachable queue, executing each object's finalisation method.

The next time the garbage collector is invoked, it sees that the finalised objects are truly garbage, since the application's roots do not point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed. The important thing to un-

Figure 30: Managed Heap after Second Garbage Collection



derstand here is that two GCs are required to reclaim memory used by objects that require finalisation. In reality, more than two collections may be necessary since the objects could get promoted to an older generation. Figure 30 shows what the managed heap looks like after the second GC.

A.2.2 Generation in .NET implementation

The **Microsoft**.NET framework uses a generational garbage collector where the generations are three. When initialised, the managed heap contains no objects. Objects added to the heap are said to be in generation 0, as you can see in Figure 31. Stated simply, objects in generation 0 are young objects that have never been examined by the garbage collector.

Now, if more objects are added to the heap, the heap fills and a garbage collection must occur. When the garbage collector analyses the heap, it builds the graph of garbage (shown here in purple) and non-garbage objects. Any objects that survive the collection are compacted into the left-most portion of the heap. These objects have survived a collection, are older, and are now considered to be in generation 1 (Figure 32).

As even more objects are added to the heap, these new, young objects are placed in generation 0. If generation 0 fills again, a GC is performed. This time, all objects in generation 1 that survive are compacted and considered to be in generation 2 (see Figure 33). All survivors in generation 0 are now compacted and considered to be in generation 1. Generation 0

Figure 31: GC with only Gen0

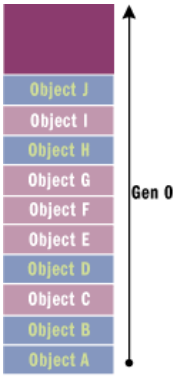
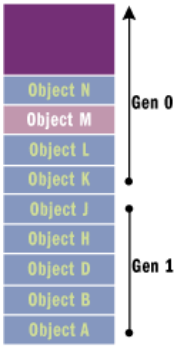
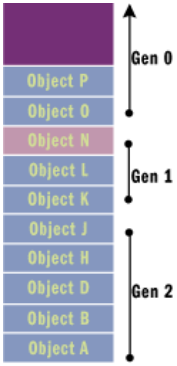


Figure 32: GC with Gen0 and Gen1



currently contains no objects, but all new objects will go into generation 0.

Figure 33: GC with Gen0, Gen1 and Gen2



Currently, generation 2 is the highest generation supported by the run-time's garbage collector. When future collections occur, any surviving objects currently in generation 2 simply stay in generation 2. There is another portion of memory which is called *Large Object Heap* (LOH) where big object, according to heuristics, are stored. While the generations are compacted during collection the LOH space is not compacted and therefore we can have fragmentation issues when dealing with this portion of memory.

A.3 Garbage Collection impact in Game application

.NET is becoming a very interesting platform to be used in game development. **Microsoft** is promoting the adoption of XNA Game Studio[1] and Mono is helping the diffusion covering other platform not covered by .NET. Memory management and garbage collection have a terrific im-

[1]XNA Developer Center http://msdn.microsoft.com/en-us/xna/default.aspx

pact on the productivity but this come to a cost and the effects they produce at run-time must be considered carefully while designing the software. When collection happens on the Gen2 section, or when objects are promoted and then destroyed[2] the application execution will be paused. While Gen0 and Gen1 are per-thread generations Gen2 and the Large Object heap are shared, this implies that when a collection is performed on shared generations all the threads must be paused.

The GC process has a priority that increases as the available system memory goes down, this means that allocation strategies are important. The support for finalisable objects should be used with caution since there is no guarantee on when they will finalised, also the finalisation thread will have to execute all the finalisation methods before returning control to application threads.

Buffers are things to be carefully inspected since big arrays will be allocated on the Large Object Heap memory. This portion of memory is never compacted and this means that a wrong allocation pattern can lead to fragmentation and thus to out of memory exceptions. The problem is even more frequent when the buffers are made by large reference types or value types. In the case of Big value types they will be allocate on the large object heap and this can accentuate the fragmentation problem. With value types we will be facing another issue since the size in byte of the storage for the array will be the number of elements times the size in bytes of the type.

For more details about .NET Garbage Collector performance hints see http://msdn.microsoft.com/en-us/library/ms973837.aspx.

[2]This problem is known as *almost long living object*.

Appendix B

Software Engineering and Architecture in Game development

A video game application is usually built with code that can be partitioned as in Figure 34.

Figure 34: Typical Game code structure

At the moment most of the game products are based on C++ language, VM based products are jumping in the market these days thanks

to **Microsoft**.NET and Novell Mono[1]

The "Specific Game Code" section contains code that is specific to a game title, here is where code realise the specific game design. Usually this portion of the code is not completely reusable unless developing sequels or extremely similar games. At this level the hardware machine is usually completely hidden since the level beneath should be providing cross platform implementations. Machine independent game code is a main goal for multi platform developer since it will reduce the development costs for n target platforms to the cost of almost 1.5 platforms.

Game Middlewares are libraries that realise physic simulation, AI, high level networking (chat system for example), quite famous frameworks are

- Havok[2]
- PhysX [3]
- Digital Molecular Matter [4]
- AiImplant [5]
- Euphoria [6]

Middleware are reusable software components and even if a lot of them are "off the shelf" companies can use them to create their own specific middleware. Sometimes proprietary components are developed for internal use and later transformed into stand alone products. The frameworks we cited are not tied to any specific game engine and, beside simulation code, they can bring specific implementation in order to leverage the hardware. PhysX for example is capable to use GPU to increase

[1]Mono platform is adopted by game industry thanks to the Unity game engine, at the moment platform supported are Pc, Mac, iPhone and Nintendo Wii

[2]Havok http://www.havok.com

[3]Ageia Physx by Nvidia http://www.nvidia.com/object/physx_new.html

[4]DMM softwarehttp://www.pixeluxentertainment.com/

[5]AiImplant http://www.presagis.com/products/simulation/aiimplant/

[6]Euphoria engine http://www.naturalmotion.com/

performance (and accuracy since it is a physic engine) while Havok is oriented on multi core architectures[7].

Generally the goal of Game middlewares is to provide functionality with high level abstractions, cross platform capabilities, execution model and high performance. The co existence of middleware and game engine in the same application is not easy since the game engine will be already hiding the hardware details, that is why game engine developers and middleware developers provide "certification" about the interoperability between products and adaptor to combine them.

Amongst game engines the most popular and definitely state of the art are:

- Unreal Engine [8]
- Gamebryo [9]
- Cryengine [10]
- Torque [11]
- Unity [12]

A game engine provides all the abstractions needed to interact with the hardware machine and so memory, disk access, graphic drivers, input devices, processors, cores and threads are remodelled and hidden in its code. A game engine is a very complex framework since it is providing features and technologies crucial to the development process.

Since most of the engines are based on C++ they usually provides support for features not present in C++ run-time as:

[7]Havok company works in collaboration with Intel and their synergy let them realise state of the art performance in physic simulation without the need of additional accelerators.

[8]Epic Games http://www.unrealtechnology.com/

[9]Emergent Game Technology http://www.emergent.net/en/Products/Gamebryo/

[10]Crytek http://www.crytek.com/technology/cryengine-3/specifications/

[11]Garage Games http://www.garagegames.com/

[12]Over The Edge http://unity3d.com/

- custom allocators for object creation
- object pools
- garbage collection
- reflection support
- object serialisation
- BCL for math, algebra and geometry

The features above combined with HW abstractions are generally the common set of functionality provided by Virtual Machines the likes of Java and .NET. Every engine provides a type system root that must be used in order to leverage the run time features of the engine.

On top of the mentioned components every engine realises execution model, plug in interface (used to connect middlewares), rendering abstractions, game logic object support (like checkpoints, missions, levels, quests, etc.). Engines comes with scripting languages that is implemented using the reflection capabilities of the engine and its type system[13].

Because of massive use of C++ templates and the size of the code base engines are distributed as source code, developers tailor them in the game development process investing a huge amount of resources since every time the engine is modified a full build of game and tools must be performed. Tools provided with the engine are built on the engine code base as well, this help the process since the object model is shared between the tools and the final game giving a huge boost to the production pipeline.

Efficiency in memory, execution and precision are crucial in the foundational components of the game engine such as mathematic libraries since they will be used to implement every other piece of code in the engine itself, middlewares, and games. Wrong design and bugs in foundational libraries will affect every aspect of the application code.

[13].NET provides scripting capabilities thanks to the DLR, a framework for implementing dynamic languages that leverage .NET reflection and light weight code generation at runtime. The DLR offers a great flexibility since it is language independent.

Appendix C

Rendering in Real Time Application

One of the most critical aspects of the game main loop is represented by the rendering step. Every cycle the current frame must be computed from the internal representation of the state of the world held, in computer games featuring 3D graphics, into appropriate data structures. An essential data structure widely adopted by computer games is the scene graph (Ebe06; scg), a graph where nodes represent the state of the various elements of the world and arcs the relations among them. Rendering is the process of visiting the data contained in the scene graph. Since the scene graph holds the whole state of the world, it may also contain objects that are not visible (either because out of the field of view or occluded by other objects) given the current position of the camera.

An obvious optimisation of this process would be to avoid the elaboration of these invisible objects, which can be significant in number if the world is large enough. The Z test (depth buffer test) is used to decide which pixel is to be displayed on the final picture and it uses the distance from of a point from the camera in the 3D space to decide the closest visible fragment to be shown. To have a depth sorted version of the scene can (potentially) avoid some drawing to happen because of a wrong order in rendering (zbu) and view frustum test can avoid sorting

something out of sight.

Rendering with a wrong order make the pixels to be drawn more than once, thus impacting on the fill rate, in this example the marked region is drawn three times because the cubes passes the depth test and so the new pixel is to be drawn in front of the previous.

Given this information we can think about a rendering algorithm as follows:

```
1 foreach (mesh in CurrentScene){
2       if (IsVisible(mesh))
3             CameraSortedList.Add(mesh);
4 }
5 foreach (mesh in CameraSortedList)
6       mesh.Render();
```

The approach is an improvement is the time consumed for the boost and the rendering of the selected objects is less than the time the execution of a complete scene rendering would consume. The culling algorithms employed by the 3D pipeline would avoid the rendering of unnecessary primitives, but there would be a greater overhead due to the requests of unneeded graphic primitives. Moreover at the application level there is more semantic information that can be used to get rid of entire objects due to considerations possible only at this level. Often semitransparent objects have to be rendered and we must extend the basic structure of the rendering to take into account the fact that several objects are viewable along the same direction because of the alpha blending.

A simple approach to address the problem of transparent objects consists in having a distance sorted list where the first element is the further and the last one the closest to the camera. More advanced techniques adopt depth peeling strategies, since sorting object is not enough for accurate rendering; a complete and correct solution would involve sorting all the triangles (or even small tessellation element (CCC87)) defining objects, though it is not feasible in real time because computationally too expensive. The algorithm that deals with transparent objects is extended as follows:

```
1 foreach (mesh in CurrentScene)
2 if (IsVisible(mesh))
```

```
3       if (mesh.Alpha)
4             CameraSortedList_Transparent.Add(mesh);
5       else
6             CameraSortedList_Solid.Add(mesh);
7
8  foreach (mesh in CameraSortedList_Solid)
9       mesh.Render();
10 foreach (mesh in CameraSortedList_ Transparent)
11      mesh.Render();
```

The data structure `CurrentScene` holds the `SceneGraph`, which is, in its simplest form a collection of every object of the game (3D models). In the algorithm we assumed that the initialisation of the `CameraSortedList_Transparent` and the `CameraSortedList_Solid` collections are responsible for deciding the sorting strategy, and that it is performed during the execution of the Add method. Usually the scene graph is adapted to the specific game logic to obtain maximum performance since it is a very large data structure and it affects the performance in accessing data. A number of techniques to treat the geometric and topological description of the game world are used, such as quadtree (Ebe06), octree (Ebe06), sector, c-bdam (GMC+06b), p-bdam (CGG+03). A common practice is to separate the collection filling from the drawing step; this is because collections are updated after the simulation code and the drawing step at time t can be performed in parallel with the simulation step at time t+1.

Just to give some performance hint, even a simple game usually has thousands of object inside the scene graph and even in the best case the opacity test condition must be evaluated, and all of these tests happen at least 30 time per second (though in reality it should not be less than 60 frame per second in a high quality commercial game). Commercial game engines are commonly used for developing products (Swe06); the most important features of these libraries are the `SceneGraph` and the main loop (which involves the update step and the rendering one).

While AOP and FOP can be used for defining most of the piece of the game (such as the `SceneGraph`) these approaches do not work as well for the rendering strategy. Although the assumption that the minimal acceptable implementation of a `SceneGraph` is a simple collection, this

cannot be assumed for the rendering strategy where the basic assumption includes the complete scenario and the code has to deal with both solid and transparent objects. The ability to discover that the dataset of the game contains only one kind of objects would allow us to remove both the unnecessary code and data structures. This would lead us to some significant results:

- No conditional tests must be performed to discover the object type, thus conditional branches will be avoided
- The binary code for the method is the minimal set of instructions needed reducing impact on the memory hierarchy and instruction cache miss
- The working set of the method is reduced to the minimum
- Often those data structures are used inside critical section, the fewest must be touched the shortest they last

In order to achieve these results game engine ships both in binary and source code, so that developers can change the engine to fit their needs. The usual approach followed is through `#ifdef` preprocessor directive to compile an ad hoc version of the engine. The data driven optimisation is so human performed at the game design step. In such scenario there is the need for a technique to describe code evolution even by removing unused features and aspects from an existing program.

Having those facilities and the ability to perform the transformation at run time would lead to both performance boost and easy maintenance of the final program, without the need to change the source code of the engine. Modifications to the game engine often involve a full rebuild of the whole system that can have a significant impact to the development process, For instance a Play Station Portable game compilation phase takes around 40 minutes to complete on a dual core architecture with 2 GB of memory, time that a developer needs to face every debugging session.

In the rest of this paper we will consider two examples from game rendering and we will discuss how these problems can be addressed us-

ing state of the art generative programming techniques. Our goal is to show that existing approaches are not capable of describing and handling these examples without a significant amount of workarounds to describe them, with a significant overhead.

The two problems are related to the configuration of a data structures used to describe the *Potentially Visible Set* (PVS) (pvs) and the scene graph. Here we are interested in controlling the structure of a type as well as the code responsible for inserting objects into the PVS. Let us assume that we start with a basic implementation of the scene graph, which is just a class defining the interface needed to build the PVS given a camera:

```
1 public class SceneGraph{
2       public Pair<PVS, PVS> ComputePVS(camera c) {}
3 }
```

A typical implementation of the `ComputePVS` method is the following:

```
1  Pair<PVS, PVS> ComputePVS(camera c) {
2        PVS transparent = new PVS(), solid = new PVS();
3        foreach (mesh in CurrentScene)
4              if (IsVisible(mesh,c))
5                    if (mesh.Alpha)
6                          transparent.Add(mesh);
7                    else
8                   solid.Add(mesh);
9        return new Pair<PVS, PVS>(solid, transparent);
10 }
```

The PVS data structure collects meshes and its definition is the following:

```
1 public class PVS {
2   void Add(Mesh m) {
3   }
4 }
```

The data structure used by the PVS to store the Mesh objects may vary depending on the kind of objects inserted using the `Add` method. This method is responsible for implementing the insertion policy, for instance the solid objects must be sorted from the nearest to the farthest, whereas for transparent objects the sorting order is the reverse.

Insertion strategy is not the only dimension of the problem: different organisations may suit better for different objects and possibly with a non homogeneous interface (the same problem arises for the scene graph depending on the kind of nodes and the semantics intended for the collection as discussed in (Ebe06)). Using generative programming techniques it is possible to synthesise the appropriated version of PVS for a given kind of objects. It would desirable to adapt PVS at run-time avoiding costs of interpretation because of the large number of objects manipulated by the set.

Ideally we would like to be able to first define the internal structure of scene graph and of the PVS with respect to the actual type of objects that must be stored in it. Once the structure has been decided it is also necessary to define the code to initialise and manage data stored within these structures. Since both scene-graph and PVS are subject to change during execution we would like to be able to adapt these components during run-time, with the goal of maximising performance. This is a form of program specialisation that takes place at run-time, and it can be convenient because the potential overhead due to code generation is amortised by the intensive work performed by the data structures.

Appendix D

Experiments Code

Listing D.1: IL code for normal computation

```
1       .maxstack 2
2       ...
3    IL_003f: ldloc.1
4    IL_0040: ldloc.0
5    IL_0041: call class Point3Df Vector3Df::op_Subtraction(
6             class Vector3Df,
7             class Vector3Df)
8    IL_0046: stloc.3
9    IL_0047: ldloc.2
10   IL_0048: ldloc.0
11   IL_0049: call class Point3Df Vector3Df::op_Subtraction(
12            class Vector3Df,
13            class Vector3Df)
14   IL_004e: stloc.s V_4
15   IL_0050: ldloc.3
16   IL_0051: ldloc.s V_4
17   IL_0053: call class Vector3Df Vector3Df::cross(
18            class Vector3Df,
19            class Vector3Df)
20   IL_0058: stloc.s V_5
21   IL_005a: ldloc.s V_5
22   IL_005c: callvirt instance void Vector3Df::Normalise()
23      ...
```

Listing D.2: IL code for normal computation inlined

```
1       .maxstack 3
2       ...
```

```
3         IL_003f: ldloc.1
4         IL_0040: callvirt instance float32 Point3Df::get_X()
5         IL_0045: ldloc.0
6         IL_0046: callvirt instance float32 Point3Df::get_X()
7         IL_004b: sub
8         IL_004c: stloc.3
9         IL_004d: ldloc.1
10        IL_004e: callvirt instance float32 Point3Df::get_Y()
11        IL_0053: ldloc.0
12        IL_0054: callvirt instance float32 Point3Df::get_Y()
13        IL_0059: sub
14        IL_005a: stloc.s V_4
15        IL_005c: ldloc.1
16        IL_005d: callvirt instance float32 Point3Df::get_Z()
17        IL_0062: ldloc.0
18        IL_0063: callvirt instance float32 Point3Df::get_Z()
19        IL_0068: sub
20        IL_0069: stloc.s V_5
21        IL_006b: ldloc.2
22        IL_006c: callvirt instance float32 Point3Df::get_X()
23        IL_0071: ldloc.0
24        IL_0072: callvirt instance float32 Point3Df::get_X()
25        IL_0077: sub
26        IL_0078: stloc.s V_6
27        IL_007a: ldloc.2
28        IL_007b: callvirt instance float32 Point3Df::get_Y()
29        IL_0080: ldloc.0
30        IL_0081: callvirt instance float32 Point3Df::get_Y()
31        IL_0086: sub
32        IL_0087: stloc.s V_7
33        IL_0089: ldloc.2
34        IL_008a: callvirt instance float32 Point3Df::get_Z()
35        IL_008f: ldloc.0
36        IL_0090: callvirt instance float32 Point3Df::get_Z()
37        IL_0095: sub
38        IL_0096: stloc.s V_8
39        IL_0098: ldloc.s V_4
40        IL_009a: ldloc.s V_8
41        IL_009c: mul
42        IL_009d: ldloc.s V_5
43        IL_009f: ldloc.s V_7
44        IL_00a1: mul
45        IL_00a2: sub
46        IL_00a3: stloc.s V_9
47        IL_00a5: ldloc.s V_5
48        IL_00a7: ldloc.s V_6
49        IL_00a9: mul
50        IL_00aa: ldloc.3
51        IL_00ab: ldloc.s V_8
52        IL_00ad: mul
```

```
53        IL_00ae: sub
54        IL_00af: stloc.s V_10
55        IL_00b1: ldloc.3
56        IL_00b2: ldloc.s V_7
57        IL_00b4: mul
58        IL_00b5: ldloc.s V_4
59        IL_00b7: ldloc.s V_6
60        IL_00b9: mul
61        IL_00ba: sub
62        IL_00bb: stloc.s V_11
63        IL_00bd: ldloc.s V_9
64        IL_00bf: ldloc.s V_10
65        IL_00c1: ldloc.s V_11
66        IL_00c3: call float32 [mscorlib]System.Math::Max(
67           float32,
68           float32)
69        IL_00c8: call float32 [mscorlib]System.Math::Max(
70           float32,
71           float32)
72        IL_00cd: stloc.s V_12
73        IL_00cf: ldloc.s V_9
74        IL_00d1: ldloc.s V_12
75        IL_00d3: div
76        IL_00d4: stloc.s V_9
77        IL_00d6: ldloc.s V_10
78        IL_00d8: ldloc.s V_12
79        IL_00da: div
80        IL_00db: stloc.s V_10
81        IL_00dd: ldloc.s V_11
82        IL_00df: ldloc.s V_12
83        IL_00e1: div
84        IL_00e2: stloc.s V_11
85        IL_00e4: ldloc.s V_9
86        IL_00e6: ldloc.s V_9
87        IL_00e8: mul
88        IL_00e9: ldloc.s V_10
89        IL_00eb: ldloc.s V_10
90        IL_00ed: mul
91        IL_00ee: add
92        IL_00ef: ldloc.s V_11
93        IL_00f1: ldloc.s V_11
94        IL_00f3: mul
95        IL_00f4: add
96        IL_00f5: conv.r8
97        IL_00f6: call float64 [mscorlib]System.Math::Sqrt(float64)
98        IL_00fb: conv.r4
99        IL_00fc: stloc.s V_12
100       IL_00fe: ldloc.s V_9
101       IL_0100: ldloc.s V_12
102       IL_0102: div
```

```
103         IL_0103: stloc.s V_9
104         IL_0105: ldloc.s V_10
105         IL_0107: ldloc.s V_12
106         IL_0109: div
107         IL_010a: stloc.s V_10
108         IL_010c: ldloc.s V_11
109         IL_010e: ldloc.s V_12
110         IL_0110: div
111         IL_0111: stloc.s V_11
112         IL_0113: ldloc.s V_9
113         IL_0115: ldloc.s V_10
114         IL_0117: ldloc.s V_11
115         IL_0119: newobj instance void Vector3Df::.ctor(
116            float32,
117            float32,
118            float32)
119         ...
```

Listing D.3: Matrix multiplication

```
1 Matrix3Df m = m1 * m2 * m3 * m4 * m5 * m6;
```

Listing D.4: Matrix multiplication specialised

```
1  Matrixf[] ms = new Matrixf[]{m1,m2,m3,m4,m5, m6};
2  Matrixf res = new Matrixf();
3
4  for (int i = 0; i < ms.Length; i++)
5  {
6     float[,] m = ms[i].m_matrix;
7     float c00 = m[0, 0];
8     float c01 = m[0, 1];
9     float c02 = m[0, 2];
10    float c03 = m[0, 3];
11
12    float c10 = m[1, 0];
13    float c11 = m[1, 1];
14    float c12 = m[1, 2];
15    float c13 = m[1, 3];
16
17    float c20 = m[2, 0];
18    float c21 = m[2, 1];
19    float c22 = m[2, 2];
20    float c23 = m[2, 3];
21
22    float c30 = m[3, 0];
23    float c31 = m[3, 1];
24    float c32 = m[3, 2];
25    float c33 = m[3, 3];
```

```
26
27      for (int r = 0; r < 4; r++)
28      {
29          float r0 = res.m_matrix[r, 0];
30          float r1 = res.m_matrix[r, 1];
31          float r2 = res.m_matrix[r, 2];
32          float r3 = res.m_matrix[r, 3];
33
34          res.m_matrix[r, 0] = (r0 * c00) + (r1 * c10)
35                                  + (r2 * c20) + (r3 * c30);
36          res.m_matrix[r, 1] = (r0 * c01) + (r1 * c11)
37                                  + (r2 * c21) + (r3 * c31);
38          res.m_matrix[r, 2] = (r0 * c02) + (r1 * c12)
39                                  + (r2 * c22) + (r3 * c32);
40          res.m_matrix[r, 3] = (r0 * c03) + (r1 * c13)
41                                  + (r2 * c23) + (r3 * c33);
42      }
43
44  }
```

Listing D.5: Baricentre Computation

```
1  Point3Df ComputeBaricentre(Point3Df[] vertices)
2  {
3        float n = vertices.Lenght;
4        float x,y,z;
5        foreach(var v in vertices)
6        {
7              x+= v.X / n;
8              y+= v.Y / n;
9              z+= v.Z / n;
10       }
11       return new Point3Df(x,y,z);
12 }
```

References

[ACC04] Giuseppe Attardi, Antonio Cisternino, and Diego Colombo. Cil + metadata > executable program. *Journal of Object Technology*, 3(2):19–26, 2004. 55, 56

[ACK03] Giuseppe Attardi, Antonio Cisternino, and Andrew Kennedy. Codebricks: code fragments as building blocks. *SIGPLAN Not.*, 38(10):66–74, 2003. 3, 50, 55, 65, 79, 84

[ali] Antialiasing and mipmap. `http://en.wikipedia.org/wiki/Anti-aliasing`. 130

[And07] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 38–58, New York, NY, USA, 2007. ACM. 130

[Ang06] Ed Angel. *Interactive Computer Graphics: A top-down approach with OpenGL Foruth Edition*. Addison Wesley, 2006. 24

[aop07] *ECOOP 2007 – Object-Oriented Programming*, volume 4609/2007 of *Lecture Notes in Computer Science*, chapter A Machine Model for Aspect-Oriented Programming, pages 501–524. Springer Berlin, August 2007. 48, 49

[asp] Aspectj, aspect-oriented programming for java. `http://aspectj.org/`. 27, 28, 66

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. 2

[BBR+04] Frederic Besson, Tomasz Blanc, Inria Rocquencourt, Cedric Fournet, and Andrew D. Gordon. From stack inspection to access control: A security analysis for libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004*, pages 61–75. IEEE Computer Society, 2004. 65, 79

[BD05] Francesco Balena and Giuseppe Dimauro. *Practical Guidelines and Best Practices for Microsoft® Visual Basic and Visual C# Developers*. Microsoft Press, 2005. 91

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM. 25

[bEGHJV94] by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison Wesley, 1994. 35

[bli] Blitz++ library for scientific computing. `http://www.oonumerics.org/blitz/`. 4

[boo] Boost c++ libraries. `http://www.boost.org/`. 4

[bTLY99] by Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, Second Edition*. Java series. Prentice Hall, 1999. 12

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM. 58

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM. 58

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, 1987. 168

[CCC05] Walter Cazzola, Antonio Cisternino, and Diego Colombo. [a]c#: C# with a customizable code annotation mechanism. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1264–1268, New York, NY, USA, 2005. ACM. 57, 80, 120

[CCH09] Walter Cazzola, Diego Colombo, and Duncan Harrison. Aspect-oriented procedural content engineering for game design. *SAC*, 2009. 120, 125

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, Reading, MA, US, June 2000. 23, 24, 27, 51, 61, 71, 102

[cec] Cecil cli file reader. http://www.mono-project.com/Cecil. 59

[CGG+03] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society. 169

[Cis] Clifilerw library. http://clifilerw.codeplex.com/. 20, 59, 62

[Cis03] Antonio Cisternino. *Multi-Stage and Meta-Programming Support in Strongly Typed Execution Engines*. PhD thesis, University of Pisa, 2003. 21, 51, 66

[CKM+01] L. N. Chakrapani, P. Korkmaz, V. J. Mooney, III, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: what can a poor compiler do? In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 176–180, New York, NY, USA, 2001. ACM. 147

[Cli08] Brian Clifton. *Advanced Web Metrics with Google Analytics*. Sybex, 2008. 145

[CMS05] Bruno Cabral, Paulo Marques, and Luís Silva. RAIL: code instrumentation for .NET. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, Santa Fe, New Mexico, 2005. 59

[CN96] Charles Consel and François Noël. A general approach for runtime specialization and its application to c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, New York, NY, USA, 1996. ACM. 25

[Cou97] Patrick Cousot. Program analysis: the abstract interpretation perspective. *SIGPLAN Not.*, 32(1):73–76, 1997. 58

[CP07a] Walter Cazzola and Sonia Pini. Aop vs software evolution: a score in favor of the blueprint. In *RAM-SE*, pages 81–91, 2007. 43, 55

[CP07b] Walter Cazzola and Sonia Pini. On the footprints of join points: The blueprint approach. *Journal of Object Technology*, 2007. 43, 55

[CPI88] H. Massalin C. Pu and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988. 26

[CTHL01] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, Queen Mary London, 2001. 15

[DCD07] Cristian Dittamo, Antonio Cisternino, and Marco Danelutto. Parallelization of c# programs through annotations. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part II*, pages 585–592, Berlin, Heidelberg, 2007. Springer-Verlag. 77

[DFSJ08] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: flexible and safe pointcut/advice bindings. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 60–71, New York, NY, USA, 2008. ACM. 29

[DP] J. Aycock D. Pereira. Instruction set architecture of mamba a new virtual machine for python. *Technical Report 2002-706-0*. 13

[Ebe06] David H Eberly. *3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics*. Number 978-0122290633 in The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann, second edition, November 2006. 92, 167, 169, 172

[ECMa] Ecma 334 c# language specification. `http://www.ecma.ch/ecma1/STAND/ecma-334.htm`. 20, 69

[ECMb] Ecma 335, common language infrastructure (cli). `http://www.ecma.ch/ecma1/STAND/ecma-335.htm`. 12, 15, 20, 58, 65, 69

[ecmc] Ecma scripting standard. `ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf`. 17

[EL03] Erik Ernst and David H. Lorenz. Aspects and polymorphism in aspectj. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 150–157, New York, NY, USA, 2003. ACM. 29

[Eng96] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 160–170, 1996. 25

[EQvCD08] Bryan Eisenberg, John Quarto-vonTivadar, Brett Crosby, and Lisa T. Davis. *Always Be Testing: The Complete Guide to Google Website Optimizer*. Sybex, 2008. 145

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004. 14

[GG08] Vincenzo Gervasi and Giacomo A. Galilei. Software manipulation with annotations in java. pages 161–184, 2008. 82

[GM02] Robert Graybill and Rami Melhem. *Power Aware Computing*. Series in Computer Science. Springer, 2002. 147

[GMC+06a] Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. *Computer Graphics Forum*, 25(3):333–342, September 2006. Proceedings of Eurographics 2006. 5

[GMC+06b] Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), sep 2006. To appear in Eurographics 2006 conference proceedings. 169

[GMP+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000. 25

[Gou08] Tom Goulding. Complex game development throughout the college curriculum. *SIGCSE Bull.*, 40(4):68–71, 2008. 4, 54

[GSS+06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006. 46

[HLL06] Ming-Yang Hou, Xi-Yang Liu, and He-Hui Liu. Fifi: an architecture to realize self-evolving of java program. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 93–93, New York, NY, USA, 2006. ACM. 3

[ISOa] Iso/iec 23270, information technology – c# language specification. `http://www.iso.org`. 20

[ISOb] Iso/iec 23271, information technology – common language infrastructure. http://www.iso.org. 12, 20

[JG02] Neil D. Jones and Arne J. Glenstrup. Program generation, termination, and binding-time analysis. *SIGPLAN Not.*, 37(9):283–283, 2002. 25

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993. 25

[Joh75] S. C. Johnson. Yacc - yet another compiler compiler. 1975. 26

[KCC00] Samuel N. Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components 2: Binary-level components. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 28–50, London, UK, 2000. Springer-Verlag. 16

[KH09] Christopher M. Kanode and Hisham M. Haddad. Software engineering challenges in game development. *Information Technology: New Generations, Third International Conference on*, 0:260–265, 2009. 54

[Kra98] A. Krall. Efficient javavm just-in-time compilation. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 205, Washington, DC, USA, 1998. IEEE Computer Society. 12

[Lew98] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Comput. Surv.*, 30(1):3–27, 1998. 16

[LiY99] *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999. 20

[LL] Mark Leone and Peter Lee. Dynamic specialization in the fabius system. *ACM Comput. Surv.*, page 23. 26

[LL94] Mark Leone and Peter Lee. Lightweight run-time code generation. Technical report, Department of Computer Science, University of Melbourne, 1994. 25

[LL96a] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 137–148, New York, NY, USA, 1996. ACM. 25

[LL96b] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *In Workshop on Compiler Support for System Software (WCSSS*, pages 8–17, 1996. 25

[Mas92] Henry Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, New York, NY, USA, 1992. 26

[MR03] James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Professional, 2003. 69

[mse] Windows embedded. http://www.microsoft.com/windowsembedded/en-us/default.mspx. 48

[MY01] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In *in Proceedings of Programs as Data Objects, Second Symposium, PADO*, 2001. 15, 21, 25

[MY02] Wan-Chun Ma and Chia-Lin Yang. *Advances in Multimedia Information Processing — PCM 2002*, volume 2532/2002 of *Lecture Notes in Computer Science*, chapter Using Intel Streaming SIMD Extensions for 3D Geometry Processing. Springer Berlin, 2002. 99

[NBA05] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In *Conference proceedings : Erfurt, Germany, September 20 - 22, 2005*, Lecture Notes in Informatics 69, pages 19–38, 2005. 47

[.ne] The .net common language runtime. http://msdn.microsoft.com/net. 51

[Ngu07] Hubert Nguyen. *GPU Gems3: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2007. 14

[NV02] James Newkirk and Alexei A. Vorontsov. How .net's custom attributes affect design. *IEEE Software*, 19(5):18–20, 2002. 20

[NVI] NVIDIA. Cuda (compute unified device architecture). http://www.nvidia.com/object/cuda_home.html. 14

[ocl] Opencl : The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/. 14

[OCV] The ocaml system - implementation. http://pauillac.inria.fr/~lebotlan/docaml_html/english/. 13, 20

[OMY01] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. Dynjava: Type safe dynamic code generation in java. In *In JSSST Workshop on Programming and Programming Languages, PPL2001, March 2001*, 2001. 16, 25

[OS99] Tim Sheard Oregon and Tim Sheard. Using metaml: a staged programming language. In *In Advanced Functional Programming*, pages 207–239. Springer-Verlag, 1999. 26

[PA02] David Pereira and John Aycock. Instruction set architecture of mamba, a new virtual machine for python. Technical report, 2002. 25

[par] The parrot virtual machine. http://dev.perl.org/perl6/. 13

[PD83] Steven Pemberton and Martin C. Daniels. *PASCAL Implementation: P4 Compiler/Compiler and Assembler Interpreter*. Computers and Their Applications. Ellis Horwood Ltd, 1983.

[PH09] Jiyong Park and Seongsoo Hong. Building a customizable embedded operating system with fine-grained joinpoints using the aox programming environment. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1952–1956, New York, NY, USA, 2009. ACM. 45

[PHEK99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, 1999. 26

[Pin07] Sonia Pini. *Blueprint: A High-Level Pattern Based AOP Language*. PhD thesis, Department of Informatics and Computer Science, Università di Genova, Genoa, Italy, 2007. 43

[Piu05] Ian Piumarta. Accessible language-based environments of recursive theories. Technical report, Viewpoints Research Institute, 2005. 48

[PS97] Enrico Puppo and Roberto Scopigno. Simplification, lod and multiresolution principles and applications, 1997. 125

[pvm] Python vm. http://www.python.org/. 13

[pvs] Potentially visible set. http://en.wikipedia.org/wiki/Potentially_visible_set. 171

[RF05] Matt Pharr Randima Fernando. *GPU Gems2: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2005. 14

[Rog97] Dale Rogerson. *Inside COM*. Microsoft Programming Series. Microsoft Press, February 1997. 16, 20

[rot] Shared source common language infrastructure 2.0 release. http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en. 59

[RPM] Rapid mind multi-core development platform. http://www.rapidmind.net/. 14

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005. 59

[scg] Scene graph in rendering. http://en.wikipedia.org/wiki/Scene_graph. 167

[SdR84] Brian Cantwell Smith and Jim des Rivieres. Interim 3-lisp reference manual. 1984. 17

[SE06] Tarek Sobh and Khaled Elleithy. *Advances in Systems, Computing Sciences and Software Engineering*, chapter Runtime support for Self-Evolving Software, pages 157–163. Springer Netherlands, 2006. 3

[Ses] Peter Sestof. Runtime code generation with jvm and clr. http://www.dina.dk/~sestoft/rtcg/rtcg.pdf. 15, 16

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007. 11

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. *Semantics, Applications, and Implementation of Program Generation*, pages 2–44, 2001. 23, 25, 27, 53

[SHH09] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1944–1951, New York, NY, USA, 2009. ACM. 48, 49

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. 25

[sml] Sml.net compiler home page. http://www.cl.cam.ac.uk/Research/TSG/SMLNET/. 18

[SNS03] David Stutz, Ted Neward, and Geoff Shilling. *Shared source CLI essentials*. O'Reilly Media, 2003. 2

[Spra] Spring framewrok. http://www.springsource.com/. 38

[sprb] Spring.net framewrok. http://www.springframework.net/. 38, 55

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*, volume 11. Addison Wesley, 1994. 17

[Swe06] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 269–269, New York, NY, USA, 2006. ACM. 90, 169

[TeaBS98] Walid Taha, Zine el-abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety (extended abstract). In *In 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929. Springer-Verlag, 1998. 26

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, 1997. 25

[TSDNP02] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José M. Piquer. Altering java semantics via bytecode manipulation. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 283–298, London, UK, 2002. Springer-Verlag. 15, 21

[tvm] Tea vm. http://www.acroname.com/brainstem/brainstem.html. 13

[Vel98] Todd L. Veldhuizen. Arrays in blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag. 25

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983. 130

[xvm] Xsltvm. http://www.gca.org/papers/xmleurope2000/papers/s35-03.html. 13

[zbu] Z buffer and z culling. http://en.wikipedia.org/wiki/Z-buffering. 167

[Zlo75] Moshé M. Zloof. Query by Example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, Anaheim, California, 1975. ACM. 67