Multi-stage and Meta-programming Support in Strongly Typed Execution Engines

ANTONIO CISTERNINO

May 2003
Abstract

Program development and execution involves a number of processing and transformation steps performed by a variety of tools such as: preprocessors, compilers, program development environments, parser generators, software installers, operating system loaders, just to mention the most common ones. Each system takes as input a program in source or object form and produces a new program, which can be used as input to another system.

This thesis develops a technique suitable to express staged computations, targeted to modern execution environments, such as the JVM and the CLR. Code fragments are introduced in programs as first class values, which can be composed by means of an operator called Bind. We discuss power of this operator showing that it is type-safe and complete: only type sound code fragments can be generated; all possible programs can be produced from a finite set of base code fragments. Binding provides meta-programming support in the runtime environment. We show that the staging operators of MetaML can be expressed in term of our mechanism. Implementation and performance issues are discussed for Bind and code values, i.e. the values representing code fragments.

An execution model for programs based on the mechanism is proposed. A program is executed not only at runtime: a program evaluator may evaluate it at different times of its lifetime (after compilation, during installation, and so on). A distinctive feature of the model is that program transformations before runtime can be expressed in the program itself.
Acknowledgments

The contribution of many people has made possible this thesis. Therefore I need to thank everyone who contributed to this result. I apologize but a single page is not enough to mention all the people that deserve it.

First of all I wish to thank my supervisor Giuseppe Attardi who always supported and encouraged me, leaving me the freedom of experimenting with many different topics. Comments from my external advisors, Andrew Kennedy and Martin Odersky have greatly improved the content of this manuscript. The initial work of this thesis has been done whilst I was visiting Microsoft Research Laboratory in Cambridge. It has been great to collaborate with Andrew Kennedy, who gave me the opportunity of working on my subject. Many people contributed in discussions to this work: Don Syme, Claudio Russo, Luca Cardelli, Cedric Fournet, Andy Gordon, Egon Börger, Giorgio Ghelli. I wish to thank also Jim Miller who gave me the opportunity of showing my work to the people in Microsoft .NET product team. The formal model described in the thesis couldn’t have been possible without the support – both moral and material – by Paolo Mancarella. I wish to thank Maria Simi for her support during these years, she always found time to spend in listening my conjectures and ideas always encouraging me when I was facing some difficulty.

There are many ways of contributing to a work like this. All my friends have helped me, in particular Ruggero, who has been always present and supported me in many ways. I want to thank my parents, and my grandparents for giving me the opportunity of becoming what I am. Last but not least I wish to thank my wife Manuela and my son Marco for the support in this eight-years-long adventure in Computer Science: without their help and patience I wouldn’t be able to conclude my studies.
3.4.1 Code Transformation ...............................................................69
3.4.2 Bind Values ........................................................................73
3.4.3 Bind\textsubscript{\textgreek{e}}: Binding Arguments with Code Values .......................................................................74
3.4.4 Bind\textsubscript{\textgreek{e}}: Inlining of Code Values ..........................................................76
3.4.5 Bind Function .......................................................................77
3.4.6 Executing Code Values ............................................................79
3.5 INTERPRETERS AND COMPILERS ..............................................80
3.6 GENERATING CODE FROM COMBINATORS ......................................87
3.7 CONSIDERATIONS ABOUT CODE AND \texttt{BIND} ..................................................89

4 PERFORMANCE AND IMPLEMENTATION .................................................93
4.1 CODE BRICKS API ......................................................................93
4.1.1 An Extension to the Model .........................................................94
4.1.2 The Switch Problem .................................................................95
4.2 READING IL FROM ASSEMBLIES ..................................................97
4.3 IMPLEMENTATION STRATEGIES ...............................................99
4.4 THE MAX-STACK PROBLEM ......................................................101
4.5 OPTIMIZING THE GENERATED CODE .........................................102
4.6 THE REVERSE POLISH NOTATION COMPILER ............................108
4.7 EXECUTION ENVIRONMENT AND CODE VALUES ........................111
4.8 CODE BRICKS PERFORMANCE: THE .NET RE CASE STUDY ................113
4.8.1 An overview ...........................................................................113
4.8.2 Implementing RE compiler with Code Bricks .........................117
4.8.3 Performance Measurements ......................................................124

5 APPLICATIONS OF CODE BRICKS ..................................................133
5.1 RUNTIME CODE GENERATION .................................................133
5.2 SUPPORT TO ASPECT ORIENTED PROGRAMMING ......................134
5.3 DOMAIN SPECIFIC LANGUAGES ...............................................137
5.4 A VARIANT OF PROOF CARRYING CODE ....................................141
5.5 SUPPORTING PROGRAMMING LANGUAGES ................................144
5.5.1 DynJava and Dynamic Code Generation ..................................145
5.5.2 Bind, for Implementing Closures .............................................147

6 META-PROGRAMMING AND STAGING ..........................................149
6.1 CODE BRICKS AS SUPPORT FOR META-PROGRAMMING ...........149
6.1.1 C++ Template Meta-programming ...........................................150
6.1.2 C++ Template Meta-programming and Code Bricks ..................153
6.2 MULTI-STAGE PROGRAMMING WITH CODE BRICKS ..................154
6.2.1 Staging: a First Example .........................................................155
6.2.2 Code Bricks to Support MetaML .............................................157
6.3 MULTI-STAGING OUT OF PROCESS ..........................................160
6.4 MULTI STAGING WITH NAMED STAGES ....................................161

7 CONCLUSIONS .............................................................................165
7.1 REVIEW .............................................................................................................. 165
7.2 SUMMARY OF RESULTS.................................................................................. 167
7.3 OPEN PROBLEMS AND PROMISING RESEARCH DIRECTIONS .................. 168

8 APPENDIX ............................................................................................................. 169

8.1 EXTENSION OF THE TYPE SAFETY THEOREM’S PROOF ......................... 169
  8.1.1 Basic Lemmas ............................................................................................ 169
  8.1.2 Eval ldloca ............................................................................................... 169
  8.1.3 Eval stloc .................................................................................................. 170
  8.1.4 Eval add .................................................................................................... 170

8.2 DEFINITION OF $\Pi \vdash A \rightarrow A'$ ............................................................. 170

8.3 INTERPRETER THEOREM PROOF ................................................................. 173

8.4 TYPE SAFETY THEOREM PROOF ................................................................. 181

8.5 THE REVERSE POLISH NOTATION COMPILER .......................................... 187

BIBLIOGRAPHY ....................................................................................................... 193
1 Introduction

Program development and execution involves a number of processing and transformation steps performed by a variety of tools such as: preprocessors, compilers, program development environments, parser generators, software installers, operating system loaders, just to mention the most common ones. Each system takes as input a program in source or object form and produces a new program, which can be used as input to another system.

Every computation stage transforms the program according to some information that often is not present in the program itself. It would be better if a program could be able to express how it should be evaluated at each stage. Unfortunately the ability of expressing how a program should be modified across stages requires an execution model capable of supporting it. Tradition assumes that the program is written, often compiled, and then executed. Strongly typed execution environments such as Java Virtual Machine [48] or Microsoft’s Common Language Runtime [20] offer the opportunity of being extended with multi-stage and meta-programming capabilities that can be exploited by the languages that shares such runtime.

1.1 Reflection, Meta-Programming and RTCG

The current programming model is based on an execution paradigm which seems to not scale well. As pointed out in [15] programmers every day reinvent the wheel because it is more optimized than the existing one. We are incapable of writing really reusable software pieces though many reuse mechanisms have been designed since long 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’ve 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 probably we would have 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. And in
this need for generic interfaces lies the main problem with code reuse models based on reusing prewritten functionalities.

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 favor of highly efficient, hand-written, non generic libraries. An evident example of this trend is three dimensional graphics [3]: 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.

Goal of generative programming [15] is to provide this reusable set of functionalities in a way that specialize libraries and components when the options are specified by the programmer.

To achieve real code optimization we need to generate programs whenever some information becomes available and 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. Runtime code generation [14], [16], [29], [42], [43], [44], [45], [49], [50], [56], [61], Meta-programming [66], [67], [74], [78], [81], Partial evaluation [34], [35], [66].

It is interesting to notice that the goal of specializing components requires some abilities:

◊ Analysis of components usage
◊ Generation of the code representing the optimized version of some portion of a program
◊ Iterated program specialization in order to make use of information when it becomes available during its lifecycle.

Every aspect of the problem influence programming systems at all levels: we need programming constructs for controlling the generation process, and the specialization of programs; the code analysis and generation support should be included in runtime libraries so that programs can rely upon.
This thesis focuses on the problem of implementing the support for runtime code generation, meta-programming and multi-stage programming. We describe an extension to a particular class of runtime support that we call strongly typed execution environments (STEE) whose relevant members are Sun JVM and Microsoft CLR.

Our work extends the reflection support provided by STEE with a code type. The instances of this type, code values, are black box objects representing the body of methods. We provide a single operator to manipulate code values called Bind which turns out to be expressive enough to offer not only a way for manipulating method bodies but also as system for runtime code generation, meta-programming and multi-stage programming. This unifying mechanism can be implemented in a very efficient way and exposed as is to programming languages.

Another interesting property of our mechanism is that relies on programming elements shared, in STEE, by programming languages and the execution engine: types and methods.

Thus a programmer can use the mechanism, which is exposed as a type in its language, to mix code written in the same language having the impression that the meta-programming system performs source to source code transformations. Besides the code manipulation is performed at binary level rather than source level.

![Diagram](image_url)

*Figure 1.1: The programmer controls the code manipulation bearing in mind a transformation performed on the source code whilst the real transformation is performed at the level of the virtual machine.*
1.2 Strongly Typed Execution Environments

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

Garbage collection, libraries of precooked functionalities are often considered a requisite for a programming language. Programming languages based on virtual machines allows programs to be run across different platforms at the only 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’t be compromised running applications.

Java has been 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, have been proposed (see for instance the introduction of [40]) but Java has been the first to have a huge impact on programming mainstream. Nowadays also Microsoft is pushing virtual-machine based programming languages based on the Common Language Infrastructure (CLI) recently standardized by ECMA [20] and ISO [33]. The core of CLI is the virtual execution system also known as Common Language Run-time (CLR).

Both JVM [48] and CLR [20] implement a multi-threaded stack-based virtual machine, that offers many services such as dynamic loading, garbage collection, Just In Time (JIT) compilation. When the 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 [55], Python [62], TEA [76], XSLTVM [80], are but few examples. An alternative to stack-based virtual machines are register-based virtual machines, these machines 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 upcoming machine of Perl 6 called Parrot [57]. Although both stack-based and register-based machines are Turing equivalent there always is a fervent debate among the implementers of which model offer better performance. In [59], 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 doesn’t 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 need that the environment is able to reflect types and their methods to running programs. JVM and CLR are good examples of these environments. We don’t really need support to inheritance; we rather should be able to close a value to a function.

A *Strongly Typed Execution Environment* (STEE) is an execution environment which implements a virtual machine which provides an extensible type-system, reflection capabilities, and the execution model guarantees that type of values can be always established and values are always 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.

### 1.3 Structure of STEE

A STEE is an execution system whose execution is type driven. Its input is a program expressed in a language which is called *intermediate language*. CLR provides the Common Intermediate Language (CIL), JVM executes the byte-code. 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 JVM or CLR.

The intermediate language expresses a program executed by a thread: the execution begin 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. 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 loads of details such as registers, stack and heap implementation. This is the reason because many researches decide to work at the level of IL/bytecode although this may introduce some overhead [10], [49], [50], [65], [75].

Figure 1.2: CLI Machine State Model

1.3.1 Intermediate Language

The intermediate language of a stack-based STEE usually provides a set of instructions to load values from the store to the operand’s stack, and to store values in the stack back into the store. The invocation of a subroutine (either a function or a method) assumes that the arguments are loaded on the stack, a call instruction performs the call and the returned value is placed on the stack in place of the arguments.
In this section we briefly introduce CIL which will be used throughout the rest of the thesis. Consider the following class which is expressed both in Java and C#:

```java
class Add {
    public int add(int i, int j) {
        return i+j;
    }
    public int add3(int i, int j, int k) {
        int z = add(i, j);
        return add(z, k);
    }
}
```

The class is compiled into intermediate language shown in Table 1.1. Both IL and bytecode express operations by loading operands on the operand’s stack and then invoking the required operation. There are few differences between the instructions we need to control, though the CLR provides a virtual machine much more complex than JVM.

**Table 1.1: Comparison of CIL and Java Bytecode**

<table>
<thead>
<tr>
<th>CIL</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>void .ctor() {</td>
<td>Method Add()</td>
</tr>
<tr>
<td>.maxstack 1</td>
<td>0 aload_0</td>
</tr>
<tr>
<td>IL_0000: ldarg.0</td>
<td>1 invokespecial #1</td>
</tr>
</tbody>
</table>
| IL_0001: call instance | <Method java.lang.Object()>
| void System.Object::.ctor() | 4 return |
| IL_0006: ret | // end of method Add::.ctor |
| } // end of method Add::.ctor | |
| int32 'add'(int32 i, int32 j) | Method int add(int, int) |
| { | 0 iload_1 |
| .maxstack 2 | 1 iload_2 |
| IL_0000: ldarg.1 | 2 iadd |
| IL_0001: ldarg.2 | 3 ireturn |
| IL_0002: add | |
| IL_0003: ret | |
| } // end of method Add::'add' | |
| int32 add3(int32 i, int32 j, int32 k) | Method int add3(int, int, int) |
| { | 0 aload_0 |
| .maxstack 3 | 1 iload_1 |
| .locals init (int32 V_0) | 2 iload_2 |
| IL_0000: ldarg.0 | 3 invokevirtual #2 |
| IL_0001: ldarg.1 | <Method int add(int, int)> |
| IL_0002: ldarg.2 | |
| IL_0003: call instance | |
| } // end of method Add::'add3' | |
The main differences at the level of the abstract machine are:

◊ CLR allows allocating non primitive values on the stack (types of these values are known as value types)
◊ Only a subset of CIL is verifiable: there are instruction for manipulating pointers (we restrict ourselves to verifiable methods)
◊ CLR type system is common rooted: value types (which include primitive values) can be boxed into heap objects whose type inherits from System.Object.
◊ CLR provides the notion of delegate that is a good approximation of functions as first class objects.

### 1.3.2 Function Objects

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

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

This declaration corresponds to the following class:

```csharp
sealed class F : System.Delegate {
    public F(object o, IntPtr m) {}
    public virtual System.IAsyncResult BeginInvoke(int32 a, int32 b, System.AsyncCallback callback, object o) {}
    public virtual int32 EndInvoke(class System.IAsyncResult r) {}
    public virtual int32 Invoke(int32 a, int32 b) {}
}
```
As explained in section 13.6 of Partition II of ECMA standard 335 [20], a delegate class should define a constructor and three methods, namely BeginInvoke, EndInvoke and Invoke. These methods don’t 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 runtime is unaware of the notion of delegate so the implementation would be less efficient. Peter Sestoft in [65] describes a possible implementation of delegates in Java. A hand-written schema for having anonymous functions is used by DynJava [56] to run code generated at runtime.

### 1.4 Related Works

The work presented in this dissertation originates mainly by the experience of the author with C++ template meta-programming [2], [15] combined with discover of MetaML [73], [74] approach.

Because of its applicability there is a huge amount of literature that can be considered to be related. However we have found only fragments of the ideas contained in this thesis in very few papers. DynJava [56] is a system which performs similar operations on Java bytecode, though our code values and bind operator are far more expressive.

Calcagno et al. [10] describe an implementation of MetaOCaml a modified version of OCaml compiler to cope with staging annotations. Their approach relies on the OCaml bytecode showing that the idea of manipulating the intermediate language represents a good compromise between source to source transformations and machine code manipulation.

In particular the idea of generating code starting from the compiled bodies of methods seems to be completely new. Our contribution is in the area of the implementation of meta-programming and multi-staging support [35], [66].

Instead of providing a long list of systems only remotely related to our work we have decided to distribute comments about related works throughout the whole thesis.
1.5 Organization and Reading Plans

This thesis is organized in two parts. Part I introduces the model of code generation, and the fundamental operator for manipulating code values, $\textit{Bind}$. A formal model for the system is presented; we show that the transformation performed on streams of IL instructions is compatible with the functional description of the operator. The implementation of code values and the transformation of the IL are discussed.

Part II discusses applications of Code Bricks API, the implementation of our code generation model. In particular relations with meta-programming and multi-staging are discussed.

1.5.1 Part I

Chapter 2 introduces the notion of code type, code values, and $\textit{Bind}$ operator with examples. Although the $\textit{Bind}$ transformation is performed on IL instructions we discuss the examples using C# to simplify the understanding of the approach from the user standpoint.

Chapter 3 introduces a formal model of code values and $\textit{Bind}$. We extend a formal model of CLR proposed by Andy Gordon and Don Syme in [26], [27] which is small enough to formally define the transformation performed for generating code. We prove that $\textit{bind}$ operator is correct and complete. It is correct in the sense that the semantics expected by the transformation is satisfied. Moreover it is shown that the code generated using $\textit{Bind}$ is type safe. It is complete in the sense that it is possible to implement a compiler using code values and the operator. The relations with Futamura [23] projections and partial evaluation are discussed.

Chapter 4 discusses some aspect of the implementation (Code Bricks). In particular are discussed schemas for optimizing the generated code. In the end there is a brief discussion about issues and benefits of extending an execution environment with the code type.

1.5.2 Part II

The second part of the thesis is about the applications of the proposed mechanisms.

Chapter 5 is about applications of Code Bricks other than meta-programming and multi-staging. We discuss:

◊ runtime code generation;
◊ use of Code Bricks for supporting Aspect Oriented Programming;
implementation of Domain Specific Languages with Code Bricks;
◊ a variant of proof carrying code based on the Bind operator;
◊ use of Code Bricks by compilers to implement dynamic code generation and closures.

Chapter 6 focuses on the use of Code Bricks as a runtime support for implementing meta-programming and multi-staging. The relations of the model with C++ template meta-programming and MetaML are discussed. In both cases we are able to translate meta-programs into Code Bricks code fragments. A new notion of multi-staged programming languages is proposed in which a staged computation can be carried out by different processes. In this wider context we propose architecture for a multi-staged architecture that uses Code Bricks for code manipulation.
PART ONE
2 Extending the Execution Environment

2.1 Reflection in STEE

*Reflection* is being often considered an essential feature of a programming language. Component systems are easier to use as long it is possible to query the structure of a component at runtime instead of at compile time: it is possible to write generic components capable of dealing with components by inspecting their structure at runtime. The evolution of both COM [64] and CORBA [47] has been towards the support of metadata associated with components and accessible at runtime.

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 [24]:

- **Introspection**: the program can access to a representation of its own internal state. This support may range from knowing the type of values at runtime to having access to a representation of the whole source program.
- **Intercession**: the representation of the state of the program can be changed at runtime. This may include the set of types used, values and the source code.

Both introspection and intercession require a mechanism, 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 runtime. C++, for instance, supports *runtime type identification* (RTTI) that allows a program to have exact information about type of objects at runtime. Stroustrup [71] strongly encourage the use of compile-time type checking relying on RTTI only in absolutely necessary cases. Besides interpreted languages tend to offer rich reflection support because the execution system holds information that compile languages throw away during compilation. Besides, Interpreters have exact information about type of values and may also
interpret code generated at runtime. ECMA Script [21], 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 [69]), that are not modeled 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 runtime support contains additional code to keep track of types at runtime. Besides, the programmer would expect abstractions compatible with the structure of the programming language’s abstract machine (unless he is interested in manipulating the state of the machine which is target of the compilation).

STEE require information on types in order to enforce type safety at runtime\(^1\). 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. 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 an STEE tend to expose types of the execution environment as types within the language. This helps the programmer reusing libraries as well to use the fat class library that is part of the system. An example of this is SML.NET [70], 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 and other types are not easily accessible (perhaps because they are mapped to non trivial CLR types).

Though method bodies are part of type definition usually they aren’t available through the reflection interface. There are several reasons for this choice:

\(^1\) 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 runtime through runtime types.
Execution engine knows only their compiled version and the source it is no longer available. The only form available is the list of the abstract machine’s instructions, which are unknown at the source language level.

It would be possible to support only introspection of code: allow changing the running code would have impact on system’s performance. Moreover the execution environment enforces properties such as type safety and protection mechanisms that would be harder to guarantee allowing intercession on method bodies.

Program source could be made available within metadata, but often companies or programmers don’t like to distribute their source code.

In synthesis an appropriate abstraction for code is available such that programs could use it in a profitable way.

In this dissertation we introduce an appropriate abstraction for representing code executed by an 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. Besides this extension doesn’t introduce any additional overhead into the execution engine and it is suitable to support meta-programming and multi-stage languages.

2.2 Code values

We introduce the type Code within the execution engine with the aim of representing a method body: a sequence of instructions with an associated signature. The signature describes the number and type of the input parameters expected by the code fragment (identified as method arguments), and its return type.

Code values are instances of type Code; these can be combined to produce new values by means of a single operator named bind. A code value can be either open or closed depending if the signature has at least one input argument or not.

In the following example a Code value representing the static method\(^2\) add is created and assigned to the variable \(c\):

\(^2\) Add method has been intentionally declared static: instance methods can be represented as code values but the this pointer should be considered. In fact instance methods are like static ones with the first argument bound to the object on which the method has been invoked. As we discuss later the issues with instance methods are
public class Test {
    static int add(int i, int j) {
        return i + j;
    }
    public static void Main(string[] args) {
        Type t = typeof(Test);
        MethodInfo m = t.GetMethod("add");
        Code c = new Code(m);
    }
}

We will use C# [4], [19], [32] for examples, though the extension is at the level of the runtime: code values relies on types and methods which are abstractions shared between C# and the execution engine. Thus code is part of the reflection provided by the runtime rather than an extension of the language.

There are two kinds of code values:

◊ Atomic values: obtained directly from methods as in the previous example
◊ Derived values: created by calling the bind operator on other values

Code values can be only partially deconstructed to match their structure: atomic values should be treated as basic blocks; derived values are expressed in terms of bind and atomic values.

With code values, code becomes first class within the runtime. Besides a code value it is not directly executable though it always represents a method. If the execution engine knows when a code value should be executed it can adopt an optimized representation for code values rather than having to generate the code for each binding operation. Moreover the only executable objects into STEE are methods: making code values executable would change the model that is strongly based on types. Instead we prefer to add an operation to convert a code value $c$ into an object which has a method whose signature is the same as $c$. In our implementation on .NET we generate a delegate from a code value:

```csharp
delegate int Sum(int, int);
```

related to access protection mechanisms thus for the moment we will consider only static methods in our examples.

3 All the examples could be expressed in Java with very few changes to those presented.
Type t = typeof(Test);
MethodInfo m = t.GetMethod("add");
Code c = new Code(m);
Sum s = c.MakeDelegate(typeof(Sum)) as Sum;
Console.WriteLine("{0} == {1}" , add(1, 1) , s(1, 1));
}

In this example $c$ represents the code of the method $add$. The code value is transformed into a delegate of type $Sum$ which is an object exposing a method $Invoke$ with the signature $\text{int} \times \text{int} \rightarrow \text{int}$ implicitly called when the arguments are applied to $s$. Delegates are exposed by CLR as classes though the execution engine treat them in a different way. In Java it is possible to do the same by implementing a class loader [65] responsible of generating a type with a method inside.

Mark Shields in his thesis [68] introduces a similar notion of code value, though in his dissertation the abstraction is at language level rather than at the level of the runtime. Moreover it is not possible to deconstruct code values, no matter how they are created. Besides, as in MetaML [74], code values could be constructed out of expressions, not only from functions.

Although the invocation of $\text{MakeDelegate}$ is needed in order to execute code values, it will be omitted in our examples assuming that code values are executable. This is only intended to make samples more readable and when a code value is invoked we really mean that $\text{MakeDelegate}$ is called first to obtain the delegate.

### 2.3 Binding

Together with $\text{Code}$ type we introduce an operator called $\text{bind}$ which is the only operation available for manipulating code values. This operator allows creating new code values out of existing ones in a way that recalls partial evaluation [34]. Binding is both correct and complete as we will show later on. It is correct with respect to types: only type-safe code could be generated. It is complete because it is possible to define atomic values that can generate all possible programs.

A code value has associated a signature which represents the interface of the code fragment it represents. Bind allows fixing the input parameters with values, other code values, or leaves them as arguments of the generated code value. Consider the following example:

```csharp
public class Test {
    static int add(int i, int j) {
        return i + j;
    }
```
public static void Main(string[] args) {
    Type t = typeof(Test);
    MethodInfo m = t.GetMethod("add");
    Code c = new Code(m);
}

Understanding the code generated by bind is simple; the following expression\(^4\) should be true for all possible expressions \(x\) and \(y\) that return an integer value:

\[
\text{add}(1, \text{add}(x, y)) == d(x, y)
\]

Roughly speaking the semantic of bind is to generate the code which behaves as if the program would invoke the methods corresponding to the atomic code values in the same way as \texttt{Bind} is used.

The \texttt{Bind} operator allows binding the parameters of a \texttt{Code} value to given values, to other code values, or to a free variable, leaving them in this case as arguments of the resulting code value. \texttt{Code.Free} represents a free variable, so for instance

\[
\text{Code } d = c.\text{Bind}(1, \text{Code.Free});
\]

corresponds to a function of one integer argument that adds one to it.

For the simple case of non functional values, the semantics of \texttt{Bind} can be defined in terms of partial application and lambda-lifting [22].

**Definition 2.1:** **Partial application** consists in the application of a curried function \(f : A_1 \rightarrow \ldots \rightarrow A_n \rightarrow R\) to its first \(m\) (\(m \leq n\)) arguments yielding a new function \(g : A_{m+1} \rightarrow \ldots \rightarrow A_n \rightarrow R\). Assuming that \(g = f(x_1, \ldots, x_m)\), \(g\) is defined by \(g(x_{m+1}, \ldots, x_n) = f(x_1, \ldots, x_m)\),

**Definition 2.2:** **Lambda lifting** is a program transformation to remove free variables. An expression containing a free variable is replaced by a function applied to that variable.

The \texttt{Bind} operator combines these two transformations as follows:

- partial application: of the code value function to the supplied arguments provided they are compatible with the signature types.
- lambda lifting: all free variables present in the expression are lifted into bound variables of the resulting function.

\(^4\) In the expression we assume that the code value \(d\) is executable; what we really mean is that we generate the executable object and we invoke the appropriate method like in the example in paragraph 2.2.
More precisely, consider a code value as described by a lambda expression. For instance the code value \(c\) corresponds to the expression:

\[ \lambda x\ y. x + y \]

Binding

\(c.\text{Bind}(1, \text{Code.Free});\)

corresponds to performing partial application of:

\[ (\lambda x\ y. x + y)\ 1\ z \]

where \(z\) is a new free variable, obtaining

\[ 1 + z \]

and then lambda lifting to obtain:

\[ \lambda z. 1 + z \]

Similarly,

\(c.\text{Bind}(\text{Code.Free}, \text{Code.Free});\)

produces

\[ \lambda t\ v. t + v \]

and hence the combination

\(c.\text{Bind}(1, c.\text{Bind}(\text{Code.Free}, \text{Code.Free}));\)

produces by partial application

\[ 1 + (t + v) \]

and then by lambda lifting

\[ \lambda t\ v. 1 + (t + v) \]

Although the lambda notation uses names for variables, the arguments in \text{Code} values are anonymous and are referred through positional numbers, like in the De Bruijn notation [17], [18] for representing binding constructs. Representing variables through De Bruijn indices avoids the need for renaming (alpha-conversion) to avoid variable capture during reduction. Nevertheless in the presentation we will still use the more traditional lambda notation since it is easier to read.

The case of functional values is dealt like in the following example:

\[ \text{Bind}(\lambda x.E, \lambda y.B) = \lambda y.(\lambda x.E)\ B) \]
A proper account would involve the use of a typed lambda calculus, since in Code we deal with typed constants, variables, expressions and methods. Of course a code value may contain constants (as the integer 1 in the example), primitive operators (like +), invocations of methods and side-effect operators like assignment, which are not present in pure lambda calculus.

Another reading of the Bind operator is that the resulting code value behaves as if the program were invoking the methods corresponding to the code values in the same way as Bind is used.

\[
d(x, y) == \text{add}(1, \text{add}(x, y))
\]

Bind can generate code values with open variables: arguments can be bound or left open using a special value indicated by Code.Free. In the example above the code value associated with \( c \) is the same as \( c.\text{Bind}(\text{Code.Free}, \text{Code.Free}) \) because all the input arguments are left open.

Another way of representing binding could be using a circuit like notation. Each atomic value is a box with an appropriate number of input and output lines. This view may help to enhance the compositional nature of derived code values.

In the following paragraphs we consider Bind operation more in detail before the introduction of a formal specification of the operator.

### 2.3.1 Signature of Bind

Bind is a polymorphic operator that behaves differently according to the type of code values to which it is applied. However, since it accepts a variable number of arguments, it is implemented as a method with accepts one argument of type Object[], with the following signature:

```csharp
public Code Bind(params object[] pars)
```

We used a C# feature to hide the construction of the array of objects: the params keyword indicates to the compiler that the method can be invoked with a list of parameters to be converted into an array. In other languages the explicit construction of the array would have been needed.

This allows expressing all possible method signatures: in fact Bind has the same signature used by reflection for dynamic method invocation. In Java instead native types would have to be wrapped into the corresponding wrapping classes (such as Integer for int type). The CLR type system is common rooted and also the boxed version of native types inherits from Object, thus the programmer may insert values of those types directly into the array.
2.3.2 Binding values

The type of each argument of a code value must be compatible with the type of the corresponding parameter. Consider the following example:

```csharp
public class Test {
    public static void foo(int i) {
        Console.WriteLine("Number is {0}", i);
    }
    public static void Main(string[] args) {
        Code c = new Code(typeof(Test).GetMethod("foo"));
        Code d = c.Bind(1);
        d(); // Invoke d
    }
}
```

Executing this code will print the following:

```
Number is 1
```

Binding the value 1 to the only argument of `foo` results in the generation of a method where the argument has been turned into a local variable bound to such value. The generated code is similar to the following method:

```csharp
public static void foo1() {
    int i = 1;
    Console.WriteLine("Number is {0}", i);
}
```

An optimized version of the code would just use the constant 1 instead of the variable i. Such optimizations can be applied while generating code, as discussed in a later section.

The previous example involved an integer value, i.e. a value type. Binding a reference type instead of a value type objects complicates the matter: a reference cannot be easily wired into the code; moreover a reference to the object must remain visible to the garbage collector to avoid that it gets collected.

To handle reference types a code value maintains an array of object references called `environment`. Binding a reference type adds the reference to the environment, and its index is used in the generated code. Consider the following example:

```csharp
public class Test {
    public class Baz {
        public int i;
        public Baz(int j) {
            i = j;
        }
    }
    public static void foo(Baz b) {
```
Console.WriteLine("Number is {0}", b.i);
}
public static void Main(string[] args) {
    Code c = new Code(typeof(Test).GetMethod("foo"));
    Baz b = new Baz(1)
    Code d = c.Bind(b);
    b.i = 2;
    d(); // Invoke d
}

In this case to the code value d is associated a reference to the instance of Baz referred by b. This reference is stored into the array of closed values associated to d called environment (similar to the lexical environment stored in a closure). The execution of the program produces the following output:

    Number is 2

Note that the output contains the updated value of b.i: the reference to the object is stored into the array associated with d. In this case the code generated is equivalent to the following:

    public static void foo2(object[] environment) {
        Console.WriteLine("Number is {0}", ((Baz)environment[0]).i);
    }

The first argument is hidden to the programmer and it is an array which contains only one element: the reference to the closed object. Closures in functional programming languages are often implemented with the same schema. As we will discuss in chapter 5 code may be taken into account to implement closures in addition to support meta-programming.

It is also worth noting that the closed values are always stored by reference, thus for value types which are wrapped in objects it isn't possible to change the value after binding (i.e. if we had used struct instead of class for Baz type the number in the output would have been 1).

### 2.3.3 Free variables and the $x+x$ problem

Binding is the process of fixing one or more input arguments of an open code value. It is possible to fix only a subset of the input arguments generating another open code value. We have already introduced the special value, `Code.Free`, which can be used to leave some argument open. In the following source the increment function is generated out of the add method:

    public class Test {

5 In CLR boxed value types; in the JVM the wrapping classes of native types.
```csharp
public static int add(int i, int j) {
    return i + j;
}

public static void Main(string[] args) {
    Code c = new Code(typeof(Test).GetMethod("add"));
    Code d = c.Bind(1, Code.Free); // λx.1+x
    Code e = c.Bind(Code.Free, 1); // λx.x+1
}
}
```

In this example both `c` and `d` represent an increment function, though the implementation is different in the two cases. Of course in this case the result is the same because addition is commutative.

To address this problem the `Bind` operator allows a class of placeholders described by the class `Free`. In the following example we generate the code that doubles a value using the `add` method previously defined:

```csharp
public static void Main(string[] args) {
    Code c = new Code(typeof(Test).GetMethod("add"));
    Free x = new Free();
    Code d = c.Bind(x, x); // λx.x+x
    Console.WriteLine("Double of 2 is {0}", d(2));
}
```

Here we introduce a named placeholder `x` used to bind the two arguments of `add` to a single input parameter. The identity is preserved through different calls to `bind` introducing a weak notion of name into the operation. The function `λx,y.2x+y` can be generated as follows:

```csharp
public static void Main(string[] args) {
    Code c = new Code(typeof(Test).GetMethod("add"));
    Free x = new Free();
    Code d = c.Bind(x, Code.Free); // λx.x.x+y
    Code e = d.Bind(x, d); // λx,y.x+x+y
}
```

Though we haven’t discussed about code composition yet, it is plain enough that the variable `e` contains the desired code value. The term `x` is shared between the two invocations of `Bind`; thus the resulting code will have two input arguments rather than three.

`Code.Free` is the anonymous placeholder and can be used to indicate that the input argument is not bound to other input arguments. Of course this placeholder is not strictly needed: a fresh placeholder generated and not referenced anymore would have the same effect. Besides having a special notation for anonymous placeholders may offer a better control to the programmer and a chance to optimize the generated code.
2.3.4 Code composition by means of Bind

So far we have discussed how Bind allows fixing the parameters of existing code values to generate new code values specialized for those parameters. If these were the only uses of code values and bind the support would have been hardly acceptable for code generation. Moreover given an open code value with \( n \) arguments it would have been possible to use Bind at most \( n \) times.

The most important feature of this operator consists in being able to combine code values in new values. Code can be composed in two different ways with Bind:

◊ Fixing input parameters of arbitrary code values with other ones

◊ Using code combinators, which are methods defined with the aim of being used by the operator.

In both cases the goal is to define a semantic of the composition which can be understood by a programmer without having to think in terms of the operations performed on the intermediate language.

As we have already done for values, we want to stay close as much as possible to the metaphor that the code generated is equivalent as if the methods would have been invoked in the same way arguments have been fixed.

2.3.4.1 Composition without code combinators

The idea of composing code by a sort of inlining of arguments that are code values has been inspired by C++ template meta-programming [15] described in section 6.1.1 and the way code is generated at compile time.

Code composition happens when an input argument of a code value is bound with another code value. The rationale is that if the type returned from a method is compatible with the input argument of a code value it is possible to do a sort of inlining. In this way we have a mean of expressing code generation based on the metaphor of method invocation that is well known to the programmer. If an open code value is used to fix an input parameter its parameters are lifted to the signature of the code value being created.

Consider again the example of the three-way add method based on the two way version:

```csharp
public class Test {
    public static int add(int i, int j) {
        return i + j;
    }
    public static void Main(string[] args) {
        Code c = new Code(typeof(Test).GetMethod("add"));
    }
}
```
```csharp
Code d = c.Bind(Code.Free, c); // λx,y,z.x+(y+z)
if (add(1, add(2, 3)) == d(1, 2, 3))
    Console.WriteLine("It works!");
}
}

Result of binding is a code value equivalent to the following method:

```csharp
public static int add1(int i, int j, int k) {
    int l = j + k;
    return i + l;
}
```

The second argument of c is bound to an open code value (that is c itself, though there is no recursion involved into the generation) with two input arguments. Thus these input arguments are lifted in place of the bound argument leading to a code value with three arguments: the first corresponds to the original, the second and the third are the inputs of the code value c.

Note that the code value bound is prefixed to the method body of add and the result is stored into a local variable. This variable is used in place of the former input argument. At first sight one may think that the following method would be better:

```csharp
public static int add2(int i, int j, int k) {
    return i + (j + k);
}
```

In add2 the definition of add is replaced at each occurrence of parameter j in the body of method add. Although in this case inlining the expression (i + j) leads to a better code (the use of a local variable is avoided), it cannot be used in general: if the code value is substituted to all the occurrences of some argument there is a risk that side-effects are produced more than once. Another drawback of inlining code where an argument is used is code bloat: the definition of the code is replaced for each occurrence leading to pathological situations in which a methods of few instructions blow up in a simple bind operation.

The four way version of add can be obtained as follows:

```csharp
public static void Main(string[] args) {
    Code c = new Code(typeof(Test).GetMethod("add"));
    Code d = c.Bind(c, c); // λw,x,y,z.(w+x)+(y+z)
    if (add(add(1, 2), add(3, 4)) == d(1, 2, 3, 4))
        Console.WriteLine("It works!");
}
```

In this case two input values are bound with a code value and the equivalent method is:
public static int add3(int w, int x, int y, int z) {
    int l = w + x;
    int m = y + z;
    return l + m;
}

Bind evaluates the arguments in a left-to-right order; this may be different from the strategy adopted by a language. Evaluation order is relevant not for values, which have been already evaluated when the array of objects has been filled, but for code values: their code is prefixed in that order and this influence the order in which side-effects may take place.

2.3.4.2 Code composition with code combinators

Composition method based on fixing code values to input parameters is easy to understand but it imposes some limit on the ability of generating code. The problem lies in the metaphor chosen: we can only link the return value of a code value to the argument of another one. Within such metaphor the only reasonable schema of composition would be code prefixing.

Suppose we are interested to combine a code value representing a while with another code value which should be its body. With the combination strategy we have seen so far there is no way to express this: an inlining of the code would be required replacing a placeholder for the body.

To extend the metaphor in a suitable way we need a new kind of placeholder. We use a type on input arguments which represents a function. Let’s start with an example that shows how to define a code combiner to express a for statement:

public delegate void Cmd(int i);

public class Test {
    public static void For(Cmd c, int par) {
        for (int i = 0; i < par; i++) c(i);
    }
    public static void Body(int j) {
        Console.WriteLine("Number is {0}", j);
    }
    public static void Main(string[] args) {
        Code c = new Code(typeof(Test).GetMethod("For"));
        Code b = new Code(typeof(Test).GetMethod("Body"));
        Code d = c.Bind(b, Code.Free);
        d(3);
    }
}

The output of the program is:
Number is 0
Number is 1
Number is 2

Code value d is equivalent to the following method:

```csharp
public static void For1(int p) {
    for (int i = 0; i < p; i++)
        Console.WriteLine("Number is {0}", i);
}
```

We use delegates (see section 1.3.2) as placeholders inside methods. A code combiner is a method that takes as input one or more delegate types. When the delegate is bound with a code value each invocation of the bound argument is replaced with the body of the code value. In the previous example the for loop uses the body of method Body instead of the delegate call.

Each delegate argument invocation is inlined with the code value bound with it. In this way it is possible to generate repeating a code fragment more than once. This is useful to perform optimizations such as loop unrolling.

If a code value is bound to a delegate argument, this argument shall only be invoked and not used in any other way (assignment or as an input to a method call). Of course when an argument is of type delegate and it is bound with a delegate instance it is treated as in all cases in which a value is bound to an input argument.

The two composition strategies discussed above can be mixed using delegates and other types as input arguments for code values.

### 2.4 Examples

We present three examples of code generation based on code values and binding operations. The first two examples show the use of the two kinds of code composition available on the same application. Third example show how a recursive function can be expressed using Bind.

#### 2.4.1 Power without code combinators

In this example we illustrate how to generate a specialized version of the $x^y$ function for a given value of $y$. The function Power generates an open code value which computes the power function for a given exponent without loops.

The algorithm consists in performing multiplications and square operations depending on the binary representation of the exponent. For instance

$$x^{11} = x^{1011} = x^{2^{1011}} \cdot x = (x^{2^{1010}} \cdot x)^2 \cdot x = ((x^2)^2 \cdot x)^2 \cdot x$$
This algorithm is optimal and it is linear in the number of bits of the exponent. A recursive method based on it is the following:

```java
static int pow(int y, int x) {
    if (y == 0)
        return 0;
    else if (y == 1)
        return x;
    else if (y % 2 == 0) {
        int v = pow(y / 2, x);
        return v * v;
    } else // y is odd and greater than 1
        return x * pow(y - 1, x);
}
```

The structure of the code generator is exactly the same: code values are used to accumulate the proper sequence of square and product operations. Method `Power` has the same structure of `pow` with the difference that `Bind` is used instead of performing the operations. This is a neat example of how turn an interpreter into a compiler by means of binding code values.

```java
class Power {
    static Code one = new Code(typeof(Power).GetMethod("One"));
    static Code id  = new Code(typeof(Power).GetMethod("Id"));
    static Code sqr = new Code(typeof(Power).GetMethod("Sqr"));
    static Code mul = new Code(typeof(Power).GetMethod("Mul"));

    public static int One(int i) {
        return 1;
    }
    public static int Id(int i) {
        return i;
    }
    public static int Sqr(int x) {
        return x * x;
    }
    public static int Mul(int x, int y) {
        return x * y;
    }

    public static Code Power(int n, Code x) {
        if (n == 0) return one.Bind(x);
        else if (n == 1) return x;
        else if (n % 2 == 0) return sqr.Bind(Power(n / 2, x));
        else return mul.Bind(x, Power(n - 1, x));
    }
    public static Code Power(int n) {
        Free x = new Free();
        return Power(n, id.Bind(x));
    }
}
```
static public void Main(string[] args) {
    Code c = Power(Int32.Parse(args[0]));
    Console.WriteLine("{0}^{1} = {2}", args[0], args[1],
        c(Int32.Parse(args[1])));
}

Power always returns an open code value with one argument of type int. If the exponent is 0 then the constant function $\lambda x.1$ is returned (method One). Of course this case only happens if the user specifies such a value as first argument of power. If the exponent is 1 the code value argument is returned because no further modification to the value is needed.

The most interesting cases happen if the exponent is greater than 1. If it is even the code for square function $\lambda x.x^2$ is used; otherwise the code for $\lambda x,y.x \cdot y$ is used. In this case the code value has two arguments that are fixed with two different code values; yet the code value that results from binding has only one argument. This is because we use a Free variable to fix the argument passed to Power: all the input of code values produced by it share the same input argument.

The code generated by Power(11) is equivalent to the following:

```csharp
public static int pow11(int x) {
    int 11 = x;       // $\lambda x.x = \text{Power}(1, x)$
    int 12 = 11 * 11; // $\lambda x.x^2 = \text{Power}(2, x)$
    int 13 = 12 * 12; // $\lambda x.(x^2)^2 = \text{Power}(4, x)$
    int 14 = x * 13;  // $\lambda x.x \cdot (x^2)^2 = \text{Power}(5, x)$
    int 15 = 14 * 14; // $\lambda x.(x \cdot (x^2)^2)^2 = \text{Power}(10, x)$
    return x * 15;    // $\lambda x.x \cdot (x \cdot (x^2)^2)^2 = \text{Power}(11, x)$
}
```

We have applied the translation for Bind which requires that a local variable is introduced when a code value is bound to an argument with type other than delegate.

Of course the resulting code may have been far better than the one shown above. Besides, the JIT may get rid of the locals simply analyzing the structure of the intermediate language. Nonetheless it is possible to optimize such kind of patterns during the code generation phase. We shall discuss optimization of generated code in chapter 4; for the moment we are interested only to expressive power of the operator.
2.4.2 Power with code combinators

In this example we discuss a slightly different implementation of the Power generator based on code combinators. The structure of the code is essentially the same already discussed, though in this case methods Sqr and Mul are implemented as combinators.

de delegate int IntExp(int i);

class Power {
    static Code one = new Code(typeof(Power).GetMethod("One"));
    static Code id = new Code(typeof(Power).GetMethod("Id"));
    static Code sqr = new Code(typeof(Power).GetMethod("Sqr"));
    static Code mul = new Code(typeof(Power).GetMethod("Mul"));

    public static int One(int i) {
        return 1;
    }
    public static int Id(int i) {
        return i;
    }
    public static int Sqr(IntExp f, int x) {
        int v = f(x);
        return v*v;
    }
    public static int Mul(IntExp f, IntExp g, int x) {
        return f(x)*g(x);
    }

    public static Code Power(int n, Code x) {
        if (n == 0) return one.Bind(x);
        else if (n == 1) return x;
        else if (n%2 == 0)
            return sqr.Bind(Power(n/2,x), Code.Free);
        else return mul.Bind(x, Power(n-1, x), Code.Free);
    }
    public static Code Power(int n) {
        return Power(n, id);
    }

    static public void Main(string[] args) {
        Code c = Power(Int32.Parse(args[0]));
        Console.WriteLine("{0}^{1} = {2}", args[0], args[1],
                           c(Int32.Parse(args[1])));
    }
}

Like in the previous version Power takes as input a code value with the same signature of IntExp. Note that in this case there is no need of introducing a
named input argument because the parameter is explicitly shared into the body of the method.

The code generated by Power(11) is the following:

```java
public static int pow11v2(int x) {
    int l1 = x;
    int l2 = l1 * l1;
    int l3 = l2 * l2;
    int l4 = x * l3 * l3;
    return x * l4 * l4;
}
```

In this case we save a local variable with respect to the former version of the program. This is because when code combinators are used the introduction of local variables is controlled by the program. We have used a local variable in Square to avoid that the same code value would have been duplicated.

Let’s consider the following definition for Square:

```java
public static int Square(IntExp f, int x) {
    return f(x) * f(x);
}
```

With such definition Power(11) would have been the following:

```java
public static int pow11v3(int x) {
    return x * (x * ((x * x)*(x * x))) * (x * ((x * x)*(x * x)));
}
```

In this case no local variables are used but the generated code is considerably bigger than the previous one, though it is faster in this example. Of course a better implementation of Power may take into account the number of expected locals and decide which version to generate.

With code combinators it is possible to get more control and generate code values that can be obtained without them. Vice-versa is not true: it is possible to generate code values using code combinators that aren’t expressed otherwise. Nonetheless it is useful to have both means of generating code values: code combinators should be defined ad-hoc whereas a normal method can be used to generate new code based on it.

### 2.4.3 Expressing recursion

Is it possible to express recursive methods using code values and binding? This is an obvious question although the answer is not so trivial. Recursive functions can be expressed with this extension of reflection by making the computation of
the fix-point explicit. Let us consider the generation of two recursive functions based on the same code combinator.

```csharp
public delegate int basef(int i);
public delegate int combiner(int i, int j);
public delegate int fun(fun d, int n);

public class Rec {
    public static int Id(int i) {
        return i;
    }
    public static int mul(int i, int j) {
        return i * j;
    }
    public static int add(int i, int j) {
        return i + j;
    }
    public static int rec(fun r, basef g, combiner h, int n) {
        if (n == 1)
            return g(1);
        else {
            int a = g(n), b = r(r, n - 1);
            return h(a, b);
        }
    }
}

the structure of the recursive function is sketched in method `fact`:

- the argument `g` represents the function to be invoked as a base of the recursion, and it is called when the argument `n` is 1;
- the argument `h` is the function for the inductive step of the recursion;

public static void Main(string[] args) {
    Code id = new Code(typeof(Rec).GetMethod("Id"));
    Code ff = new Code(typeof(Rec).GetMethod("rec"));
    Code gg = new Code(typeof(Rec).GetMethod("mul"));
    Code hh = new Code(typeof(Rec).GetMethod("add"));

    Code m = ff.Bind(Code.Free, id, gg, Code.Free);
    Code a = ff.Bind(Code.Free, id, hh, Code.Free);
    fun f = (fun)m.MakeDelegate(typeof(fun));
    Console.WriteLine("{0}! = {1}", args[0],
                      f(f, Int32.Parse(args[0])));
    f = (fun)a.MakeDelegate(typeof(fun));
    Console.WriteLine("S(0, {0}) = {1}", args[0],
                      f(f, Int32.Parse(args[0])));
}
```
Recursion cannot be expressed into code because \texttt{rec} will be used to get a code value to be bound with the function for the base and inductive steps of recursion. We made the process explicit: it is known that a code value can be converted into a delegate; thus we fix \texttt{g} and \texttt{h} with code values and then we convert the code value into a delegate of type \texttt{fun}. The function itself is passed as the first argument leading to a recursive call.

With the same combinator we are able to generate the code to evaluate the factorial and the sum of the first \( n \) natural numbers. To generate factorial we use product as inductive step whereas for sum we use addition.

This is the way of expressing recursion using code combinators. Without them things are much easier though there is no real benefit in binding: the code values are bound to a function that corresponds to the first invocation of recursion, all subsequent calls are to the original method without code prefix.

Consider the following example:

```csharp
public class Test {
    public static int fact(int n) {
        return n == 0 ? 1 : n * fact(n - 1);
    }
    public static int add(int i, int j) {
        return i + j;
    }
    public static void Main(string[] args) {
        Code a = new Code(typeof(Test).GetMethod("add"));
        Code f = new Code(typeof(Test).GetMethod("fact"));
        Code af = f.Bind(a);

        Console.WriteLine("({0} + {1})! = {2}", 1, 2, af(1, 2));
    }
}
```

The output of the program is:

\[(1 + 2)! = 6\]

Although the semantics of binding is preserved the code generated for \texttt{af} is equivalent to the following:

```csharp
public static int af1(int i, int j) {
    int ll = i + j;
    return ll == 0 ? 1 : Test.fact(ll - 1);
}
```

Thus only the first execution involves the code generated, afterward method \texttt{fact} will be invoked.
2.5 Towards a formal definition of Bind

So far we have introduced code values and binding by means of examples. We have used C# to sketch the structure of the generated code although the operator manipulates code fragments at STEE level.

Let us consider again the following example:

```csharp
public class Test {
    public static int add(int i, int j) {
        return i + j;
    }
    public static void Main(string[] args) {
        Code c = new Code(typeof(Test).GetMethod("add"));
        Code d = c.Bind(Code.Free, c); // λx,y,z.x+(y+z)
        if (add(1, add(2, 3)) == d(1, 2, 3))
            Console.WriteLine("It works!");
    }
}
```

In 2.3.4.1 we have described the code value \( d \) in terms of a C# method. Although description level was adequate to understand the semantics of the operator, the transformation is performed at IL level and now we should discuss the real transformation.

A plausible translation of method `add` in IL is the following:

```csharp
.method public static int code_add(int, int) {
    ldarg.0
    ldarg.1
    add
    ret
}
```

The two input arguments are loaded onto operand stack; the invocation of the operation `add` takes the topmost two arguments of the stack and replaces them with their sum. The `ret` operation returns the result of the addition to the method caller.

When the atomic code value `c` is built, the body of `add` is loaded from the binary file and is associated with it. Together with the body the signature, information about local variables, and the environment (empty in this case) are created:

```csharp
.method public static int code_c(Environment, int, int) {
    ldarg.1
    ldarg.2
    add
    ret
}
```
When `Bind` is invoked on `c` signature, local variables and environment are generated for `d`. The body of `d` is the following:

```java
.method public static code_d(Environment, int, int, int) {
    .locals init (int v_0)
    ldarg.2
    ldarg.3
    add
    stloc.0
    ldarg.1
    ldloc.0
    add
    ret
}
```

As already discussed the third argument of `c` (the second of `add`) is replaced by a local variable. The code bound with it is prefixed to the body and the result is stored into the local variable. The loading of the local variable is used in place of that of the argument.

In general the binding process consists in locating the places where an input argument is used and replacing with an appropriate sequence of instructions. The process starts from method bodies present into the compiled programs, exposed by our extension as atomic values. This is an essential aspect of binding: the abstraction of method is shared among the execution environment and programming languages supporting it; thus the programmer perceives that the operator operates at language level whereas the transformation is performed at the intermediate language level.

In order to guarantee that the semantics of binding can be understandable not only in terms of the instructions, we claim that the transformation performed on the code preserve the semantics of “method invocation”. This property is not trivial and we should be proved.

In the next chapter a more formal definition of code values and bind will be given. The expressive power of the operator bind is also discussed in terms of the STEE intermediate language.
3 A Formal Model for Code Type

3.1 Introduction

We have introduced an informal model of a transformation on code that allows building code out of precooked methods that are used as input. A programming language such as C# has been used to describe the semantics of methods produced by operator `Bind`.

Nevertheless the real code transformation is performed on the output of the C# compiler – or whatever other compiler targeting CLR. As shown in Figure 1.1 the code generation is performed by reading IL instruction from the binary file and through a particular transformation the desired IL code is generated. Besides the primitive mechanism is such that it can be used also directly in programming languages. This is possible because STEE have a significant number of semantic objects shared between programming languages and their compiled version. For instance, we rely on runtime methods and types that are exposed as is in programming language to ease code reuse and dynamic loading.

How do we are sure that the code produced using `Bind` has the semantics of the programs that we have sketched in C#? Is the code generated using binding well formed? Under the hypothesis that the methods used to build the initial code values are correct with respect to types, do we are guaranteed that the output of `Bind` preserves this correctness? Is there a set of methods that can be used to generate any program?

In this chapter we are interested in defining a transformation that operates on IL combining method bodies in such a way that their execution is equivalent to the application of methods used. For instance, let \( f(x, y) \) and \( g(z) \) be methods; assuming that types are compatible we expect that the result of binding \( g(z) \) to the first argument of \( f(x, y) \) will output the code whose execution is equivalent to \( f(g(z), y) \).

Once the transformation is defined, we should show that it does the right thing. We introduce a formal framework which is a significant subset of the CLR in order to give a precise definition of the transformation and prove interesting properties.
This chapter may result hard to read, this is because the transformation is complex and we have introduced several concepts to model code values. The formal framework is based on a model of CLR proposed by Syme and Gordon [26], [27]. We have extended this framework to model aspects relevant to our code transformation and absent in the original model.

A formal definition of the code transformation performed by Bind is given. We focus our attention to properties of code generated by means of Bind; in particular we show that the partial application semantics expected is fulfilled. Moreover the transformation deals correctly with types generating only well formed and type safe method bodies. Finally we define a set of methods that allows generating the code of programs expressed in an imperative language.

In the rest of the thesis we will discuss how a primitive such as Bind can be used to express the typical operations required to manipulate programs and to implement operators required by programming languages with support to multi-staged computations.

3.2 A model of the execution engine

To investigate the properties of Code type we introduce a formal model of the execution engine. Target of our study will be the code generated using bind operator.

We model the execution engine extending the model of IL used by Gordon and Syme [26], [27] to prove type safety of a subset of common intermediate language called Baby IL (BIL). Our extensions are required to introduce IL instructions manipulated by binding. We use a model of CLI rather than JVM because the Java virtual machine is essentially a subset of common language runtime.

In the rest of this section we introduce the syntax and the semantics of extended BIL. Extensions made to the original model will be indicated and the type safety theorem’s proof extended appropriately.

3.2.1 Types and Values

All BIL methods run in an execution environment that contains a fixed set of classes. Each class specifies types for a set of field variables, and signatures for a set of methods. Each object belongs to a class. The memory occupied by each object consists of values of each field specified by its class. Methods are shared among all objects of a class (and possibly other classes). Objects of all classes may be stored boxed in a heap, addressed by heap references. Object of certain
classes – known as values classes – may additionally be stored unboxed in the stack or as fields embedded in other objects.

We consider five sets:

◊ the set of class names \( c \in \text{Class} \)
◊ the set of value class names \( vc \in \text{ValueClass} \subseteq \text{Class} \)
◊ the set of field names \( f \in \text{Fields} \)
◊ the set of method names \( m \in \text{Methods} \)
◊ the set of static methods \( sm \in \text{StaticMethods} \subseteq \text{Methods} \)

Moreover is assumed that

\[ \text{System.Type} \in \text{Class} - \text{ValueClass} \]

Types describe objects, fields, methods’ arguments and result and intermediate results computed during the evaluation of method bodies. Types are defined using the following syntax:

\[ A, B \in \text{Type} ::= \text{void} | \text{int32} | \text{class } c | \text{value class } vc | A\& \]

The type \text{void} describes absence of data and it is used as in C only to specify that a method returns no value.

The type \text{int32} represents a 32 bit integer value; though in BIL these values are used only for conditional and while loops, in our extension a single arithmetic exception, the sum, is included to represent the class of arithmetic instructions in the method body.

A reference class describes a pointer to a boxed object (i.e. heap allocated) and is defined using class type constructor. Value classes are defined by means of value class type constructor and describe unboxed objects (i.e. a sequence of words representing the fields of the class). To each value class \( vc \) is associated a boxed class \( \text{class } vc \) intended to store its object in the heap.

The pointer type \( A\& \) is intended to describe a pointer to data of type \( A \), which may be stored either in the heap or the stack.

The following restrictions are imposed on pointer types (pointer confinement policy):

◊ No field may hold a pointer
◊ No method may return a pointer
◊ No pointer may be stored indirectly via another pointer

An important use of pointers in IL is to allow arguments and results to be passed by reference. These conditions are intended to avoid dangling pointers while pass by reference is still possible.
The following predicate is used to test if a type is pointer free:

\[ \text{pointerFree}(A) \equiv \neg \exists B. A = B & \]

Methods are identified by signatures:

\[ \text{sig} \in \text{Sig} ::= B \quad \text{m} \quad (A_1, \ldots, A_n) \]

A signature includes the name of a method \( m \), the type of its arguments \( A_1, \ldots, A_n \) and the type \( B \) of returned value. A method name can be used more than once in a class but the whole signature should be unique in the class.

The execution environment provides the inheritance relation among types. We introduce the subtyping relation \( (A <: B) \) which is the least relation satisfying the following rules:

\[
\begin{align*}
A & <: A & \text{(Sub Refl)} \\
\text{c inherits c'} & \quad \text{class c <: class c'} & \text{(Sub Class)}
\end{align*}
\]

The relation \( A \) inherits \( B \) holds if \( A \) inherits from \( B \); moreover because it is a transitive relation so it is \( A <: B \).

The execution environment consists of three components – a function \( \text{fields}(c) \), a function \( \text{methods}(c) \), a relation \( \text{c inherits c'} \) – that satisfies the following axioms:

\[
\begin{align*}
\text{fields} & \in \text{Class} \to (\text{Field} \to \text{fin Type}) &\quad \text{fields of a class} \\
\text{methods} & \in \text{Class} \to (\text{Sig} \to \text{fin Body}) &\quad \text{methods of a class} \\
\text{inherits} & \subseteq \text{Class} \times \text{Class} &\quad \text{class hierarchy} \\
\text{c inherits c} & \quad \text{(Hi Refl)} \\
\text{c inherits c' \land c' inherits c''} & \to \text{c inherits c''} & \text{(Hi Trans)} \\
\text{c inherits c' \land c' inherits c} & \to \text{c = c'} & \text{(Hi Antisym)} \\
\text{c inherits System.Object} & \quad \text{(Hi Root)} \\
\text{c inherits d} & \to \text{f \in dom(fields(d))} \\
\text{f \in dom(fields(c)) \land fields(c)(f) = fields(d)(f)} & \quad \text{(Hi fields)} \\
\text{c inherits d} & \to \text{dom(methods(d)) \subseteq dom(methods(c)) - StaticMethods} & \text{(Hi methods)} \\
\text{c inherits vc} & \Rightarrow \text{c = vc} & \text{(Hi Val)} \\
\text{pointerFree(fields(c)(f))} & \quad \text{(Good fields)} \\
\text{B m(A_1, \ldots, A_n) \in dom(methods(c))} & \Rightarrow \text{pointerFree(B)} & \text{(Good methods)}
\end{align*}
\]

\[ ^a \text{Although IL syntax disallows multiple inheritance, it happens that the axioms allow a class to inherit from two superclasses that are incomparable according to the inheritance relation.} \]
We indicate with $\rightarrow_{fin}$ the set of finite mapping from field names and method signatures to types. If $\text{fields}(c) = f_i \rightarrow A_i^{f_i} \in 1..n$, the class $c$ has exactly the set of field names $f_i, ..., f_n$ with types $A_i, ..., A_n$ respectively. The same applies to methods: if $\text{methods}(c) = \text{sig}_i \rightarrow b_i^{f_i} \in 1..n$ the class has exactly methods with signatures $\text{sig}_i, ..., \text{sig}_n$, implemented by the bodies $b_i, ..., b_n$ respectively.

Axioms $Hi$ define the inheritance in the system: derived classes inherit fields and methods from super classes. The hierarchy is common rooted in $\text{System}$.Object and no class can be derived from a value class. The axioms labeled with $\text{Good}$ impose the constraints on pointer types.

### 3.2.2 Method bodies

BIL is a deterministic, single-threaded, class-based object oriented language. It doesn’t provide a model for exception handling. Though it is possible to make bind operator work with exceptions, in our formal model we will take into account a simplified version of it.

Some instruction requires indicating a method or the class constructor’s as argument. A reference to either a method or a constructor for class $c$ is obtained by adding $c::$ as prefix of the name in the signature. We also need labels to indicate targets of branches.

We indicate these elements as follows:

- $\Diamond \quad L$ \hspace{1cm} (Label)
- $\Diamond \quad M ::= B c::m(A_1, ..., A_n)$ \hspace{1cm} (Method reference)
- $\Diamond \quad K ::= \text{void} c::.ctor(A_1, ..., A_n)$ \hspace{1cm} (Constructor reference)

The syntax of BIL is the following:

- $i4$ \hspace{1cm} (32 bit signed integer)
- $a, b \in \text{BodyInstr} ::= i4$ \hspace{1cm} (body instructions)
- $\text{ldc.i4} i4$ \hspace{1cm} (load integer)
- $a \text{btrue} L_1 \text{bfalse} L_2 \text{b} L_1 \text{br} L_2$ \hspace{1cm} (conditional)
- $L_1::a \text{btrue} L_2 :: b \text{br} L_1 L_2$ \hspace{1cm} (while loop)

---

7 The notation $f_i \rightarrow A_i^{f_i}$ express a finite map. Let $\text{dom}(f_i \rightarrow A_i^{f_i}) = \{ f_i, ..., f_n \}$, assuming that $f_i = f_j \Rightarrow i = j$. Let $(f_i \rightarrow A_i^{f_i})(f) = A_i$ if there exists $i$ such that $f_i = f$; undefined otherwise.

8 It is assumed only one constructor per class. The constructor has the following signature:

- $\text{void} \ .\ctor(A_1, ..., A_n)$

Constructors can only be invoked to create a new instance of a class: $\ .\ctor \notin \text{Methods}$. 
\( a \ b \)  (sequencing)
\( a \ ldind \)  (load indirect)
\( a \ b \ stind \)  (store indirect)
\( lda rga \ j \)  (load argument address)
\( a \ starg \ j \)  (store into argument)
\( ldloca \ j \)  (load local address)
\( a \ stloc \ j \)  (store into local)
\( a_1 \ldots a_n \ newobj \ K \)  (create new object)
\( a_0 \ a_1 \ldots a_n \ callvirt \ instance \ M \)  (call on boxed object)
\( a_0 \ a_1 \ldots a_n \ call \ instance \ M \)  (call on unboxed object)
\( a_0 \ a_1 \ldots a_n \ call \ M \)  (call static method)
\( a \ ldflda \ A \ c::f \)  (load field address)
\( a \ b \ stfld \ A \ c::f \)  (store into field)
\( a \ box \ vc \)  (copy value to heap)
\( a \ unbox \ vc \)  (fetch pointer to value)
\( a \ b \ add \)  (add two integers)

\textit{Body} ::= \textit{.locals} (A_0, \ldots, A_k) \textit{BodyInstr}  (method body)

The syntax of BIL has been designed to be as closest as possible to the intermediate language of CLR. It is assumed that the labels used in both cases don’t appear in any of the sub-expressions. To simplify the syntax of conditional and while loop the following definitions are employed:

\[
\begin{align*}
\text{a b cond } & \equiv \text{a brtrue L1 } \ b \ br \ L2 : \ b1 : L2 : \\
\text{a b while } & \equiv L1 : \ a \ brfalse L2 : b \ br \ L1 : L2 :
\end{align*}
\]

As discussed in [27] this schema of branching is not enough to express all possible branching schemas that can be expressed in IL. Nonetheless bind operation doesn’t affect branching structure, so it is enough for our purpose to adopt the same model.

We have added four instructions to the original model: \texttt{ldloca}, \texttt{stloc}, \texttt{add} and \texttt{call}.

The first two instructions are used to manipulate local variables. Local variables are needed to define the transformation performed by bind; their role is very similar to method arguments’, though they are accessible only from the method body. Each local variable has a type and can store only values compatible with it; like arguments they are numbered starting from 0 and are allocated on the stack.
The call instruction has been added to support invocation of static methods. Static methods are modeled like instance methods with the only difference that there is no self reference passed as hidden argument to the method.

Three instructions can be derived from BIL:

\[
\begin{align*}
  a \text{ldfld } A c::f &\equiv a \text{ldflda } A c::f \text{lind} \\
  a \text{ldarg } j &\equiv a \text{ldarga } j \text{lind} \\
  a \text{ldloc } j &\equiv a \text{ldloca } j \text{lind}
\end{align*}
\]

Local variables have been modeled as an extension of the body. We use a notation similar to arguments in the stack frame. Local variables are always initialized. We assume that:

\[.\text{locals}(A_0, ..., A_k) \Rightarrow \forall i \in 0..k, \text{pointerFree}(A_i)\]

Thus local variables can only store values or reference to boxed objects. We also assume that local variables are always initialized. Unboxed objects are assumed containing only zero; pointers to boxed objects are initialized to the special value 0 which indicate that the reference to the heap is invalid.

### 3.2.3 Evaluation of Method Bodies

The memory model consists of a heap of objects and a stack of method invocation frames. Each method frame is formed by two vectors: local variables and input arguments.

Objects in the heap can be referred through heap references. Moreover we need to model pointers to types and return values. We will refer to those elements as follows:

\[
\begin{align*}
p, q &\quad \text{(heap reference)} \\
p\text{tr} &\equiv\ 
\begin{cases} 
\quad p &\quad \text{(pointer)} \\
\quad (i, j) &\quad \text{(pointer to boxed object)} \\
\quad [i, j] &\quad \text{(pointer to argument } j \text{ of frame } i) \\
\quad \text{ptr}.f &\quad \text{(pointer to local } j \text{ of frame } i) \\
\quad u, v &\quad \text{(result)} \\
\quad 0 &\quad \text{(void)} \\
\quad i4 &\quad \text{(integer)} \\
\quad f_i \rightarrow u^{i \in 1..n} &\quad \text{(value: unboxed object)}
\end{cases}
\end{align*}
\]

Note that the result may be an unboxed object which is the sequence of its field values chained together.

The memory model is defined as follows:
\[o ::= c[f_i \to_{\text{fin}} u_i^{\epsilon_{1..n}}] \quad \text{(boxed object)}\]
\[h ::= p_i \to_{\text{fin}} o^{\epsilon_{1..n}} \quad \text{(heap)}\]
\[fr ::= .\text{args}(u_0, \ldots, u_n) .\text{locals}(l_0, \ldots, l_m) \quad \text{(frame)}\]
\[s ::= fr_1 \ldots fr_n \quad \text{(stack)}\]
\[\sigma ::= (h, s) \quad \text{(store)}\]

It is assumed that boxed objects (i.e. objects in heap) are labeled with the class \(c\) to which they belong. This is needed for method invocation: a pointer to a boxed object of type \(A\) can refer to any boxed object of type \(B\) such that \(B \subset A\).

Consider for instance the heap
\[h = p \to c[f_1 \to 1, f_2 \to (g \to 3)]\]
It contains a single object of class \(c\) referenced by \(p\). There are two fields into the object named \(f_1\) and \(f_2\). The first field contains the integer 1, and the second the unboxed object \(g \to 3\). Field \(f_2\) should have as type a value class with a single field whose name is \(g\).

The following stack contains two frames:
\[s = .\text{args}(p, p.f_2.g) .\text{locals}(1) .\text{args}(p, (1, 1), [1, 0]) .\text{locals}()\]
The top of the stack is \(.\text{args}(p, (1, 1), [1, 0]) .\text{locals}()\) which is associated to a method body with no local variables, one object argument and two pointers to integers to the first argument\(^9\) and the first local.

To make it easier to access the store two functions are used:
\[\text{lookup}(\sigma, \text{ptr}) \quad \text{(lookup \text{ptr} in store \sigma)}\]
\[\text{update}(\sigma, \text{ptr}, v') \quad \text{(update store \sigma at \text{ptr} with result \(v'\))}\]

To define these functions it is necessary the notion of \(\text{path}\) which is a possibly empty sequence of field names:
\[f^\circ ::= f_i \ldots f_n \quad \text{(sequence of fields, written \(\epsilon\) if \(n = 0\)}\]
The notion of path helps to manipulate value classes that can be possibly nested. We use a couple of auxiliary functions to lookup and update values in value classes:
\[\text{lookup}(v, \epsilon) \equiv v\]
\[\text{lookup}(f_i \to u_i^{\epsilon_{1..n}}, f^\circ) \equiv \text{lookup}(u_j, f^\circ) \quad \text{where} \quad j \in 1..n\]
\[\text{update}(v, \epsilon, v') \equiv v'\]
\[\text{update}(f_i \to u_i^{\epsilon_{1..n}}, f^\circ, v') \equiv (f_i \to \text{update}(u_j, f^\circ, v'), f_i \to u_i^{\epsilon_{(1..n)\setminus\{j\}}}) \quad \text{for} \quad j \in 1..n\]

\(^9\) We assume that both frames refer to instance methods of class \(c\) and that \(p\) refers to the object in the heap \(h\)
Now we can define \textit{lookup} and \textit{update} functions for accessing the store.

The \textit{lookup} function is overloaded for each type of pointer we have introduced. It works by accessing \textit{heap} or \textit{stack} depending on the pointer, and the \textit{lookup} function for values is used to walk through unboxed objects. The definition of lookup is the following:

\[
\text{lookup}((h, s), p.f^+) = \text{lookup}(f_i \rightarrow u^{e_l..e_{l+n}}, f^+) \\
\text{where } h(p) = c[f_i \rightarrow u^{e_l..e_{l+n}}]
\]

\[
\text{lookup}((h, s), (i, j).f^+) = \text{lookup}(v_i.f^+) \\
\text{where } s = \text{fr}_1...\text{fr}_m \text{ with } i \in 1..m \\
\text{and } \text{fr}_i = .\text{args}(v_{0..n}).\text{locals}(l_0, ..., l_k) \text{ with } j \in 1..n
\]

\[
\text{lookup}((h, s), [i, j].f^+) = \text{lookup}(l_i.f^+) \\
\text{where } s = \text{fr}_1...\text{fr}_m \text{ with } i \in 1..m \\
\text{and } \text{fr}_i = .\text{args}(v_{0..n}).\text{locals}(l_0, ..., l_k) \text{ with } j \in 1..k
\]

Let us consider the store \(\sigma = (h, s)\) of the previous example:

\[\text{lookup}(\sigma, (1, 0)) = p\]
\[\text{lookup}(\sigma, p.f_i.g) = 3\]
\[\text{lookup}(\sigma, [1, 0]) = 1\]

The \textit{update} function is similar to \textit{lookup} and its purpose is to have a single way of updating the store whatever pointer is used. Its definition is similar to \textit{lookup}:

\[
\text{update}((h, s), p.f^+, v') \equiv ((h - p), p \rightarrow c[\text{update}(f_i \rightarrow u^{e_l..e_{l+n}}, f^+, v')]\], s) \\
\text{where } h(p) = c[f_i \rightarrow u^{e_l..e_{l+n}}]
\]

\[
\text{update}((h, s), (i, j).f^+, v') \equiv \(h, \text{fr}_1...\text{fr}_m .\text{args}(v_{0..n}, \text{update}(v_i.f^+, v'), ..., v_n).\text{locals}(l_0, ..., l_k)...) \text{fr}_m\) \\
\text{where } s = \text{fr}_1...\text{fr}_m \text{ with } i \in 1..m \\
\text{and } \text{fr}_i = .\text{args}(v_{0..n}).\text{locals}(l_0, ..., l_k) \text{ with } j \in 1..n
\]

\[
\text{update}((h, s), [i, j].f^+, v') \equiv \(h, \text{fr}_1...\text{fr}_m .\text{args}(v_{0..n}, \text{update}(l_i.f^+, v'), ..., l_k)... \text{fr}_m\) \\
\text{where } s = \text{fr}_1...\text{fr}_m \text{ with } i \in 1..m \\
\text{and } \text{fr}_i = .\text{args}(v_{0..n}).\text{locals}(l_0, ..., l_k) \text{ with } j \in 1..k
\]

As an example of \textit{update} let us to consider the following expressions evaluated in the store \(\sigma\) (in bold is indicated the modified value):

\[
\text{update}((h, s), p.f_i.g, 0) \equiv \(p \rightarrow c[f_i \rightarrow 1, f_2 \rightarrow (g \rightarrow 0)]\), \text{args}(p, p.f_i.g).\text{locals}(1).\text{args}(p, (1, 1), [1, 0]).\text{locals}() \\
\text{update}((h, s), (2, 1), 2) \equiv
\]
\[(p \rightarrow c[f_1 \rightarrow 1, f_2 \rightarrow (g \rightarrow 2)],
\quad \text{args}(p, p, p, g).\text{locals}(1) .\text{args}(p, (1, 1), [1, 0]).\text{locals}())
\]

\[\text{update}((h, s), [1, 0], 3) \equiv
\quad (p \rightarrow c[f_1 \rightarrow 1, f_2 \rightarrow (g \rightarrow 3)],
\quad \text{args}(p, p, p, g).\text{locals}(3) .\text{args}(p, (1, 1), [1, 0]).\text{locals}())\]

The operational semantics of method bodies is a formal judgment \(\sigma \vdash b \mapsto v \cdot \sigma'\) meaning that the body \(b\), evaluated in the initial store \(\sigma\), evaluates the value \(v\) leaving the store \(\sigma'\).

The following rules are about control flow of the execution engine:

\[
\sigma \vdash \text{ldc}\cdot i4 \mapsto i4 \cdot \sigma \quad \text{(Eval ldc)}
\]

\[
\sigma \vdash a \mapsto u \cdot \sigma' \quad \sigma' \vdash b \mapsto v \cdot \sigma'' \quad \sigma \vdash a b \mapsto v \cdot \sigma'' \quad \text{(Eval Seq)}
\]

\[
\sigma \vdash a \mapsto 0 \cdot \sigma' \quad \sigma' \vdash b_0 \mapsto v \cdot \sigma'' \quad \sigma \vdash a b_0 \mapsto v \cdot \sigma'' \quad \text{(Eval Cond Else)}
\]

\[
\sigma \vdash a \mapsto i4 \cdot \sigma' \quad i4 \neq 0 \quad \sigma' \vdash b_1 \mapsto v \cdot \sigma'' \quad \sigma \vdash a b_1 \mapsto v \cdot \sigma'' \quad \text{(Eval Cond Then)}
\]

\[
\sigma \vdash a \mapsto 0 \cdot \sigma' \quad \sigma \vdash a b \mapsto u \cdot \sigma \quad \text{(Eval While 0)}
\]

\[
\sigma \vdash a \mapsto i4 \cdot \sigma' \quad i4 \neq 0 \quad \sigma' \vdash b \mapsto u \cdot \sigma'' \quad \sigma' \vdash a b \mapsto u \cdot \sigma''' \quad \sigma \vdash a b \mapsto u \cdot \sigma''' \quad \text{(Eval While 1)}
\]

Control flow is expressed in a standard way as in language IMP [79]; there is no boolean type thus integers are used instead.

The store is manipulated through \(\text{ld}xxx\) and \(\text{st}xxx\) instructions. The rules which express their behavior rely on \(\text{update}\) and \(\text{lookup}\) functions introduced so far:

\[
\sigma \vdash a \mapsto ptr \cdot \sigma' \quad \text{(Eval ldind)}
\]

\[
\sigma \vdash a \text{ ldind} \mapsto \text{lookup}(ptr) \cdot \sigma' \quad \text{(Eval ldind)}
\]

\[
\sigma \vdash a \mapsto ptr \cdot \sigma' \quad \sigma' \vdash b \mapsto v \cdot \sigma'' \quad \sigma \vdash a b \text{ stdind} \mapsto 0 \cdot \sigma \quad \text{(Eval stind)}
\]

\[
\sigma \vdash a \text{ stdind} \mapsto 0 \cdot \sigma' \quad \sigma' \vdash b \mapsto \text{update}(\sigma'', ptr, v) \quad \text{(Eval starg)}
\]

\[
\sigma \vdash a = (h, f_1 \ldots f_i) \quad \text{(Eval ldarga)}
\]

\[
\sigma \vdash a \text{ ldarga} j \mapsto (i, j) \cdot \sigma \quad \text{(Eval ldarga)}
\]

\[
\sigma \vdash a \text{ starg} j \mapsto 0 \cdot \text{update}(\sigma', (i, j), u) \quad \text{(Eval starg)}
\]
Now we consider the rules for reference types (i.e. \( c \not\in \text{ValueClass} \)). These rules control object’s creation, which should produce a boxed object, and method invocation that should model the run-time dispatch of method bodies\(^\text{10}\).

\[
K = \text{void} \ c::\cdot\text{ctor}(A_1, \ldots, A_n)
\]

\( c \not\in \text{ValueClass} \)

\[
\text{fields}(c) = f_i \rightarrow_{\text{fin}} A_{i}^{\text{fin}, 1..n} \quad \sigma \vdash a_i \mapsto v_i \cdot \sigma_{i+1} \quad \forall i \in 1..n
\]

\[
\sigma_{n+1} = (h, s) \quad p \not\in \text{dom}(h) \quad h' = h, p \rightarrow c[f_i \rightarrow_{\text{fin}} \text{Eval} v_i^{\text{fin}, 1..n}]
\]

\( \sigma \vdash a_1 \ldots a_n \text{ newobj } K \mapsto p \cdot (h', s) \)  (Eval newobj)

\[
M = B \ c::m(A_1, \ldots, A_n)
\]

\( B \ m(A_1, \ldots, A_n) \in \text{StaticMethods} \)

\[
\sigma_0 \vdash a_0 \mapsto p_0 \cdot (h_1, s_i) \quad h_2(p_0) = c'[f_i \rightarrow_{\text{fin}} u_i^{\text{fin}, 1..n}]
\]

\[
(h_1, s) \vdash a_i \mapsto v_i \cdot (h_{i+1}, s_{i+1}) \quad \forall i \in 1..n
\]

\[
\text{methods}(c')(B \ m(A_1, \ldots, A_n)) = \text{.locals}(A_0, \ldots, A_i) b
\]

\[
(h_{n+1}, s_{n+1}.\text{args}(p_0, v_1, \ldots, v_n).\text{locals}(l_0, \ldots, l_i))
\]

\[
\vdash b \mapsto v' \cdot (h', s' f')
\]

\( \sigma_0 \vdash a_0 a_1 \ldots a_n \text{ callvirt instance } M \mapsto v' \cdot (h', s') \)  (Eval callvirt)

In (Eval newobj) the constructor simply assigns the fields of the object that is boxed on the heap evaluating \( a_i, \ldots, a_n \); the reference to the newly allocated object is returned. The signature of the constructor is only needed to indicate the type of its arguments (which are the types of class \( c \) fields').

The (Eval callvirt) rule evaluates the first argument \( a_0 \) to get the reference \( p_0 \) on which the method should be invoked. The reference \( p_0 \) refer to the instance on which the method should be called. It may happen that the method reference \( M \) is of a class \( c \) whilst \( p_0 \) refers to an object of class \( c' \); in this case we should find the appropriate method body which is associated to \( c' \). In this way a subtype may override the implementation of a method inherited from a supertype. The other arguments are evaluated and allocated into a new stack frame which is added in top of the stack. The body \( b \) of the method is evaluated in the

---

\(^{10}\) For run-time dispatch we mean that the body retrieved for a method comes from the class labeling the boxed object rather than the class to which belongs the signature (the bind of method body is not known until runtime).

\(^{11}\) Typing rules will impose that \( c' <: c \) (see the rule (Body Subsum)).
resulting store. Finally the frame is removed from the stack and the modified store returned\(^\text{12}\).

Value types have similar rules for creating unboxed objects and method invocation. In addition they have two additional instructions that allow building a boxed object out of an unboxed one and vice versa. Boxing is the mechanism that allows using callvirt to invoke methods inherited from `System.Object` by value types. In such a way even integer values can be stored on the heap using the boxed version of `int32`.

\[\begin{align*}
\forall c \in \text{ValueClass} \\
K &= \text{void } vc :: \text{ctor}(A_1', ..., A_n') \\
\text{fields}(vc) &= f_i \rightarrow \text{fin } A_i^{=1..n} \\
\sigma \vdash a_i &\mapsto v_i \cdot \sigma_i, \quad \forall i \in 1..n \\
\sigma \vdash a_1 ... a_n \text{ newobj } K &\mapsto \left(f_i \rightarrow \text{fin } v_i^{=1..n}\right) \cdot \sigma_{n+1} \\
\end{align*}\]

\[\begin{align*}
\forall c \in \text{ValueClass} \\
M &= B \text{ vc :: } m(A_1, ..., A_n) \\
B \text{ } m(A_1, ..., A_n) &\notin \text{ StaticMethods} \\
\sigma \vdash a_0 &\mapsto \text{ptr } \cdot (h_1, s_1) \\
(h_1, s_1) \vdash a_i &\mapsto v_i \cdot (h_{i+1}, s_{i+1}), \quad \forall i \in 1..n \\
\text{methods}(vc)(B \text{ } m(A_1, ..., A_n)) &= \text{ .locals}(A_0, ..., A_k) \\
\sigma \vdash b &\mapsto v' \cdot (h', s', fr') \\
\sigma \vdash a_0 a_1 ... a_n \text{ call instance } M &\mapsto v' \cdot (h', s') \\
p &\notin \text{ dom}(h') \\
\sigma \vdash a &\mapsto \text{ptr } \cdot (h', s') \\
\text{lookup}((h', s'), \text{ptr}) &= f_i \rightarrow \text{fin } v_i^{=1..n} \\
\sigma \vdash a \text{ box } vc &\mapsto p \cdot (h', p \rightarrow vc[f_i \rightarrow \text{fin } v_i^{=1..n}]), s) \\
\sigma \vdash a &\mapsto p \cdot \sigma' \\
\sigma \vdash a \text{ unbox } vc &\mapsto p \cdot \sigma' \\
\end{align*}\]

Creation of unboxed objects simply returns the new object as result value; it could be stored either on the stack or on the heap as field of a boxed object.

Moreover, because value classes cannot derive from other classes we don’t have to face the problem of run-time dispatch. Thus we locate the body of method \(m\) which is known to belong to \(vc\) and it is evaluated in a store with an additional frame on the stack which contains the result of evaluation of the arguments. Note that the self argument should still evaluate to a pointer.

\[\text{Note that the stack can be modified by means of pointers to local variables and arguments that can be passed as arguments.}\]
Instances of value types cannot be allocated in the heap as is; they should be inserted in a boxed object. There are two ways to do it:

◊ by inserting the value in a field of a boxed object
◊ by creating a wrapper boxed object to contain it

The box instruction is used to create the wrapper of the value that should be stored in the heap. The inverse operation is called unbox and its semantics may appear to be curious: the reference of the boxed object, that is the input, is returned as the output of the instruction. As a matter of fact the unbox operation return a pointer that can be used by lookup as p.e. The typing judgment rules will rely on the instruction to convert the reference into a pointer to a value.

To access and modify fields of classes we use the following two instructions:

$$\sigma \vdash a \mapsto ptr \cdot \sigma'$$
$$\sigma \vdash a \; ldflda \ A \ c :: f \mapsto ptrf \cdot \sigma'$$

(Eval ldflda)

$$\sigma \vdash a \mapsto ptr \cdot \sigma' \cdot b \mapsto v \cdot \sigma''$$
$$\sigma \vdash a \; stfld \ A \ c :: f \mapsto 0 \cdot update(\sigma'', ptrf, v)$$

(Eval stfld)

Both instructions evaluates a to a ptr to an object either boxed or unboxed. The instruction ldflda returns a pointer to the field’s value, and stfld store the specified value (obtained by evaluating b) into the field.

Static methods are invoked both on boxed and unboxed objects as follows:

$$M = B \ c :: m(A_1, \ldots, A_n)$$
$$\begin{align*}
B \ m(A_1, \ldots, A_n) & \in \textit{StaticMethods} \\
\sigma_0 = (h_1, s_1) & \vdash a_i \mapsto v_i \cdot (h_{i+1}, s_{i+1}) \quad \forall i \in 1..n \\
\text{methods}(c)(B \ m(A_1, \ldots, A_n)) & = \textit{.locals}(A_0', \ldots, A_k')b \\
(h_{n+1}, s_{n+1} \cdot \textit{args}(v_1, \ldots, v_n) \cdot \textit{locals}(l_0, \ldots, l_k)) & \vdash b \mapsto v' \cdot (h', s' \textit{fr'})
\end{align*}$$

(Eval call)

In this version of call there is no self argument in the arguments allocated on the new stack frame. Moreover there is no instance thus the body of the method is provided by the class declared in the method reference.

Finally we present the trivial rule for add instruction:

$$\sigma \vdash a \mapsto v_1 \cdot \sigma' \sigma \vdash b \mapsto v_2 \cdot \sigma''$$
$$\sigma \vdash a \; b \; \textit{add} \mapsto v \cdot \sigma'''$$

(Eval add)
3.2.4 Typing Method Bodies

We are interested in studying the \texttt{Bind} operator with respect to the type system. Our aim is to show that the code values generated with binding are sound with respect to types. To reason about type properties of generated code we describe the type system for BIL.

A \textit{type frame}, $Fr$, is a description of the values’ types of the current stack frame and it has the form:

$$ Fr ::= .\text{args}(A_0, ..., A_n).\text{locals}(A_1', ..., A_k') \quad \text{(frame)} $$

A typing judgment that a body ($\text{BodyInstr}$) evaluates to a result of a given type under the hypothesis that the current stack frame matches the type frame $Fr$:

$$ Fr \vdash b : B $$

An additional assumption is made about the execution environment: the method bodies conform to their signature.

$$ c \in \text{Class} \land \text{ValueClass} \land m(A_i, ..., A_n) \not\in \text{StaticMethods} \land \quad \text{(Ref methods)} $$

$$ \text{methods}(c)(B m(A_i, ..., A_n)) = .\text{locals}(A_0', ..., A_k') b \Rightarrow $$

$$ .\text{args}(\text{class } c, A_i, ..., A_n).\text{locals}(A_0', ..., A_k') \vdash b : B $$

$$ vc \in \text{ValueClass} \land B m(A_i, ..., A_n) \not\in \text{StaticMethods} \land \quad \text{(Val methods)} $$

$$ \text{methods}(vc)(B m(A_i, ..., A_n)) = .\text{locals}(A_0', ..., A_k') b \Rightarrow $$

$$ .\text{args}(\text{value class } vc, A_i, ..., A_n).\text{locals}(A_0', ..., A_k') \vdash b : B $$

$$ c \in \text{Class} \land B m(A_i, ..., A_n) \in \text{StaticMethods} \land \quad \text{(Stat methods)} $$

$$ \text{methods}(c)(B m(A_i, ..., A_n)) = .\text{locals}(A_0', ..., A_k') b \Rightarrow $$

$$ .\text{args}(A_i, ..., A_n).\text{locals}(A_0', ..., A_k') \vdash b : B $$

The following rule is important because allows using expressions of subtype $B$ in a context expecting expressions of a super-type of $B$:

$$ Fr \vdash b : B \quad B \ll B' \quad \text{(Body Subsum)} $$

The typing rules for control flow are:

$$ \text{Fr} \vdash \text{lde}.i4 \ i4 : \text{int32} \quad \text{(Body lde)} $$

$$ \text{Fr} \vdash a : \text{void} \quad \text{Fr} \vdash b : B \quad \text{(Body Seq)} $$

$$ \text{Fr} \vdash a : \text{int32} \quad \text{Fr} \vdash b_0 : B \quad \text{Fr} \vdash b_1 : B \quad \text{Fr} \vdash a \ b_0 \ b_1 \ \text{cond} : B \quad \text{(Body Cond)} $$
\[
\frac{Fr \vdash a : \text{int32} \quad Fr \vdash b : \text{void}}{Fr \vdash a \ b \text{ while} : \text{void}} \quad \text{(Body While)}
\]

The type \textbf{void} is used to guarantee to indicate that a body returns no value. In practice this is equivalent to always return something that is ignored. This happens, for instance, in the rule (Eval Seq). The conditions used in \textit{cond} and \textit{while} requires that the guard is of type integer.

In the following rules we should pay attention to implement the pointer confinement policy stated in section 3.2.1. Let’s start with typing rules for pointer types:

\[
\frac{Fr \vdash a : A \&}{Fr \vdash a \ l\text{dind}: A} \quad \text{(Body l\text{dind})}
\]

\[
\frac{\text{pointerFree}(A) \quad Fr \vdash a_1 : A \& \quad Fr \vdash a_2 : A}{Fr \vdash a_1 a_2 \ s\text{tind} : \text{void}} \quad \text{(Body s\text{tind})}
\]

The (Body s\text{tind}) rule requires that \(A\) is pointer free in order to implement the third condition which imposes that no pointer can be used to store another pointer.

Arguments are manipulated using pointers; thus the typing rules for \textit{l\text{darga}} and \textit{s\text{targ}} should check that the index of the argument is valid and the pointer is of the right type. Similar rules define typing for access to local variables.

\[
\begin{align*}
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\end{align*}
\]

\[
\begin{align*}
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\end{align*}
\]

\[
\begin{align*}
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\text{\it j} \in 0..n & \quad \text{\it j} \in 0..n & \\
\end{align*}
\]

Now we consider typing rules for reference types; for typing judgment, though, we cannot share rules between reference and value types as we’ve done for the operational semantics (with the only exception of static method invocation).

\[
\begin{align*}
K = \text{void} & \quad c::\text{ctor}(A_0, \ldots, A_n) \\
\text{fields}(c) = f_1 \rightarrow_{\text{fin}} A_i^{i..n} & \\
Fr \vdash a_i : A_i & \quad \forall i \in 1..n \quad c \in \text{Class -ValueClass} \\
Fr \vdash a_1 \ldots a_n \text{newobj } K : \text{class } c & \\
\end{align*}
\]

(Ref \text{newobj})
\[ B \ m(A_1, \ldots, A_n) \in \text{dom}(\text{methods}(c)) \]
\[ B \ m(A_1, \ldots, A_n) \not\in \text{StaticMethods} \]
\[ \vdash a_0 : \text{class } c_0 : A_i : \forall i \in 1..n \quad \text{(Ref callvirt)} \]
\[ \vdash a_0 \ a_1 \ldots a_n \ \text{callvirt} \ \text{instance } B \ c :: m(A_1, \ldots, A_n) : B \]
\[ \text{fields}(c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \]
\[ \vdash a : \text{class } c \quad j \in 1..n \quad \text{(Ref ldflda)} \]
\[ \vdash a : \text{class } c \quad \text{fields}(c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \quad \text{(Ref stfld)} \]

The rules for value types are similar to those for reference types, though in this case we use pointers rather than references.

\[ K = \text{void } v_c :: \text{ctor}(A_1, \ldots, A_n) \]
\[ \vdash a : A_i : \forall i \in 1..n \quad \text{(Val newobj)} \]
\[ \vdash a_0 \ a_1 \ldots a_n \ \text{newobj} \ K : \text{value class } v_c \]
\[ \text{B } m(A_1, \ldots, A_n) \in \text{dom}(\text{methods}(v_c)) \]
\[ \text{B } m(A_1, \ldots, A_n) \not\in \text{StaticMethods} \]
\[ \vdash a_0 : \text{value class } v_c \quad a_0 : A_i : \forall i \in 1..n \quad \text{(Val call)} \]
\[ \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \]
\[ \vdash a : \text{value class } v_c \quad j \in 1..n \quad \text{(Val ldflda)} \]
\[ \vdash a : \text{value class } v_c \]
\[ \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \]
\[ \vdash a : \text{value class } v_c \quad \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \quad \text{(Val stfld)} \]
\[ \vdash a : \text{value class } v_c \quad \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \]
\[ \vdash a : \text{value class } v_c \quad \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \quad \text{(Val box)} \]
\[ \vdash a : \text{class } v_c \quad \text{fields}(v_c) = f_i \rightarrow \text{fin } A_i^{\#1..n} \quad \text{(Val unbox)} \]

Note that the axiom (Good fields) makes superfluous adding the pointerFree(–) to the premise of (Ref stfld), (Val stfld) and (Val box) rules.

As we discussed the (Eval unbox) rule would be useless because it acts as identity with respect to the store. The instruction unbox is relevant to the type-system: the rule (Val unbox) is used to keep track that the reference to the heap
should be changed from a reference to a boxed object to a pointer to an unboxed one.

The only rule shared between value and reference types is the following:

\[
B \, m(A_i, \ldots, A_n) \in \text{dom}(\text{methods}(c))
\]

\[
B \, m(A_i, \ldots, A_n) \in \text{StaticMethods}
\]

\[
Fr \vdash a_i : A_i \quad \forall i \in 1..n
\]

\[
Fr \vdash \text{call } B \, c \, m(A_i, \ldots, A_n) : B
\]

(\text{Body call})

Finally we should consider the typing rule for the \text{add} instruction:

\[
Fr \vdash a : \text{int32} \quad Fr \vdash b : \text{int32}
\]

\[
Fr \vdash a \, b \, \text{add} : \text{int32}
\]

(\text{Body add})

### 3.2.5 Typing the Memory Model

We are interested in typing the store of the execution environment in order to guarantee that operations are always invocated on values of appropriate types.

As we have already introduced heap, stack and store; we can define their types as follows:

\[
H := p_i \rightarrow c^{i \in 1..n}
\]

(heap type)

\[
S := Fr_1 \ldots Fr_m
\]

(stack type)

\[
\Sigma := (H, S)
\]

(store type)

In order to assign a type to each element of the memory we introduce a number of conformance judgments:

\[
\Sigma \vdash u : A
\]

in \(\Sigma\) result \(u\) has type \(A\)

\[
H \vdash o : c
\]

in \(H\) object \(o\) has class \(c\)

\[
H \vdash h
\]

heap \(h\) conforms to \(H\)

\[
\Sigma \vdash fr : Fr
\]

frame \(fr\) conforms to \(Fr\)

\[
\Sigma \vdash \sigma
\]

store \(\sigma\) conforms to \(\Sigma\)

Conformance rules for results are the following:

\[
H(p) = c \quad c \text{ inherits } c'
\]

(Res Ref)

\[
(H, S) \vdash p : \text{class } c'
\]

\[
H(p) = vc
\]

(Ptr Ref)

\[
\text{ie} \, 1..m \quad Fr_i = \text{.args}(A_0, \ldots, A_n) \cdot \text{locals}(A'_0, \ldots, A'_k) \quad j \in 1..n
\]

\[
(H, Fr_1 \ldots Fr_m) \vdash (i, j) : A_{i\&}
\]

(Ptr Arg)
\[
\begin{align*}
\text{class } c \lor A &= \text{value class } c' \\
\sum \vdash \text{ptr}: A & \quad \text{fields}(c) = f_i \rightarrow_{\text{fn}} A_i^{i..n} \quad j \in 1..n \\
\sum \vdash \text{ptr}, f_j: A_{ij}\& \\
\sum \vdash 0: \text{void} \\
\sum \vdash i4: \text{int32} \\
\text{fields}(vc) &= f_i \rightarrow_{\text{fn}} A_i^{i..n} \quad \sum \vdash v_i: A_i \quad \forall i \in 1..n \\
\sum \vdash f_i \rightarrow_{\text{fn}} v_i^{i..n}: \text{value class } vc \\
\end{align*}
\]

The rule \(\text{(Res Ref)}\) assigns a reference type \(\text{class } c'\) to a heap reference \(p\), so long as \(c'\) is a super-class of the actual class of the object referred to by \(p\).

The rule \(\text{(Ptr Ref)}\) assigns a pointer type to a heap reference \(p\) that refers to a value that is boxed on the heap.

These two rules can assign both a reference type and a pointer type to a heap reference to a value class. \(\text{(Res Ref)}\) is needed to type references constructed by \text{box} instruction. \(\text{(Ptr Ref)}\) is needed to type pointers constructed by the \text{unbox} instruction.

\(\text{(Ptr Arg)}\) and \(\text{(Ptr Loc)}\) assign types to local variable and argument pointers. The rule \(\text{(Ptr Field)}\) assigns a pointer type to the pointer referring field \(f_j\). The last three rules assign a type to the possible values.

Now we can state the conformance rule for objects:

\[
\begin{align*}
\text{fields}(c) &= f_i \rightarrow_{\text{fn}} A_i^{i..n} \\
(H, \emptyset) \vdash v_i: A_i \quad \forall i \in 1..n \\
H \vdash c[f_i \rightarrow v_i^{i..n}]: c \\
\end{align*}
\]

This rule defines when a heap object is well-typed. The preconditions require that the values \(v_i\) of the object should be typed with an empty stack. Thus a field cannot hold a stack pointer since the rules \(\text{(Ptr Arg)}\) and \(\text{(Ptr Loc)}\) assume a non-empty stack type.

The conformance rule for heaps is:

\[
\begin{align*}
H &= p_i \rightarrow c_i^{i..n} \\
H \vdash o_i: c_i \quad \forall i \in 1..n \\
H \vdash p_i \rightarrow o_i^{i..n} \\
\end{align*}
\]

\text{(Con Heap)}
This rule defines when a heap conforms to the heap type $H$. A similar rule is defined for stack frames:

$$
\sum \ni \in 0..n \ni A_i \ni \forall i \\
\sum \ni \in 0..k \ni A_i' \ni \forall j \\
\sum \ni .\text{args}(u_0, \ldots, u_n) . \text{locals}(v_0, \ldots, v_k) : \\
.\text{args}(A_0, \ldots, A_n) . \text{locals}(A_0', \ldots, A_k')
$$

(Con Frame)

Finally we can define when a store conforms to a store type:

$$
H \ni h \ni (H, F_{r_1} \ldots F_{r_n}) \ni fr \ni \forall i \ni 1..n \\
(H, F_{r_1} \ldots F_{r_n}) \ni (h, fr_1 \ldots fr_n)
$$

(Con Store)

The rule asks that the heap is conforming to a store type and each stack frame to a frame type. In order to type check the stack frame $fr_i$ we shouldn’t take into account any subsequent frame that would be younger than the current one.

### 3.2.6 Evaluation Respects Typing

The following theorem asserts that, if a program satisfies the restrictions on type structure imposed in section 3.2.1 and the typing rules for method bodies in section 3.2.4, then its evaluation according to the rules in section 3.2.3 can lead only to conformant intermediate states as defined in section 3.2.5.

Let $H \ni H'$ mean that $\text{dom}(H) \ni \text{dom}(H')$ and $H(p) = H'(p)$ for all $p \ni \text{dom}(H)$. This relation indicates that the heap type can only grow preserving the references and their type.

**Theorem 3.1:** If $(H, S Fr) \ni \sigma$ and $Fr \ni b : B$ and $\sigma \ni b \ni v \cdot \sigma'$ then there exists a heap type $H'$ such that $H \ni H'$ and $(H', S Fr) \ni v : B$ and $(H', S Fr) \ni \sigma'$.

The proof of the theorem is fully available in [27]; in Appendix 8.1 we discuss the changes needed to the proof due to our extension of the model.

### 3.3 Code Type

Now that we have introduced the formal model of the runtime on which we rely upon, we can introduce the formal definition of code type. As we already pointed out a code value is formed by three components: a signature; an environment which keeps track of bound values and a body of instructions.

**Definition 3.1:** Let $b \ni \text{Body}$, $s \ni \text{Sig}$ and $e = .\text{env}((e_0, v_0), \ldots, (e_n, v_n))$ an ordered list of pairs $(e, v) \ni \text{Int} \times \text{Val}$. A code value $cv$ is a triple $cv \ni < s, b, e >$.

A code value can be obtained from a method of a class using the function $\text{methodof}$:
A similar lemma can be stated for value classes.

We use the special name \( \mu \) in signatures of code values because to represent them it is irrelevant the method’s name. The code value is enough to associate the signature to the body.

It is worth noting that \textit{methodof} allow us building code values either out of instance or static methods. This is because \textit{call instance} and \textit{callvirt} instructions simply add an additional hidden argument to the method. The following lemma states that if we consider an instance method and the corresponding static method with the self argument explicit with the same body the two commands are equivalent.

\textbf{Lemma 3.1:} If \( c \in \text{Class} - \text{ValueClass} \), and \( c', c'' \in \text{Class} \), and \( B \text{~} m_1(A_1, ..., A_n) \in \text{methods}(c) \), and \( B \text{~} m_2(class\ c, A_1, ..., A_n) \in \text{methods}(c') \), and \( B \text{~} m_3(A_1, ..., A_n) \notin \text{StaticMethods} \), and \( B \text{~} m_4(class\ c, A_1, ..., A_n) \in \text{StaticMethods} \), and \( \sigma \vdash a_0 \mapsto p_0 \cdot (h, s) \) \( h(p_0) = c'[f_i \rightarrow \tau_i (u^{\xi_i -\cdot})] \), and \( \text{methods}(c')(B \text{~} m_1(A_1, ..., A_n)) = \text{methods}(c')(B \text{~} m_2(class\ c, A_1, ..., A_n)) \), then \( \sigma \vdash a_0 \ a_1 \ldots \ a_n \ \text{callvirt instance} \ M \mapsto v \cdot \sigma' \iff \sigma \vdash a_0 \ a_1 \ldots \ a_n \ \text{call} \ M' \mapsto v \cdot \sigma'; \) where \( M = B \text{~} m_1(A_1, ..., A_n) \) and \( M' = B \text{~} m_2(class\ c, A_1, ..., A_n) \).

\textbf{Proof.} The proof is trivial. \( \Rightarrow \) Applying \((\text{Eval callvirt})\) we get that the store in which the body \( b \) gets evaluated is \( (h_{n+1}, s_{n+1}.\text{args}(p_0, v_1, ..., v_s).\text{locals}(l_0, ..., l_i)) \). Because \( \sigma \vdash a_0 \mapsto p_0 \cdot (h_i, s_i) \) the store in which \( b \) gets evaluated is the same in the application of \((\text{Eval call})\). \( \Leftarrow \) The first argument \( a_0 \) should evaluate to a reference \( p_0 \) thus we can apply \((\text{Eval callvirt})\) evaluating \( b \) in the same store as in \((\text{Eval call})\) and we get the same final store.

A similar lemma can be stated for value classes.

\textbf{Lemma 3.2:} If \( v_c \in \text{ValueClass} \), and \( c \in \text{Class} \), and \( B \text{~} m_1(A_1, ..., A_n) \in \text{methods}(v_c) \), and \( B \text{~} m_2(\text{value class\ } v_cA, A_1, ..., A_n) \in \text{methods}(c) \), and \( B \text{~} m_3(A_1, ..., A_n) \notin \text{StaticMethods} \), and \( B \text{~} m_4(\text{value class\ } v_cA, A_1, ..., A_n) \in \text{StaticMethods} \), and

\[
\text{methods}(v_c)(B \text{~} m_1(A_1, ..., A_n)) = \text{methods}(c)(B \text{~} m_2(\text{value class\ } v_cA, A_1, ..., A_n)),
\]
then \( \sigma \vdash a_0 a_1 \ldots a_n \) call instance \( M \mapsto v \cdot \sigma' \Leftrightarrow \sigma \vdash a_0 a_1 \ldots a_n \) call \( M' \mapsto v \cdot \sigma' \); where \( M = B c :: m(A_1, \ldots, A_n) \) and \( M' = B c' :: m(c, A_1, \ldots, A_n) \).

**Proof.** It is almost identical to the proof of Lemma 3.1.

Although BIL allows defining, for every instance method, an equivalent static version, this isn’t true in the CLR. Access control mechanisms ensure that if a member of class \( c \) is private then it could be accessed only from methods defined in the class itself. In this context making explicit the self argument in a static method isn’t enough to get the equivalence with an instance one because the latter may have access to members which aren’t accessible to the former.

### 3.4 Binding Code Values

In this section we introduce the bind operation on code values. The essential operation of bind is to fix a parameter of an existing code value with a value that is *compatible*.

Binding is a complex operation which involves the three components of a code value. We need to manipulate:

- the signature to add or reduce its arguments
- the environment to add values bound to a code value
- the body to generate the new code out of the existing one

In this section we introduce the **Bind** function which is the fundamental tool to manipulate code values. As discussed in chapter 1 this is a complex operation that depends on the type of arguments and the values used to bind them. To separate different aspects we provide three functions:

- \( \text{Bind}^v : \text{Code} \times \text{Int} \times \text{StoreType} \times \text{Val} \rightarrow \text{Code} \)
- \( \text{Bind}^\pi : \text{Code} \times \text{Int} \times \text{Code} \rightarrow \text{Code} \)
- \( \text{Bind}^\chi : \text{Code} \times \text{Int} \times \text{Code} \rightarrow \text{Code} \)

The function \( \text{Bind}^v \) expresses the binding of an argument \( i \) to a value. The second function \( \text{Bind}^\pi \) expresses code combination without code combiners. The last function, \( \text{Bind}^\chi \), is the code combination with code combiners.

#### 3.4.1 Code Transformation

The manipulation of method bodies requires the ability of applying a set of renaming rules to change the way in which the program access local variables and arguments. In particular we need the following kinds of mapping rules:

- local variable \( i \) becomes local variable \( j \) (\( i \neq j \))
◊ argument $i$ becomes argument $j \ (i \neq j)$
◊ argument $i$ becomes local variable $j$
◊ type of argument $i$ has an `Invoke` method whose invocations are to be replaced with a sequence of instructions

We give the definition of a transformation which maps a sequence of BIL instructions into another one by means of a set of rules built on the structure of BIL.

The transformation that should be applied is defined as a finite map $\Pi = (l, i) \rightarrow_{\text{fin}} (l', j, b)$, where $l \in \{ \text{loc}, \text{arg} \}$, $l' \in \{ \text{loc}, \text{arg}, \text{inline} \}$, $i, j \in 0..n$, $b \in \text{BodyInstr}$, assuming $n$ the highest possible index for argument or local variable$^{13}$.

Now we can define the transformation of a sequence of BIL instructions. The transformation is expressed as follows$^{14}$:

$$\Pi \vdash b \rightarrow b'$$

The full definition of the transformation is given in appendix 8.1, we provide here the most important rules. The transformation simply looks for instructions which load or store arguments or local variables; in such cases if the instruction should be changed a substitution will be performed.

The larger part of instructions are left untouched by the transformation, the following three rules are reported as significant examples of this fact. The rules apply the transformation to their constituents and simply rebuild the transformed statement.

\[
\begin{align*}
\Pi \vdash \text{ldc\.} i4 & \rightarrow \text{ldc\.} i4 \\
\Pi \vdash a & \rightarrow a' \quad \Pi \vdash b & \rightarrow b' \\
\Pi \vdash a \ b & \rightarrow a' \ b'
\end{align*}
\]
(Ren ldgc)

\[
\begin{align*}
\Pi \vdash a & \rightarrow a' \quad \Pi \vdash b_0 & \rightarrow b_0' \quad \Pi \vdash b_1 & \rightarrow b_1' \\
\Pi \vdash a \ b_0 \ b_1 \ \text{cond} & \rightarrow a' \ b_0' \ b_1' \ \text{cond}
\end{align*}
\]
(Ren Cond)

Let us consider the rules that perform the real transformation on the instruction sequence. There is a rule for each one of the possible outcomes of transformation $\Pi$. The following rules don’t modify the instruction because either the argument or the local variable isn’t involved in the transformation:

\[
\begin{align*}
\Pi \vdash \text{ldarga} \ j & \rightarrow \text{ldarga} \ j \\
(\text{arg}, j) & \notin \text{dom}(\Pi)
\end{align*}
\]
(Ren ldarga)

---

$^{13}$ We assume $(l, j, \varepsilon) = (l, j)$, where $\varepsilon$ is the empty sequence.

$^{14}$ We indicate with $\bot$ a transformation that couldn’t be performed
\[(\text{loc}, j) \notin \text{dom}(\Pi)\]
\[\Pi \vdash \text{ldloca} j \rightarrow \text{ldloca} j \quad (\text{Ren ldloca})\]

\[(\text{arg}, j) \notin \text{dom}(\Pi)\]
\[\Pi \vdash \text{starg} j \rightarrow \text{starg} j \quad (\text{Ren starg})\]

\[(\text{loc}, j) \notin \text{dom}(\Pi)\]
\[\Pi \vdash \text{stloc} j \rightarrow \text{stloc} j \quad (\text{Ren stloc})\]

Mapping arguments on other arguments is useful for renaming: often bind requires that indexes of arguments are shifted to introduce or eliminate arguments from signature.

\[\Pi(\text{arg}, j) = (\text{arg}, i)\]
\[\Pi \vdash \text{ldarga} j \rightarrow \text{ldarga} i \quad (\text{Ren ldarga arg})\]

\[\Pi(\text{arg}, j) = (\text{arg}, i)\]
\[\Pi \vdash \text{starg} j \rightarrow \text{starg} i \quad (\text{Ren starg arg})\]

Often we need to rename local variables too: different bodies use indexes from 0 to \(n\) to access local variables; during instruction merging we should avoid clashes.

\[\Pi(\text{loc}, j) = (\text{loc}, i)\]
\[\Pi \vdash \text{ldloca} j \rightarrow \text{ldloca} i \quad (\text{Ren ldloca loc})\]

\[\Pi(\text{loc}, j) = (\text{loc}, i)\]
\[\Pi \vdash \text{stloc} j \rightarrow \text{stloc} i \quad (\text{Ren stloc loc})\]

An argument can be the bound to the result of a code value; in this case a local variable would hold the value that would have been into the argument. As we’ve done for renaming we provide a pair of rules that simply transform ldarga and starg into ldloca and stloc respectively.

\[\Pi(\text{arg}, j) = (\text{loc}, i)\]
\[\Pi \vdash \text{ldarga} j \rightarrow \text{ldloca} i \quad (\text{Ren ldarga loc})\]

\[\Pi(\text{arg}, j) = (\text{loc}, i)\]
\[\Pi \vdash \text{starg} j \rightarrow \text{stloc} i \quad (\text{Ren starg loc})\]

Last but not least we introduce a rule to support the code combinators: the invocation of a special method is replaced by a body that uses local variables instead of arguments.
This transformation assumes a special pattern in order to perform the transformation. The argument \( i \) is used as placeholder to locate the invocations that should be replaced. Any other use of the argument should be forbidden. The following rules make the transformation fails:

\[
\Pi (\text{arg}, i) = (\text{inline}, k, b) \quad \Pi + a_i \rightarrow a_i' \\
\Pi + \text{ldarg } i \ a_1 \ldots a_n \text{ callvirt instance } M \rightarrow \\
a_i' \text{ stloc } k_1 \ldots a_n' \text{ stloc } k_n \ b' \\
\text{(Ren Invoke)}
\]

We have assumed the \text{Invoke} method because CLR exposes delegates as sub-classes of \text{System.Delegate} with such method. When a delegate refers to a method; the invocation of \text{Invoke} results in the invocation of the target method.

Let us consider the code value \(<\text{int32 }\mu(\text{int32}), \ .\text{locals}(\text{int32}) \ b, \ .\text{env}()>\) where

\[
b = \\
\text{ldc.i4 } 0 \\
\text{stloc } 0 \\
\text{ldarg } 0 \\
\text{ldarg } 0 \\
\text{ldloc } 0 \\
\text{add} \\
\text{stloc } 0 \\
\text{ldc.i4 } -1 \\
\text{ldarg } 0 \\
\text{add} \\
\text{starg } 0 \\
\text{while} \\
\text{ldloc } 0
\]

This is the body of a static function which takes as input a number \( n \) and returns the sum of the first \( n \) integers\(^{15}\).

\(^{15}\)We've used a loop rather than the direct formula to get a non trivial body.
Now suppose that we want to produce a code value which calculates the sum of the first \( n \) integers where \( n \) is the output of another code value \(<\text{int32} \ \mu(), \ \text{locals()} \ b', \ \text{env()} >\). Using bind we can fix the first argument to such code value. As discussed in the previous chapter our strategy for generating the new body consists in prefixing \( b \) with \( b' \) and turning the input argument into a local variable. We should ensure that the new body refers local variables properly; a possible solution can be the following:

\[
b' \ \text{stloc} \ 0 \ b''
\]

We use the local variable 0 to store the result of \( b' \); this means that we should rename the local variable used by \( b \) from 0 to 1. Thus the mapping to produce \( b'' \) is \( \Pi = \{ (\text{arg}, 0) \rightarrow (\text{loc}, 0), (\text{loc}, 0) \rightarrow (\text{loc}, 1) \} \) and \( \Pi \uparrow b \rightarrow b'' \).

\[
b'' = \begin{align*}
&\text{ldc.i4 0} \quad \text{(Ren ldc)} \\
&\text{stloc 1} \quad \text{(Ren stloc loc)} \\
&\text{ldloc 0} \quad \text{(Ren ldarg loc)} \\
&\text{ldloc 0} \quad \text{(Ren ldarg loc)} \\
&\text{ldloc 1} \quad \text{(Ren ldloc loc)} \\
&\text{add} \quad \text{(Ren add)} \\
&\text{stloc 1} \quad \text{(Ren stloc)} \\
&\text{ldc.i4 -1} \quad \text{(Ren ldc)} \\
&\text{ldloc 0} \quad \text{(Ren ldarg loc)} \\
&\text{add} \quad \text{(Ren add)} \\
&\text{stloc 0} \quad \text{(Ren starg loc)} \\
&\text{while} \quad \text{(Ren While)} \\
&\text{ldloc 1} \quad \text{(Ren ldloc loc)}
\end{align*}
\]

Of course we could also have used the local variable 1 for the argument; in this case the transformation map would have been smaller though the cost of the transformation process would have been essentially the same.

### 3.4.2 Binding Values

Now that the code transformation function has been defined we can define the \( \text{Bind}_v \) function. When a value is bound to an argument the environment is used to store it. The environment is a sort of stack frame which preserves the values needed by the code instructions.

Code values can survive to method calls: we should not allow that the pointers confinement policy stated in 3.2.1. The environment is a sort of object thus
we should forbid that a pointer can be bound to an argument. The definition of $\text{Bind}_v$ is the following:

$$
\text{Bind}_v(c, i, \sum, v) = \\
\langle B \mu(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n), .\text{locals}(A_0', \ldots, A_{i-1}', A_{i+1}', \ldots, A_n'), b', \\
.\text{env}((e_0, v_0), \ldots, (e_m, v_m), (k + 1, v)) \rangle \\
$$

where $i \in 0..n$,

$c = \langle B \mu(A_0, \ldots, A_n), .\text{locals}(A_0', \ldots, A_n') b, .\text{env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle$,  

$\sum \models v : A_i$,  

$\text{pointerFree}(A_i)$,  

$\Pi \vdash b \rightarrow b'$,  

$\Pi = \{ (\text{arg}, i) \rightarrow (\text{loc}, k + 1), (\text{arg}, i+1) \rightarrow (\text{arg}, i), \ldots, (\text{arg}, n) \rightarrow (\text{arg}, n-1) \}$. 

The function operates on the $i$th argument of the code value. The function requires that in the store $\sum$ the value $v$ has the same type $A_i$. Moreover the type $A_i$ should be pointer free; otherwise we may save a pointer to the current stack frame that in future may be no longer available.

The $i$th argument is removed from the signature and a new local variable is introduced with index $k+1$. The local variable will contain the value $v$ that is saved in the environment together its index. The body is changed accordingly using the code transformation $\Pi$.

### 3.4.3 $\text{Bind}_\pi$ Binding Arguments with Code Values

$\text{Bind}_\pi$ function defines the composition of two code values together. The function corresponds to the code generation schema discussed in 2.3.4.1. The input consists of a code value $c$, an integer that specifies the index of the argument that should be fixed with a code value $c'$.

A new code value is produced based on $c$; a new local variable is introduced and the instructions of $c'$ precede those of $c$. The new local variable is used instead of the argument within the body of $c$.

The function is defined as follows:

$$
\text{Bind}_\pi(c, i, c') = \\
\langle B \mu(A_0, \ldots, A_{i-1}, A_i', \ldots, A_n'), .\text{locals}(A_0', \ldots, A_{i-1}', A_i', \ldots, A_n'), b'' \text{stloc } j b''', \\
.\text{env}((e_0, v_0), \ldots, (e_m, v_m), (e_0' + k + 1, v_0'), \ldots, (e_m' + k + 1, v_m')) \rangle \\
$$

where $i \in 0..n$,

$c = \langle B \mu(A_0, \ldots, A_n), .\text{locals}(A_0', \ldots, A_{i-1}', A_i', \ldots, A_n'), b, .\text{env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle$,  

$c' = \langle A_i \mu(A_0', \ldots, A_n'), .\text{locals}(A_0'', \ldots, A_{i-1}'', A_i'', \ldots, A_n'') b', \\
.\text{env}((e_0', v_0'), \ldots, (e_m', v_m')) \rangle$. 

\[ j = k + k' + 2, \]
\[ \Pi = \{ (\text{arg}, 0) \rightarrow (\text{arg}, i), \ldots, (\text{arg}, n') \rightarrow (\text{arg}, i + n'), \]
\[ (\text{loc}, 0) \rightarrow (\text{loc}, k + 1), \ldots, (\text{loc}, k') \rightarrow (\text{loc}, k + k' + 1) \}, \]
\[ \Pi' = \{ (\text{arg}, i + 1) \rightarrow (\text{arg}, i + n' + 1), \ldots, (\text{arg}, n) \rightarrow (\text{arg}, n + n' - 1), \]
\[ (\text{arg}, i) \rightarrow (\text{loc}, j) \}, \]
\[ \Pi \vdash b' \rightarrow b'', \]
\[ \Pi' \vdash b \rightarrow b'' \]

The idea behind this definition is that the code generated should behave as if the method defined by \( c \) is executed and the \( i \)th argument is the result of invoking the method \( c' \). Of course we should guarantee that the return type of \( c' \) is \( A_i \).

Because \( c' \) may require input arguments, we should lift its arguments into the signature of the resulting code value. This process of lifting arguments into the outermost signature is similar to lambda lifting [22] and is used to guarantee that there are no open terms inside a code value.

Our choice has been to replace the \( i \)th argument of the signature with the arguments of the signature of \( c' \). We could have chosen to simply append the new arguments at the end of the signature, though in this way it’s easier to understand the meaning of arguments. Consider for instance the following invocation:

\[ c(v_0, \ldots, v_{i-1}, c'(v_0', \ldots, v_{n'}'), v_{i+1}, \ldots, v_n) \]

If we read the arguments left to right we get the same order of the signature generated by our function.

We can’t inline the definition of \( c' \) where the \( i \)th argument of \( c \) is used because \( c' \) could produce side effects that can cumulate if its code is executed more than once. Moreover \( c' \) could be a method with a large number of instructions and replacing each instruction \( \text{ldarga} \ i \) would lead to a code bloat. Thus we take the definition of \( c' \) and we do a sort of inlining: the value produced by \( c' \) is stored into a local variable which is used instead of the \( i \)th argument in the body of \( c \).

Although the operation is simple, we should pay attention to rewire the access to local variables and arguments properly. In particular the local variables of \( c' \) are shifted by \( k + 1 \) and its arguments by \( i - 1 \); the body is changed accordingly by \( \Pi \). The body of \( c \) should be updated because the arguments with index greater than \( i \) should be renamed because of lifting and all the accesses to argument \( i \) should be replaced with accesses to a new local variable that contains the value calculated by \( c' \): the transformation \( \Pi' \) is used perform the update.
3.4.4 \textit{Bind}_x Inlining of Code Values

The last function we define for binding code values is \textit{Bind}_x whose purpose is to offer control over inlining of code. This function is crucial to express code combinators that otherwise the \textit{Bind} operator wouldn’t be able to express.

The idea is that if we assume a function type as input argument, then the method body will contain only invocations to the function object. As discussed in section 1.3.2 function objects are characterized through classes or interfaces which exposes a method whose name is known. In stack based machines it is fairly easy to find how a function object passed as argument is used: the invocation requires that the argument is loaded on the operands’ stack before the arguments. After the code responsible for loading the stack there is the \texttt{callvirt} instruction which performs the invocation.

A function type in our formal framework is a class containing a method named \texttt{Invoke} with the appropriate signature. If an argument of such type is bound with a code value \(c’\) with the same signature of \texttt{Invoke} and the argument is used only to invoke that method, then each invocation is replaced by the body of \(c’\). The function is defined as follows:

\[
\text{\textit{Bind}}_x(c, i, c’) = \begin{cases} 
\langle B \mu(A_0, \ldots, A_{i-1}, A_i, \ldots, A_n), \\
\text{.locals}(A_0’, \ldots, A_{k’}, A_0’’, \ldots, A_{k’’}, A_{i’’}’, \ldots, A_{n’’}) b’’ \\
\text{.env}((e_0, v_0), \ldots, (e_m, v_m), (e_{0’} + k + 1, v_{0’}), \ldots, (e_{m’} + k + 1, v_{m’})) \rangle 
\end{cases}
\]

where \(i \in 0..n\),

\[
c = \langle B \mu(A_0, \ldots, A_k), \text{.locals}(A_0’, \ldots, A_{k’}) b, \text{.env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle,
\]

\[
c’ = \langle B’ \mu(A_0’’, \ldots, A_{n’’}), \text{.locals}(A_0’’, \ldots, A_{n’’}) b’’, \text{.env}((e_{0’’}, v_{0’’}), \ldots, (e_{m’’}, v_{m’’})) \rangle,
\]

\[
A_i = \text{class } \texttt{cl},
\]

\[
B’ \text{.Invoke}(A, A_1’, \ldots, A_{n’’}) \in \text{dom}(\text{methods}(\texttt{cl})),
\]

\[
B’ \text{.Invoke}(A, A_1’, \ldots, A_{n’’}) \in \text{StaticMethods},
\]

\[
\prod = \{ (\text{loc}, 0) \rightarrow (\text{loc}, k + 1), \ldots, (\text{loc}, k’) \rightarrow (\text{loc}, k + k’ + 1),
\]

\[
(\text{arg}, 1) \rightarrow (\text{loc}, k + k’ + 1), \ldots, (\text{loc}, n’) \rightarrow (\text{loc}, k + k’ + n’) \},
\]

\[
\prod’ = \{ (\text{arg}, i) \rightarrow (\text{inline}, k + k’, b’’’) \},
\]

\[
\prod’ + b’ \rightarrow b’’’,
\]

\[
\prod’ + b \rightarrow b’’
\]

The function removes the \(i\)th argument from the signature of \(c\). Type \(A_i\) should be a class that contains an instance method called \texttt{Invoke}. There is another assumption that is less explicit: the argument \(i\) should be used only to call the \texttt{Invoke} method, otherwise the transformation of body \(b\) will fail.
The signature of $c'$ should be the same of \texttt{Invoke} so that we can do some sort of inlining of its invocations.

Local variables are extended to make room for the local variables used by $b'$. Moreover local variables are added to use instead of the input arguments of $b'$.

The $\Pi$ transformation defines how $b'$ should be converted in $b'''$. The environment of the resulting code contains the values needed by $c$ and $c'$. Local variable indexes in the environment coming from $c'$ are shifted according to other transformations.

Finally $\Pi'$ uses the \textit{inline} mapping to perform the inlining of the transformed body $b'''$ into $b$.

### 3.4.5 \textbf{Bind Function}

In previous sections we have introduced the three functions $\text{Bind}_v$, $\text{Bind}_\pi$ and $\text{Bind}_x$. These functions are used to define the $\text{Bind}$ function which is the fundamental operation to manipulate code values.

Before introducing $\text{Bind}$ we should discuss how free variables are modeled. As discussed in section 2.3.3 binding requires some notion of names to share arguments. We introduce the $\text{Free}$ set whose elements represents an argument that is not bound.

In this definition of $\text{Bind}$ we assume a slightly different version of placeholders for open arguments: in our implementation objects of class \texttt{Free} have an identity which persists across different bindings; in the formal model the identity of \texttt{Free} elements is preserved only for the current invocation of $\text{Bind}$. Of course if we arguments cannot be shared across different $\text{Bind}$ we should explicitly propagate the sharing.

The $\text{Bind}$ function is the following:

$$
\text{Bind}: \text{StoreType} \times \text{Code} \times \bigcup_i \text{(val} \cup \text{Free} \cup \text{Code}) \rightarrow \text{Code}
$$

We have defined $\text{Bind}$ as a function with a variable number of arguments. Perhaps our definition should ensure that its definition matches the signature of the code value to which arguments should be bound.

$\text{Bind}$ has a recursive definition which performs binding of input arguments in a right to left order. First of all the function transforms the code value to resolve argument sharing. We rely on the function $\text{Share}$ for this task:

$$
\text{Share}: \text{Code} \times \bigcup_i \text{(val} \cup \text{Free} \cup \text{Code}) \rightarrow \text{Code} \times \bigcup_i \text{(val} \cup \text{Free} \cup \text{Code})
$$

$\text{Share}(c, v_0, \ldots, v_n) = (\text{Share}(c', v_0, \ldots, v_i, v_{i+1}, \ldots, v_n), v_0, \ldots, v_i, v_{i+1}, \ldots, v_n)$
where \( i \in 0..n, j \in 0..n, i < j, v_i, v_j \in \text{Free} \), \( v_i = v_j \), \( A_i = A_j \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
c' = < B \mu(A_o, ..., A_{i+1}, A_{i+2}, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b' \cdot \\
\quad \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
\Pi = \{ (\text{arg}, i) \rightarrow (\text{arg}, i) \},
\]
\[
\Pi + b \rightarrow b'
\]
\[
\text{Share}(c, v_0, ..., v_n) = (c, v_0, ..., v_n) \quad \forall i, j \in 0..n, v_i, v_j \in \text{Free}, v_i = v_j \Rightarrow i = j
\]

\( \text{Share} \) function simply eliminates arguments bound to the same element of \( \text{Free} \). Its definition is recursive and the base case happens when there are no more arguments to merge. We keep track of the values eliminated so that the association between values and arguments of the code value is preserved.

Now we define \( \text{Bind} \):

\[
\text{Bind}(\Sigma, c,v_i, ..., v_n) = \text{Bind}(\Sigma, \text{Share}(c,v_0, ..., v_n))
\]
where \( \forall i \in 0..n \), \( v_i \in \text{Free} \), \( \exists i, j \in 0..n, v_i, v_j \in \text{Free}, v_i \neq v_j \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
\text{Bind}(\Sigma, c, v_0, ..., v_n) = c
\]
where \( \forall i \in 0..n, v_i \in \text{Free} \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
\text{Bind}(\Sigma, c, v_0, ..., v_n) = \text{Bind}(\text{Bind}(\Sigma, c, i, v_i), v_0, ..., v_i-1, v_i+1, ..., v_n)
\]
where \( i \in 0..n, v_i \in \text{Free}, \forall j \in 0..n, v_j \in \text{Free} \Rightarrow i \leq j \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
\Sigma \vdash v_i : A_i
\]
\[
\text{Bind}(\Sigma, c, v_0, ..., v_n) = \text{Bind}(\text{Bind}(\Sigma, c, i, v_i), v_0, ..., v_{i-1}, v_i, v_{i+1}, ..., v_n)
\]
where \( i \in 0..n, v_i \in \text{Code}, \forall j \in 0..i, v_j \in \text{Free} \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
v_i = < A_i \mu(A_o'', ..., A_{n''}) \cdot \text{locals}(A_o''', ..., A_k''') b' \cdot \\
\quad \text{env}((v_0', u_0'), ..., (e_m', u_m')) >,
\]
\[
\varphi \in \text{Free}, \forall j \in 0..n', \varphi \vdash \varphi \Rightarrow h = l, \forall h \in 1..n', l \in 0..n, \varphi \neq v_i
\]
\[
\text{Bind}(\Sigma, c, v_0, ..., v_n) = \text{Bind}(\text{Bind}(\Sigma, c, i, v_i), v_0, ..., v_{i-1}, v_i, v_{i+1}, ..., v_n)
\]
where \( i \in 0..n, v_i \in \text{Code}, \forall j \in 0..i, v_j \in \text{Free} \)
\[
c = < B \mu(A_o, ..., A_n) \cdot \text{locals}(A_o', ..., A_k') b \cdot \text{env}((v_0, u_0), ..., (e_m, u_m)) >,
\]
\[
v_i = < B \mu(A_o'', ..., A_{n''}) \cdot \text{locals}(A_o''', ..., A_k''') b' \cdot \\
\quad \text{env}((v_0', u_0'), ..., (e_m', u_m')) >,
\]
\[
A_i = \text{class } cl,
\]
\[
B \text{Invoke}(A_o'', ..., A_{n''}) \in \text{dom}(\text{methods}(cl))
\]
\[
B \text{Invoke}(A_o''', ..., A_{k'''}) \in \text{StaticMethods}
\]
We assume that the conditions to apply a definition of \textit{Bind} are checked in order in order to make the definition more readable.

The function is defined recursively. The base step is when the values are all members of \textit{Free} and no sharing is possible. In this case we simply return the code value. While the same element of \textit{Free} is repeated in \( v_1, \ldots, v_n \) \textit{Share} is applied to generate the code that shares arguments.

Then the values are searched in a left to right order for values that aren’t marked free. When a non-free value is found the appropriate transformation is applied depending on the type of the value.

The recursion is well founded: \textit{Share} reduces the number of shared \textit{Free} elements until no more are left; after the elimination of shared arguments the function reduce by one the number of values that aren’t \textit{Free} and possibly introduce only fresh elements of \textit{Free} in the new invocation.

3.4.6 Executing Code Values

Since we are interested in proving properties about the execution of code generated by means of binding, we should define how the execution environment should execute a code value.

We extend the \texttt{call} instruction to allow execution of code values. Because of Lemma 3.1 and Lemma 3.2 we can consider only static methods in our discussion. For each instance method we can consider its static version.

The new rule is the following:

\[
\text{sig} = B \mu(A_0, \ldots, A_n) \\
\text{c} = \langle \text{sig}, .\text{locals}(A_0', \ldots, A_n') b, .\text{env}((v_0, u_0), \ldots, (v_m, u_m)) \rangle \\
\sigma = (h_0, s_0) \mapsto v_i \mapsto (h_{i+1}, s_{i+1}) \quad \forall i \in 0..n \\
(h_{m+1}, s_{m+1}).\text{args}(v_0, \ldots, v_n).\text{locals}(l_0, \ldots, l_i) \mapsto b \mapsto v \cdot (h', s' f' r') \\
l_{j} = u_{j} \quad \forall j \in 0..m \\
\] 

\[ \sigma \vdash a_0 \ldots a_n \text{call c} \mapsto v \cdot (h', s') \quad \text{(Eval Code)} \]

The rule has the same structure of (Eval \texttt{call}), although in this case we should consider the values we have stored into the environment. Thus we init the locals that have been introduced during the code generation process with the value saved in \texttt{env}.

We should also extend the typing judgment with a rule similar to (Body \texttt{call}):
3. A Formal Model for Code Type

\[ \text{sig} = \text{B } \mu(A_0, \ldots, A_n) \]
\[ c = \langle \text{sig}, \text{.locals}(A_0', \ldots, A_i') \rangle b, \text{.env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle \]
\[ \forall i \in 0..n \quad \text{Fr} \vdash a_i : A_i \]
\[ \text{Fr} \vdash a_0 \ldots a_n \text{ call } c : \text{B} \quad \text{(Body call)} \]

Note that we should guarantee that the type of the value stored in the environment is compatible with the local we use to store it; otherwise the type safety theorem wouldn’t be true anymore.

We notice that the environment is extended only by \textit{Bind}, which relies on the conformant judgment for values to associate a local variable to its value stored in the environment. Thus the local variables in the stack frame are initialized properly.

### 3.5 Interpreters and Compilers

So far we have assumed that a way to understand how the code generated through \textit{Bind} works is to think to a method invocation whose constituents are values and, when a code value is bound to an argument, method calls.

Let us consider the following method:

\begin{verbatim}
int32 add(int32, int32) ∈ StaticMethods
M = int32 cl::add(int32, int32)
methods(cl)(int32 add(int32, int32)) =
   .locals()
  ldarg 0
  ldarg 1
  add
\end{verbatim}

Now we can generate the three way add function by means of code values and \textit{Bind}:

\begin{verbatim}
a = methodof(int32 cl::add(int32, int32))
ϕ, ϕ, ϕ ∈ Free
C = Bind(∑, a, ϕ) = Bind(∑, Bind₃(a, 0, a), ϕ, ϕ, ϕ) = Bind(∑, a’, ϕ, ϕ, ϕ) = a’
a’ = Bind₃(a, 0, a) =
   \langle \text{int32 } \mu(\text{int32, int32, int32}), \text{.locals(int32)} b, \text{.env()} \rangle
b =
   ldarg 0
   ldarg 1
   add
   stloc 0
   ldloc 0
\end{verbatim}
To understand the behavior of the code generated through \textit{Bind} we say: because we\textacute{'}ve bound the first argument of \texttt{add} with a function which returns an integer value we would expect that the following equivalence hold:

\[ \sigma \vdash a_0 \ a_1 \ a_2 \ \texttt{call} \ c \mapsto \nu \cdot \nu' \iff \sigma \vdash a_0 \ a_1 \ \texttt{call} \ M \ a_2 \ \texttt{call} \ M \mapsto \nu' \]

If we use a functional notation we expect that the following is true\textsuperscript{16}:

\[ \texttt{add}(\texttt{add}(a_0, a_1), a_2) = \texttt{Bind}(\sum, a, \texttt{Bind}(\sum, a, \varphi_0, \varphi_1), \varphi)(a_0, a_1, a_2) \]

We observe that if we syntactically replace the string “\texttt{Bind}(\sum, a)” with “\texttt{add}(“ and we replace the free placeholders with the corresponding values we get exactly the same expression on the left. Thus the code generated with \textit{Bind} is in some sense a compiled version of the expression on the left.

Our aim is to prove that our intuition about this property of \textit{Bind} is true for each application of the operator. This property is fundamental because makes the operator usable by the programmer. Now we try to characterize the property more formally and prove that holds.

The idea is that code values are created out of an existing method by means of \textit{methodof} or by means of \textit{Bind}. We introduce the notion of ground form which is when a code value is expressed only in terms of applications of \textit{Bind} and code values obtained with \textit{methodof}.

\textbf{Definition 3.2:} A code value \( c \in \text{Code} \) is said to be in ground form if

\[ c = \texttt{Bind}(\sum, \textit{methodof}(B \ c::m(A_1, \ldots, A_n), \nu_0, \ldots, \nu_n), \forall i \in 1..n, \nu_i \in \text{Code} \Rightarrow \nu_i \text{ is in ground form.} \]

It is worth noting that code values obtained with \textit{methodof} can be expressed in ground form by a simple application of \textit{Bind}’s definition:

\[ M = B \ c::m(A_1, \ldots, A_n) \]

\[ \textit{methodof}(M) = \texttt{Bind}(\sum, \textit{methodof}(M), \varphi_1, \ldots, \varphi_n), \varphi \in \text{Free} \]

Not every code value can be expressed in ground form; this is because the ground form imposes that the code value whose arguments should be bound must be obtained through \textit{methodof}. The notion of ground form helps us to characterize a subset of the \textit{Code} set which we are interested in.

\[ \textsuperscript{16} \text{We have used the definition of } \textit{Bind} \text{ to say that } \texttt{Bind}(\sum, a, \varphi_0, \varphi_1) = a. \]
**Proposition 3.1:** It is possible to build code values that cannot be expressed in ground form using only code values of the form $\text{methodof}(M)$ and a number of application of $\text{Bind}$.

*Proof.* Let $M = \text{int} \ c::\text{inc(int)}$ be a method with the following body:

```
.locals() ldarg 0 ldc.i4 1 add
```

$M$ is the increment function. Consider the code value (in ground form) obtained by applying the increment function to itself $n$ times:

$$c = \text{Bind}(\sum, \text{methodof}(M), \text{Bind}(\sum, \text{methodof}(M), \text{Bind}(\ldots, \phi)))$$

We fix $n$ so that the number of instructions of $c$ is larger than whatever code value obtained with $\text{methodof()}$. Thus there is no method $M'$ such that $\text{method}(M') = c$. Consider the following code value:

$$c' = \text{Bind}(\sum, c, 1)$$

The code value $c'$ cannot be expressed in ground form because, by definition of $c$, there isn’t a method $M'$ such that $\text{methodof}(M') = c$. This concludes the proof.

The ground form of a code value is a combination of $\text{Bind}$ applications, values and $\text{methodof}$ applications. Like any function application we can picture it as a tree whose depth depends on the maximum number of nested applications of $\text{Bind}$. The leaves of the tree are either values or results of $\text{methodof}$.

Note that in the ground form we always assume that code values obtained through $\text{methodof}$ are forced to be enclosed in a $\text{Bind}$ call with all arguments bound to free elements to make explicit their unbound arguments.

![Figure 3.1: Ground form tree.](image)

Let us consider the code value $c$ of the previous example. Its ground form coincide with its definition because $a = \text{methodof}(M)$. The corresponding tree is shown in Figure 3.1: each internal node in the tree represents an application of
**Bind.** The first child is the code value being manipulated\(^{17}\) and the others are values bound to arguments. Elements of set Free represent arguments that are still unbound in the generated code.

We assume that the code values in the leaves are obtained from static methods. Lemma 3.1 and Lemma 3.2 guarantee that we don’t loose generality in making this assumption.

From the ground form of a code value it is possible to derive an expression which we call the *interpreted form*. We need a helper function that, given the ground form of a code value, produces a sequence of BIL instructions.

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
\text{ldarg } j \text{ Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i+1, j+1, k)
\]
\[
\text{where } v \in \text{Free}, \forall l \in 0..n, v_l = v_i \Rightarrow i \leq l
\]

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
\text{ldarg } l \text{ Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i+1, j, k)
\]
\[
\text{where } v \in \text{Free},
\]
\[
\exists l \; v_l = v_i \forall m \in 0..n, v_l = v_m \Rightarrow l \leq m
\]

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
\text{ldloc } k \text{ Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i+1, j, k+1)
\]
\[
\text{where } v \in \text{Val}
\]

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
\text{Interpret}(v_i, 0, j, k) \text{ Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n),
\]
\[
i+1, j', k'
\]
\[
\text{where } v \in \text{Code},
\]

**Bind** would apply **Bind**\(_x\) to **v\(_i\)**.

\(j'\) and \(k'\) the maximum \(j\) and \(k\) generated by the first **Interpret** application

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
\text{newobj void } cl::\text{ctor}()
\]
\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i+1, j, k)
\]
\[
\text{where } v \in \text{Code},
\]

**Bind** would apply **Bind**\(_x\) to **v\(_i\)**.

\(\sigma\)\(_+\) \(a_1 \ldots a_n\) call **v\(_i\)** \(\mapsto\) \(v \cdot \sigma\)
\[
\sigma\)\(_+\) \(a_1 \ldots a_n\) callvirt \(B' \ cl::\text{Invoke}(A_1, \ldots, A_n) \mapsto v \cdot \sigma'
\]

\[
\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \ c::m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =
\]
\[
call B c::m(A_0, \ldots, A_n)
\]

\(^{17}\) We have omitted \(\Sigma\) from the arguments shown in the tree.
where \( i = n + 1 \)

The function \texttt{Interpret} may look ugly; still it simply tries to capture the process of converting a code value in ground form into a tree of function applications. It corresponds to the syntactic transformation we have spotted in the \texttt{add} example.

The indices \( i, j \) and \( k \) are used to iterate over the arguments and to keep track of argument and local variable indices used. We distinguish between the two possible bindings of code values to arguments. In the first case we assume that \texttt{Bind} would apply \texttt{Bind}_{\pi} and in this case we recursively continue in our generation process. In the second case we assume that a splice-in would occur; we postulate the existence of a class with an \texttt{Invoke} method whose execution is equivalent to our code value\textsuperscript{18}.

\begin{definition}
Let \( c \in \texttt{Code} \) a code value expressed in ground form, the interpreted form of \( c \) is a code value whose signature is the same of \( c \) and the body is built as follows:
\begin{itemize}
  \item \texttt{.locals} \( (A_0', ..., A_k') \) where \( A_0', ..., A_k' \) are derived as in the definition \texttt{Bind}
  \item The body is \texttt{Interpret}(\( c, 0, 0, 0 \))
  \item The environment is \texttt{.env} \( ((0, v_{e0}), ..., (k, v_{ek})) \) where \( \forall i \in 0..k, e_i \in 1..n \) and \( v_i \in \texttt{Val} \)
\end{itemize}

We would have preferred to define a method as the interpreted form of a ground form. Unfortunately we should cope with values that are in the environment and should be loaded into local variables. Besides if we use a code value the (\texttt{Eval Code}) rule guarantee that local variables are initialized in this way.

Apart from these technical details the important aspect of the interpreted form is that its body corresponds to the expression that calls a method instead of generating the code. Let us consider the interpreted form in our example of the three ways \texttt{add}:

\begin{align*}
\texttt{Interpret}(&\texttt{Bind}(\sum, a, \texttt{Bind}(\sum, a, \varphi_0, \varphi_1), \varphi_2), 0, 0, 0) = \\
\texttt{Interpret}(&\texttt{Bind}(\sum, a, \varphi_0, \varphi_1), 0, 0, 0) \\
\texttt{Interpret}(&\texttt{Bind}(\sum, a, \texttt{Bind}(\sum, a, \varphi_0, \varphi_1), \varphi_2), 1, 2, 0) = \\
\texttt{ldarg 0 ldarg 1 call M}
\end{align*}

\textsuperscript{18} This is because when \texttt{Bind}_{\pi} is applied the argument of the function is intended to be a function object: the invocation of the method \texttt{Invoke} corresponds to invoke the method which implements the method.
Interpret(Bind(∑, a, Bind(∑, a, ϕ₁, ϕ₂), ϕ₂), 1, 2, 0) =
1darg 0
1darg 1
call M
1darg 2
call M

Now that we have captured the essence of our intuitive interpretation of binding we can state and give a proof the correctness theorem which essentially states that Bind operates as we would expect generating the code which preserves the semantics of the interpreted version. The theorem is closely related to the first Futamura projection [23]: Bind is an operator which outputs a code equivalent to a particular application of function whose arguments can be the result of another application. Thus the Bind operation can be considered a tool to implement a partial evaluator.

**Theorem 3.2 (Interpreter theorem):** Let \( c \in Code \) a code value in ground form, let \( c' \in Code \) be the interpreted form of \( c \); the following equivalence holds:

\[
\sigma \vdash a_0 \ldots a_n \text{call } c \Rightarrow v \cdot \sigma' \iff \sigma \vdash a_0 \ldots a_n \text{call } c' \Rightarrow v \cdot \sigma'
\]

The proof is based on induction of the tree’s depth associated to the ground form. Details of the proof are reported in appendix 8.3.

The Interpreter theorem is crucial to the design of Bind because it states that our intuition about the behavior of the code generated through the operator is correct. It also defines the limits of this interpretation: the theorem holds only for code values that can be expressed in ground form. To fully understand the scope of this restriction we consider now the code values that can’t be expressed in ground form. Consider the following binding:

\[
c = \text{Bind}(\Sigma, \text{methodof}(B c::m(A₀, A₁), \phi, v₀), v₁)
\]

We could be tempted to say that the binding is equivalent to the ground form:

\[
\text{Bind}(\Sigma, \text{methodof}(B c::m(A₀, A₁)), v₁, v₀)
\]

The intuition is that we simply change the order in which values are bound to a code value. Now suppose that both \( v₀ \) and \( v₁ \) are code values that produce some side effect. Because Bind composes code values by prefixing one to each other (by means of Bind) the application order in this case may lead to a different order in which side effects take place.

In particular the ground form we’ve supposed being equivalent to \( c \) will output a code where the code coming from \( v₁ \) is before the code from \( v₀ \); in \( c \) the
same code values appears in the opposite order. Thus in general the two expressions aren’t equivalent.

If we have a look at the structure of $c$ we may notice that it isn’t in ground form because the code value whose arguments are bound isn’t of the form `methodof`. Thus we can’t build the interpreted form and get the equivalence of the generated code.

Nonetheless we can still extend the result to all code values: a code value isn’t in ground form if the target of some binding inside it is the result $c$ of `Bind`. If $c$ is in ground form we can consider the theorem to understand its behavior, and then we may consider the method equivalent to it and apply again the theorem.

**Corollary 3.1:** Let $c, d \in \text{Code}$, $c = \text{Bind}(\Sigma, d, v_0, \ldots, v_n)$, $d$ is in ground form, $\forall i \in 0..n, v_i \in \text{Code} \Rightarrow v_i$ is in ground form. Let also $M = \text{B cl}::m(A_0, \ldots, A_n)$ such that $\sigma^{+} \cdot \text{call} \ M \mapsto v \cdot \sigma' \iff \sigma^{+} \cdot \text{call} \ d \mapsto v \cdot \sigma'$. The following equivalence holds:

$$\sigma^{+} \cdot \text{call} \ c \mapsto v \cdot \sigma' \iff \sigma^{+} \cdot \text{call} \ \text{Bind}(\Sigma, \text{methodof}(M), v_0, \ldots, v_n) \mapsto v \cdot \sigma'$$

**Proof.** The proof is straightforward.

This corollary simply states that we can consider the result of `Bind` of a ground form as if it has been obtained through `methodof` of a special method. Thus given a code value that cannot be expressed in ground form we can define an equivalent code value that in some sense can be considered its ground form. Thus code values that cannot be expressed in ground form don’t retain enough information to control further binding of arguments.

In the rest of this section we state a theorem that grant us that `Bind` can generate only type safe code. This result is quite straightforward, though it is important because if the programmer should emit the instructions may it can generate code that cannot be properly typed leading to a runtime error.

**Theorem 3.3:** Let $c = < B \ \mu(A_0, \ldots, A_n), \ .\text{locals}(A_0', \ldots, A_k') \ b, \ .\text{env}((e_0, v_0), \ldots, (e_m, v_m))>$, then $\ .\text{args}(A_0, \ldots, A_k).\text{locals}(A_0', \ldots, A_k') \vdash b : B$.

The proof of the theorem is based on a structural induction on the definition of `Bind`, `Bind_{\mu}`, `Bind_{\pi}` and `Bind_{\chi}`. This theorem ensures that the transformations we perform on method bodies are consistent to those made to the signatures.
3.6 Generating Code from Combinators

So far we have focused our study on the behavior of code values. Our aim has been to show that the code generated by the particular transformation performed by $\textit{Bind}$ is correct with respect the model we have described in chapter 1.

All our statements have been of the conditioned on having some method that we use to get some functionality through $\textit{methodof}$. Now we are interested in understanding if we can define a finite set of methods that allow us to generate all kinds of programs\(^{19}\).

In this section we show a simple example of how compile a simple imperative language starting from a finite number of combinators. The result is relevant because it guarantees that we can build a general purpose compiler based on a fixed set of methods that should be available into the compiler. Again this result closely recall the second Futamura projection \([23]\): we can exploit code values and $\textit{Binding}$ to generate a compiler for a specific language.

We consider the IMP language:

\[
\begin{align*}
\textit{Expr} & ::= \textit{Var} + \textit{Exp} \mid \textit{Var} \\
\textit{Cmd} & ::= \textit{Var} = \textit{Expr} \mid \text{if} (\textit{Expr}) \textit{Cmd} \textit{else} \textit{Cmd} \mid \textit{while} (\textit{Expr}) \textit{Cmd} \mid \textit{Cmd} ; \textit{Cmd}
\end{align*}
\]

We introduce a code value for each command and expression:

\[
\begin{align*}
\textit{seq} = & < \textit{void } \mu (\textit{class Cmd, class Cmd}) , \\
& .\textit{locals}() \\
& \text{ldarg} 0 \text{ callvirt void Cmd::Invoke()} \\
& \text{ldarg} 1 \text{ callvirt void Cmd::Invoke()}, \\
& .\textit{env}() > \\
\textit{While} = & < \textit{void } \mu (\textit{class Expr, class Cmd}) , \\
& .\textit{locals}() \\
& \text{ldarg} 0 \text{ callvirt int32 Expr::Invoke()} \\
& \text{ldarg} 1 \text{ callvirt void Cmd::Invoke()} \\
& \textit{while}, \\
& .\textit{env}() > \\
\textit{If} = & < \textit{void } \mu (\textit{class Expr, class Cmd, class Cmd}) , \\
& .\textit{locals}() \\
& \text{ldarg} 0 \text{ callvirt int32 Expr::Invoke()} \\
& \text{ldarg} 1 \text{ callvirt void Cmd::Invoke()}
\end{align*}
\]

\(^{19}\) We intend programs whose behavior is the same, of course we can’t generate all the possible combinations of BIL instructions.
We assume that both `Expr` and `Cmd` have an `Invoke` method with the appropriate signature. These classes are used like delegates and we rely on `Bind` to replace the invocations of `Invoke`. The class `Var` is used to represent a variable whose content is kept in the field `val`.

Given an IMP program we can generate the compiled version by means of `Bind` and the code values defined above. Let us consider the following command:

\[ v = 2; \text{if } (v) \text{ then } v = 1 \text{ else } v = v + 1 \]

The compiled version of the program is calculated by the following composition of code values:

\[
\begin{align*}
    a &= \text{Bind}(\Sigma_{\text{assign}}, v, \text{Bind}(\Sigma_{\text{id}}, 2)) \\
    t &= \text{Bind}(\Sigma_{\text{assign}}, v, \text{Bind}(\Sigma_{\text{id}}, 1)) \\
    e &= \text{Bind}(\Sigma_{\text{assign}}, v, \text{Bind}(\Sigma_{\text{add}}, v, \text{Bind}(\Sigma_{\text{id}}, 1))) \\
    \text{Bind}(\Sigma, \text{seq}, a, \text{Bind}(\Sigma, \text{If}, v, t, e))
\end{align*}
\]

We rely on a boxed object to store the variable `v` of the IMP program.

The main problem in defining compilers in this way is how to represents types. A more important problem is how to access types that have been defined outside the program. As we’ve discussed in section 2.4.3 recursion can be expressed by introducing an additional argument which is a reference to the code value itself.
Anyway we have been able to define a finite (small) number of code combinators that are general enough to compile a Turing equivalent language. Of course the quality of generated code wouldn’t be really exciting though it is possible to spot several patterns that can be optimized during the code generation process.

### 3.7 Considerations about Code and Bind

In this chapter we have presented a formalization of the code type and the Bind operator. We rely on a model of an execution environment which is a substantial subset of the CLR. How is related this result with the real implementation?

First of all we should mention the fact that not every aspect of Code has been modeled: in the implementation we have a notion of converting a code value into an executable object which is exposed as a delegate; in the model a code value doesn’t keep track of how it has been obtained while in the implementation we retain the structure of the Bind calls (see chapter 4). We have introduced in the execution model a special version of the call instruction to execute code values; besides creation and Bind of code value are outside BIL.

The implementation represents code values as instances of Code class, Bind is a method of this class and there is a method should be called to be able to get the executable object corresponding to the code value.

Nonetheless we were interested in the properties of the code generated using Bind, and the model has turned out to be expressive enough to expose all the details needed in our investigations.

Besides we should point out that we have extended our results to instance methods relying on Lemma 3.1 and Lemma 3.2. Moreover in the real implementation there is no such easy interchange between static and instance methods: access control to class members may forbid a static method to access a private field of another class.

The definition of $\textit{Bind}_x$ relies on a weaker condition than the one of our implementation. We rely on the notion of delegate to represent functions passed as arguments. A delegate type fulfills all the conditions imposed on the argument’s type in order to apply $\textit{Bind}_x$ though the opposite is not true. However adding delegates to the model would have complicated the model without improving the model. Moreover the condition we have used can be applied also to JVM whereas the notion of delegate is limited to CLR.

Another element that should be taken into account is the role of store type $\Sigma$ that we have used in $\textit{Bind}$. The definition of $\textit{Bind}$ requires that at runtime the
operation should be able to check values’ types. We use the store type as a substitute of the reflection that is available in the execution environment.

Let us consider now how our results can be extended to the real execution engine. First of all we should mention that the \textit{Bind} transformation performs only local changes in method bodies. The consideration is that we substitute something with something equivalent, so that if the methods we use to get the code values are correct then the transformation preserves correctness. Also in the real implementation the only instructions manipulated are the same described in the formal model. This is the reason because the branching model of BIL doesn’t introduce any problem in the transformation.

Besides the implementation should cope with details that aren’t described in the model. To bake a code value into a method it is necessary to calculate the maximum height of the operands’ stack required by the method. We discuss later on the technique we use to calculate this value. Moreover in the model we simply assume that the labels of \textit{while} and \textit{cond} are always fresh to avoid name clashes. Intermediate Language expresses branches as relative offsets, in bytes, between the branch instruction and its target. The code generator should fix all the branch instructions with the appropriate offsets after the transformation.

BIL doesn’t provide a \texttt{ret} instruction as in IL; this may appear a relevant difference that could create problems in our implementation. Fortunately we can simply replace each occurrence of \texttt{ret} in method body with a branch and a load which is very close to the BIL solution which leaves the return value on top of the operands’ stack, though it is implicit.

Both the model and the implementation don’t consider exception handling. We believe that we can trivially extend our approach to code with exception handling. In the formal model we should consider blocks of instructions organized in trees rather than simple lists of instructions. In the implementation we can keep close to the current implementation because exceptions are represented as ranges over a single list of IL instructions.

So far we have shown how \textit{Bind} makes it easier to compile partial applications. Besides it is less easy, though possible, to write compilers relying on a fixed set of combinators. The main problem we have to face is that types are sealed in signatures: while we restrict ourselves to use a fixed set of types for the target language we get in troubles when we want to compile arbitrary types. If the execution engine offers support for generic types, like the generic CLR described in [37], we can extend our rules to combine the generative power on type of parametric polymorphism with the ability of generating method
bodies. Unfortunately we can’t rely on parametric polymorphism implemented at language level as in GJ [9].

In the following chapters we consider applications of code values and *Bind*. We will switch back to the notation used in chapter 1 to express code values and binding. Still our model allows us to just write binding expressions without having to explain their output in term of code.
4 Performance and Implementation

4.1 Code Bricks API

Code generation by means of Bind requires that one or more code values are obtained from existing methods. Thus the natural way of exposing the code generation mechanisms is as an extension of the reflection. Both Java and CLR exposes information about types through a set of classes; there is a class that describes each component of a type: fields, constructors, methods and so on.

The natural way of building code values seems to be having some way to ask for the code value associated with the descriptor of the method – that would be instance of MethodInfo on CLR (in System.Reflection) and Method on JVM (in java.lang.reflect).

We define our implementation an API because it is more than a mechanism: it is directly exposed to programming languages as a set of classes. Besides we don’t provide any special language to use it. Of course the API may give the illusion of being an extension of the language, because in STEE types are elements shared between languages and the execution environment.

The API is built around the Code class whose principal members are listed below:

```csharp
class Code {
    // methodof(m)
    public Code(MethodInfo m) {}

    // Bind
    public Bind(params object[]) {}

    // Signature
    public ParameterInfo[] GetParameters() {}
    public Type ReturnType { get {} }

    // Environment
    private object[] environment;

    // Body
    private Type[] locals;
    private ILInstr[] body;
}
```
The first constructor of the class corresponds to the `methodof` function introduced in section 3.3. The code value is built starting from the body and the signature of the method described by the `MethodInfo` object passed as argument.

The `Bind` method implements the `Bind` function; its signature has already been discussed in section 2.3.1.

The `GetParameters` method returns an array of parameters’ descriptors, which are objects of `ParameterInfo` class. This method together with the property `ReturnType` allows accessing the signature of the code value.

We know that the signature is only the first component of a code value. Though we will discuss later in more detail the data structures used for representing the body and the environment of a code value we have included three private variables to hold the values closed with `Bind` (the variable environment); the type of the local variables of the body and its instructions.

The set of objects of class `Free` represents the elements of the `Free` set introduced in the model. The use of the `Free` property has been discussed in section 2.3.3. This property it isn’t strictly needed: we could have used `new Free()` instead of `Code.Free`. Besides the most common situation is that parameters aren’t shared; thus through this property we avoid to allocate useless instances of the `Free` class.

### 4.1.1 An Extension to the Model

The `Code` class described so far is almost the same we have discussed and it represents the core of Code Bricks. By the way this has been the signature of the class in the first implementation of the API. During a second implementation we have found a better way to represent code values, as we will discuss later, and we have been able to extend the interface described above with the following members:

```csharp
// Structure inspection
public bool FromMethod { get {} }
public object[] BindArguments { get {} }
```

The first method is able to tell whether the code value has been obtained from a `MethodInfo` or as result of a binding operation. `BindArguments` method allow a sort of inspection of a code value: if it has been built out of a method an array

---

20 `ParameterInfo` is an abstract class defined in the `System.Reflection` namespace and used to describe the arguments of methods.
of Free objects is returned; the argument of the Bind invocation which produced the code value otherwise.

This is not just an additional feature of the implementation. The ability of inspecting the structure of the code value introduces a fairly significant change in the model. In this extension of the original model properties discussed so far still hold: we only add the ability of inspecting how a code value has been obtained in terms of basic constituents.

The most relevant change is that all the code values obtained through Bind application can be deconstructed. This is an interesting property because it offers the ability of performing pattern matching operations on the code values. Of course this is a coarse grain model of pattern matching, and it doesn’t provide any mean of looking inside precompiled methods.

Inspection of code values can be very useful: a component which receives a code value can decide its behavior depending on its structure; it can also check the constituents of the code value to enforce security checks.

Consider for instance a component which requires running a code that should be guaranteed to have a certain structure. It could ask for a code value and then check it has been built in the proper way.

4.1.2 The Switch Problem

In the previous chapter we have shown that with code values and Bind it is even possible to write a simple compiler. This is indeed true, but we only know that we are generating the code equivalent to some program.

How much control does offer Bind over the code generated? Although the model is coarse grain it is possible to output IL programs which aren’t too bad. This is because we are merging chunks of code produced by compilers. The only inefficiencies introduced relate to the introduction of local variables to join blocks of instructions and, of course, a lack of global optimizations.

Nonetheless we had experienced a reasonable control over the output of Bind, though the programmer should figure out how the transformation works to produce the expected code. Moreover it isn’t strictly required to compile methods using a programming language: if necessary it is possible to write the IL method with the desired code, and then read it back using code.

The ability of mixing chunks of precompiled code implies that IL instructions cannot be modified by the programmer. This hypothesis is acceptable because almost all CIL instructions accept a fixed number of arguments. Unfortunately there is a single but relevant exception: the switch instruction. With bind we
cannot build a switch instruction with a number of cases that is known only when the code is generated.

Of course if we know in advance the number of possible branches of our switch instructions we can write a combiner like the following one:

```csharp
delegate void Case(int v);
public static void Switch(int v, Case a, Case b) {
    switch(v) {
    case 0: a(v); break;
    case 1: b(v); break;
    }
}
```

Unfortunately we can’t combine the Switch method with itself to get a bigger switch. Moreover the integer values used in the case instructions should be known at compile time.

From the expressivity standpoint this isn’t a real issue: a switch can be represented as a chain of if-then-else. Besides the switch instruction exists because it allows a faster execution. We introduce an additional constructor to the `Code` class to generate a switch statement with an arbitrary number of case blocks:

```csharp
public Code(params Code[] sw) {}
```

This constructor checks that all the code values contained in `sw` have the same signature. The integer labels are the same of the indices of the array. The schema is like the following:

```csharp
static T Switch(int v, A1 a1, ..., An an) {
    T ret;
    switch (v) {
    case 0: ret = sw[0](v, a1, ..., an) break;
    case 1: ret = sw[1](v, a1, ..., an) break;
    ...
    case m: ret = sw[m](v, a1, ..., an) break;
    }
    return ret;
}
```

We assume that all the code values in `sw` have the signature:

```csharp
T μ(int, A1, ..., An)
```

The IL generated relies on the `switch` instruction which expects a variable number of arguments\(^{21}\):

\(^{21}\) The switch instruction in CLR is quite different from its counterpart in the JVM. The labels are labeled implicitly from 0 to n whereas in the JVM there is a table that should be provided that performs the mapping between labels and offsets.
switch n t₀ ... tₙ

The first integer \( n \) indicates how many branches should be considered. After this number follows a list of integer that represents the targets of the jumps for each case. To execute the instruction the value \( v \) on top of the operands stack and then the control flow is transferred to the instruction at offset \( t_v \).

Binding is not able to produce such code because it just filters the instructions. The sources of IL instructions are compiled methods that can contain just instances of the \texttt{switch} instruction for some value of \( n \). Thus the real problem is the fact that a variable number of arguments is allowed on this particular instruction.

Luckily \texttt{switch} is the only instruction of the runtime with a variable number of arguments. Adding the constructor described above it’s enough to support the generation of switch statements using Code Bricks.

4.2 Reading IL from Assemblies

To implement Code Bricks we should be able to read the IL instructions of the method bodies. Unfortunately the reflection support doesn’t provide any mean to get such information. There is no \texttt{GetILStream()} method in the \texttt{MethodInfo} class neither a method to get the number and the type of the local variables.

Reading the IL instructions from a .NET binary file is not straightforward: the unit of deployment of the code is the assembly. An assembly contains the definition of an arbitrary number of types. Its binary format is based on a variant of Windows PE executable format\(^{22}\).

We have developed a reader of .NET assemblies to be able to access to method bodies [11]. The reader has been written in C# and it has been designed to guarantee the maximum performance in Bind implementation.

We briefly describe the format of an assembly so that we can discuss the performances of code generation with Code Bricks.

The binary format of an assembly \(^{20}\) contains the definition of a set of types together with the code of their methods. Inside the file the metadata are organized into a relational database which comprises a number of tables. Each table contains integers that typically index other tables. There are four heaps that contain the data of the program: strings, identifiers, 128 bits GUIDs\(^{23}\), and other information such as the signatures of the local variables. Finally IL is stored in

\(^{22}\) The PE file format is based on the COFF standard.
\(^{23}\) GUID stands for Globally Unique Identifier. GUIDs are widely used in COM to give unique names to components.
another portion of the file and linked by the Method table which holds all the methods defined into the assembly.

![Figure 4.1: Organization of the Assembly file](image)

Figure 4.1 shows the portion of the assembly file which is needed to locate the information about method bodies. TheTypeDef table contains the types defined in the assembly. References among types and assemblies are expressed using indices to the appropriate tables.

Each string, like the name of the class or the method is stored in the Strings heap. The User Strings heap contains the strings used by the program\(^{24}\).

From the entry in the TypeDef table it is possible to find the list of the methods of the class: the list is simply a contiguous segment of rows of the Method table. A method entry has a relative virtual address (RVA) to the beginning of the stream of instructions of its body. The Signature field refers a location in the Blob heap where is stored the signature of the local variables of the method.

We use memory mapping to have the maximum performance accessing the file. A set of types has been defined to read the memory in the right way. Because of the use of pointers to the memory require the unsafe mode. We have reduced the unsafe code to the minimum by defining a class which exposes the memory as an array of bytes.

The reflection doesn’t provide any support in finding the index into the Method table given the MethodInfo object. Moreover when we access the tables

---

\(^{24}\) There are two heaps for strings because the program strings are mostly expressed in ASCII and can be compressed using the UTF-8 encoding. The strings used by the program are stored in Unicode 16 because the heap may contain all possible strings.
in the file we should find the appropriate reflection objects. The library has been
designed to provide low-level access to the file with helpers that can be used to
fill the gap between the raw file and the reflection objects.

The second implementation of `Bind` is able to transform the stream of bytes
without having to decode the instructions. This greatly improves the perform-
ance of the code generation process. In particular if the execution environment
is aware of the code values: the stream of bytes can be directly sent to the
loader.

### 4.3 Implementation Strategies

Our first implementation of Code Bricks has been quite close to the model we
have described in chapter 1. When a code value was created we read the IL in-
structions from the assembly and stored in memory. Each invocation of `Bind`
returned a new code value containing the instructions obtained by the trans-
formation.

The instructions were kept in memory as objects to simplify the transforma-
tion and keep track of the arguments. When the `MakeDelegate` method was in-
voked the instructions were packed in a byte array ready to be passed to the
loader of the runtime.

Of course the approach wasted a certain amount of resources, mainly mem-
ory, because we had adopted an eager approach to the generation: Code objects
aren’t directly executable, so it isn’t necessary to generate the body at each in-
vocation of `Bind`. Thus we have changed our strategy keeping track of the bind-
ing relations among code values and the code is generated only when the code
value should be converted into an executable object.

In the lazy solution we should solve the problem of combining more than
one transformation at once. In particular we should combine all the transforma-
tions \( \prod \) because a transformation may rename an argument that afterwards is
bound to a value.

Let us consider the following example:

```csharp
class Test {
    public static int add(int i, int j) { return i+j; }
    public static int mul(int i, int j) { return i*j; }
    public static void Main(string[] args) {
        Code a = new Code(typeof(Test).GetMethod("Add"));
        Code m = new Code(typeof(Test).GetMethod("Mul"));
        // \( \lambda x, y, x \cdot x + y \cdot z \)
        Code a3 = a.Bind(a, Code.Free);
        // \( \lambda x, y, z, w \cdot x \cdot y + (z \cdot w) \)
        Code am = a3.Bind(Code.Free, Code.Free, m);
    }
}
```
The code value \( \text{am} \) has associated the tree shown in Figure 4.2. The data structures are linked together in the same way.

The transformations involved in the two \texttt{Bind} operations are indicated in Figure 4.2 and are the following (we indicate also the identities):

\[
\begin{align*}
\Pi_a &= \{ (\text{arg}, 0) \to (\text{loc}, 0), (\text{arg}, 1) \to (\text{arg}, 2) \} \\
\Pi_{a2} &= \{ (\text{arg}, 0) \to (\text{arg}, 0), (\text{arg}, 1) \to (\text{arg}, 1) \} \\
\Pi_{a3} &= \{ (\text{arg}, 0) \to (\text{arg}, 0), (\text{arg}, 1) \to (\text{arg}, 1), (\text{arg}, 2) \to (\text{loc}, 1), (\text{loc}, 0) \to (\text{loc}, 0) \} \\
\Pi_m &= \{ (\text{arg}, 0) \to (\text{arg}, 2), (\text{arg}, 1) \to (\text{arg}, 3) \}
\end{align*}
\]

We note that the transformation \( \Pi_{a3} \) manipulates the second argument which implies that, for the code coming from \( \Pi_a \) the real transformation should be:

\[
\Pi_a = \{ (\text{arg}, 0) \to (\text{loc}, 0), (\text{arg}, 1) \to (\text{loc}, 1) \}
\]

Thus we can combine the transformations so that the global transformation avoids multiple changes of the same instruction.

Each leaf in the tree, bound to a method, acts like a cursor that allows iterating over the instructions of the method. Each internal node represents a binding. We have associated a cursor which returns the next instruction of the body of the related sub-tree.
This approach is iterated and the cursors are initialized with the maps coming from the upper levels of the tree. In this way we avoid multiple renaming of the same instruction.

To fully describe the implementation we should consider how the environment of the code value is implemented. In the formal model we keep the value outside the store and we load into the local variables the values before the execution.

Of course this strategy, though convenient from a formal standpoint, cannot be easily implemented. We use an array of objects, when the argument would have been loaded we load the \( i \)th element of the array. If the argument is read and modified we introduce a local variable initialized at the beginning of the method.

### 4.4 The Max-Stack Problem

To generate the code of a method, the execution engine requires a bit of information: the maximum height of the operand’s stack. The calculation of this information requires a simulation of the stack in order to find the maximum height. The `System.Reflection.Emit.ILGenerator` class of the share source CLI [51] performs this calculation.

Besides in our case we already have the information about max-stack for each method we read. It would be better if we are able to reuse this information to derive the new values for the generating code.

If we take another look to the tree of Figure 4.2 we may notice that we could exploit the knowledge of the transformation together with the information about max-stack.

We combine code using two transformations: \( \text{Bind}_\pi \) and \( \text{Bind}_\chi \). IL requires that a method always returns with the stack of operands empty. We know that \( \text{Bind}_\pi \) adds the body of a code value as a prefix: we can compute the maximum value between the max-stack of the code that is bound to the argument and the target of the binding.

<table>
<thead>
<tr>
<th>Code Value</th>
<th>Max-Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>2</td>
</tr>
<tr>
<td>a(^3)</td>
<td>2</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
</tr>
</tbody>
</table>
The Table 4.1 contains the max-stack value for the four code values of the previous example. It is trivial to see that the max-stack value for both a and m is 2: the code simply loads two arguments on the stack and perform an operation.

When BindC operates some method call is replaced with the body of the method. We don't know the height of the stack when the callvirt instruction is called, though we know that there are n arguments of the Invoke method25. Thus an approximation of the max-stack in this case would be the following:

\[
\text{max-stack}(c.\text{Bind}(v_1, \ldots, v_n)) = \max_i(v_i) - \min_i(n_i) + \max-stack(c)
\]

where

- \(\max_i(v_i)\) is the maximum for max-stack of the inlined code values \(v_i\) or 0
- \(\min_i(n_i)\) is the min. number of arguments required by inlined code values

In this case we are taking a worst case approach: often the callvirt instruction is performed when the stack is not full. This isn't a real problem because the max-stack information is used by the runtime to dimension the stack frame on the stack.

Nonetheless we can improve this calculation. When a code value is built on a method, the constructor scans its body and tries to establish the height of the stack at each invocation of callvirt that could be target of an inlining. The information is retained into the code value and used only during the code generation process.

Finally we should consider the influence of the BindC transformation on the max-stack value. We know that the environment is stored into an array thus we can translate loading and storing of bound arguments into loading and storing into an array element. This requires us to use two additional instructions to load the array reference and the index of the element to be manipulated. Because these instructions cannot cumulate their effect of the operand stack we simply need to add 2 to the max-stack value if the binding involves BindC transformations.

4.5 Optimizing the Generated Code

The quality of the generated code highly depends on the code read from the method bodies. Consider for instance the code generated by the C# compiler for the add method:

```csharp
.method public hidebysig static
    int32  'add'(int32 i, int32 j) cil managed
{
```

25 There is also the self argument that will be eliminated in the transformation.
The body of the method is different from what we would have expected: there is a branch instruction to the next instruction and the value, already on top of the stack is stored in a local variable and then reloaded. Of course if we enable the optimization switch we get:

```csharp
.method public hidebysig static
    int32  'add'(int32 i, int32 j) cil managed
{
    // Code size       4 (0x4)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  add
    IL_0003:  ret
} // end of method MainClass::'add'
```

Although the two methods do the same thing the code is somewhat different. When we use the Bind operator the code can propagate its inefficiencies. For instance the three way add would be the following if we take the first version of add:

```csharp
.method public hidebysig static
    int32  'add3'(object[] c, int32 i, int32 j, int32 k) cil managed
{
    .maxstack  2
    .locals init (int32 V_0, int32 V_1, int32 V_2)
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  add
    IL_0003:  stloc.2
    IL_0004:  br.s    IL_0006
    IL_0006:  ldloc.0
    IL_0007:  ret
} // end of method MainClass::'add3'
```
The first argument of the generated method is the reference to the array of the environment. In this case the environment is empty. All the signatures of code values have as the first argument an array of objects. The argument will be hidden to the user when the code value is transformed in its executable form.

The bind operation tends to propagate the inefficiencies in the generated code leading to a larger code in this case. Let us consider the output of `Bind` if we use the optimized version of `add`:

```csharp
.method public hidebysig static
int32 'add3'(object[] c, int32 i, int32 j, int32 k) cil managed
{
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000:  ldarg.1
    IL_0001:  ldarg.2
    IL_0002:  add
    IL_0003:  stloc.0
    IL_0004:  ldloc.0
    IL_0005:  ldarg.3
    IL_0006:  add
    IL_0007:  ret
} // end of method MainClass::'add'
```

This code is considerably smaller and uses two local variables less than the previous version. In general the local variables used tend to grow up with binding.

We notice that also in this improved version of `add3` we still introduce inefficiency because the result of the first sum is stored in a local variable that is loaded again for the second sum and never reused.

It is plain that applying systematic transformations to code chunks would introduce inefficiencies. Besides we are generating intermediate language rather than machine code; and our output is subject to a second compilation performed by the JIT compiler. Thus we have some concrete hope that the JIT could get rid of most of the inefficiencies introduced by our generation process.

Nonetheless a JIT compiler should generate code in a timely fashion, and it cannot perform the full set of optimizations performed by a traditional compiler’s back-end. It would be better if we are able to get rid of many patterns that systematically arose by the code composition.

It is worth noting that we have improved the calculation of max-stack property by exploiting the knowledge about the building blocks of our code. Can we extend the same approach to other optimizations?
Let us consider the following example of generating the increment function out of the add code by applying the simple Bind transformation and specifying the constant 1 as the first argument:

```
.method public hidebysig static
    int32 'inc'(object[] v, int32 i) cil managed
{
    .maxstack 2
    IL_0000: ldarg.0
    IL_0001: ldc.i4.0
    IL_0002: ldelem.ref
    IL_0003: unbox [mscorlib]System.Int32
    IL_0004: ldind.i4
    IL_0005: ldarg.1
    IL_0006: add
    IL_0007: ret
} // end of method MainClass::'inc'
```

The value 1 is stored as a boxed object in the array containing the environment associated with the code value. Instructions from IL_0000 to IL_0004 represent the access to the array and the unboxing of the integer value.

A first optimization we can do is reckoning that the type of the bound argument is integer and hard-wire into the code the value 1. In this case would have been the following:

```
.method public hidebysig static
    int32 'inc'(object[] v, int32 i) cil managed
{
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldc.i4.1
    IL_0001: stloc.0
    IL_0002: ldloc.0
    IL_0003: ldarg.1
    IL_0004: add
    IL_0005: ret
} // end of method MainClass::'inc'
```

In this case we avoid the overhead of boxing and unboxing the constant 1, though we should introduce a local variable.

Looking at the code of `add` we notice that there is no need of introducing the local variable because the argument used to store values by method body, thus we can simply replace the instruction that loads the argument with `ldc.i4.1`.

In this case we are able to improve the quality of the generated code simply by analyzing the body of the method. In this case we have recognized that an argument was read only, and this information allowed us to avoid the introduction of a local variable.
In general we can keep statistics about the use of arguments and local variables within a method body. These statistics are used during binding to adopt better strategies during the transformation. In some sense we use the code transformation and we partial evaluate it.

Table 4.2: Features and their use in optimization

<table>
<thead>
<tr>
<th>Feature</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only arguments</td>
<td>No need of allocating a local variable</td>
</tr>
<tr>
<td>Last use of a local variable</td>
<td>The local can be reused afterwards</td>
</tr>
<tr>
<td>Number of $\text{Bind}_\chi$ candidates</td>
<td>Code size estimation</td>
</tr>
<tr>
<td>Pass by reference arguments</td>
<td>Sharing of the storage</td>
</tr>
<tr>
<td>Reduction of locals used in $\text{Bind}_\pi$</td>
<td>To avoid to introduction of local variables during inlining of code values</td>
</tr>
<tr>
<td>$\text{Bind}_\chi$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 indicates the features we’re looking for whilst we look at the body of a method. This information are stored in the code value itself and updated in the derived code values if necessary.

Of course we could perform some control-flow analysis and other typical optimization techniques used by compilers [1]. Though we are interested in runtime code generation, so the code generation time cumulates with time to JIT the code and it can’t be long. The features mentioned above can be deduced with a single pass of the code and greatly improve the quality of the generated code.

An interesting consequence of this approach is that checking properties of code values that follow the same structure as binding can be computed on the constituents and then propagated through binding. A trivial example is the max-stack value.

Another interesting example is an estimate of the size of the generated code: we know that $\text{Bind}_\pi$ simply prefixes a method to another one leading to a code whose size is essentially the sum of constituents; in the case of $\text{Bind}_\chi$ we should know how many invocation of a delegate are present, then we can calculate the size of the resulting code.

We can only overestimate the size of the resulting code: during the code generation process instruction are changed leading to possible changes in the dimension of the code. However the estimate would be fairly accurate and this would speed up the code generation process because we can dimension the buffer where we store the generated code appropriately.

The last feature in Table 4.2 may seem useless. As shown in the model the transformation relies on the fact that we look for the loading of a function object
on which is invoked a special method. The base schema requires the introduction of a local variable for each argument required by the function.

Suppose that all the arguments are read only in the function to be inlined in place of the call. This is a frequent situation if we write code combinators. Moreover often the arguments of a call are local variables rather than expressions: in this case it’s a pity to introduce a lot of useless local variables. Therefore we look for all the situations in which the number of instructions between the loading of the function object and the callvirt instruction coincide with the arguments’ number of the function. In this case we don’t introduce any local variable and we use the instruction that loads the \( i \)th argument wherever the argument is required. Consider the following method fragment (assuming that the third argument is a delegate named MyDelegate):

\[
\begin{align*}
&\text{ldarg.2} \\
&\text{ldloc.1} \\
&\text{ldarg.s 5} \\
&\text{ldarg.1} \\
&\text{callvirt int32 MyDelegate::Invoke(int32, System.String, int32)}
\end{align*}
\]

Suppose that we bind to the third argument the method:

```csharp
public static int M(int i, string s, int j) {
    Console.WriteLine("{0}\ni = {1}\nj = {2}\n", s, i, j);
    return i+j;
}
```

If we apply the standard \( \text{Bind}_f \) we get the following code from the transformation:

\[
\begin{align*}
&\text{ldloc.1} \\
&\text{ldarg.s 5} \\
&\text{ldarg.1} \\
&\text{stloc.5} \\
&\text{stloc.4} \\
&\text{stloc.3} \\
&\text{// Code of M here with} \\
&\text{//\(\{(arg, 0)\rightarrow(loc, 3), (arg, 1)\rightarrow(loc, 4), (arg, 2)\rightarrow(loc, 5)\}\)}
\end{align*}
\]

Now if we adopt the strategy described above we are able to discover that argument 0 becomes ldloc.1, and so on. Thus we can output the following code:

\[
\begin{align*}
&\text{ldloc.1} \\
&\text{ldarg.s 5} \\
&\text{ldarg.1} \\
&\text{// Code of M here with} \\
&\text{//\(\{(arg, 0)\rightarrow(loc, 1), (arg, 1)\rightarrow(arg, 5), (arg, 2)\rightarrow(arg, 1)\}\)}
\end{align*}
\]
In conclusion the code generation process can be very fast because we read chunk of IL instruction and we merge them with changes that can be performed by an automata. A lazy approach helps us to save both memory and CPU-time avoiding the unneeded generation of the code at each bind operation.

The generated code depends on the quality of the code of the methods used to build the code values. Indeed the code coming from methods has been optimized by the compiler at compile time, thus we can generate good code at runtime exploiting a computation performed at compile time. Of course the optimization of the code can be perceived in methods with non trivial bodies. Besides we introduce some systematic inefficiency into generated code: mainly unneeded local variables and redundant instructions. The JIT compiler may be able to get rid of these problems, though if we help the result would be probably better. Also because we know the properties of the generated code whilst the JIT should treat the code as the normal output of a compiler.

Although we don’t have much time to spend in code optimization we are able to greatly reduce the locals used and the introduction of useless instructions. The optimization can be implemented by keeping statistics about building blocks of code (methods) at the cost of a scan of the method body when the code value is created.

### 4.6 The Reverse Polish Notation Compiler

As an example of the code generated using Code Bricks we have written a calculator that compiles expressions expressed in reverse polish notation. The full source of the compiler is included in appendix 8.5. The calculator has the following syntax:

```
stmt ::= letf var expr | let var expr | expr
expr ::= expr expr [/*-*/] | term
term ::= num | var | term* var
```

The calculator supports variables and function definition. The program compiles the expression using code values and the print the generated code. Are supported both integers and floating point numbers. We don’t go into the details of the implementation, though it is worth to have a look to the methods we use to build the code values corresponding to expressions:

```
// Code values
public static double add(double i, double j){ return i+j;
}
public static int add(int i, int j) { return i+j; }
```
public static double mul(double i, double j) { return i*j; }
public static int mul(int i, int j) { return i*j; }
public static double sub(double i, double j) { return i-j; }
public static int sub(int i, int j) { return i-j; }
public static double div(double i, double j) { return i/j; }
public static int div(int i, int j) { return i/j; }
public static double conv(int i) { return i; }

Each operation has two methods: one for dealing with integers and the other to deal with double; their use should be evident by now. We have one more method that should be considered: conv.

The conv method seems to be the identity method, though it does something. It performs an implicit type conversion from int to double. The CLR requires that all the type conversions are explicit. Its definition is the following:

```
.method public hidebysig static
float64 conv(int32 i) cil managed
{
    // Code size       3 (0x3)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  conv.r8
    IL_0002:  ret
} // end of method MainClass::conv
```

This method is used to cast an integer into a double if the result of an operation is integer and should be used in an operation involving a double. An example is the following expression:

```
3.1 6 2 / +
```

The code value obtained from conv is used as follows:

```
if (l.ReturnType != r.ReturnType) {
    MethodInfo cv = this.GetType().GetMethod("conv");
    if (args == typeof(int))
        l = new Code(cv, l);
    else
        r = new Code(cv, r);
    args = typeof(double);
}
```

This is a neat example of the level of control that Code Bricks offers to the programmer. We are able to control from C# details like type conversion.

We include the output of the program given two expressions: a sum of two integers and a function definition. Let’s start with the first expression:
The expression $2 + 2$ generates a code value that represents the constant function 4. The code generator isn’t able to discover that the first three instructions evaluates to a constant. A step of partial evaluation would be required in order to find and optimize these situations. We can insert a rule in the automata which generates the code, but the more rules we have to check the less is the generation speed. If the JIT compiler can be able to find it the real execution of the function is to load the value 4 on the stack.

The calculator allows us to define functions. To be precise we only generate functions: closed terms simply are constant functions. The following function shows an example of argument sharing: the program is able to recognize that two arguments are shared and generates the appropriate code.

```csharp
> let f = x y + x *
.method public static valuetype [mscorlib]System.Double
code787(object[] A_0, valuetype [mscorlib]System.Double A_1, valuetype [mscorlib]System.Double A_2)
{
    .maxstack 2
    .locals init (valuetype [mscorlib]System.Double V_0)
    IL_0: ldarg.1
    IL_1: ldarg.2
    IL_2: add
    IL_3: stloc.s 0
    IL_4: ldloc.0
    IL_5: ldarg.1
    IL_6: mul
    IL_7: ret
}
```

The instructions IL_3 and IL_4 are useless: the version of Code Bricks used to generate this example wasn’t able to spot the problem and avoid the introduction of a local variable.
Note that functions always assume double arguments because there is no way to indicate the types in the language.

4.7 Execution Environment and Code Values

So far we have discussed how the `Bind` operator can generated code. In this section we discuss how to produce an executable object starting from the code generated using code values.

STEE are built around the notion of type: methods are just part of types and to be executed, require that the type is load by the execution environment. However both JVM and CLR provides a mean of runtime code generation: in Java it is necessary to define a class loader that assembles the bytecode for a class and create it by calling the `defineClass` method of `ClassLoader` class; besides CLR provides the namespace `System.Reflection.Emit` that contains a number of classes to dynamically creates assemblies, classes and methods.

Code values are somewhat outside this picture: they are more similar to functions than methods. To be precise code values are in general closures; in fact there is the environment that containing the objects closed with binding. Moreover we may want execute a code value at any time, so we cannot assume of creating a class with more than a code value. Besides we can’t execute a block of code without creating a type, and a major change would be required to the runtime to change this assumption.

The first question to be answered is how a code value can be seen as an element of the runtime. We have already answered to this question: we can use delegates on the CLR or their surrogate in Java [65]. We need delegates because a code value can’t be modeled as a simple pointer to function: we should store the reference to the environment somewhere. Besides a delegate is a pair: a reference to an object, and a pointer to a method.

If the execution environment is not aware of code values the only viable strategy is to generate a class for each code value that should be executed. The class has a field inside which is the environment associated with the code value. A method is generated inside the class and the generated code is stored in it. An instance of the class is created and the environment properly initialized. A delegate to the method is created.

In this way we don’t break the execution model though we introduce several inefficiencies in the execution of the generated code:

- The code generation requires an additional overhead to create the structure of the new class;
The execution environment should allocate several data structures to load and manage the type whereas our interest is only into a method.

When the generated code becomes useless it is hard to dispose because we should unload the type.

Of course these inefficiencies depend on how many code values should be executed by a program. However all the applications that need to generate some code, use it and throw away may get unneeded overhead.

The problem is that STEE are designed to cope with objects that are used and disposed, types are loaded and methods compiled with the idea that they will be needed. This assumption is indeed plausible when classes are essentially handwritten, but it fails if the program automatically generates the code for itself.

CLR provides a method called `SwapMethodBody` which allows redefining the method’s body of a dynamically created class\(^{26}\). However this method doesn’t help us, because we can’t reuse a class unless we are sure that there are no more delegates referring to the old definition of the method.

We can improve the performance of the compilation of code values by making aware the execution environment of the code values. We can preserve the same interface that is coherent with the type based execution model whilst we can reduce the overhead exploiting our knowledge of the semantics of Code Bricks.

The first thing to avoid is the creation of a type for each code value: the runtime can implement a special internal type which is allowed to be extended with new methods. Then, if the execution environment provides the notion of delegate, we can create a delegate using this hidden type and the newly created method. Moreover the garbage collector may keep track of the delegate objects associated to code values. When a method becomes useless it could be eliminated from the methods of this special type and the corresponding code unloaded from the heap used by the JIT to create the methods.

We are working to extend the shared source implementation of CLI from Microsoft to support code values. We are interested in measuring how the performance of the system improves also for programs that needs to generate small functions to execute only once (like the top level of an interpreter).

---

\(^{26}\) Assuming that the method to be replaced has the same signature of the one declared in the class.
4.8 Code Bricks Performance: the .NET RE Case Study

So far we have discussed possible strategies for implementing Code Bricks library. The intuition is that the lazy approach and the cut and paste of IL pre-compiled would be effective to achieve fast runtime code generation. Moreover we could also expect that the quality of generated code wouldn’t be that bad because we rely on blocks of code produced by a compiler which, at compile time, performs a number of optimizations. Besides potential global optimizations cannot be performed and some inefficiencies are introduced by the join strategy adopted to reconnect code fragments.

In this section we discuss a case study in which the implementation of Code Bricks has been used to implement a compiler for regular expressions to speed up pattern matching at runtime. Although this is a classic example we felt that the toy application wouldn’t be enough to test generation speed and performance of generated code. Thus we have decided to re-implement the regular expression compiler in .NET emitting IL code using Code Bricks rather than Reflection.Emit as it does actual implementation. This is a good setup for testing our library: we can compare the benefits of compiling code with respect to regular expression interpreter – also shipped with RE namespace – and compare execution performance against the code generated by the hand crafted compiler used by the library.

We first describe the approach followed to turn the RE interpreter into a compiler, and how generated code is related to its counterpart generated by the RE library compiler. Then we discuss in far more detail a subset of the compiler that is used for regular expressions of the form A|B.

4.8.1 An overview

A classic example of dynamic code generation is the dynamic generation of programs that matches regular expressions. Finding a match for a pattern in a large text can be very expensive, in particular if the pattern to find is known only at runtime.

Libraries for regular expressions like PCRE [58] often provide a notion of compilation of the pattern. The string of the pattern is parsed and converted in an internal form which represents the definition of the automata associated with it. An interpreter of the automata is responsible to executing the pattern matching against a text in input.

CLR is shipped with a namespace, System.Text.RegularExpressions, containing classes that provide pattern matching capabilities. The parser of regular expressions in .NET produces a bytecode which can be either inter-
interpreted or compiled. The parser corresponds to PCRE compiler and produces the description of the automata using the language implemented by the interpreter and the compiler.

Whilst the interpreter simply interprets the automata against the input, the compiler dynamically generates a class using System.Reflection.Emit. The generated class implements the automata and can be considerably faster because its code gets compiled by JIT.

The source code of the whole RE package is available with the Shared Source CLI [51] and it is contained in the directory

`fx/src/regex/system/text/regularexpressions`

The interpreter is defined in the file `regexinterpreter.cs` and the compiler in the file `regexcompiler.cs`. The interpreter and the compiler share the same structure: a loop iterates over the opcodes produced by the parser. For each opcode the interpreter updates its internal state whereas the compiler emits IL instructions in the body of a method in the target class; both the interpreter and the output of the compiler implement the same interface.

The aim of the case study was to show that code generation with Code Bricks offers enough control on the generated code. Thus we have reverse engineered the compiler to derive the C# instructions that could have produced the needed code.

The case study has been quite successful: the source of the original RE compiler is more than three thousands lines of code. We have rewritten it using our model in about 1,400 lines of code that generate code quite similar to that produced by the original version.

Of course with Code Bricks the compiler could have been generated starting from the interpreter as we’ve done in the power example in section 2.4.1 or the reverse polish notation calculator in section 8.5. In this section we show as an example the process adopted to produce the compiler based on Code Bricks rather than System.Reflection.Emit.

Let us consider for instance the Getmark operand which is implemented within the interpreter as follows:

```csharp
case RegexCode.GetMark:
    Stackframe(1);
    Track(Stacked(0));
    Textto(Stacked(0));
    Advance();
    continue;
```
We could define a Code value for each method and combine them in the same way is done in the interpreter. By inlining the functions we get the following code fragment\(^\text{27}\) which would be equivalent to the code generated using binding:

```csharp
++runstackpos;
runtrack[--runtrackpos] = runstack[runstackpos - 1];
runtrack[--runtrackpos] = runcodepos;
runextpos = runstack[runstackpos - 1];
```

The regular expression compiler in SSCLI exploits the CLR facilities in the Reflection.Emit namespace for generating IL code at runtime. Instead of generating an automaton equivalent to the given regular expression and interpreting it, the compiler generates the IL code that emulates the automaton.

The questions are: how far is this code from the hand-coded version of the compiler? Do we get any benefit in doing things by hand?

To answer to these questions we have reverse engineered the output of the compiler by replacing sequence of IL instructions by C# statements. In this way we can write a compiler based on Code Bricks by packing these instructions in methods used to build code values; the output will be the same code of the hand-written compiler. A rather surprising outcome of the experiment has been that we’ve derived a source really close to the interpreter. Thus we could have defined the compiler as a dynamic partial specialization of the interpreter.

We have been successful in reducing the size of the compiler because a sequence of IL instructions map into a single high level instruction. Let us consider a fragment of the compiler:

```csharp
internal void PopStack() {
    _ilg.Emit(OpCodes.Ldloc_S, _stackV);
    _ilg.Emit(OpCodes.Ldloc_S, _stackposV);
    _ilg.Emit(OpCodes.Dup);
    _ilg.Emit(OpCodes.Ldc_I4_1);
    _ilg.Emit(OpCodes.Add);
    _ilg.Emit(OpCodes.Stloc_S, _stackposV);
    _ilg.Emit(OpCodes.Ldelem_I4);
}
```

Such code is equivalent to the following C# instruction:

```csharp
stack[stackpos++];
```

It may seem that a single function for an instruction can be a waste of resources. In fact the way the compiler is written is driven by the need of dominating the

\(^{27}\) The Advance method simply increments the “program counter” of the interpreter and it isn’t needed in the compiled version.
complexity instructions of the virtual machine. The Getmark operand is compiled as follows:

```c
case RegexCode.Getmark:
    ReadyPushTrack();
    PopStack();
    Dup();
    Stloc(_textposV);
    DoPush();
    Track();
    break;
```

We have already shown how PopStack corresponds to a single instruction. The functions ReadyPushTrack, DoPush and Track are similar to PopStack. The equivalent C# code if the following:

```c
int s = stack[stackpos++];
track[--trackpos] = s;
textpos = s;
track[--trackpos] = i;
```

Note how this code fragment is close to the one we would have obtained by deriving the compiler from the interpreter.

The value \( i \) is an integer generated during compilation which indicates a state of the automaton, and it is used in a while loop with a big switch inside. In the interpreter the index of the current instruction is used instead.

A compiler would generate almost the same IL code from the C# code fragment above. Note that we have introduced a local variable which in the original code is unneeded because there is the possibility of duplicating the value on top of the stack (\( \text{Dup} \) call).

Once spotted the instructions we want to use in our target program we pack them into a method:

```c
static void GetMark(ref int[] stack, ref int stackpos,
    ref int[] track, ref int trackpos,
    ref int textpos) {
    int s = stack[stackpos++];
    track[--trackpos] = s;
    textpos = s;
    track[--trackpos] = i;
}
```

Code fragments are loaded using Code values and combined with the techniques shown so far.
4.8.2 Implementing RE compiler with Code Bricks

The SSCLI RE library is an articulated library which comprises a parser for regular expressions Perl like, an interpreter of parsed expressions and a compiler which generates IL code on the fly and executes JIT code equivalent to the automata.

The class RegexRunner is abstracts the concept of executor and hides the rest of the library from the particular kind of executor used (interpreted or compiled).

![Figure 4.3: Inheritance hierarchy of the modified RE library](image)

We have added a new class that inherits from RegexRunner class to the library. Each class that derives from RegexRunner should implement three methods among which there is the Go() method which implement the code to test if at the current position in the input string there is a match of the pattern.

Class Regex selects the appropriate runner depending on options specified to the class constructor:

```csharp
if (factory != null)
    runner = factory.CreateInstance();
else if ((roptions & RegexOptions.CodeCompiled) == 0)
    runner = new RegexInterpreter(code, UseOptionInvariant() ? CultureInfo.InvariantCulture : CultureInfo.CurrentCulture);
else
```

The RegexOptions.CodeCompiled implies that static method Build generates the code value inside and generates a type indicated as CodeRegexCompiled.
The runner instance interprets the pattern expressed as a program for an interpreter using 41 opcodes defined in RegexCode class. An opcode can be modified using one modifier out of five that are available.

The program generated for patterns of the form $A \| B$ is the following:

0: RegexCode.Lazybranch 7
1: RegexCode.Setmark
2: RegexCode.Lazybranch 5
3: RegexCode.Multi 0
4: RegexCode.Goto 6
5: RegexCode.Multi 1
6: RegexCode.Capturemark 0 -1
7: RegexCode.Stop

The execution model of this bytecode assumes two stacks: a stack (runstack) for parameters and one for backtracking (runtrack). The LazyBranch instruction simply remembers where to jump on the backtracking stack. Setmark is used to set the matching mark to the current text position. The instruction Multi is responsible for matching the string $A$ in the first case, $B$ in the second.

In case of failure the interpreter signals backtrack which causes the pop from runtrack of the reference to the instruction responsible for the push. The interpreter modifies the opcode by adding the modifier RegexCode.Back. Our goal is to visit the program and generate its equivalent IL version.

The loop of the interpreter looks like the following:

protected override void Go() {
    Goto(0);
    for (;;) {
        switch (Operator()) {
            case RegexCode.Stop:
                return;
            case RegexCode.Nothing:
                break;
            case RegexCode.Goto:
                Goto(Operand(0));
                continue;
            ...
        }
        Backtrack();
    }
}
The main loop selects the operation by means of a switch; if an operation needs to backtrack it simply breaks the switch case. A fairly obvious compilation strategy is to reproduce the same loop in IL with a different switch with a partial evaluated version of the code needed to cope with each operand. Moreover the `RegexCode.Back` modifier affects only used instructions so we can generates only needed backtracking code.

The schema produced by standard compiler looks like the following:

```
I0:  // Code for Lazybranch 7
I1:  // Code for SetMark
...
Backtrack:  
    switch B0, B1, B2, B3
B0: // Code for Lazybranch | Back 7
```

In this case the loop is implemented using branches as the compiler does with loops. A label for each backtrack operation is needed because we can’t push a branch label on the backtrack stack.

The compiler builds the IL related to each instruction and accumulates the instruction inside an `ILGenerator` object which is used to fill the body of a dynamically created method.

We have adopted a similar strategy in compilation, though we can’t produce code fragments with branch instructions to undefined labels. This is because methods should be always well formed. How can we express thus the same loop? We use a while and a switch assuming that each operator indicates the number of the next instruction. Our skeleton will look like the following:

```
int i = 1;
while (i != -1)
    switch (i) {
    case 0: // Backtrack code
        EnsureStorage();
        i = runtrack[runtrackpos++];
        break;
    case 1: // Lazybranch 7  
        ...
        break;
    ...
    }
```

By iterating the technique illustrated in the previous section we can produce the following program that is the program we want to generate (assuming the pattern `ab|cd`):
protected override void Go() {
    int i = 1;
    while (i != -1)
        switch (i) {
        case 0:
            EnsureStorage();
            i = runtrack[runtrackpos++];
            break;
        case 1: // RegexCode.Lazybranch 14
            runtrack[--runtrackpos] = runtextpos;
            runtrack[--runtrackpos] = 9;
            i = 2;
            break;
        case 2: // RegexCode.Setmark
            runstack[--runstackpos] = runtextpos;
            runtrack[--runtrackpos] = 10;
            i = 3;
            break;
        case 3: // RegexCode.Lazybranch 9
            runtrack[--runtrackpos] = runtextpos;
            runtrack[--runtrackpos] = 11;
            i = 4;
            break;
        case 4: { // RegexCode.Multi 0
            if (2 > runtextend - runtextpos) {
                i = 0;
                break;
            }
            i = runtext[runtextpos] == 'a' ?
                (runtext[runtextpos + 1] == 'b' ? 0 : -1) : -1;
            if (i == -1) {
                i = 0;
                break;
            }
        }
        runtextpos += 2;
        i = 5;
        break;
    }
    i = 7;
    break;
    case 6: { // RegexCode.Multi 1
        if (2 > runtextend - runtextpos) {
            i = 0;
            break;
        }
        i = runtext[runtextpos] == 'c' ?
            (runtext[runtextpos + 1] == 'd' ? 0 : -1) : -1;
        if (i == -1) {
            i = 0;
        }
    }
The most interesting cases are 4 and 6 corresponding to Multi instructions. The effect of Multi instruction is an attempt to match a sequence of characters against a sequence of characters in the pattern. The parser stores these strings in a variable so that the compiler knows that 0 corresponds to the string "ab" and 1 to "cd". We note that the test of the string match is performed using a sequence of tests rather than a while loop. This is because at compile time we know the string thus we can unroll the loop to speedup the string match.

We rely on two delegate types to express splice-in:

deedlete int IR(RegexRunner re);
deelete int IRI(RegexRunner re, int i);
The following code builds the appropriate code value for the specific instance of a `Multi` operation:

```csharp
    public static int _MultiSkel(RegexRunner re, int len, IR checkstring, int go) {
        if (len > re.runtextend - re.runtextpos) return 0;
        if (checkstring(re) == -1) return 0;
        re.runtextpos += len;
        return go;
    }
    public static Code MultiSkel = new Code(typeof(CodeRegexCompiler).GetMethod("_MultiSkel"));

    public static int _CharCheck(RegexRunner re, int pos, int ch) {
        return re.runtext[re.runtextpos + pos] == ch ? 0 : -1;
    }
    public static Code CharCheck = new Code(typeof(CodeRegexCompiler).GetMethod("_CharCheck"));

    public static int _CharChkMid(RegexRunner re, int pos, int ch, IR rest) {
        return re.runtext[re.runtextpos + pos] == ch ? rest(re) : -1;
    }
    public static Code CharCheckMid = new Code(typeof(CodeRegexCompiler).GetMethod("_CharChkMid"));

    public Code Multi(int operand, int instnum) {
        string s = runstrings[operand];
        Code c = CharCheck.Bind(Code.Free, s.Length - 1, (int)s[s.Length - 1]);
        for (int i = s.Length - 2; i >= 0; i--)
            c = CharCheckMid.Bind(Code.Free, i, (int)s[i], c);
        return MultiSkel.Bind(Code.Free, s.Length, c, instnum);
    }
```

The `Multi` method unrolls the string combining `CharCheck` and `CharCheckMid`. We rely on splice-in to produce the appropriate sequence of char tests. The skeleton is then filled with the appropriate test.

The corresponding code in the RE Compiler is the following:

```csharp
    int i;
    String str;

    str = _strings[Operand(0)];
```
Ldc(str.Length);
Ldloc(_textendV);
Ldloc(_textposV);
Sub();
BgtFar(_backtrack);

// unroll the string
for (i = 0; i < str.Length; i++) {
    Ldloc(_textV);
    Ldloc(_textposV);
    if (i != 0) {
        Ldc(i);
        Add();
    }
    Callvirt(_getcharM);
    if (IsCi())
        CallToLower();
    Ldc((int)str[i]);
    BneFar(_backtrack);
}

Ldloc(_textposV);
Ldc(str.Length);
Add();
Stloc(_textposV);
break;

This version is indeed less readable of our version and relies on a significant amount of code to prepare reflection objects.

We have done the same for all the instructions of the program. Then we should join the code values into a switch and then splice the switch code into the while loop. The following code performs these operations:

    public static void _Pump(RegexRunner re, IRI sw) {
        int i = 1;
        while (i != -1) i = sw(re, i);
    }

    public static Code Pump = new
        Code(typeof(CodeRegexCompiler).GetMethod("_Pump"));

    public Code CompileCode() {
        ArrayList sw = new ArrayList();
        sw.Add(BacktrackCode);
        numInstr = 0;
        for (int codepos = 0; codepos < runcodes.Length;
            codepos += RegexCode.OpcodeSize(runcodes[codepos]))
            numInstr++;
    }
label = numInstr + 1; // 0 is the backtrack!

for (int codepos = 0; codepos < runcodes.Length; codepos += RegexCode.OpcodeSize(runcodes[codepos])) {
    runcodepos = codepos;
    GenOneCode(runcodes[codepos], sw);
}

for (int i = 0; i < _notecount; i++) {
    BacktrackNote n = _notes[i];
    if (n._flags != 0) {
        runcodepos = n._codepos;
        GenOneCode(runcode._codes[n._codepos] | n._flags, sw);
    }
}

Code[] ops = new Code[sw.Count];
sw.CopyTo(ops);

return Pump.Bind(Code.Free, new Code(ops));

The compilation method simply invokes in the appropriate order the GenOneCode method which generates the code related to the opcodes and appends it to sw. All code values in sw have the signature int c(RegexRunner re) thus can be joined in a switch using the appropriate constructor.

Finally we should point out that the compiled code is limited to the automata that checks for match in position runtextpos of the runtext string which is the input string. It is the code in RegexRunner class inherited by all three classes which is responsible of calling Go() and advancing the pointer in the input string.

4.8.3 Performance Measurements

We have recently finished a first implementation of Code Bricks library so we are able to provide some preliminary results of performance compared to the SSCLI RE library. The library is around ten thousands lines of C# code\(^{28}\) and it is indeed far from being mature. Besides most of the optimizations and imple-

\(^{28}\) The library relies on memory mapping to achieve fast access to binary files containing IL. The C# code (around 200 lines of code) implementing the abstraction of a mapped file as an array of bytes is operating system dependent (by means of the standard PInvoke mechanism provided by CLI standard to invoke code in DLLs). The rest of the library should run on every implementation of CLI standard. In particular the library can be used on SSCLI on MacOS X, Free BSD and Linux.
mentation strategies discussed in this chapter have been already implemented but far more can be done.

We have conducted benchmark on both the debugging and release version of the library. It is quite interesting to study the debugging setup because it is a sort of worst case for code generated using Code Bricks; thus we focus our attention on the debugging case rather than the release case.

To understand why code compiled without optimization is far more interesting than the release version let us consider the code generated by C# compiler for the _CharCheck method in debug mode:

```csharp
.method public hidebysig static int32 _CharCheck(
    class CodeTest.RegularExpressions.RegexRunner re,
    int32 pos,
    int32 ch) cil managed
{
    // Code size 31 (0x1f)
    .maxstack 3
    .locals init ([0] int32 CS$00000003$00000000)
    IL_0000: ldarg.0
    IL_0001: ldfld string CodeTest.RegularExpressions.RegexRunner::runtext
    IL_0006: ldarg.0
    IL_0007: ldfld int32 CodeTest.RegularExpressions.RegexRunner::runtextpos
    CodeTest.RegularExpressions.RegexRunner::runtext
    IL_000c: ldarg.1
    IL_000d: add
    IL_000e: callvirt instance char [mscorlib]System.String::get_Chars(int32)
    IL_0013: ldarg.2
    IL_0014: beq.s IL_0019
    IL_0016: ldc.i4.ml
    IL_0017: br.s IL_001a
    IL_0019: ldc.i4.0
    IL_001a: stloc.0
    IL_001b: br.s IL_001d
    IL_001d: ldloc.0
    IL_001e: ret
} // end of method CodeRegexCompiler::CharCheck
```

The output of compiler with the optimization turned on is the following:

```csharp
.method public hidebysig static int32 _CharCheck(
    class CodeTest.RegularExpressions.RegexRunner re,
    int32 pos,
    int32 ch) cil managed
{
    // Code size 26 (0x1a)
    .maxstack 3
    .locals init ([0] int32 CS$00000003$00000000)
    IL_0000: ldarg.0
    IL_0001: ldfld string CodeTest.RegularExpressions.RegexRunner::runtext
    IL_0006: ldarg.0
    IL_0007: ldfld int32 CodeTest.RegularExpressions.RegexRunner::runtextpos
    CodeTest.RegularExpressions.RegexRunner::runtext
    IL_000c: ldarg.1
    IL_000d: add
    IL_000e: callvirt instance char [mscorlib]System.String::get_Chars(int32)
    IL_0013: ldarg.2
    IL_0014: beq.s IL_0019
    IL_0016: ldc.i4.ml
    IL_0017: br.s IL_001a
    IL_0019: ldc.i4.0
    IL_001a: stloc.0
    IL_001b: br.s IL_001d
    IL_001d: ldloc.0
    IL_001e: ret
} // end of method CodeRegexCompiler::CharCheck
```
The main difference between the two versions is that in the former method a useless local variable is introduced as well as an unneeded branch at the end of the method.

When we run the RE compiler based on Code Bricks in debug mode we get worst IL code and the binding tend to propagate these useless instructions, in particular with code splice-in. Moreover when we unroll the check of the string we introduce a new local variable for each character as well as a pair of `stloc/ldloc` instructions.

Our test environment consists of an input text stored in a file. We have used an input text obtained by repeating six times the plain text version of “I Promessi Sposi”, Alessandro Manzoni’s novel, to obtain a file of around 7.5Mb. We have executed a simple match on this text varying several parameters and taking elapsed time for interpreter, RE compiler and out compiler based on Code Bricks. We have adopted the following parameters:

◊ Pattern of the form $A|B|C|...$
◊ Pattern that (don’t) produces a match
◊ Pattern length

For each setting of these parameters we have measured three intervals: compilation time (not for the interpreter), the time to get the answer from `IsMatch` a first time and a second time. We measure two runs on the same regular expression instance because in the first run the JIT compiles the `Go()` method introducing some overhead with respect to subsequent runs. We have also included the number of IL instructions produced by the two compilers.
The following tables report a run of the three version of RE matcher for several patterns. The Len and Found columns indicate the length of the pattern and if it is found; these columns are intended to help evaluating data. Time is measured in milliseconds. We note that the difference between patterns that matches and those who don’t is that in the former case the matching code halts to the first match; in the latter the whole file is scanned.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Len</th>
<th>Found</th>
<th>Constr.</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>2</td>
<td>1</td>
<td>300</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Renzo</td>
<td>5</td>
<td>1</td>
<td>320</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>Azzecca-garbugli</td>
<td>15</td>
<td>1</td>
<td>310</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Renzo</td>
<td>Azzecca-garbugli</td>
<td>21</td>
<td>1</td>
<td>351</td>
<td>110</td>
</tr>
<tr>
<td>Renzo</td>
<td>Lucia</td>
<td>Azzecca-garbugli</td>
<td>27</td>
<td>1</td>
<td>320</td>
</tr>
<tr>
<td>Xx</td>
<td>2</td>
<td>0</td>
<td>320</td>
<td>180</td>
<td>140</td>
</tr>
<tr>
<td>Xxxxx</td>
<td>5</td>
<td>0</td>
<td>351</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>5</td>
<td>0</td>
<td>320</td>
<td>12899</td>
</tr>
<tr>
<td>xxx</td>
<td>xyx</td>
<td>7</td>
<td>0</td>
<td>330</td>
<td>12889</td>
</tr>
<tr>
<td>Xxxxx</td>
<td>xxyyxyyy</td>
<td>14</td>
<td>0</td>
<td>340</td>
<td>12838</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>yz</td>
<td>8</td>
<td>0</td>
<td>320</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>yz</td>
<td>kk</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.3: Performance of IsMatch using the interpreter

The interpreter is clearly the bottom line. We notice that when we use alternatives in the pattern the performance of the interpreter increase significantly. This is because the sequence matching is implemented by a simple while which is optimized by the JIT. When we use ‘|’ the interpreter should perform backtracking checks and its performance struggles.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Len</th>
<th>Found</th>
<th>Instr.</th>
<th>Constr.</th>
<th>Comp.</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>2</td>
<td>1</td>
<td>172</td>
<td>431</td>
<td>110</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Renzo</td>
<td>5</td>
<td>1</td>
<td>193</td>
<td>601</td>
<td>120</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Azzecca-garbugli</td>
<td>15</td>
<td>1</td>
<td>270</td>
<td>431</td>
<td>110</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Renzo</td>
<td>Azzecca-garbugli</td>
<td>21</td>
<td>1</td>
<td>338</td>
<td>451</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>Renzo</td>
<td>Lucia</td>
<td>Azzecca-garbugli</td>
<td>27</td>
<td>1</td>
<td>406</td>
<td>501</td>
<td>120</td>
</tr>
<tr>
<td>Xx</td>
<td>2</td>
<td>0</td>
<td>172</td>
<td>471</td>
<td>110</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>Xxxxx</td>
<td>5</td>
<td>0</td>
<td>193</td>
<td>451</td>
<td>100</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>5</td>
<td>0</td>
<td>219</td>
<td>461</td>
<td>110</td>
<td>1552</td>
</tr>
<tr>
<td>xxx</td>
<td>xyx</td>
<td>7</td>
<td>0</td>
<td>233</td>
<td>481</td>
<td>100</td>
<td>1682</td>
</tr>
<tr>
<td>Xxxxx</td>
<td>xxyyxyyy</td>
<td>14</td>
<td>0</td>
<td>282</td>
<td>491</td>
<td>110</td>
<td>1592</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>yz</td>
<td>8</td>
<td>0</td>
<td>266</td>
<td>451</td>
<td>100</td>
</tr>
<tr>
<td>Xx</td>
<td>xy</td>
<td>yz</td>
<td>kk</td>
<td>11</td>
<td>0</td>
<td>313</td>
<td>521</td>
</tr>
</tbody>
</table>
Table 4.4: Performance of IsMatch using the RE compiler

The hand written version of the compiler is good in performance and the cost of compilation is quite independent from the pattern.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Len</th>
<th>Found</th>
<th>Instr.</th>
<th>Constr.</th>
<th>Comp.</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re</td>
<td>2</td>
<td>1</td>
<td>257</td>
<td>310</td>
<td>40</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>Renzo</td>
<td>5</td>
<td>1</td>
<td>296</td>
<td>310</td>
<td>40</td>
<td>400</td>
<td>10</td>
</tr>
<tr>
<td>Azzecca-garbugli</td>
<td>15</td>
<td>1</td>
<td>439</td>
<td>300</td>
<td>90</td>
<td>731</td>
<td>40</td>
</tr>
<tr>
<td>Renzo</td>
<td>Azzecca-garbugli</td>
<td>21</td>
<td>1</td>
<td>585</td>
<td>351</td>
<td>40</td>
<td>711</td>
</tr>
<tr>
<td>Renzo</td>
<td>Lucia</td>
<td>Azzecca-garbugli</td>
<td>27</td>
<td>1</td>
<td>731</td>
<td>391</td>
<td>60</td>
</tr>
<tr>
<td>Xx</td>
<td>2</td>
<td>0</td>
<td>257</td>
<td>340</td>
<td>30</td>
<td>2904</td>
<td>2013</td>
</tr>
<tr>
<td>Xxxxx</td>
<td>5</td>
<td>0</td>
<td>296</td>
<td>320</td>
<td>40</td>
<td>2834</td>
<td>1973</td>
</tr>
<tr>
<td>xx</td>
<td>xy</td>
<td>5</td>
<td>0</td>
<td>364</td>
<td>351</td>
<td>30</td>
<td>3836</td>
</tr>
<tr>
<td>xxx</td>
<td>xyx</td>
<td>7</td>
<td>0</td>
<td>390</td>
<td>351</td>
<td>40</td>
<td>3795</td>
</tr>
<tr>
<td>xxxxx</td>
<td>xxxyxxxy</td>
<td>14</td>
<td>0</td>
<td>481</td>
<td>361</td>
<td>40</td>
<td>4426</td>
</tr>
<tr>
<td>xx</td>
<td>xy</td>
<td>yz</td>
<td>8</td>
<td>0</td>
<td>471</td>
<td>340</td>
<td>30</td>
</tr>
<tr>
<td>xx</td>
<td>xy</td>
<td>yz</td>
<td>kk</td>
<td>11</td>
<td>0</td>
<td>578</td>
<td>340</td>
</tr>
</tbody>
</table>

Table 4.5: Performance of IsMatch using the Code Bricks compiler

The compiler written using Code Bricks in debug mode is slower than the hand written compiler. If we compare the column with the number of IL instructions generated by the compiler we notice that our version generates a larger amount of additional instructions. This is due to the useless instructions introduced by the C# compiler in debug mode. In fact output of our compiler for pattern with short sequences is faster than the output of long character sequences: this is due to the unrolling of the string into the code that leads to a greater number of bind operations which replicates useless instructions.

Besides we note that the compilation time is in favor of Code Bricks though in this implementation the code is generated in bytes and then decoded to put it into method body using Reflection.Emit.

In Figure 4.4 we have reported the performance of the three matchers in the second run. It appears evident from the figure that, even if the output of our compiler is less efficient than the hand-written one, the trend is that the two compiled versions behave in a similar way. These results are pretty encouraging because we have compared the worst case approach for our generator. Also in this condition the generated code exhibits a reasonable performance.

At this stage of the implementation of the library we cannot produce the release version of the code. Our guess is that the performance line will be almost overlapped to the one of the RE compiler. To support our guess we have im-
implemented the output of the compiler simulating by hand the behavior of Code Bricks starting from the optimized IL code. The results have shown that the performance have been even better than the RE compiler, though by an insignificant fraction of time.

![Comparison of the three approaches.](image)

**Figure 4.4: Comparison of the three approaches.**

We are still working to generate a full showcase of RE compiler using Code Bricks library. At the moment we are mimicking the hand written version of the compiler (which is just the partial evaluated version of the interpreter). We believe that the smaller amount of code and its readability will contribute in spotting other optimization that would have been hard to find and implement in the compiler that generates IL directly.

Finally we observe that, under the hypothesis that Code Bricks library is correct, we are sure that all possible RE patterns will be compiled correctly generating well formed and type safe IL.
PART TWO
5 Applications of Code Bricks

5.1 Runtime Code Generation

Runtime Code Generation (RTCG), also known as Dynamic Code Generation (DCG), has been a research topic for many years, but its practical applications have been somewhat limited. The ability of generating code at runtime is obviously a good idea: it is possible to exploit the information available to the program for specializing itself. Runtime partial specialization of programs is the goal of many systems: ‘C [61], DynJava [50], [56], DyC [29], VCODE [16], Dynamic Caml [49], FABIUS [42], [43], [44], Dynamo [45], Tempo [14].

The overall goal of program specialization is to optimize the execution of a program exploiting information available at runtime. Unfortunately the code generation introduces some overhead in the program execution. A good example of RTCG is the JIT compiler: it should compile programs on the fly trying to produce efficient code in a small amount of time.

There are three dimensions to take into account in RTCG:

1. the generation of some internal form that represents the output
2. the optimization strategy used for generating the executable code
3. the language used to express the code generation (bytecode vs. assembly)

Each dimension has some influence on the performance of the generated code and the time needed to generate it. When the goal of RTCG is partial specialization of an application, it is possible to adopt some mechanism based on templates to reduce the time needed for generating the code.

Once the intermediate form has been produced from the information available at runtime, the code should be generated. At this point the executable code should be generated, and several optimizations can be performed: if the code is not optimized we may not get any benefit by specialization of the code; besides we may loose the speedup spending the time in code optimization.

The applicability of these code generation systems depends also on their degree of portability. If the system generates machine code the system will depend on some architecture; moreover the optimizations will depend on the ar-
chitecture and becomes more complex. Besides if the system can rely on a code generation library such as VCODE [16] which provides a sort of JIT compiler which maps a simple intermediate language into machine language.

The lesson learned by the systems mentioned above is that indeed it is worth to try specializing code at runtime. If the specialization is applied to meaningful applications may significantly improve the performance of programs.

For programs executed by a virtual machine such as Java, C#, or ML it is possible to generate the intermediate language. This approach has many advantages: the virtual machine is optimized for fast execution of the bytecode; the intermediate language is more expressive than the naked processor and the translation in assembly is maintained and updated with architectures for sure. The code generation is automatically portable. DynJava [56] and Dynamic Caml [49] have chosen this approach.

Is it still convenient to generate bytecode and compile it on the fly? In [49], [50], [56] are discussed the performance of programs with and without partial specialization and the benchmarks still shows that specialization greatly improves performance. In particular it is relevant the benchmark of DynJava because the system uses one of the code transformations for specializing code.

Code Bricks can be used for generating code at runtime: the generation process is quite fast because we simply merge lists of IL instructions; we do little global optimization exploiting the statistics derived analyzing the method bodies; and we generate IL.

We have discussed in section 4.8 the use of Code Bricks library to generate code at runtime. In particular the library generates code faster than a program that outputs IL directly instructions.

5.2 Support to Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a new direction in programming proposed by researchers from Xerox Palo Alto Research Center [5]. The main observation that lies behind AOP is that, though the separation of concerns is a fundamental engineering principle, programming languages don’t provide full support for it.

Programming languages tend to stress the functional aspect of the program, where the behavior 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 that are orthogonal to program functionalities but are essential. Thus, with the exception of toy programs, we can’t 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*.

There are several well-known examples of aspects: synchronization, security, debugging, are but few examples. The research group that proposed AOP has developed a language based on Java called AspectJ [5], [38]. With AspectJ a programmer can define places where aspects should be inserted. These points are known as join points and can be expressed using the *pointcut* construct. Consider the following class:

```java
class Point {
    private int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    void setX(int v) { this.x = v; }
    void setY(int v) { this.y = v; }
    int getX() { return x; }
    int getY() { return y; }
}
```

Suppose we want to add an aspect that requires adding some code before executing the assignment in the `setX` and `setY` methods. First of all we should define the join points where the aspect should be inserted. We can use *designators* to indicate the position of join points. A *pointcut* defines the join points combining designators with and, or and not operators:

```java
pointcut setter(): target(Point) &&
    (call(void setX(int)) ||
    call(void setY(int)));
```

In the example are used two designators: `target` that is used to indicate that the target of the operation is an object of the given class; `call` indicates that the operation should be a call to a method with the declared signature.

An aspect allows inserting code before or after a join point. For instance we can check the bounds of the point with the following aspect:

```java
aspect BoundsChecker {
    pointcut setter(int v): target(Point) && args(v) &&
        (call(void setX(int)) ||
        call(void setY(int)));
    before(int v) : setter(v) && args(v) {
```
if (v < 0 || v > 500)
    throw new ArgumentException("Wrong coordinate!");
}

AspectJ compiles the class Point and injects the declared aspects into the calls to its set methods. The compilation of AspectJ programs is made in two phases: pointcuts are located checking the conditions expressed with designators; the code specified in the definition of the aspect is injected before or after the pointcut.

AspectJ is not the only attempt to introduce aspects into programming languages. AspectiX [30] for example is an open architecture based on CORBA [13], [47] which allows adding aspects to CORBA services.

Code Bricks can be used to implement a system for AOP which is language independent. The specification of join points could be expressed in terms of objects available at CIL level. Not every pattern of AspectJ can be matches looking at the compiled version of a program; nonetheless it would be possible to define aspects without having to use a specific language. Using the CLIFileReader library [11] it is possible to analyze the body of methods contained into an assembly, and reconstructing the assembly file with the modified code as shown by the sample application developed by Donal Lafferty [41]. The application reads all the types defined in the input assembly and rewrite their definition in a new assembly.

Code values can be used to inject code before or after code fragments. Let us suppose that we are able to generate a method which contains a method call when we spot the call whilst analyzing code. This is possible if the code is verifiable because we can locate when the stack is loaded with the arguments to be passed to a function.

Under this hypothesis we may have been able to derive the following method from the call to the setX method:

```java
static void setXCall(Point p, int x) {
    p.setX(x);
}
```

Let us consider the following code combinator:

```java
delegate void Set(Point p, int i);
void checkBounds(Set s, Point p, int v) {
    if (v < 0 || v > 500)
        throw new ArgumentException();
    s(p, v);
}
```
In general we can always write a code combinator which adds some aspect to a code fragment. The $\text{Bind}_X$ is the transformation which helps us in inserting code before or after other code.

A programmer can also explicitly annotate its programs using code values to mark join points in a program. Then he can use $\text{Bind}$ to modify the code value and add some aspect.

Nonetheless Code Bricks can be also used to implement tools for supporting AOP.

### 5.3 Domain Specific Languages

DSLs allow developing software for a particular domain quickly and effectively. By capturing more closely the semantics of an application domain, DSL programs are generally easier to write, reason about and maintain compared to equivalent programs written in general purpose languages. On the other hand, there may be a significant overhead in creating the infrastructure needed to support a DSL, particularly vital features like debugging, tracing or profiling. Nonetheless a large part of a DSL is not domain specific, hence the task is time consuming.

Hudak [31] suggests addressing this problem by embedding the DSL into another language creating a DSEL (Domain Specific Embedded Language). Lisp macros have often been used for this purpose. The Jakarta Tool Suite [8] allows building DSLs in an extensible superset of Java that provides Abstract Syntax Tree constructors and hygienic macros. A software generator uses the DSL to create the program of interest.

Embedding a DSL in an existing host language allows inheriting all its standard mechanisms and facilities, including compilers and tools, and to integrate fully with the host language. However the domain specific parts of the language are not handled as well as in a native implementation, since the compiler has no knowledge of the domain and the new features are handled as a new layer on top of the existing language, often losing significantly in performance. Techniques like partial evaluation [34] might be helpful to eliminate interpretation steps by transforming further the generated code into code operating at the innermost level. Unfortunately partial evaluation is not suitable yet for fully automated code generation tasks.

The goal of embedding a special purpose language in a general purpose language can be achieved also using the technique of template metaprogramming [15]. Exploiting C++ templates, one can write template meta-programs that are executed during compilation by the type checker. The technique can be used to
perform code selection and code generation at compile time. Specific applications are code configuration, especially in libraries, and code optimization. Blitz++ [78] provides a sophisticated DSL for matrix algebra built through template metaprogramming. The library achieves high performance in matrix computations by performing a number of optimizations (unrolling, fusion, partial evaluation) that rely on detailed knowledge of the domain and by specializing code to each particular combination of the arguments to a function.

Code Bricks can be used to implement domain specific languages: the typical problem of implementing DSLs without adopting a generative approach is performance. In complex application domains the operators provided by a DSL are often quite expensive and significant optimizations can be made exploiting the semantics of the domain.

The matrix computations domain represents a good example of an application domain where the knowledge of the domain can significantly improve the execution speed. Consider for example a simple implementation of Matrixes in C#:

```csharp
class Matrix {
    int rows, cols;
    double[,] m;
    Matrix(int r, int c) {
        m = new double[r, c];
        rows = r;
        cols = c;
        // Init the matrix
    }
    static Matrix Add(Matrix a, Matrix b) {
        Matrix ret = new Matrix(a.rows, a.cols);
        for (int i = 0; i < a.rows; i++)
            for (int j = 0; j < a.cols; j++)
                ret.m[i, j] = a.m[i, j] + b.m[i, j];
        return ret;
    }
}

It is plain that the following expression doesn’t take advantage of the knowledge about the applicative domain:

```csharp
Matrix m = Matrix.Add(Matrix.Add(Matrix.Add(m1, m2), m3), m4);
```

Each invocation of `Add` requires the allocation of a temporary matrix, and we pay a double loop for each invocation of the method. If the matrixes are large the operation can be really expensive or even can’t be computed.

In C++ it is possible to generate the optimal code exploiting template metaprogramming (see section 6.1.1) as shown in GMCL example in [15]. The tech-
nique exploits parametric types to represent the expression tree as a type: the
previous expression would be associated to the type

Add<Add<Add<Matrix, Matrix>, Matrix>, Matrix>

The ‘+’ operator transforms the sum into a type and holds the information about
the matrixes involved in the expression. When the parametric type should be
converted into a Matrix type the library generates the code generating a single
nested loop that adds four elements at once.

Is it possible to use Code Bricks to do something similar? We can try to pur-
sue the same approach and generate the function that adds four matrixes using
code values and Bind. We can also provide a syntax which hides the details of
generation to the user of the class.

We can try the same approach as C++: the Add method could generate the
code which simply adds two elements of the matrix. We introduce a method to
cast the generated code value into a matrix; this method will add the loops to
the generated code and executes the code value. The class matrix becomes the
following:

delegate double MatrixElement(int i, int j);
delegate Matrix Fun();

class Matrix {
  int rows, cols;
  double[,] m;
  Matrix(int r, int c) {
    m = new double[r, c];
    rows = r;
    cols = c;
    // Init the matrix
  }
  static Code Add(Matrix m1, Matrix m2) {
    if (m1.rows != m2.rows || m1.cols != m2.cols)
      throw new ArgumentException();

    return addMM.Bind(m1, m2, Code.Free, Code.Free);
  }
  static Code Add(Matrix m, Code c) {
    Matrix m2 = c.Environment[0] as Matrix; // Bind
    inspection!

    if (m.rows != m2.rows || m.cols != m2.cols)
      throw new ArgumentException();

    return addMC.Bind(m, c, Code.Free, Code.Free);
  }
  static Matrix ToMatrix(Code c) {
Matrix m = c.Environment[0] as Matrix; // Bind inspection!
Matrix ret = new Matrix(m.rows, m.cols);
Compute f = loop.Bind(ret, c).MakeDelegate(typeof(Fun));
return f();
}

// Code values
static double AddMM(Matrix m1, Matrix m2, int i, int j) {
    return m1.m[i, j] + m2.m[i, j];
}
static double AddMC(Matrix m, MatrixElement f, int i, int j) {
    return m.m[i, j] + f(i, j);
}
static Matrix Loop(Matrix m, MatrixElement f) {
    for (int i = 0; i < m.rows; i++)
        for (int j = 0; j < m.cols; j++)
            m.m[i, j] = f(i, j);
    return m;
}

// Build the code values
static Code addMM =
    new Code(typeof(Matrix).GetMethod("AddMM"));
static Code addMC =
    new Code(typeof(Matrix).GetMethod("AddMC"));
static Code loop =
    new Code(typeof(Matrix).GetMethod("Loop"));

This new version of the Matrix class can be used as follows to calculate the expression above:

Matrix r = Matrix.ToMatrix(Matrix.Add(
    Matrix.Add(Matrix.Add(m1, m2), m3), m4));

The expression is surprisingly similar to the one used in the simplest version of class Matrix. Besides the execution is very different: Add accumulates the code needed to the expression into a code value. ToMatrix adds the loop to the sum of the elements, and executes the code. The ability of looking inside code values help us to access the bound arguments to check that the matrixes are compatible.

The code that gets executed is similar to the following:

object[] Env = new object[]{m1, m2, m3, m4};
ret = new Matrix(ml.rows, ml.cols);
for (int i = 0; i < ret.rows; i++)
    for (int j = 0; j < ret.cols; j++)
        ret.m[i, j] = (Env[0] as Matrix).m[i, j] +
Note that the environment is implemented as an array of objects, so we have to access it with a cast at each loop. This can be avoided by introducing a local variable of type `Matrix` in `AddMM` and `AddMC` methods.

In this example we showed how to implement an efficient DSL using Code Bricks. We can provide a set of methods representing the operations on the objects in the application domain. The methods don’t implement the operations needed but keeps track of the calls using Code objects. When the expression of the domain language should be executed the code is generated.

The ability of generating code by combining precompiled methods is helpful to develop DSLs since it allows defining meaningful code fragments in the language. And the generation process focuses only on their combination rather than the emission of billions of simple instructions that make the code hard to maintain and modify.

It is our belief that Code Bricks is a mechanism of the runtime that can be used to support programming languages in the implementation of multi-stage and meta-programming constructs. Nevertheless the ability of accessing the mechanism through an API makes it available to all programming languages, though at the cost of a clumsy syntax.

Although this solution is viable in many situations, there is still an inefficiency that can be eliminated: domain expressions are often known at compile time; in this case we could perform the generation at compile time. This is the strategy adopted by Blitz++ [78], GMCL [15], and Template Haskell [67]. In the next chapter we discuss the applications of Code Bricks to staging; it is possible to use code values before runtime by partial specialization of Intermediate Language.

5.4 A Variant of Proof Carrying Code

Proof Carrying Code (PCC) [54] is a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied (possibly in binary form) by an untrusted source. For this to be possible, the untrusted code producer must supply with the code a safety proof that attests to the code’s adherence to a previously defined safety policy. The host can then easily and quickly validate the proof without using cryptography and without consulting any external agents.
Figure 5.1: The architecture of an implementation of PCC [12]

A schema of a proof carrying code system is shown in Figure 5.1. The architecture of the system [12], [53] assumes that the programmer writes Java programs. The code producer and the code consumer agree on some safety policy, usually established by the host. A special compiler outputs the native code of a program together with annotations intended to simplify the procedure of generating the verification condition module. The code producer generates also a proof of the property requested by the code consumer in order to execute the code.

The code consumer receives the annotated native code and the proof and it is able to quickly check the proof against the annotated code.

The goal of PCC is to enable hosts to execute arbitrary code without relying on certification mechanisms which are subject to the problem of key management and related to PKI [52]. The mechanism would be indeed welcome in the everyday Internet downloading, because nowadays we install software downloaded from the net assuming that it does what it says to do.
In PCC schema the code producer is usually a developer whereas the host is a computer where the software should be installed. The architecture assumes that the programmer can spend some resource in using a PCC compiler because his programs would be easily distributed.

Let us consider a different framework: a server where clients need to run programs to achieve some goal. A possible approach is to send the application to execute on the server; perhaps relying on some framework for mobile code. The problem is that on the server we want to be granted that the program doesn’t harm by any mean the running state; though in this case the state should include other processes and any other resource accessed by the program.

Indeed we can use the PCC architecture, though it would be hard to express conditions about services rather than execution integrity. With Code Bricks is possible to implement a schema resembling PCC with the only difference that there isn’t any code producer at all! The idea is borrowed from section 4.4 and it is a generalization\(^\text{29}\). The schema works as depicted in Figure 5.2.

![Figure 5.2: A code generation schema based on Code Bricks](image)

The host provides a set of static methods representing a set of functionalities. These methods can be considered the operators of a DSL as well as the basic functions of an abstract machine.

\(^{29}\) The original idea arose during discussions with Don Syme whilst the author was internship at Microsoft Research Laboratory in Cambridge U.K.
When the operators are defined it is proven that some property $P$ holds for all the methods. The property $P$ should be such that if $P(m_1)$ and $P(m_2)$ then $P(\text{Bind}(\sum, m_1, v_1, ..., v_{i-1}, m_2, v_{i+1}, ..., v_n))$ for each possible position of $m_2$ in the binding. If we are able to define a property $P$ with such characteristics we can exploit the compositional structure of $\text{Bind}$ and assert the property for each possible combination of the code fragments.

Once defined one or more properties, the server publishes the operators so that the interested clients can read these specifications. A client that needs running a specific program can generate its description in terms of the published operators and send it back to the server.

The server can generate the requested program by creating the code values out of the published methods and combine them as specified by the client. In the end the generated program is executed.

This schema recalls PCC, though in this case only the specification is sent across the network. Because of the structure of Code Bricks, the set of published methods can be neither complete nor even Turing equivalent. The simplicity of defining new methods to extend the set of operators makes the schema plausible.

The whole architecture relies on the ability of finding a property that propagates with $\text{Bind}$ applications. Are these properties difficult to find? We have already discussed some of them: max-stack, if arguments are read-only, the overall size of the generated code, and others. We conjecture that for many properties, especially of security, the requested condition holds, though further investigations are required in this direction.

### 5.5 Supporting Programming Languages

We have already pointed out that we have exposed code values and binding as an API (an extension of the reflection) which is more of a mechanism. In fact throughout this thesis we have given plenty of examples in C# using code values and $\text{Bind}$.

Nonetheless Code Bricks is also a mechanism on which programming languages can rely upon. In the next chapter we will discuss how code values can support programming languages with meta-programming and multi-staging facilities. In this section we focus our attention on support of dynamic code generation and we discuss the opportunity of using Code Bricks for compiling closures for functional programming languages.
5.5.1 DynJava and Dynamic Code Generation

As an example of exploiting Code Bricks to implement new languages we discuss how the dynamic code generation of DynJava [56] can be implemented using code values and binding.

In DynJava the user can declare code specifications to define a dynamic code fragment, whose syntax is similar to cspec in 'C [61]. There are kinds of code specifications, statement specifications and expression specifications. A code specification begins with backquote, followed by a context specification and either a statement or an expression. Consider the following examples:

(1) is a statement specification because it represents a code fragment enclosed in curly braces preceded by its signature. The return type of a statement can be indicated like in (2). (3) is an expression specification.

It is possible to declare variables that represent statement and expression specifications. For instance

Code specifications can be combined together using @v which corresponds to inline the code fragment into another one. Consider the following example:

When a code specification is embedded by @, the context specification is always checked against surrounding code and the context of outer specification, to ensure that the composed type code is safe. In the above example c1 is embedded c2 generating a code equivalent to c3.

Note that the embedding is syntactic so that g in c1 is the same name used in c2 and after embedding they will refer to the same object. The language also
allows primitive values to be bound lifted in code fragments using a $ sign, like in the following example:

```java
String message = "Hello";

code_spec<String x> c1 =
    '{ System.out.println($message + ", " + x + "!"); };

code_spec<return void> c2 =
    '{ String x = "Michael"; @c1; return; }
```

DynJava is implemented by inserting placeholders into the bytecode which are intercepted at runtime by a special class loader and replaced with their definition. It is not evident in [56] if it is possible to accumulate code at runtime, though from the description of implementation it seems that this is not allowed because the generator is activated by the class loader where the method definition should be known. Moreover there is no object at runtime which can identify a code fragment.

DynJava can be easily compiled in IL if we can rely on Code Bricks: statement and expression specifications can be represented by code values. The transformation supported by DynJava is $Bind_x$ and is easily implemented with $Bind$. Also the $ operator corresponds to a limited version of $Bind_v$ because only native values can be lifted in specifications.

The previous example can be rewritten with Code Bricks as follows:

```java
//...
String message = "Hello";
Code c1 = new Code(typeof(Test).GetMethod("__t1"));
Code c = new Code(typeof(Test).GetMethod("__t2"));
//...
delegate void __s(string s);
static void __t1(string message, string x) {
    Console.WriteLine("{0}, {!}!", message, x);
}
static void __t2(__s c1) {
    c1("Michael");
}
```

Thus a DynJava compiler would have to generate the statements and the code to assemble them at runtime.

The execution of dynamic methods in DynJava relies on a special class loader which essentially implements the delegate type constructor: an abstract class defines the signature of the method and dynamic methods are inserted in derived classes. The Code Bricks hides these details providing the simple MakeDelegate method.
The only issue in mapping DynJava into code values is that statement specifications can be a fragment of a method. In particular a specification may contain `break` and `continue` statements which will bound outside the current specification. Consider for example the following statement:

```csharp
code_spec<break; int x> c1 = '{ if (x == 5) break; }
```

For this kind of specifications the compiler should inline the specification whenever is needed: code values cannot refer elements outside their scope and the all the information should pass through the metaphor of function invocation.

5.5.2 \textit{Bind} for Implementing Closures

The transformation \textit{Bind} could be used to support the implementation of closures in programming languages. Currently CLR doesn’t provide a real support for implementing closures, thus different programming languages implement the same concept in different way reducing the opportunity of interoperate through the runtime.

Don Syme proposes ILX [71] as an extension of CIL with specific instructions designed to support typical elements of functional programming languages. ILX is then compiled to IL using a common implementation schema. Among the extension proposed there are closures. CLR does provide delegates which are similar to closures: a delegate object represents a pair \((\text{obj}, \text{meth})\) where \text{obj} is a reference to an object and \text{meth} is a pointer to a method. In some sense the object represents the environment of the closure that is referred by the method.

Unfortunately there is a link between the method and the object that make closures hard to implement by means of delegates: the class of the object should contain the method referred by the delegate. This connection between the environment and the code implies that a compiler should generate a considerable number of types because environments are different. Compilers for .NET actually implement closures as static methods, and among their arguments it is included the environment.

Code values represent anonymous computations and can be also used for implementing closures. In particular the \textit{Bind} transformation closes values to a function. Perhaps the current implementation of Code Bricks could introduce unneeded overhead: the creation of the code value requires that the body of a method is analyzed and statistics stored into the code value. If we need only to bind values to a method this could be avoided reducing the overhead needed for implementing closures.
Nevertheless the notion of code value is suitable to represent also closures within the runtime. Assuming that the runtime is extended to support code value it is possible to optimize the use of code values as closures. This unifies two concepts in a single data type simplifying the design of the runtime and reducing the number of mechanisms provided to support the implementation of different programming languages.
6 Meta-programming and staging

6.1 Code Bricks as Support for Meta-programming

In this section we discuss how Code Bricks can be used for meta-programming. Meta-programs are programs which represent and manipulate other programs, including themselves. Programs manipulated by meta-programs are called object programs.

Meta-programming systems can be characterized in many different ways [66]:

◊ Static vs. runtime: if the meta-language is executed before or during runtime;
◊ Manually vs. automatically annotated: programs may be explicitly annotated by the programmer to distinguish the program from meta-program; otherwise a system can implicitly determine which portions of a program should be executed;
◊ Homogeneous vs. heterogeneous: does the meta-language coincide with the object language?

Code values and Bind contribute to extending an STEE to support meta-programming in an efficient way. What kind of support may offer Code Bricks to programming languages that need to implement meta-programming constructs?

In the previous chapter we discussed how to generate code using code values and Bind. The ability of generating programs and executing them is indeed part of meta-programming. It can be used by programming languages that provide some sort of eval function. An example can be the JScript compiler included in SSCLI which compiles the string passed to eval into an IL program and then executes it.\(^\text{30}\)

Nevertheless the other side of meta-programming is the ability of manipulating programs; Code Bricks allows the introspection the structure of programs only if they are code values. How can we manipulate programs by mean of code values?

\(^{30}\) See the file sscli\jscript\engine\eval.cs in the SSCLI distribution.
In section 5.3 we’ve discussed how to manipulate a sequence of method calls to generate the code to optimize the sum of matrices. We have inferred the structure of an expression implicitly by looking at the order in which method calls were performed. The ability of using Code Bricks from multiple languages makes difficult to characterize it as meta-programming system.

To implement the operator that optimizes the sum of \( n \) matrices we have followed the same approach used by C++ template meta-programming [2]. As a matter of fact the model of code values and \textit{Bind} operator has been in part inspired by this technique; in particular the way in which code generation is expressed recalls \textit{Bind}.

In the rest of this section we discuss how C++ template meta-programming works and how code values and \textit{Bind} can be complemented with other features provided by STEE to obtain a similar meta-programming framework.

### 6.1.1 C++ Template Meta-programming

C++ supports generic programming through the \textit{template} mechanism, which allows defining parameterized classes and functions. Templates together with other C++ features constitute a Turing-complete, compile-time sublanguage of C++. C++ can be considered as a two-level language [15] since a C++ program may contain both static code, which is evaluated at compile time, and dynamic code, which is executed at runtime. Template meta-programs [15] are the part of a C++ source that is executed during compilation.

C++ template meta-programming is static because the computation is performed at compile time; it is automatic because the programmer doesn’t provide any particular annotation for identifying the meta-program; it is heterogeneous because the meta-language differs from the object language.³¹

Here is an example of a parametric wrapper for a generic type T, which we actually use for creating instances of dynamically defined classes:

```cpp
template <class T>
class Item {
public:
    T value_; 

    Item(T value) : value_(value) { }
    T const & operator =(T const &newval) { } 

³¹ We consider the meta-language different from the object language because the meta-language exploits types to perform computations. Thus equivalent constructs are expressed in a different way; besides the meta-language is a subset of the object language.
Within a template, the parameter type can be used as an ordinary type, for instance for declaring variables. When a template instantiation occurs in a program, the compiler generates an appropriate version of the class. In the above example, Item<int> produces an instance of the class Item where T is int. Integer and Boolean constants are also allowed as template parameters.

A template definition may involve other templates. When a template must be instantiated, the compiler must resolve all the templates involved in its definition. Since template definitions can be recursive, the type checker module of the compiler must be able of recurring on type definitions until a base definition is found and all the types can be resolved. This forces the type checker to perform recursive computations, thus achieving Turing completeness.

Template meta-programming exploits the computation performed by the type checker to execute code at compile time. This technique is used mostly for code selection and code generation at compile time. Its applications are mainly code configuration, especially in libraries, and code optimization [15].

A meta-program consists of meta-functions, which are template classes that exploits the capabilities of compiler of inlining and calculate constant expressions to select and generate code.

A useful meta-function is If:

```cpp
template <bool g, class Then, class Else>
struct If { typedef Then VAL; }; 

template <class Then, class Else>
struct If<false, Then, Else> { typedef Else VAL; }; 
```

When the guard of the meta-function is false, the more specific definition of type If is matched and the VAL type is the same as the Else type. Otherwise the first definition is chosen and the VAL type is the one specified by the Then parameter.

Using If one can select at compile time alternative implementations depending on a type. For example the following code selects between two different ways to perform a search on a column, depending on whether the column has and index or not:

```cpp
template <class T> struct SearchWithIndex {
    static int search(const T& q) { ... }};
```
template <class T> struct SearchNoIndex {
    static int search(const T& q) { ... }};

typedef If<IsA<Attribute, IndexedAttribute>::VAL,
    SearchWithIndex<Attribute>,
    SearchNoIndex<Attribute> >::VAL> Searcher;

Attribute query(...);
Searcher::search(query);

Meta-functions may be defined recursively. Here is a function computing factorial at compile time:

    template <>
    struct Fact<0> { enum { VAL = 1 }; };
    template <int n>
    struct Fact { enum { VAL = n * Fact<n-1>::VAL }; };

    cout << "Factorial of 7 is " << Fact<7>::VAL << endl;

Meta-programs can perform partial evaluation (also known as *currying*), i.e. an evaluation done at compile time, given only some of the arguments of a function, that produces a function that accepts the remaining arguments at run time.

Partial evaluation produces optimized code. In the following example, by exploiting the inlining capabilities of the compiler, the generated code computes the power of an integer through a sequence of multiplications rather than through a loop:

    template <>
    struct Power<1> {
        static double power(double n) { return n; } };
    template <int m>
    struct Power {
        static double power(double n) {
            return m * Power<m - 1>::power(n); } };

    for (int i = 0; i < 8; i++)
    cout << "Cube of 4: " << Power<3>::power(i) << endl;

The code generated for Power<3>::power() is indeed:

    static double Power<3>::power(double n) { return n * n * n; }

Several useful meta-functions can be defined for testing types and other common meta-programming tasks [77], for instance:

◊ whether T is a class: IsClass<T>
◊ whether T is a pointer type: IsPointer<T>
◊ whether T is a constant type: IsConst<T>
whether T and U are the same type: Equals<T, U>
whether U is a base class for T: IsA<T, U>

Data structures like lists and trees can be defined using compound types. Here is a list type:

```cpp
struct Nil { typedef Nil car; typedef Nil cdr; }
template <class h, class t = Nil> struct Cons { typedef h car; typedef t cdr; };

typedef Cons<int, Cons<char>> List1;
```

Using recursion and the test functions described above one can program arbitrary recursive functions on data structures.

### 6.1.2 C++ Template Meta-programming and Code Bricks

C++ template meta-programming relies on types to express computations. Meta-programs can generate code by exploiting the ability of the compiler of inline methods and the code is accumulated because the inlining process can be recursive. The meta-program can inspect the program using meta-functions that have access to the information used by the compiler. It is even possible to implement support for reflection using this information, see for instance [6], [7].

<table>
<thead>
<tr>
<th>Table 6.1: C++ meta-programming and Code Bricks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C++</strong></td>
</tr>
<tr>
<td>Inserting values in generated code</td>
</tr>
<tr>
<td>Code generation as iterated inlining of methods</td>
</tr>
<tr>
<td>Implicit analysis of code structure by mean of function application</td>
</tr>
<tr>
<td>Code selection depending on the nature of types</td>
</tr>
</tbody>
</table>

Table 6.1 shows how the elements of C++ template meta-programming can be expressed using a STEE that supports code values and binding.
However this way of meta-programming with Code Bricks is at runtime implying an additional overhead to the program execution. Besides, with C++ it is possible to perform the specialization the program at compile time.

The meta-programming system based on Code Bricks is homogeneous because the operations transform IL programs into IL programs. Moreover code values are built using methods that expressed in programming languages that compile down to IL. Thus the transformation can be also considered homogeneous at the language level.

### 6.2 Multi-stage Programming with Code Bricks

The traditional execution model of programming languages assumes that the program is written and the executed. The evolution of systems has introduced a number of programs that manipulate a program before the “official” runtime: dynamic loaders, configuration systems, installers and many others.

It is not uncommon that a program takes as input another program and generates code. This happens, for instance, with yacc [1], [46], where the program is expressed as an annotated grammar which is converted into a program that is compiled. With the diffusion of generative programming [15], [36] approach the use of generators will increase significantly.

Perhaps a new execution model for program should be considered to describe programs evaluation [81]. In this new model a program is processed by a sequence of processors which takes as input a program and generates a new program. Each evaluation step is called stage (see Figure 6.1).

![Figure 6.1: Execution model of a for staging](image)

Each processor $P_i$ can use additional resources to manipulate the input program. In this model it is possible to describe most of the lifetime of nowadays programs. Web applications, for instance, fits in this model because usually the web server executes a program and outputs a program to be executed by the Web browser.
Multi-stage programming languages are characterized by annotation mechanisms which indicates for each stage which portion of the program should be executed [66], [73].

Lisp programmers have annotated their programs since the long time with macros. Quasi-quotation is the mechanism used by Lisp programmers to define code templates that should be filled when some information becomes available. In [28] Paul Graham characterizes Lisp macros as a unique mechanism by which “Lisp programmers can, and often do, write programs to write their programs for them”.

Although Lisp macros are a form of multi-stage program: the execution of a macro generates the code that is inserted where the macro is invoked and executed at the next stage. There is a problem with Lisp macros as a multi-stage mechanism: quasi-quotation is unaware of the special needs of variables and binding forms; it doesn’t ensure that variables occurring in a back-quoted expression are bound to the rules of static scoping.

Nevertheless Lisp macros have been extensively used and have influenced subsequent works. In particular the quasi-quotation has been used in C++ template meta-programming (where methods are the back-quoted code and arguments are the comma expressions\(^\text{32}\)); MetaML [73] uses a similar notation though a code data-type is introduced. In both cases the quasi-quotation mechanism is checked to ensure that the code generation is not just syntactically expanded in a context leading to potential problems.

As discussed in the previous chapter, there have been many attempts of generating code that requires minimal support at runtime. If it would be possible to write programs generating other programs in these frameworks we would have a framework for multi-stage programming.

In the rest of this section we discuss how to implement staged computations using Code Bricks. Moreover it is discussed how code values can be used for implementing multi-staged programs written in MetaML.

### 6.2.1 Staging: a First Example

Code Bricks allows expressing staged computations, as in languages like MetaML: at each stage a Code value is run and the result of its evaluation is a new Code value.

Consider the following example:

\(^{32}\)In LISP comma is used to specify expressions that shouldn’t be considered quoted in a quoted list.
delegate void B();
class FuncPair {
    Code M1; // We assume it with the same signature of B
    Code M2; // We assume it with the same signature of B
}

static void combine(B a, B b) {
    a();
    b();
}

static Code selectFunc(int v, FuncPair p) {
    return (v % 2 == 0) ? M1 : M2;
}

static Code job(int n, FuncPair p) {
    Code c = new Code(combine);
    Code ret = c.Bind(Code.Free, selectFunc(0, p));

    for (int i = 1; i < n + 1; i++)
        ret = ret.Bind(c.Bind(Code.Free, selectFunc(i, p)));

    ret = re.Bind(selectFunc(n + 1, p));
    return ret;
}

static void Main(string[] args) {
    FuncPair p = new FuncPair();
    p.M1 = ...; // Load it with a code value
    p.M2 = ...; // Load it with a code value
    Code c = (new Code(job)).Bind(9, p);
    //...
    Code d = c(); // First stage
    d(); // Second stage
}

We can see two stages in this program:

◊ Inside `Main` the code value `c` is executed
◊ The output of the execution of the first stage is the code value `d`
  which is then executed

In the example the output of the first stage is the code corresponding to 11
method calls of `M1` and `M2` alternated.

Of course the example is very simple; nonetheless it shows the essence of
representing a staged computation using code: we use a method which outputs
a new code value. Once executed the new value may produce another code
value and so on.
6.2.2 Code Bricks to Support MetaML

MetaML [73] is a programming language which provides specific annotation for controlling when a portion of code is executed. Code is introduced as data type so that the program can manipulate it; the idea of representing code values with type `Code` presented in this thesis comes from MetaML.

MetaML provides four staging annotations:

1. Brackets `<>` construct a code fragment
2. Escape `~` combines code fragments
3. Run `run` executes a code fragment
4. Lift `lift` constructs a code fragment from a ground value

Let us consider the use of these annotations. For each annotation we discuss how can be represented using Code Bricks.

Brackets are used to delay the execution of expressions. Consider the following example:

```plaintext
-| val result0 = 1 + 5;
val result0 = 6 : int
-| val code0 = < 1 + 5 >;
val code0 = < 1 %+ 5 > : <int>
```

The `%` sign is used to indicate that in the code value the `'+` is not free.

We use a code value to represent a delayed computation:

```csharp
static int _code0() { return 1 + 5; }
// ...
Code code0 = new Code(typeof(SomeType).GetMethod("_code0"));
```

Note that the type of the code value can be deduced because MetaML infers the type of code values.

Escape is used to combine bracketed expressions into larger ones. Consider for instance:

```plaintext
-| val code1 = <(~code0, ~code0)>
val code1 = <(1 %+ 5, 1 %+ 5)> : <int * int>
```

The same transformation can be achieved with code values using `Bind_

```csharp
delegate int __code0();
struct IntPair { public int x; public int y; }
static IntPair _code1(__code0 c1, __code0 c2) {
    IntPair ret = new IntPair();
    ret.x = c1();
    ret.y = c2();
    return ret;
}
```
// ...
Code c = new Code(typeof(SomeType).GetMethod("_code1");
Code code1 = c.Bind(code0, code0);

Note that neither CLR nor JVM, have the Pair type constructor. We could have defined Pair of objects but we would have to pay an extra boxing. With generic CLR Error! Reference source not found. we would be able to have Pair<int, int>.

The run annotation is used to execute a code fragment:
-| val result1 = run code0
val result1 = 6 : int

The run operator is trivial to implement with code values:
__code0 f = code0.MakeDelegate(typeof(__code0));
int result1 = f();

Lifting allows the injection values of ground type into a value of type code:
-| val code2 = lift 6;
val code2 = <6> : <int>
-| val code3 = lift 1 + 5;
val code3 = <6> : <int>

As shown in the code3 lift annotation doesn’t delay the execution. We can easily lift values using Bind:

static int lift(int i) { return i; }
// ...
Code c = new Code(typeof(SomeType).GetMethod("lift");
Code code2 = c.Bind(6);
Code code3 = c.Bind(1 + 5);

Note that with Bind we can bind delegates to values whereas lift cannot be used with functions.

Staged computations can be expressed by nesting the delay annotation:
-| <<5 + 5>>;
val it = <<5 %+ 5>> : <<int>>

The type of <<int>> is a code value which, when it is run, returns a code value that executed return an integer value.

Applying the conversion schema discussed above we have the following mapping into code values:

static int __stage2() { return 5 + 5; }
static Code __stage1(Code c) { return c; }
// ...
Code c = new Code(typeof(SomeType).GetMethod("__stage2");
Code c2 = new Code(typeof(SomeType).GetMethod("__stage1");
Code it = c2.Bind(c);

Note that, as in the example in section 6.2.1 we generate a code value which generates another code value and so on. Each application of run eliminates a bracket in MetaML and the execution of the code value returns the code value to be executed at the next stage.

If escape is used it is considered tied to the innermost delay annotation in which it appears:

|- val code5 = <<(5+5, ~(code2)>>;
val code5 = <<(5%+5, ~(code2)>> : <<int*int>>
|- val code6 = run code5;
val code6 = <(5%+5, 6) : <int*int>

Cross stage persistence allows code fragments to capture variables defined in outer fragments. Consider the following example:

let val a = 1+4 in <72+a> end

In this case the code fragment computed is <72%+%a> where %a indicates that a refers to a value generated in a previous stage. The value associated to a should be considered a constant. Though Cross Stage Persistency seems to be difficult to implement on a bytecode machine [10] it is trivial using Bind:

```c
static int CSP(int a) { return 72 + a; }
// ...
int a = 1 + 4;
Code c = new Code(typeof(SomeType).GetMethod("CSP"));
Code csp = c.Bind(a);
```

Of course a compiler which translates MetaML into code values and binding should ensure that the CSP safety is properly applied.

As we have shown MetaML annotations can be easily translated relying on Code Bricks. An ML compiler for .NET could be modified to support the MetaML syntax and generate code values and bindings.

The conversion schema we have discussed so far allows us to express MetaML staging with Code Bricks. Is the code generated efficient? What about the code generation process? The code generated uses only Bindx and Bindv thus no unneeded local variables are introduced. The generated code would be quite fast, though the generation process may require some resource because ML code fragments can be small and the number of code objects involved not irrelevant. Assuming that the compiler partially expands the code values it is possible to reduce this effect.
6.3 Multi-staging out of Process

In the model of staged computations there is no assumption about that the run of a stage should be performed into the same process of the previous run. As a matter of fact if the staging model doesn’t account for out of process staging the usefulness of the staging mechanism is significantly reduced.

Staged systems are useful because in the life-cycle of a program there are different stages where computations can be performed to partial specialize a program. As noted in [15] the information available about a program increases as the program crosses the various stages as shown in Figure 6.2.

![Figure 6.2: Typical lifecycle of an application.](image)

Template meta-programming and two level languages [15], [67] can exploit all the information available in a program source. Then the only application for in-process multi-staging would be RTCG and resource usage optimization.

Another issue with staging annotations like those used in MetaML and Code Bricks is that stages are anonymous. Why should a programmer use staging annotations if he can’t make any assumption about the resources available at some stage?

Besides the ability of expressing out-of process multi-staging would allow to implement many systems as single programs. An installer for instance can be expressed as a transformation that should be executed at install time. Instead of using a DSL we could define the installer as a portion of the program annotated as “install time”.

In order to be able to express multi-staging out of process, we need to be able to store the output of a stage into a program that will be executed at the next stage. Moreover the transformation should generate a code capable of generating another program.
Code Bricks seems to be the proper mechanism for implementing out of process multi-staging. It allows to transforming IL into IL whilst the transformation can be understood at language level. In the next section we propose a model for multi-staging out-of-process in which stages are labeled with names used to select the portions of code to be executed.

### 6.4 Multi Staging with Named Stages

As shown by Donal Lafferty [41] with its demo of CLIFileReader library it is possible to implement a program which reads an assembly and write another one which contains the same types. We can easily imagine adding some filter to this code that performs a variant of the *Bind* transformation.

We propose to write a special module that implements the notion of stage and is responsible for evaluating all the code labeled with the same name of the stage. The execution model is shown in Figure 6.3.

![Figure 6.3: Schema of the Stage Execution Environment](image)

Let us suppose that it is possible to write such module. Then we can put it in the standard stages of lifecycle as shown in Figure 6.4.

![Figure 6.4: Named Stages](image)
Each traditional stage is enriched (or replaced) by a Stage Execution Environment initialized with the appropriate label which is responsible of selecting the code that should be executed.

Now we should define how a single specialization block works. The Stage EE is simply an instance of the STEE whose execution is driven by a meta-program which manipulates the input program for generating the object program.

To indicate which portion of a program should be executed by Stage EE we rely on the extensible metadata provided by CLR. We introduce dummy methods that simply have the appropriate signature and an empty body. A method that is labeled with a custom attribute of type `StageAttribute` which indicates when the method should be executed and which method is responsible for generating the code value that should replace its invocation. Let us consider again the matrices domain:

```java
public class Matrix {
    double[,] m;
    Matrix(int c, int r) { ... }

    [Stage("CompileTime", "Matrix", "AddCode")]
    Matrix Add(Matrix a, Matrix b) { return null; }
    ...
}
```

In this case the `Add` method is labeled with “CompileTime”; this means that all the invocations of such method should be executed by StageEE with the label “CompileTime”. Every top level invocation of the method should be replaced with the code value (that should have a compatible signature) produced by running the `AddCode` in the class `Matrix`.

How can we merge a sequence of `Add` invocations? We note that in STEEs it is fairly easy to deduce the arguments passed to a function by looking just at the intermediate language. Thus we can generate a code value which represents the various arguments as the result of the application of `Bind`. Thus the top level method call can inspect its arguments and decide what code should be generated in the very similar way as `ToMatrix` method.

We note that additional stages can be added as needed: a program that is incapable of exploiting the information available at that stage will pass the stage unaltered. Besides if a program requires a stage that is not present may block the execution on before runtime.

---

33 Also JVM metadata are extensible, though the extension is not exposed to the language or from reflection.
The model proposed in this section is still in the design phase: it is not evident how we should manipulate method calls in order to provide the maximum amount of information that is made available to the meta-program.
7 Conclusions

In this chapter we review and discuss our work and its possible applications. We outline future work, and draw some conclusions about the scope of our research.

7.1 Review

This thesis describes a model and an implementation for code generation by partial application on a particular class of execution environments that we have called strongly typed execution environment.

We have described and discussed a particular schema for building code values out of precompiled methods. Code values can be combined together, like Lego bricks, to build new code values. The particular transformation we’ve used for combining code values has been proven correct and complete. It is correct because it generates code that behaves as the partial application of the arguments to a function. The operator can generate only type safe code avoiding the generation of broken code at runtime.

The Bind operator is in fact the combination of three different transformations called Bind$_\phi$, Bind$_\chi$, and Bind$_\pi$. The operator is also complete in the sense that we have been able to find a finite number of primitive code values (expressed as methods) that can be combined to compile programs of an imperative language.

With code values and Bind it is possible to implement transformations that closely recall the first and the second Futamura projections.

In chapter 4 we’ve discussed the implementation of Code Bricks, which is the library that implements the code type and the Bind operator described the previous chapters. The implemented transformation is essentially a superset of the one described in the formal model with the introduction of several optimizations to improve the quality of generated code. Nonetheless the formal model captures the essential traits of the transformation making of some use the formal properties proven within the model.

There is only one aspect that hasn’t been captured by the formal model: access control to members. In the formal model we can treat in the same way code
coming from instance and static methods; besides in the real CLR the self reference may have access to private and protect members and a static method may have not.

The lazy implementation of the transformation is very effective and helped us to realize that there are properties that can be derived from the constituents of binding.

Code values can be executed by constructing a sort of closure that contains the reference to the generated code and the reference to the environment. We use delegates to represent the executable version of a code value so that the type system should not be changed to introduce a new element.

In chapter 5 we’ve discussed the applications of Code Bricks different from meta-programming. In particular we discussed RTCG based on code values and binding. We provided a case study on .NET regular expression compiler; we showed how we’ve reverse engineered the compiler in order to write code values and bindings that would produce the same IL as output. It turned out that the C# code obtained by this process is very similar to the code of the interpreter; this result may indicate that the technique of turning interpreters in compilers can be effective with Code Bricks.

We have discussed the possible use of code values and bind for implementing Aspect Oriented Programming tools. The code transformation that inserts some code before or after a code value is easy to be expressed using a single code combinator and the Bind$\chi$ transformation.

Domain Specific Languages can be implemented with Code Bricks along the same lines of C++ template meta-programming. The fact that operators of the DSL can coincide with simple methods makes it easy implementation of DSLs with Code Bricks.

PCC is based on the assumption that the host computer defines a safety policy that can be verified without requiring any digital signature mechanism. We propose a similar schema in which we exploit the nature of code generation with Bind to guarantee that a set of operators can be published on the Web and assembled on demand being guaranteed that a property $P$ holding for all the operators is extended to all possible code values obtained by their combination.

Chapter 5 concludes with a brief discussion of how Code Bricks can be targeted by compilers to implement non trivial support such as DCG and closures by means of code values and bind.

Chapter 6 is devoted to meta-programming and multi-stage programming with Code Bricks. We’ve discussed how to do meta-programming in the same way as C++ template meta-programming does. Then we focused on multi-stage
programming and we have shown how the four staging annotations of MetaML can be translated in expressions based on Code Bricks.

We discussed the limits of what we’ve called in-process-staging with respect to its counterpart called out-of-process staging. The latter form of staging provides a framework that allows us to refocus the current software lifecycle as an application of out and in process multi-staging application. We concluded the chapter with a proposal of a general architecture for out-of-process multi-staging based on Code Bricks.

The architecture is based on a module that is capable of filtering an assembly and executes a subset of method invocations labeled with custom attributes. The result of the process is a new assembly that has been partially specialized by executing all the calls of methods annotated with a label equal to the one used to start the Staging EE.

7.2 Summary of Results

We summarize the key results as follows:

◊ The extension proposed to the reflection in term of code values and Binds seems to be quite effective and useful;
◊ The Bind transformation has interesting properties and is fast in generating code;
◊ The way we defined the transformation allows control over the operator at language level though the transformation is performed at IL level;
◊ The code values generated with code values and binding are type-safe;
◊ It is possible to turn interpreters into simple compilers simplifying the efficient implementation of DSLs;
◊ Code values can be deconstructed allowing a certain degree of pattern matching against runtime generated code;
◊ Partial evaluation of code generation: we rely on precompiled (and hopefully optimized) methods exploiting compile time to generate code fragments to be combined at runtime;
◊ Cross-language interoperability: it is possible to exploit the “common language” nature of CLR and use methods compiled in one language from another language, playing the strengths of each language;
There are many applications of the operator that seems to be adequate for being used as support for implementing languages that require RTCG, meta-programming and multi-staging capability.

Code Bricks can be used to implement the Staged EE which is the building block of out-of-process multi-staged systems.

7.3 Open Problems and Promising Research Directions

There are many issues and applications of Code Bricks that should be investigated. In particular we should perform additional studies to determine the real range of applicability of the results which seems to be quite promising.

Interesting questions and future research directions are the following:

- How to integrate parametric polymorphism with code values: the signature could be made explicit using type parameters. Moreover we could define generic combiners which are instantiated during binding accordingly to expected types;
- Writing compilers using Code Bricks: is it possible? How good is the generated code? How could we improve the model to allow compiler writers to exploit the already existing compilers?
- Runtime configuration of applications: big applications with complex interfaces could specialize themselves depending on the configuration reducing the working set needed the application to be executed;
- Web applications can be seen as out-of-process multi-stage applications: is it possible to express dynamic pages as the first stage of a staged application?
- What are the limits of the model? How can we implement code combinators for expressing code generation for type-less scripting languages?
- How can we improve the quality of generated code? Is it possible to move the transformation inside the JIT compiler? Do we get any real benefit?
8 Appendix

8.1 Extension of the Type Safety Theorem’s proof

In this section we extend the proof of the Theorem 3.1 provided by Gordon and Syme in [27] to the rules we have added to the model to deal with local variables and static methods. Because of our extension of stack frames the whole proof should have been reviewed, though the changes are additive and doesn’t affect the existing proof.

We have added four rules to the execution system: (Eval ldloca), (Eval stloc), (Eval call) and (Eval add). The (Eval call) case in the proof is almost identical to (Eval call instance) with the exception of the first argument of the method that in (Eval call) has no restriction.

8.1.1 Basic Lemmas

In this section are reported facts used in the extension of the proof.

Lemma 8.1: If \( \sum \not\in v : A \) and \( A <: A' \) then \( \sum \in v : A' \).

Proof. See Lemma 3 in appendix A.1 in [27]. \( \square \)

Lemma 8.2: The relation \( H \leq H' \) is reflexive and transitive (that is, for all \( H, H', H'' \), \( H \leq H \), and, if \( H \leq H' \) and \( H' \leq H'' \) then \( H \leq H'' \)).

Proof. See Lemma 9 in appendix A.1 in [27]. \( \square \)

8.1.2 Eval ldloca

\[
\sigma = (h, f_{r1} \ldots f_{ri}) \\
\sigma \vdash \text{ldloca } j \mapsto [i, j] : \sigma
\]

(Eval ldloca)

By assumption, \((H, S Fr) \vdash (h, f_{r1} \ldots f_{ri})\) and \(Fr \vdash \text{ldloca } j : B\). Because of \(Fr \vdash \text{ldloca } j : B\) we must have \(j \in 0..k\) and \(A_{j:s} <: B\) where \(Fr = .\text{args}(A_0, \ldots, A_n) .\text{locals}(A_0', \ldots, A_k')\). Because of \((H, S Fr) \vdash (h, f_{r1} \ldots f_{ri})\) we must have \(S = Fr_{r1} \ldots Fr_{ri} \) and \(Fr = Fr_{ri} \) for some \(Fr_{r1}, \ldots, Fr_{ri}\). By (Ptr Loc), \(i \in 1..l\) and \(Fr_i = .\text{args}(A_0, \ldots, A_n) .\text{locals}(A_0', \ldots, A_k')\) and \(j \in 0..n\) imply \((H, Fr_{r1} \ldots Fr_{ri}) \vdash [i, j] : A_{j:s}\). By Lemma

\[\text{34} \text{ The original (Eval call) has been renamed in (Eval call instance).}\]
8.1, this and \( A'_i <; B \) imply that \((H, S Fr) \vdash [i, j] : B\). Take \( H' = H \). By Lemma 8.2, \( H \leq H' \). We conclude \( H \leq H' \) and \((H', S Fr) \vdash [i, j] : B \) and \((H', S Fr) \vdash \sigma \).

8.1.3 Eval stloc

\[
\sigma \vdash a \mapsto u \cdot \sigma' \quad \sigma' = (h', fr_1 \ldots fr_i)
\]

By assumption, \((H, S Fr) \vdash \sigma\) and \( Fr \vdash a \) stloc \( j \) : \( B \). Because of \( Fr \vdash a \) stloc \( j \) : \( B \), we must have \( Fr = .\text{args}(A_0, \ldots, A_n) .\text{locals}(A_0', \ldots, A_i') \) and \( Fr \vdash a : A'_i \) and \( j \in 0..k \) and \( \text{void} <; B \). By induction hypothesis, \((H, S Fr) \vdash \sigma \) and \( Fr \vdash a : A'_i \) and \( \sigma \vdash a \mapsto u \cdot \sigma' \) imply there exists a heap type \( H' \) such that \( H \leq H' \) and \((H', S Fr) \vdash u : A'_i \) and \((H', S Fr) \vdash \sigma' \). Because of \((H', S Fr) \vdash (h', fr_1 \ldots fr_i)\), we must have \( S = Fr_i \ldots Fr_{i+1} \) and \( Fr = Fr_i \) for some \( Fr_i, \ldots, Fr_{i+1} \). By (Ptr loc), \( i \in 1..l \) and \( Fr_i = .\text{args}(A_0, \ldots, A_n) .\text{locals}(A_0', \ldots, A_i') \) and \( j \in 0..k \) implies \((H', Fr_1 \ldots Fr_i) \vdash [i, j] : A'_i \) \&. By Lemma 8.2, \((H', S Fr) \vdash (h', fr_1 \ldots fr_i)\), and \((H', S Fr) \vdash [i, j] : A'_i \). By (Res void), \((H', S Fr) \vdash 0 : \text{void} \) and then by Lemma 8.1, \( \text{void} <; B \) implies \((H', S Fr) \vdash 0 : B \). We conclude \( H \leq H' \) and \((H', S Fr) \vdash \text{update}(\sigma', [i, j], u)\).

8.1.4 Eval add

\[
\sigma \vdash a \mapsto v_1 \cdot \sigma' \quad \sigma \vdash b \mapsto v_2 \cdot \sigma'' \quad v = v_1 + v_2
\]

By assumption, \((H, S Fr) \vdash \sigma\) and \( Fr \vdash a \) \& add \( : \text{int32} \). Because of \( Fr \vdash a \) \& add \( : \text{int32} \) and (Body add), we must have \( Fr \vdash a \) \& int32 \( : \text{int32} \) and \( Fr \vdash b \) \& int32 \( : \text{int32} \). By induction hypothesis, \((H, S Fr) \vdash \sigma\) and \( Fr \vdash a \) \& int32 \( : \text{int32} \) and \( \sigma \vdash a \mapsto u \cdot \sigma' \) imply there exists a heap type \( H' \) such that \( H \leq H' \) and \((H', S Fr) \vdash u : \text{int32} \) and \((H', S Fr) \vdash \sigma' \). By induction hypothesis, \((H', S Fr) \vdash \sigma' \) and \( Fr \vdash b \) \& int32 \( : \text{int32} \) and \( \sigma' \vdash b \mapsto v \cdot \sigma'' \) imply there exists a heap type \( H'' \) such that \( H' \leq H'' \) and \((H'', S Fr) \vdash v : \text{int32} \) and \((H'', S Fr) \vdash \sigma'' \). By Lemma 8.2, \( H \leq H' \) and \( H' \leq H'' \) imply \( H \leq H'' \). We conclude \( H \leq H'' \) and \((H'', S Fr) \vdash v : \text{int32} \) and \((H'', S Fr) \vdash \sigma'' \).

8.2 Definition of \( \Pi \vdash a \mapsto a' \)

In this section we give the full definition of the transformation on BIL instruction sequences:

\[
\Pi \vdash \text{ldc.i4} \rightarrow \text{ldc.i4}
\]
\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b \rightarrow b' \]
\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b_0 \rightarrow b'_0 \quad \Pi \vdash b_1 \rightarrow b'_1 \]
\[ \Pi \vdash a \quad b \quad \text{cond} \rightarrow a' \quad b'_0 \quad b'_1 \quad \text{cond} \]  
(Ren Seq)

\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b \rightarrow b' \]
\[ \Pi \vdash a \quad b \quad \text{while} \rightarrow a' \quad b'_0 \quad b'_1 \quad \text{while} \]  
(Ren While)

\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b \rightarrow b' \]
\[ \Pi \vdash a \quad b \quad \text{stind} \rightarrow a' \quad b'_0 \quad b'_1 \quad \text{stind} \]  
(Ren ldind)

\[ K = \text{void} \circ \text{ctor}(A_1, ..., A_n) \]
\[ \Pi \vdash a_i \rightarrow a'_i \quad \forall i \in 1..n \]
\[ \Pi \vdash a_0 \ldots a_n \text{newobj} K \rightarrow a'_0 \ldots a'_n \text{newobj} K \]
\[ M = B \circ :m(A_1, ..., A_n) \]
\[ B \circ m(A_1, ..., A_n) \in \text{StaticMethods} \]
\[ \Pi \vdash a_i \rightarrow a'_i \quad \forall i \in 0..n \]
(Ren newobj)

\[ \Pi \vdash a_0 \ldots a_n \text{callvirt} \text{instance} M \rightarrow a'_0 \ldots a'_n \text{callvirt} \text{instance} M \]
\[ M = B \circ :m(A_1, ..., A_n) \]
\[ B \circ m(A_1, ..., A_n) \in \text{StaticMethods} \]
\[ \Pi \vdash a_i \rightarrow a'_i \quad \forall i \in 0..n \]
(Ren callvirt)

\[ \Pi \vdash a_0 \ldots a_n \text{call} \text{instance} M \rightarrow a'_0 \ldots a'_n \text{call} \text{instance} M \]
\[ M = B \circ :m(A_1, ..., A_n) \]
\[ B \circ m(A_1, ..., A_n) \in \text{StaticMethods} \]
\[ \Pi \vdash a_i \rightarrow a'_i \quad \forall i \in 1..n \]
(Ren call instance)

\[ \Pi \vdash a \rightarrow a' \]
\[ \Pi \vdash \text{a} \quad \text{box} \rightarrow \text{a}' \quad \text{box} \]  
(Ren box)

\[ \Pi \vdash a \rightarrow a' \]
\[ \Pi \vdash a \quad \text{unbox} \rightarrow \text{a}' \quad \text{unbox} \]  
(Ren unbox)

\[ \Pi \vdash a \rightarrow a' \]
\[ \Pi \vdash a \quad \text{ldfld} A \\ c:f \rightarrow a' \quad \text{ldfld} A \\ c:f \]  
(Ren ldflda)

\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b \rightarrow b' \]
\[ \Pi \vdash a \quad b \quad \text{stfld} A \\ c:f \rightarrow a' \quad b'_0 \quad b'_1 \quad \text{stfld} A \\ c:f \]  
(Ren stfld)
\[ \Pi \vdash a \rightarrow a' \quad \Pi \vdash b \rightarrow b' \] (Ren add)

\[ \Pi \vdash \text{add} a \quad \Pi \vdash \text{add} \rightarrow a' \quad \text{add} b' \] (Ren add)

\[ \Pi((\text{arg}, j)) = (\text{arg}, i) \] (Ren \text{ldarga} \ \text{arg})

\[ \Pi \vdash \text{ldarga} j \rightarrow \text{ldarga} i \] (Ren \text{ldarga} \ \text{loc})

\[ \Pi((\text{arg}, j)) = (\text{arg}, i) \] (Ren \text{ldarga})

\[ \Pi \vdash \text{ldarga} j \rightarrow \text{ldarga} j \] (Ren \text{ldarga})

\[ \Pi((\text{arg}, j)) = (\text{loc}, i) \] (Ren \text{starg} \ \text{arg})

\[ \Pi \vdash \text{starg} j \rightarrow \text{starg} i \] (Ren \text{starg} \ \text{loc})

\[ \Pi((\text{loc}, j)) = (\text{loc}, i) \] (Ren \text{ldloca} \ \text{loc})

\[ \Pi \vdash \text{ldloca} j \rightarrow \text{ldloca} i \] (Ren \text{ldloca})

\[ \Pi((\text{loc}, j)) = (\text{loc}, i) \] (Ren \text{stloc} \ \text{loc})

\[ \Pi \vdash \text{stloc} j \rightarrow \text{stloc} j \] (Ren \text{stloc})

\[ M = B :: \text{Invoke}(A_0, ..., A_n) \]

\[ B \text{ Invoke}(A_0, ..., A_n) \in \text{StaticMethods} \]

\[ \Pi((\text{arg}, i)) = (\text{inline}, k, b) \quad k_i = k + i \quad \Pi \vdash a_i \rightarrow a'_i \]

\[ (\text{arg}, i) \rightarrow \text{fin} (\text{loc}, k) \forall i \in 1..n) \quad \Pi \vdash b \rightarrow b' \] (Ren \text{Invoke})

\[ \Pi \vdash \text{ldarg} a_i \quad \ldarg a_i \ldarg a' \text{ callvirt instance } M \rightarrow \]

\[ a'_i \quad \text{stloc} k_i \quad ... \quad a'_i \quad \text{stloc} k_n \quad b' \]

\[ \Pi((\text{arg}, i)) = (\text{inline}, k, b) \] (Ren \text{check ldarga})

\[ \Pi \vdash \text{ldarga} i \rightarrow \bot \] (Ren \text{check starg})

\[ \Pi((\text{arg}, i)) = (\text{inline}, k, b) \] (Ren \text{check starg})
8.3 Interpreter Theorem Proof

The proof of this theorem is quite complex because the equivalence between the execution of the interpreted form and the code value holds only at the end of the call. In fact most of the store is the same also whilst the code values are evaluated though there is a difference on the frames allocated on top of the stack.

We introduce a definition of equivalence to be used during the evaluation. This notion of equivalence implies the store equivalence at the end of execution of the call.

**Definition 8.1:** Let \( \sigma = (h, s) \) be a store, the class of stores of the form \( (h, s f r_0 ... f r_n) \) are said to be call equivalent to \( \sigma \).

We introduce some basic lemmas used in the proof of the theorem. The essence of transformation performed by \textit{Bind} is to perform some calculation that produces a value that should be used in place of an argument. The value is stored in a local variable instead of the argument and the body is modified to use it instead of the original argument. The following lemma ensures that this kind of transformation guarantees call equivalence with respect to a given store.

**Lemma 8.3:** Let \( b \) a sequence of instructions in which neither ldloca \( j \) nor stloc \( j \) occur. Let \( \Pi = \{ (arg, i) \rightarrow (loc, j) \} \) and \( \Pi \vdash b \rightarrow b' \). If \( (h, s ... .args(v_0, ..., v_n).locals(l_0, ..., l_i)) \vdash b \leftrightarrow v \cdot (h', s' ... fr') \) and \( (h, s ... .args(v_0, ..., v_n).locals(l_0', ..., l_{i+1}', v, l_{i+1}', ..., l_{k'}) \vdash b' \leftrightarrow v \cdot (h'', s'' ... fr'') \) then \((h', s' fr')\) and \((h'', s'' fr'')\) are call equivalent with respect to the store \((h', s')\), thus \( h'' = h' \) and \( s'' = s' \).

**Proof.** The proof is by structural induction on the definition of code transformation. We restrict the proof to the instructions that interact with the store: ldind, stind, ldarga, starg, ldloca, stloc, ldfld and stfld.

(1dind)

By (Eval ldind) we know that \( \sigma \vdash a \ \text{ldind} \Rightarrow \text{lookup}(ptr) \cdot \sigma' \) under the hypothesis that \( \sigma \vdash a \Rightarrow \text{ptr} \cdot \sigma' \). Let \( a' \) be obtained by applying \( \Pi \), the execution of \( \sigma' \vdash a' \ \text{ldind} \Rightarrow \text{lookup}(ptr') \cdot \sigma'' \). By induction hypothesis \( \sigma \) and \( \sigma' \) are call equivalent as well as \( \sigma' \) and \( \sigma'' \).

There are only three ways to obtain the pointer \( \text{ptr} \) by means of ldarga, ldloca and ldflda. We know that \( a \) and \( a' \) differ only by the instructions loading and storing the \( i \)th argument which are converted into instructions that load and store the \( j \)th local variable. Thus \( \text{ptr} \) and \( \text{ptr}' \) will be different
only in this case and will point to the $i$th argument for $a$ and to the $j$th local for $a'$. In both cases the value returned by `lookup` will be the same: the local variable $j$ is unused in $b$ thus the only changes are performed in $a'$ only as it is the $i$th argument.

(stind)

By (Eval `stdind`) we know that $\sigma \vdash a \ c \ stdind \Rightarrow 0 \cdot \text{update}(\sigma', \text{ptr}, v)$ if $\sigma \vdash a \Rightarrow \text{ptr} \cdot \sigma' \text{ and } \sigma \vdash c \Rightarrow v \cdot \sigma'$. Let $\prod \vdash a \rightarrow a'$ and $\prod \vdash c \rightarrow c'$. Thus $\sigma'' \vdash a \ c \ stdind \Rightarrow 0 \cdot \text{update}(\sigma'', \text{ptr'}, v)$ if $\sigma'' \vdash a \Rightarrow \text{ptr} \cdot \sigma'' \text{ and } \sigma'' \vdash c \Rightarrow v \cdot \sigma''$. By induction hypothesis $\sigma$ and $\sigma''$ are call equivalent as well as $\sigma'$ and $\sigma''$. Like in the previous case $\text{ptr}$ and $\text{ptr'}$ would be the same unless it has been obtained through `ldarga` in $a$. Again in both cases the same value $v$ is stored into the argument or in the local variable. The resulting state will be call equivalent.

(ldarga)

We consider only the case in which the argument of `ldarga` is $i$. In this case the instruction is replaced by $\prod$ with `ldloca` $j$. In all other cases in both $b$ and $b'$ `ldarga` instructions behave in the same way. Both instruction doesn’t affect the store so the stores are call equivalent and remain so after executing the instruction.

(starg)

As in the previous case we restrict the proof to argument $i$ which is the only index replaced by `stloc` $j$. By (Eval `starg`) $\sigma \vdash a \ loc \ i \Rightarrow 0 \cdot \text{update}(\sigma', (f, i), u)$, if $\sigma \vdash a \Rightarrow u \cdot \sigma'$ and $\sigma' = (h', s' \ldots f r)$. Besides by (Eval `stloc`) $\sigma'' \vdash a \ stloc \ i \Rightarrow 0 \cdot \text{update}(\sigma''', [f', i], u)$, if $\sigma'' \vdash a \Rightarrow u \cdot \sigma'''$ and $\sigma''' = (h', s' \ldots f r')$. By induction hypothesis $\sigma$ and $\sigma'$ are call equivalent as well as $\sigma'$ and $\sigma''$. After the execution the store changes only in the topmost frame on the stack preserving the call equivalence of the result.

(ldloca)

These instructions are unaffected by the code transformation (with exception for local $j$), though the store is left unchanged by their execution.

(stloc)

These instructions affect only the topmost frame thus the call equivalence is trivially granted.

(ldflda)
The address of a field is the same in both $b$ and $b'$ thus the same pointer is returned. Perhaps this instruction doesn’t affect the store so the call equivalence is preserved.

\[(\text{stfld})\]

By (Eval \text{stfld}) we have $\sigma \mathbin{\vDash} a \text{stfld} A \mathbin{:} f \mapsto 0 \cdot \text{update}(\sigma', \text{ptr} \cdot f, v)$ if $\sigma \mathbin{\vDash} a \mapsto \text{ptr} \cdot \sigma'$ and $\sigma' \mathbin{\vDash} c \mapsto v \cdot \sigma'$. By induction hypothesis the store $\sigma'$ is call equivalent executing both $b$ and $b'$. The pointer $\text{ptr}$ should be the same in both cases thus the value $v$ is stored in the same location in the heap in both cases. Thus the call equivalence is preserved. □

A similar lemma can be stated for renaming of argument values; this operation often occurs during code generation.

**Lemma 8.4:** Let $b$ a sequence of instructions in which neither $\text{ldarga}$ nor $\text{starg}$ occur. Let $\Pi = \{(\text{arg}, i) \mapsto (\text{arg}, j)\}$ and $\Pi \mathbin{\vDash} b \mapsto b'$. If $(h, s \ldots \text{args}(v_0 \ldots, v_l).\text{locals}(l_0 \ldots, l_k)) \mapsto b \mapsto v \cdot (h', s' \ldots \text{fr'})$ and $(h, s \ldots \text{args}(v_0 \ldots, v_{l-1}, v_j, v_{l+1} \ldots, v_k).\text{locals}(l_0', \ldots, l_k')) \mapsto b' \mapsto v \cdot (h'', s'' \ldots \text{fr''})$ then $(h', s' \text{fr'})$ and $(h'', s'' \text{fr''})$ are call equivalent with respect to the store $(h', s')$, thus $h'' = h'$ and $s'' = s'$.

**Proof.** Is similar to the proof of Lemma 8.3. □

We observe that the proof of Lemma 8.3 and Lemma 8.4 can be easily generalized to more general transformations as stated in the following lemma.

**Lemma 8.5:** Let $b$ a sequence of instructions and $\Pi$ a code transformation such that $\Pi \mathbin{\vDash} b \mapsto b'$, $\sigma \mathbin{\vDash} b \mapsto v \cdot \sigma'$, $\sigma' \mathbin{\vDash} b' \mapsto v \cdot \sigma''$ and $\sigma$ is call equivalent to $\sigma''$ and $\sigma'$ is call equivalent to $\sigma'''$. Let $i$ be an index of a valid argument and $j$ the index of a valid argument or local variable unused in $b$, such that the content of argument $i$ in the topmost stack frame is the same of the argument or local variable $j$ in the same frame. Consider the code transformation $\Pi'$ obtained by adding $(\text{arg}, i) \mapsto (\text{arg}, j)$ or $(\text{arg}, i) \mapsto (\text{loc}, j)$; let $\Pi' \mathbin{\vDash} b \mapsto b''$. If $\sigma' \mathbin{\vDash} b'' \mapsto v \cdot \sigma''$ then $\sigma'$ is call equivalent to $\sigma''$.

**Proof.** The proof is similar to the proof of Lemma 8.3. □

We can now prove the interpreter theorem. The essence of the theorem is that replacing the application of $\text{Bind}$ with normal application we get the body of a method equivalent to the code generated by $\text{Bind}$. The notion of call equivalence is very useful because allow us to dig into the execution of the call being able of speaking about a notion of equivalence during the execution of the two code values.
Theorem 8.1: Let \( c \in \text{Code} \) a code value in ground form, let \( c' \in \text{Code} \) be the interpreted form of \( c \); the following equivalence holds:
\[
\sigma \vdash \text{call} \ c \iff v \cdot \sigma' \iff \sigma \vdash \text{call} \ c' \iff v \cdot \sigma'
\]

Proof. The proof is based on the depth of the tree associated to the ground form associated to \( c \). The base case is the following:
\[
c = \text{Bind}(\Sigma, \text{methodof}(B \ cl::m(A_0, \ldots, A_n)), \varphi_0, \ldots, \varphi_n),
\]
where \( \varphi_i \in \text{Free} \) and \( i \in 0..n \),
\[
\text{methods}(cl)(B m(A_0, \ldots, A_n)) = \text{.locals}(A'_0, \ldots, A'_k) \ b
\]
In this case
\[
b' = \text{Interpret}(c, 0, 0, 0) = \text{ldarg} \ 0 \ldots \text{ldarg} \ n \ \text{call} \ cl::m(A_0, \ldots, A_n)
\]
\[
c' = <B \ \mu(A_0, \ldots, A_n), \ \text{.locals}() \ b, \ \text{.env}() >
\]

\( \Rightarrow \) We assume that \( \sigma \vdash \text{call} \ c \iff v \cdot \sigma' \); by \((\text{Eval Code})\) we know that \( \sigma = (h_0, s_0), (h_i, s_i) \vdash a_i \iff v_i \cdot (h_{i+1}, s_{i+1}) \ \forall i \in 0..n, (h_{n+1}, s_{n+1}.\text{args}(v_0, \ldots, v_n).\text{locals}(l_0, \ldots, l_i)) \vdash b \iff v \cdot (h', s') \iff \sigma = (h', s')^{35} \).

We apply \((\text{Eval Code})\) to \( c' \) so that \( b' \) is evaluated in the store \((h_{n+1}, s_{n+1}.\text{args}(v_0, \ldots, v_n).\text{locals}())\). We observe that \( b' \) is essentially another call so we should apply \((\text{Eval call})\): the arguments are just loaded into the new stack frame so the body \( b \) of \( cl::m \) is evaluated in the store where the heap is unchanged and the stack is loaded with the additional stack frame \( .\text{args}(v_0, \ldots, v_n).\text{locals}(l_0, \ldots, l_i) \). The stored in which \( b \) is evaluated is call equivalent to the one in which \( b' \) is evaluated. We observe that \((h_{n+1}, s_{n+1}.\text{args}(v_0, \ldots, v_n).\text{locals}()).\text{args}(v_0, \ldots, v_n).\text{locals}(l_0, \ldots, l_i)) \vdash b \iff v \cdot (h', s' \text{'fr')} \iff \sigma'.

\( \Leftarrow \) Is similar to the previous one.

We note that \text{Interpret} is defined in a way such that each case emits the code corresponding to an argument that should be passed to the method \( c::m \). Our proof aims to show that the body \( b \) of method \( m \) is executed in an store prepared by \( c' \) such that the transformation on \( b \) in \( c \) is call equivalent with respect to the store \( \sigma \).

The proof of equivalence of the execution of \( c \) and \( c' \) is conducted by structural induction on the definition of code transformation defined in

\[35\] Note that the environment of \( c \) is empty in this case.
section 8.2. As in the base case, we note that the execution of \( c' \) implies that \( b \) is evaluated in a store call equivalent from the one in which \( b' \), the transformed copy of \( b \), is evaluated: an additional frame is present in the first case, and it is used by \( c' \) to prepare the appropriate environment for the execution of \( b \).

The initial store is \( \sigma = (h, s) \), by applying (Eval Code) we know that the evaluation of \( a_0, \ldots, a_n \) leading to the resulting store \((h_{n+1}, s_{n+1})\). For \( c' \) the stack frame allocated is \( fr' \); for \( c \) it is allocated \( fr \). The two frames have the same arguments but the array of locals in \( fr' \) is smaller than the one in \( fr \). At the end of the body generated by Interpret there is a call to the method which loads an additional frame on the stack \( fr'' \).

We consider only the relevant cases in the proof: the instructions modified by Bind. In each case we show that the transformation performed by Bind on \( b \) affects the execution in the same way as \( c' \) prepares the execution environment for it.

\((\text{Bind}_v)\)

The definition of Interpret when a value is found is the following:

\[
\text{Interpret}(\text{Bind}(\sum, \text{methodof } (B \ c :: m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) = \\
1dloc k' \text{ Interpret}(\text{Bind}(\sum, \text{methodof } (B \ c :: m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i+1, j, k+1)
\]

where \( v \in \text{Val} \)

By definition of interpreted form we know that \( l \) is initialized with \( v_i \) (stored into .env member of the code value). Thus \((h_{n+1}, s_{n+1} fr') \vdash 1dloc k' \mapsto v_i \cdot (h_{n+1}, s_{n+1} fr')\) implies that \( a_i \) in (Eval call) evaluates to \( v_i \) and the \( i \)th argument in .args is bound to \( v_i \). Bind relies on \( \text{Bind}_v \) in this case which is defined as follows:

\[\text{Bind}_v(c, i, \sum, v) = \]
\[<B \mu(A_0, \ldots, A_i, A_{i+1}, \ldots, A_n), \text{locals}(A_0', \ldots, A_i', A_i) b', \]
\[.\text{env}((e_0, v_0), \ldots, (e_m, v_m), (k+1, v)) > \]
\[\text{where } i \in \{0..n\}, \]
\[c = <B \mu(A_0, \ldots, A_n), \text{locals}(A_0', \ldots, A_i') b, \text{env}((e_0, v_0), \ldots, (e_m, v_m)) >, \]
\[\sum \vdash v : A_i, \]
\[\text{pointerFree}(A_i), \]
\[\prod b \rightarrow b', \]
\[\prod = \{ (arg, i) \rightarrow (loc, k + 1), (arg, i+1) \rightarrow (arg, i), \ldots, (arg, n) \rightarrow (arg, n-1) \} \]

In this case the argument \( i \) is mapped into local variable \( k+1 \) and all the other arguments are shifted back by one position.
Applying Lemma 8.5 we can conclude that the store after the execution of $b'$ is call equivalent to the store resulting from the execution of $b$.

$\text{(Bind}_a\text{)}$

The definition of $\text{Interpret}$ in case an argument is bound to a code value whose return value should be used as input argument is the following:

$$\text{Interpret}(\text{Bind}(\Sigma \text{methodof}(B \colon c \colon m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i, j, k) =$$

$$\text{Interpret}(\text{Bind}(\Sigma \text{methodof}(B \colon c \colon m(A_0, \ldots, A_n)), v_0, \ldots, v_n), i + 1, j', k')$$

where $v_i \in \text{Code}$,

$\text{Bind}$ would apply $\text{Bind}_a$ to $v_i$, $j'$ and $k'$ the maximum $j$ and $k$ generated by the first $\text{Interpret}$ application

The corresponding definition in $\text{Bind}$ is based on $\text{Bind}_a$:

$$\text{Bind}_a(c, i, c') =$$

$$\langle B \mu(A_0, \ldots, A_i, A_i', \ldots, A_n', A_i, \ldots, A_0),$$

$$.\text{locals}(A_0', \ldots, A_i', A_i'', \ldots, A_n'', A) b' stloc j b''',$$

$$.\text{env}((e_0, v_0), \ldots, (e_m, v_m), (e_0' + k + 1, v_0'), \ldots, (e_m' + k + 1, v_m')) >$$

where $i \in 0..n$,

$$c = \langle B \mu(A_0, \ldots, A_i), .\text{locals}(A_0', \ldots, A_i', A) b, .\text{env}((e_0, v_0), \ldots, (e_m, v_m)) >,$$

$$c' = \langle A_i \mu(A_0', \ldots, A_i''), .\text{locals}(A_0'', \ldots, A_i'') b',$$

$$.\text{env}((e_0', v_0'), \ldots, (e_m', v_m')) >,$$

$$j = k + k' + 2,$$

$$\Pi = \{ (\text{arg}, 0) \rightarrow (\text{arg}, i), \ldots, (\text{arg}, n') \rightarrow (\text{arg}, i + n'),$$

$$(\text{loc}, 0) \rightarrow (\text{loc}, k + 1), \ldots, (\text{loc}, k') \rightarrow (\text{loc}, k + k' + 1) \},$$

$$\Pi' = \{ (\text{arg}, i + 1) \rightarrow (\text{arg}, i + n' + 1), \ldots, (\text{arg}, n) \rightarrow (\text{arg}, n + n' - 1),$$

$$(\text{arg}, i) \rightarrow (\text{loc}, j) \},$$

$$\Pi + b' \rightarrow b'''$$

In this case $\text{Interpret}$ generates the interpreted form of code value $v_i$ whose result is used as $i$th argument for $b$. We note that binding tree associated with $v_i$ is less deep than $c$ so we can rely on induction hypothesis. We observe that the interpreted form of $v_i$ is executed in the same store of its body with the only difference that local variables of $v_i$ are shifted in $c$.

Thus the execution of the interpreted form leads to a store call equivalent to the store produced by execution of $b'''$ in $c$ (the two stores differ only by the topmost frames on the stack as discussed before). They also produce the same value $v$ as result of execution. The value $v$ is bound to the $i$th argument.
in the stack frame in which $b$ will be executed in the interpreted version. When executed $c$ the same value $v$ is stored into a local variable $j$. By Lemma 8.5 we can conclude that the store after the execution of $b''$ is call equivalent to the store at end of the execution of $b$. We can conclude that the transformation $\text{Bind}_\pi$ produces the same changes to the store as the interpreted form: during execution the top stack frames will be slightly different but in the end the resulting state will be the same.

$(\text{Bind}_\chi)$

The definition of $\text{Interpret}$ corresponding to $\text{Bind}_\chi$ is the following:

$$\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \text{::}m(A_0, ..., A_n)), v_0, ..., v_n), i, j, k) =$$

newobj void $cl$::ctor()

$$\text{Interpret}(\text{Bind}(\Sigma, \text{methodof}(B \text{::}m(A_0, ..., A_n)), v_0, ..., v_n), i + 1, j, k)$$

where $v \in \text{Code},$

$\text{Bind}$ would apply $\text{Bind}_\chi$ to $v,$

$$\sigma + a_1 ... a_n \text{ call } v_i \mapsto v \cdot \sigma' \Rightarrow$$

$$\sigma + a_0 ... a_n \text{ callvirt } B' \text{::} \text{Invoke}(A_1, ..., A_n) \mapsto v \cdot \sigma'$$

This definition postulates the existence of an instance method named $\text{Invoke}$ of a class $cl$. The invocation of this method is equivalent to the execution of the code value. This definition approximates the notion of delegate in the runtime. The method $\text{Invoke}$ can be easily defined by means of $\text{Interpret}$ that, by induction hypothesis, exists. Note that we need an object of class $cl$ in order to invoke the method inside the definition of $c::m$.

The definition of $\text{Bind}_\chi$ is the following:

$$\text{Bind}_\chi(c, i, c') =$$

$$< B \mu(A_0, ..., A_i, A_{i+1}, ..., A_n),$$

$$\text{.locals}(A_0', ..., A_i', A_{i}''', ..., A_n''),$$

$$A_i'') b'',$$

$$\text{.env}((e_0, v_0), ..., (e_m, v_m), (e_0' + k + 1, v_0'), ..., (e_m' + k + 1, v_m')) >$$

where $i \in \mathbb{N},$

c = $< B \mu(A_0, ..., A_i), \text{.locals}(A_0', ..., A_i') b, \text{.env}((e_0, v_0), ..., (e_m, v_m)) >,$

c' = $< B' \mu(A_1', ..., A_n'), \text{.locals}(A_0''', ..., A_{i+1}'') b',$$

$$\text{.env}((e_0', v_0'), ..., (e_{m'}', v_{m'}')) >,$$

$A_i = \text{class } cl,$

$B' \text{::} \text{Invoke}(A_0, A_1', ..., A_n') \notin \text{dom}(\text{methods}(cl)),$

$B' \text{::} \text{Invoke}(A_0, A_1', ..., A_n') \in \text{StaticMethods},$

$$\Pi = \{ (\text{loc}, 0) \rightarrow (\text{loc}, k + 1), ..., (\text{loc}, k') \rightarrow (\text{loc}, k + k' + 1),$$

$$(\text{arg}, 1) \rightarrow (\text{loc}, k + k' + 2), ..., (\text{loc}, n') \rightarrow (\text{loc}, k + k' + n' + 1) \},$$
\[ \Pi' = \{ \text{arg}_i \to (\text{inline}, k + k' + 1, b''') \}, \]
\[ \Pi' b' \to b''', \]
\[ \Pi' b \to b'' \]

The \textit{i}th argument passed to \texttt{c::m} is an instance of class \texttt{cl}. Inside \texttt{b} the only use of such argument is to invoke the method \texttt{Invoke}, otherwise \texttt{Bind} would fail in generating the code by (Ren check ldarga) and (Ren check starg).

We note that arguments of the inlined code value are mapped by \texttt{Bind}_\pi into local variables starting from \( k + k' + 1 \). We know that the transformation is applied to the following pattern by (Ren \texttt{Invoke}):

\texttt{ldarg i \ldots a_n callvirt instance B c::Invoke(A_0, \ldots, A_n)}

Applying (Eval callvirt) we know that if \( a_1, \ldots, a_n \) evaluate to \( a_0, \ldots, a_0 \) that correspond to the arguments in the stack frame in which the body of \texttt{Invoke} is executed. In \texttt{c}, by (Ren \texttt{Invoke}), these values are stored into local variables and the reference to arguments are renamed in the same way. By application of Lemma 8.5 we can conclude that the execution of method invocation is call equivalent to execute the inlined body.

(\textit{Share} and \textit{Free})

The definition of \texttt{Interpret} in these cases are:

\texttt{Interpret(Bind(\Sigma, methodof(B c::m(A_0, \ldots, A_n))), v_0, \ldots, v_n), i, j, k) =}

\texttt{ldarg j Interpret(Bind(\Sigma, methodof(B c::m(A_0, \ldots, A_n))), v_0, \ldots, v_n), i+1, j+1, k) where v_\in Free, \forall l \in 0..n, v_l = v_i \Rightarrow i \leq l}

\texttt{Interpret(Bind(\Sigma, methodof(B c::m(A_0, \ldots, A_n))), v_0, \ldots, v_n), i, j, k) =}

\texttt{ldarg l Interpret(Bind(\Sigma, methodof(B c::m(A_0, \ldots, A_n))), v_0, \ldots, v_n), i+1, j, k) where v_\in Free,}

\[ \exists l \forall v_i = v_l \forall m \in 0..n, v_l = v_m \Rightarrow l \leq m \]

By applying \texttt{Bind} definition we know that if \( \phi \in Free \) is left open then the transformed body still use the argument possibly renamed. Thus the free arguments of \texttt{c} are the arguments bound to \textit{Free} values, possibly lifted by \texttt{Bind}_\pi.

We note that the arguments resulting from \texttt{Bind} are enumerated in order of appearance. Thus \texttt{Interpret}'s arguments appear in the same way as those in \texttt{c}. If all \textit{Free} elements in the invocation are different the argument \( j \) is mapped to argument \( i \) in the call of \texttt{c::m}. It is trivial to verify that \texttt{Bind} rename arguments in the same way so that the value loaded in the two cases is the same.
In case the same element of $\textit{Free}$ is used in $\textit{Bind}$ the function $\textit{Share}$ is applied: again we note that the effect of $\textit{Share}$ is to share the argument with the minimum index. Again the second rule of interpret applies the same policy preserving the correctness of execution.

## 8.4 Type Safety Theorem Proof

**Theorem 8.2:** Let $c = B \mu(A_0, ..., A_n), .\text{locals}(A_0', ..., A_k') b, .\text{env}((e_0, v_0), ..., (e_m, v_m)) >$, then $.\text{args}(A_0, ..., A_n) .\text{locals}(A_0', ..., A_k') \vdash b : B.$

**Proof.** Let us assume that $c$ is a ground form, the proof can be trivially extended to all code values introducing a method for each ground form as it has been done in Corollary 3.1.

The proof is by induction on the depth of the bind tree associated to $c$. The base case is when

$$c = \text{Bind}(\Sigma, \text{methodof}(B m(A_0, ..., A_n)), \varphi_1, ..., \varphi_n),$$

where $\varphi \in \textit{Free}$ and $i \in 0..n$,

$$\text{methods}(cl)(B m(A_0, ..., A_n)) = .\text{locals}(A_0', ..., A_k') b$$

By (Stat methods) we can conclude

$$\.\text{args}(A_0, ..., A_n) .\text{locals}(A_0', ..., A_k') \vdash b : B.$$

The induction step should consider the four possible cases in the definition of $\textit{Bind}$ depending of the function applied: $\textit{Share}$, $\textit{Bind}_0$, $\textit{Bind}_\sigma$ and $\textit{Bind}_\chi$.

**(Share)**

The $\textit{Bind}$ definition when $\textit{Share}$ function is used is the following:

$$c = \text{Bind}(\Sigma, c', v_0, ..., v_n) = \text{Bind}(\Sigma, .\text{Share}(c', v_0, ..., v_n))$$

where $\forall i \in 0..n \; v_i \epsilon \textit{Free}, \exists i, j \in 0..n, v_i, v_j \epsilon \textit{Free}, v_i \neq v_j$

$$c' = B \mu(A_0'', ..., A_n''), .\text{locals}(A_0'', ..., A_k'') b'', .\text{env}((e_0, u_0), ..., (e_m, u_m)) >$$

By induction ipothesis $.\text{args}(A_0, ..., A_n) .\text{locals}(A_0', ..., A_k') \vdash b' : B$. The aim of $\textit{Share}$ is to change the body of $c'$ in order to share arguments. To prove that $\textit{Share}$ preserves the types we rely on induction on the number of pairs that should be collapsed by the function.

The base case is trivial because $c$ is equal to $c'$:

$$\textit{Share}(c, v_0, ..., v_n) = (c, v_0, ..., v_n) \quad \forall i, j \in 0..n, v_i, v_j \epsilon \textit{Free}, v_i = v_j \Rightarrow i = j$$
Given that the property holds for the code value \( \text{Share}(c', v_0, ..., v_i, ..., v_n) \) we should show that the property applies to \( \text{Share}(c, v_0, ..., v_n) \) obtained as follows:

\[
\text{Share}(c, v_0, ..., v_n) = \\
(\text{Share}(c', v_0, ..., v_i, ..., v_n), v_0, ..., v_i, ..., v_n)
\]

where \( i \in 0..n \), \( j < i \), \( v_i \in \text{Free} \), \( v_i = v_i \). \( A_i = A_j \),

\[
c = \langle B \mu(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \rangle b . \text{env}((e_0, u_0), ..., (e_m, u_m)) >,
\]

\[
c' = \langle B \mu(A_0, ..., A_0', A_1', ..., A_k') . \text{locals}(A_0', ..., A_1') \rangle b',
\]

\[
\begin{array}{c}
\Pi = \{ (\text{arg}, j) \rightarrow (\text{arg}, i) \}, \\
\Pi \vdash b \rightarrow b'
\end{array}
\]

We note that the typing judgment has the same structure of the code transformation defined in 8.2. We rely on structural induction to prove that if \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash b : B \) then \( \text{args}(A_0, ..., A_1', A_2', ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash b' : B \). Because the code transformation changes instructions depending on the content of \( \Pi \). In this case the only instructions changed are those related to arguments that should be shared.

We know that \( \Pi = \{ (\text{arg}, j) \rightarrow (\text{arg}, i) \} \) and \( A_i = A_j \); the instruction \( \text{ldarga} j \) is transformed by (Ren \( \text{ldarga} \) \( \text{arg} \)) into \( \text{ldarga} i \). By (Body \( \text{ldarga} \)) \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash \text{ldarga} j : A_i \) and, because \( A_i = A_j \) we can conclude that \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash \text{ldarga} i : A_i \). Thus the typing judgment of \( b \) is left unchanged by applying the code transformation.

The same is applied to \( \text{starga} \); by (Body \( \text{starga} \)) we know that \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash \text{starga} j : \text{void} \) under the hypothesis that \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash a : A_i \); because \( A_i = A_j \) we derive that \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash a : A_i \) and we can apply (Body \( \text{starga} \)) to derive \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash \text{starga} i : \text{void} \) which is the instruction changed by (Ren \( \text{starga} \) \( \text{arg} \)). Again the typing judgment is left unchanged by the application of code transformation.

We can conclude that if the code value \( c \) is obtained by applying \( \text{Bind} \) to the result of \( \text{Share} \) then \( \text{args}(A_0, ..., A_n) . \text{locals}(A_0', ..., A_k') \vdash b : B \) where \( b \) is the body of the code value.

\((\text{Bind}_c)\)

Definition of \( \text{Bind} \) for values is the following:

\[
c = \text{Bind}(\Sigma, c', v_0, ..., v_n) = \text{Bind}(\text{Bind}_c(i, \Sigma, v_i), v_0, ..., v_{i+1}, v_{i+1}, ..., v_n)
\]

where \( i \in 0..n, v_i \in \text{Free}, \forall j \in 0..n, v_i \in \text{Free} \Rightarrow i \leq j \)
\[ c' = \langle B \, \mu(A_0, \ldots, A_0), . \text{locals}(A_0', \ldots, A_k') b', . \text{env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle, \]
\[ \Sigma \vdash v : A_i \]

We recall the definition of \textit{Bind}:

\[ \text{Bind}(c', i, \Sigma, v) = \]
\[ \langle B \, \mu(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0), . \text{locals}(A_0', \ldots, A_i') b, . \text{env}((e_0, v_0), \ldots, (e_m, v_m), (k + 1, v)) \rangle \]
\[ \text{where } i \in 0..n, \]
\[ c' = \langle B \, \mu(A_0, \ldots, A_0), . \text{locals}(A_0', \ldots, A_i') b', . \text{env}((e_0, v_0), \ldots, (e_m, v_m)) \rangle, \]
\[ \Sigma \vdash v : A_i \]
\[ \text{pointerFree}(A_i), \]
\[ B \, \mu(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0) \vdash b : B. \]

By induction hypothesis \[ \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i') \vdash b : B. \]

We rely on structural induction as we’ve done in the previous case for \textit{Share}. We note that arguments are shifted in order to fix references to arguments following \( i \).

We consider the transformation \( \langle \text{arg}, i \rangle \rightarrow (\text{loc}, k + 1) \). We know that because (Body \( \text{ldarga} \)) \[ \text{args}(A_0, \ldots, A_0). \text{locals}(A_0', \ldots, A_i') \vdash \text{ldarga} i : A_i; \] the new frame is \[ \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \] so that \( A_{i+1} = A_i \). By (Ren \( \text{ldarga} \, \text{loc} \)) we know that the instruction is replaced by \( \text{ldloca} k+1 \). By (Body \( \text{ldloca} \)) we conclude that \[ \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \vdash \text{ldloca} k+1 : A_i. \]

Thus the transformation preserves the typing judgment. By (Body \( \text{starga} \)) we know that \[ \text{args}(A_0, \ldots, A_0). \text{locals}(A_0', \ldots, A_i') \vdash a \, \text{starga} i : \text{void} \] under the hypothesis that \[ \text{args}(A_0, \ldots, A_0). \text{locals}(A_0', \ldots, A_i') \vdash a : A_i. \]

By induction hypothesis \[ . \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \vdash a : A_i \] and \( A_{i+1} = A_i \), thus we can conclude by (Body \( \text{stloc} \)) and (Ren \( \text{starg} \, \text{loc} \)) that \[ . \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \vdash a \, \text{stloc} k+1 : A_i. \]

Again the code transformation preserves the typing judgment.

We consider the transformation \( \langle \text{arg}, i+1 \rangle \rightarrow (\text{arg}, i \rangle \), the other cases are treated in the same way. By (Body \( \text{ldarga} \)) \[ \text{args}(A_0, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \vdash \text{ldarga} i+1 : A_{i+1}; \] In the new stack frame \[ \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \] the \( i \)th argument has type \( A_{i+1} \). By (Ren \( \text{ldarga} \, \text{arg} \)) \[ \text{ldarga} i+1 \] is changed into \( \text{ldarga} i \). We derive that \[ . \text{args}(A_0, \ldots, A_{i-1}, A_{i+1}, \ldots, A_0). \text{locals}(A_0', \ldots, A_i', A_i) \vdash \text{ldarga} i : A_{i+1}. \]

Thus the typing judgment is
preserved. By (Body starг) \( \text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash a : A_i \)
implies \( \text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash \text{starg } i+1 : \text{void} \). By induction hypothesis \( \text{args}(A_0, \ldots, A_i, A_{i+1}, \ldots, A_n).\text{locals}(A_0', \ldots, A_k', A_i') \vdash a : A_{i+1} \), by (Ren starг arg) and (Body starг) \( \text{args}(A_0, \ldots, A_i, A_{i+1}, \ldots, A_n)\) .\text{locals}(A_0', \ldots, A_k', A_i') \vdash \text{starg } i : \text{void} \). Again the typing judgment is preserved.

\((\text{Bind}_x)\)

Code prefix in \( \text{Bind} \) is expressed as follows:

\[
c = \text{Bind}(\sum_{i} c', v_0, \ldots, v_n) = \text{Bind}(\text{Bind}(c', i, v_i), v_0, \ldots, v_{i+1}, \phi, \ldots, \phi_i, v_{i+1}, \ldots, v_n)
\]
where \( i \in 0..n, v \in \text{Code}, \forall j \in 0..i, v \in \text{Free} \),

\[
c' = B \mu(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') b', \text{env}(e_0, \ldots, (e_m, u_m)) >,\]

\[
v_i = A_i \mu(A_0'', \ldots, A_n''), \text{locals}(A_0''', \ldots, A_k'''', A_i') b''', \text{stloc } j b''''',\]

\[
\text{env}(e_0, \ldots, (e_m, u_m)) >,\]
\( \phi \in \text{Free}, \forall j \in 0..n', \phi = \phi \Rightarrow h = l, \forall h \in 1..n', l \in 0..n. \phi \neq v_i \)

We recall the definition of \( \text{Bind}_x \):

\[
\text{Bind}(c', i, c') =
B \mu(A_0, \ldots, A_{i+1}, A_i', \ldots, A_n', A_{i+1}, \ldots, A_n),
\text{locals}(A_0', \ldots, A_k', A_i') b' \text{stloc } j b''',\]
\[
\text{env}(e_0, \ldots, (e_m, u_m), (e_0' + k + 1, v_0'), \ldots, (e_m' + k + 1, v_m')) >,
\]
where \( i \in 0..n \),

\[
c' = B \mu(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') b', \text{env}(e_0, \ldots, (e_m, u_m)) >,\]

\[
c''' = A_i \mu(A_0'', \ldots, A_n''), \text{locals}(A_0''', \ldots, A_k'''', A_i') b''',\]
\[
\text{env}(e_0, \ldots, (e_m, u_m)) >,\]

\[
j = k + k' + 2,
\]

\[
\Pi = \{ (\text{arg}, 0) \rightarrow (\text{arg}, i), \ldots, (\text{arg}, n') \rightarrow (\text{arg}, i + n'),\]
\[
(\text{loc}, 0) \rightarrow (\text{loc}, k + 1), \ldots, (\text{loc}, k') \rightarrow (\text{loc}, k + k' + 1) \},
\]

\[
\Pi' = \{ (\text{arg}, i + 1) \rightarrow (\text{arg}, i + n' + 1), \ldots, (\text{arg}, n) \rightarrow (\text{arg}, n + n' - 1),\]
\[
(\text{arg}, i) \rightarrow (\text{loc}, j) \},
\]

\[
\Pi \vdash b'' \rightarrow b''',\]

\[
\Pi' \vdash b' \rightarrow b'''''
\]

By induction hypothesis \( \text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash b' : B \) and \( \text{args}(A_1', \ldots, A_n'), \text{locals}(A_0'', \ldots, A_k'''', A_i') \vdash b'' : A_i \). We should prove that \( \text{args}(A_0, \ldots, A_{i+1}, A_i', \ldots, A_n', A_{i+1}, \ldots, A_n).\text{locals}(A_0', \ldots, A_i', A_0'', \ldots, A_k'''', A_i') \vdash b'' \text{stloc } j b'''' : B \). First of all we focus on the proof of \( \text{args}(A_0, \ldots, A_{i+1}, A_i', \ldots, A_n', A_{i+1}, \ldots, A_n).\text{locals}(A_0', \ldots, A_i', A_0'', \ldots, A_k'''', A_i') \vdash b'' : A_i \).
We observe that $b'''$ is the result of applying $\Pi$ to $b''$. Arguments are shifted by $i$ and locals by $k+1$. The schema for the proof is the same of the previous cases and it is based on the observation that $A_o'''$ is located in $i$th position in the new stack frame, and the same happens for subsequent arguments. The same applies to local arguments that are appended to locals starting from index $k+1$: the $(k+1)$th local’s type is $A_o'''$ and so on. In the case of locals we shall use (Body ldloc), (Ren ldloc), (Body stloc) and (Ren stloc) instead of their argument’s counterpart.

By (Body stloc) we can conclude that $\text{.args}(A_0, ..., A_{i-1}, A_i', ..., A_n', A_{i+1}, ..., A_n).\text{locals}(A_o', ..., A_k', A_o''', ..., A_k'''', A_i) \vdash b''' \text{ stloc } j: \text{void}$. Now we should prove that $\text{.args}(A_0, ..., A_{i-1}, A_i', ..., A_n', A_{i+1}, ..., A_n).\text{locals}(A_o', ..., A_k', A_o''', ..., A_k'''', A_i) \vdash b''': B$.

Again $b''''$ is obtained by applying $\Pi'$ to $b'$, so we can rely on the induction hypothesis. We notice that arguments from $i+1$ to $n$ are shifted by $n'$ and again we can follow the proof schema of the $\text{Bind}_v$ case. The $i$th argument is transformed by $(\text{arg}, i) \rightarrow (\text{loc}, j)$; because $A_i = A_i$ by definition we can proceed as we did in $\text{Bind}_v$ proving that $\text{.args}(A_0, ..., A_{i-1}, A_i', ..., A_n', A_{i+1}, ..., A_n).\text{locals}(A_o', ..., A_k', A_o''', ..., A_k'''', A_i) \vdash b''': B$.

Now we can apply (Body Seq) and conclude $\text{.args}(A_0, ..., A_{i-1}, A_i', ..., A_n', A_{i+1}, ..., A_n).\text{locals}(A_o', ..., A_k', A_o''', ..., A_k'''', A_i) \vdash b'''' \text{ stloc } j \ b''''' : B$.

($\text{Bind}_X$)

The last case of $\text{Bind}$ definition is the following:

$c = \text{Bind}(\Sigma, c', v_0, ..., v_n) = \text{Bind}(\text{Bind}_X(c', i, v), v_0, ..., v_{i-1}, v_{i+1}, ..., v_n)$

where $i \in 0..n$, $v \in \text{Code}$, $\forall j \in 0..i$, $v \in \text{Free}$,

c' = < B \mu(A_0, ..., A_n), \text{.locals}(A_o', ..., A_k') b', \text{.env}((v_0, uv_0), ..., (v_n, uv_n)) >,

vi = < B \mu(A_o''', ..., A_n'''), \text{.locals}(A_o''', ..., A_k''') b'', \text{.env}((v_0', uv_0'), ..., (v_n', uv_n')) >,

Ai = \text{class cl},

B \text{ Invoke}(A_o''', ..., A_n''' \in \text{dom}(\text{methods}(cl))),

B \text{ Invoke}(A_o''', ..., A_n''' \in \text{StaticMethods})

We recall the definition of $\text{Bind}_X$:

$\text{Bind}_X(c', i, c'') =$

< B \mu(A_0, ..., A_{i-1}, A_{i+1}, ..., A_n),

\text{.locals}(A_o', ..., A_k', A_o''', ..., A_k''', A_i''', ..., A_n'''') b'''',

\text{.env}((v_0, uv_0), ..., (v_n, uv_n), (v_0' + k + 1, v_0'), ..., (v_n' + k + 1, v_n')) >
where \( i \in 0..n, \)
\[
c' = \ll A(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') c' \rr, \]
\[
c'' = \ll B \mu(A_i''', \ldots, A_n''') \rr \ll A_0'', \ldots, A_k''\rr b''', \]
\[
\text{env}((e_0', v_0'), \ldots, (e_m', v_m')) >, \]
\[
A_i = \text{class } cl, \]
\[
B' \text{ Invoke}(A_0, A_i'', \ldots, A_n'') \in \text{dom}(\text{methods}(cl)), \]
\[
B' \text{ Invoke}(A_0, A_i'', \ldots, A_n'') \notin \text{StaticMethods}, \]
\[
\prod = \{ (\text{loc}, 0) \rightarrow (\text{loc}, n + 1), \ldots, (\text{loc}, k') \rightarrow (\text{loc}, k + k' + 1), \}
\]
\[
\prod' = \{ (\text{arg}, 0) \rightarrow (\text{inline}, k + k', b''') \}, \]
\[
\prod + b'' \rightarrow b''', \]
\[
\prod' + b' \rightarrow b''' \]

By induction hypothesis \( .\text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash b' : B \) and \( .\text{args}(A_i'', \ldots, A_n'') .\text{locals}(A_0'', \ldots, A_k'') \vdash b'' : B' \). We split the proof in two steps. First of all we prove that \( .\text{args}(A_0, \ldots, A_i, A_i, \ldots, A_n).\text{locals}(A_0', \ldots, A_i, A_i', A_i'', \ldots, A_k'') \vdash b''' : B' \) under the hypothesis that \( b'''' \) is obtained by applying \( \prod \) to \( b'' \). The second step is aimed to show that \( .\text{args}(A_0, \ldots, A_i, A_i, \ldots, A_n).\text{locals}(A_0', \ldots, A_i, A_i', A_i'', \ldots, A_k'') \vdash b'''' : B, \) under the hypothesis that \( b'''' \) is obtained through \( \prod' \) from \( b' \).

The first step can be proven like previous cases because the code transformation simply renames locals and maps the arguments into locals. It is enough to note that the types are added to the locals array in the frame in the same order: the \( i \)th local of \( b'' \) is added in position \( k+i+1 \). After the locals comes the arguments that are moved from arguments to locals. The proof is similar to those used for \text{Bind}_e \) and \text{Share}.

The code transformation on \( b' \) affects only a sequence of instructions of the body as stated by (Ren Invoke): \( \prod \vdash \text{laddr } i \ a_i \ldots \ a_n \ \text{callvirt instance } c::\text{Invoke} \rightarrow a_i \text{ stloc } k+k'+1 ... \ a' \text{ stloc } k+k'+n' \ b''' \) where \( a' \) are obtained as \( \prod \vdash a_i \rightarrow a_i'. \) By (Ref callvirt) and the induction hypothesis\(^{36} \) we know that \( .\text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash a_i : A_i'. \) By induction hypothesis \( .\text{args}(A_0, \ldots, A_i, A_i, \ldots, A_n).\text{locals}(A_0', \ldots, A_k', A_i', A_i'', \ldots, A_k'') \vdash a_i' : A_i'. \) Moreover the \( .\text{args}(A_0, \ldots, A_n).\text{locals}(A_0', \ldots, A_k') \vdash \text{laddr } i \ a_i \ldots \ a_n \ \text{callvirt instance } B' \ c::\text{Invoke}(A_i, A_i'', \ldots, A_n'') : B'. \)

By applying (Body stloc) and (Body Seq) we can conclude that \( .\text{args}(A_0, \ldots, A_i, A_i, \ldots, A_n).\text{locals}(A_0', \ldots, A_i', A_i'', \ldots, A_k'') \vdash a_i' \text{ stloc} \). 

---

\(^{36}\) We refer to the innermost induction on the typing judgment.
Another application of (Body Seq) let us to conclude that 
\[ \text{.args}(A_0, ..., A_{i-1}, A_i, ..., A_n) \text{.locals}(A_0', ..., A_i', A_0'', ..., A_i'', A_0''', ..., A_n''') \vdash \text{stloc } k+k'+1 ... a_n' \text{ stloc } k+k'+n' : \text{void}. \]
Thus the typing judgment is preserved by code transformation.

In all cases of \textit{Bind} definition we conclude that the code transformation preserves the typing judgment of the constituents concluding the proof of the theorem. □

### 8.5 The Reverse Polish Notation Compiler

```csharp
using System;
using CodeAPI;
using System.Collections;
using System.Reflection;

namespace Reverse {
    class RPNCalc {
        private Hashtable environment;
        private Hashtable free;

        public RPNCalc() {
            environment = new Hashtable();
            free = null;
        }

        public string eval(string input) {
            int i = 0;
            input = input.Trim();
            if (input.IndexOf("letf ") == 0) {
                free = new Hashtable();
                i = 5;
                while (i < input.Length && input[i] == ' ') i++;
                if (i == input.Length)
                    return "Invalid function statement!";
                input = input.Substring(i);
                i = 0;
                while (i < input.Length && input[i] != ' ') i++;
                if (i == input.Length)
                    return "Invalid function statement!";
                string varName = input.Substring(0, i);
                while (i < input.Length && input[i] != ' ') i++;
                if (i == input.Length)
                    return "Invalid function statement!";
                input = input.Substring(i);
                environment[varName] = parse(ref input, false);
                Console.WriteLine((Code)environment[varName]);
            } else if (input.IndexOf("let ") == 0) {
                // Further code for handling 'let' statements...
            } else { // Handling other tokens...
                // Further code for handling other tokens...
            }
        }
    }
}
```
i = 4;
while (i < input.Length && input[i] == ' ') i++;
if (i == input.Length)
    return "Invalid function statement!";
input = input.Substring(i);
i = 0;
while (i < input.Length && input[i] != ' ') i++;
if (i == input.Length)
    return "Invalid function statement!";
string varName = input.Substring(0, i);
while (i < input.Length && input[i] == ' ') i++;
if (i == input.Length)
    return "Invalid function statement!";
input = input.Substring(i);
environment[varName] = parse(ref input, true);
return environment[varName].ToString();
}
else {
    Console.WriteLine(parse(ref input, true)); return "";
}

return "";

Code parse(ref string input, bool resolve) {
Code l = null, r = null;
string mn = null;
input = input.Trim();
switch (input[input.Length - 1]) {
case '+': mn = "add"; goto build;
case '-': mn = "sub"; goto build;
case '*': mn = "mul"; goto build;
case '/': mn = "div";
    build:
        input = input.Substring(0, input.Length - 1);
        r = parse(ref input, resolve);
        l = parse(ref input, resolve);
        Type args = l.ReturnType;
        if (l.ReturnType != r.ReturnType)
            if (args == typeof(int))
                l = c_conv.Bind(l);
            else {
                r = c_conv.Bind(r);
                args = typeof(double);
            }
        mn += (args == typeof(double) ? "_d" : "_i");
    Code c = ops[mn] as Code;
    return c.Bind(l, r);
}

int i = input.Length - 1;
while (i >= 0) {
    if ("+*/- ".IndexOf(input[i]) != -1) {
        i++;
        break;
    }
    i--;
}

string term = (i == -1) ?
    input : input.Substring(i);
input = (i == -1) ? "" : input.Substring(0, i);
if (!char.IsDigit(term[0])) {
    if (resolve && !environment.ContainsKey(term))
        throw new ArgumentException("Unknown variable");
    else {
        if (!resolve){ // Assume it as a parameter
            if (!free.ContainsKey(term))
                free[term] = new Free(typeof(double));
            return
                ((Free)free[term]).Type == typeof(double) ?
                c_id_d.Bind(free[term]) :
                c_id_i.Bind(free[term]);
        } else {
            Object val = environment[term];
            // Function
            if (val.GetType() == typeof(Code)) {
                Code fun = (Code) val;
                object[] pars = new object[fun.Arity];
                for (int j = 0; j < fun.Arity; j++)
                    pars[j] = parse(ref input, resolve);
                return fun.Bind(pars);
            } else { // Value
                return val is double ?
                    c_id_d.Bind(val) : c_id_i.Bind(val);
            }
        }
    }
}

} else {
    try {
        return c_id_i.Bind(int.Parse(term));
    } catch(Exception) {
        return c_id_d.Bind(double.Parse(term));
    }
}

// Code values
public static double add_d(double i, double j){
    return i+j;
public static int add_i(int i, int j) { return i+j; }  
public static double mul_d(double i, double j) { 
    return i*j; }  
public static int mul_i(int i, int j) { return i*j; }  
public static double sub_d(double i, double j) { 
    return i-j; }  
public static int sub_i(int i, int j) { return i-j; }  
public static double div_d(double i, double j) { 
    return i/j; }  
public static int div_i(int i, int j) { return i / j; }  
public static double conv(int i) { return i; }  
public static double id_d(double d) { return d;  }  
public static int id_i(int i) { return i; }  
public static Code c_add_d = 
    new Code(typeof(RPNCalc).GetMethod("add_d"));  
public static Code c_add_i = 
    new Code(typeof(RPNCalc).GetMethod("add_i"));  
public static Code c_mul_d = 
    new Code(typeof(RPNCalc).GetMethod("mul_d"));  
public static Code c_mul_i = 
    new Code(typeof(RPNCalc).GetMethod("mul_i"));  
public static Code c_sub_d = 
    new Code(typeof(RPNCalc).GetMethod("sub_d"));  
public static Code c_sub_i = 
    new Code(typeof(RPNCalc).GetMethod("sub_i"));  
public static Code c_div_d = 
    new Code(typeof(RPNCalc).GetMethod("div_d"));  
public static Code c_div_i = 
    new Code(typeof(RPNCalc).GetMethod("div_i"));  
public static Code c_id_d = 
    new Code(typeof(RPNCalc).GetMethod("id_d"));  
public static Code c_id_i = 
    new Code(typeof(RPNCalc).GetMethod("id_i"));  
public static Code c_conv = 
    new Code(typeof(RPNCalc).GetMethod("conv"));  
public static Hashtable ops;  
static RPNCalc() {  
    ops = new Hashtable();  
    ops["add_d"] = c_add_d;  
    ops["add_i"] = c_add_i;  
    ops["mul_d"] = c_mul_d;  
    ops["mul_i"] = c_mul_i;  
    ops["sub_d"] = c_sub_d;  
    ops["sub_i"] = c_sub_i;  
    ops["div_d"] = c_div_d;  
    ops["div_i"] = c_div_i;  
    ops["id_d"] = c_id_d;  
    ops["id_i"] = c_id_i;  
    ops["conv"] = c_conv;  
}
ops["div_i"] = c_div_i;
}

public static void Main(string[] args) {
    RPNCalc calc = new RPNCalc();
    while (true) {
        Console.Write("> ");
        string line = Console.ReadLine();
        Console.WriteLine(calc.eval(line));
    }
}
Bibliography


