Design and Implementation of a Multi-stage, Object-oriented Programming Language

GREGORY MICHAEL NEVEROV
B. App. Sci. (Math), B. I.T.

A thesis submitted for the degree of
Doctor of Philosophy
in Computer Science
2007

School of Software Engineering and Data Communications
Queensland University of Technology
Brisbane, Australia
Keywords

Programming languages; Multi-stage programming; Object-oriented programming; Meta programming; Run-time code generation; Polytypic programming; Type systems

Abstract

Multi-stage programming is a valuable technique for improving the performance of computer programs through run-time optimization. Current implementations of multi-stage programming do not support run-time type introspection, which is a significant feature of modern object-oriented platforms such as Java and C#. This is unfortunate because many programs that use type introspection in these languages could be improved with multi-staging programming.

The aim of this research is to investigate the interaction between multi-stage programming and object-oriented type introspection. This is done by the invention of a new programming language that is a multi-stage extension to C#. The language is capable of expressing traditional multi-stage programs as well as a new style of multi-stage programs that incorporate type introspection, most notably polytypic algorithms such as object serialization. A compiler for the language is implemented and freely available. The language is significant because it is the first object-oriented, multi-stage language; the first attempt to combine type introspection with multi-stage programming; and the first exploration of polytypic programming in a multi-stage context.
# Contents

1 Introduction ......................................................... 1  
  1.1 Contribution .................................................. 3  
  1.2 Outline ......................................................... 4  

2 Related work ....................................................... 6  
  2.1 Background .................................................... 6  
    2.1.1 Syntax-based meta programming .......................... 9  
    2.1.2 Semantics-based meta programming ..................... 10  
    2.1.3 Summary .................................................. 13  
  2.2 Literature review ............................................... 13  
    2.2.1 Compile-time meta programming ......................... 13  
    2.2.2 Partial evaluation ...................................... 14  
    2.2.3 Multi-stage languages .................................. 16  
    2.2.4 LISP ..................................................... 18  
    2.2.5 Object-oriented meta-programming languages ........... 19  
    2.2.6 Higher-level imperative code generation ................ 21  
    2.2.7 Generic programming .................................... 22  
    2.2.8 Object-oriented reflection ............................... 23  
    2.2.9 Run-time code generation APIs .......................... 24  

3 Imperative multi-stage programming ................................ 27  
  3.1 Motivation .................................................... 28  
  3.2 Language overview ............................................ 29  
    3.2.1 Staging constructs ....................................... 29  
    3.2.2 Staging example .......................................... 31  
    3.2.3 Multi-stage type system ................................ 35  
    3.2.4 Automatic variable renaming ............................. 36  
    3.2.5 Programming methodology ............................... 36  
    3.2.6 Testing and debugging ................................. 39
3.3 Issues

3.3.1 Scope extrusion

3.3.2 Syntactic issues

3.3.3 Assignment

3.3.4 Cross-stage persistence semantics

3.3.5 Code persistence

3.3.6 Staged array access

3.3.7 Memory management

3.4 Results

3.4.1 Example applications

3.4.2 Performance data

3.5 Conclusion

4 Polytypic multi-stage programming

4.1 Motivation

4.2 Introduction

4.2.1 The Reflection API

4.2.2 Staging reflection

4.2.3 Type checking reflection

4.2.4 Plan

4.3 Polytypic algorithms

4.3.1 Basic serialization

4.3.2 Recursive serialization

4.3.3 Recursive function generation

4.3.4 Polytypic method signatures

4.3.5 Code type co-variance

4.4 Dynamically-typed approach

4.4.1 Basic serialization

4.4.2 Recursive serialization

4.4.3 Revisiting the parametric signature

4.4.4 Conclusion

4.5 Statically-typed approach

4.5.1 Parametric reflection types
Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature: __________________________

Date: ____________________________
Acknowledgments

A Ph.D. is more than a thesis, a diploma and a lesson in the craft of research. For me it has been a challenging and significant period of my life and I wish to thank those who have helped me along the way.

Paul Roe, my supervisor, for his reliable advice, practical encouragement and well-tested patience. Wayne Kelly, my associate supervisor, for his valued second opinion at critical times.

The students of the lab, Richard Mason, Dominic Cooney, Douglas Stockwell, Jiro Sumitomo, Alexander Streit, Asbjørn Rygg, Simon Kent, Aaron Searle, Darryl Cain, Jens Tröger and Joel Pobar, with whom I have shared so many enlightening, fascinating, and humorous discussions.

Don Syme for the wonderful opportunity to do an internship at Microsoft Research, Cambridge in the summer of 2006.

Ravi Jain for hosting my internship at Google, Mountain View in 2007.

Simeon Cran for inviting me to visit Microsoft, Redmond in 2004.

Jorge Cham for writing Piled Higher and Deeper.

The Australian government for supporting this work through an Australian Postgraduate Award.

My parents and brother and sisters for everything else.
1 Introduction

The three most important things in computer science are abstraction, abstraction, abstraction.— *Paul Hudak*

Computer programs are written to perform particular tasks: the larger the program the wider the set of tasks it can perform. Programmers write programs by mapping the tasks they need to perform into the abstraction mechanisms of a programming language. For example a program that solves systems of linear equations could, if written in an object-oriented language, use classes to represent matrices and vectors and use a method to represent the action of solving the equation.

Programmers build larger or more general programs by using more layers of abstraction. Abstraction is good because it facilitates software reuse and structures a program so that programmers can more readily comprehend and modify it. The downside of increased abstraction is that it often impairs program performance. Consider a software library constructed to perform a very wide range of tasks and an application programmer who wants to use the library to perform only a very specific task. Most of the library’s functionality is unneeded and yet the part that is used still carries the burden of extra abstraction (more function calls, more indirect access to data, more memory usage, etc.)

What is needed is a programming environment or tool that encourages programmers to be abstract (as abstraction leads to improved reuse and maintainability) but eliminates the performance cost typically associated with abstraction. To this end various ideas have successfully addressed the problem, such as macros, partial evaluation and multi-stage languages. In particular multi-stage languages, the most recent attempt at the problem, give a higher degree of programmer control and type safety than previous attempts.

MetaOCaml is a multi-stage programming extension to the functional language OCaml. MetaOCaml is statically typed and allows the programmer to control the order of evaluation within a program. This means that when a general program is being used for a specific task, the parts of the program not engaged in the task can
be evaluated ahead of time to leave a program specialized for doing just the task at hand.

Object-oriented languages such as C# and Java use classes as the primary means of abstraction. Whilst statically typed on one level they also allow dynamically typed programming through run-time type information and type introspection (also known as the reflection API). These features are an integral part of the languages and are specifically supported by their underlying virtual machines. Despite this support, type introspection is usually a slow operation and often a performance bottleneck for programs that use it.

Type introspection allows programmers to write very abstract programs that operate on any type of object by analysing its structure at run-time. Such programs are called polytypic programs and examples include serialization, equality comparison and deep copying. Polytypic programs in object-oriented languages utilize type introspection so they typically run slowly because of its poor performance.

Type introspection is a fundamental part of the abstraction capability of the object-oriented languages C# and Java. Therefore any attempt at comprehensively eliminating the performance cost of all superfluous abstraction in these languages must address their ability for type introspection. Existing theory and practice in multi-stage programming provides no mechanism for incorporating run-time type information into the staging of a program. If multi-stage programming could interact with type introspection then it would be a means of efficient polytypic programming and a new application for multi-stage programming.

The aim of this research is to investigate the interaction between multi-stage programming and object-oriented type introspection to enable the creation of staged, polytypic programs.
1.1 Contribution

The contribution of this thesis is the design and implementation of a multi-stage, object-oriented programming language.

This contribution is significant for a number of reasons. Firstly, all multi-stage languages to date have been functional languages. This work is the first example of multi-stage language theory applied to a general-purpose, imperative or object-oriented environment. Secondly, the high degree of static typing offered by multi-stage languages is an advancement on the type safety of meta-programming languages in the object-oriented world. Thirdly, this work goes beyond current multi-stage programming techniques as it incorporates type introspection into the staging process.

In plain, concrete terms: traditional multi-stage programming is extended to cover the reflection API of C# or Java. This means the use of reflection in a program can be partially-evaluated away. A programmer can write code that uses reflection and at run-time code is generated to perform the program’s functionality but without the use of reflection.

This work has resulted in the creation of the programming language Metaphor: a multi-stage programming extension to C#. Metaphor is implemented as a compiler targeting Microsoft’s Common Language Runtime platform. The compiler is freely available for download.

Notable points about Metaphor:

- Metaphor transposes multi-stage programming concepts to an imperative, object-oriented language.
- Metaphor is the first multi-stage object-oriented or imperative language.
- Metaphor is the first object-oriented, meta-programming language to give a high degree of static typing.
The object-oriented virtual machine setting is significant because of the central role type reflection plays in the virtual machine and the way this language addresses interaction with it.

Metaphor also supports cross-stage persistence across process boundaries, which is useful for saving generated code and mobile applications.

This work has resulted in three publications [33–35]. The Metaphor compiler and sample programs are available for download at [http://www.fit.qut.edu.au/~neverov/](http://www.fit.qut.edu.au/~neverov/).

### 1.2 Outline

Section 2 provides a background on meta-programming languages, including multi-stage languages which are a distinct type of meta-programming language. It describes related work on functional multi-stage programming languages as well as meta-programming and run-time code generation systems for imperative and object-oriented languages.

Section 3 introduces the Metaphor language—a multi-stage programming extension to C#. It highlights a number of areas where the application of multi-stage language theory from the functional realm was not straightforward because of characteristic differences in the imperative, object-oriented base language, C#. The successful creation of the Metaphor language and its implementation enables programmers to employ existing techniques from multi-stage programming theory in the development of object-oriented programs and further establishes multi-stage programming as a practical and widely-applicable programming technique.

Finally, Section 4 extends the basic notion of multi-stage programming to encompass type introspection, thus enabling the creation of efficient polytypic programs. It compares three approaches for integrating C#’s reflection API with multi-stage programming, each approach offering a varying degree of static type safety. Across the approaches, increasing static type safety comes with the cost of increased language complexity and so a suitable compromise is found. This extension allows program-
urers to use a familiar Reflection API from C#/Java to produce staged, polytypic programs.
2 Related work

2.1 Background

A *meta program* is a very broad term that can be applied to any program that manipulates program code in some way. This task is quite generic and vague and consequently a wide variety of approaches have been used to implement it. Sheard [9] gives a good taxonomy of meta-programming systems. One identifying characteristic of a meta program is whether the code manipulation occurs at compile-time or run-time.

The program that a meta program manipulates is sometimes referred to as an *object program*. Because a meta program manipulates an object program there are two programming languages involved: the language of the meta program and the language of the object program. In some cases these two languages are the same. A meta-programming system can be seen as, and is often motivated as, an extension to a base language. The base language becomes the object language in the meta-programming system and the extension itself becomes the meta language.

There are two broad reasons why programmers need meta programming:

1. to express in the meta program an algorithm that could not be otherwise expressed in the object language, and

2. to translate a meta program into an object program to improve its efficiency.

Examples of 1

In C++ (without templates) it is not possible to write a class for a linked list of any possible element type. An immediate solution to the problem could be to write out a separate linked list class for every element type of interest. This is a very tedious and mechanical process, but it can be automated using C++ templates. A paramet-
ric linked list class can be specified in the template language and when the template
(meta program) executes it will generate a C++ class for a linked list of the parameter-
ized type. Hence templates (the meta language) are able to express parametric classes
which could not be expressed in C++ (the object language) alone.

Many applications, especially web applications, need to interface with a database.
Most programming languages do not have database access as part of their funda-
mental semantics so code needs to be written in the language to access the database.
Since databases are self-describing (i.e., contain metadata) it is theoretically possible
to write a single program that can access any data from a database. In practice how-
ever this can be difficult to realize especially if the language has a static type system
like C# or Java as there is no way to statically express the dynamic type information
gathered from the database. This problem arises in many mainstream languages and
is commonly solved by an external code generation tool. This tool reads the database
metadata and generates class definitions or other source code for manipulating the
database from a chosen language.

Haskell has a number of standard type classes (Show, Eq, Ord, etc.) that a data type
can implement to gain some basic functionality in the language, e.g., showing as a
string or comparing for equality. It is tedious for a programmer to implement these
type classes when in many cases a generic implementation induced by the structure
of the type is sufficient. However writing such a generic definition is not possible in
some versions of Haskell (such as in Haskell 98). The first attempt to solve this prob-
lem was the introduction of the deriving clause. The deriving clause tells the compil-
er which type classes (from a fixed set of basic ones) need to be implemented and the
compiler automatically generates the code for them. In this way the deriving clause
acts as a meta program that brings about the generation of regular Haskell code.

The problem with the deriving approach is that it only works with a small number
of built-in type classes and is not user extensible. More sophisticated meta-program-
ing systems have since been created for Haskell such as generics and templates, and
these are discussed in detail in Section 2.2.7.
Examples of Pan \([13]\) and Vertigo \([12]\) are systems for creating computationally-derived images. These systems use a mathematical description to generate 2D images and 3D images respectively. The programmer expresses the mathematical description using Haskell. Although the programmer appears to be writing in Haskell, the Haskell program actually serves as a meta program that generates a judiciously selected subset of C. This object language can be highly optimized by constant folding, function in-lining and common sub-expression elimination to a degree that could not be achieved in an ordinary C program. The object program also runs with much greater efficiency than the Haskell program. Hence the meta-programming process has improved the efficiency of the meta language, if only in a special case.

Multi-stage languages (see Section 2.2.3) are a meta-programming system where the meta and object languages are the same. A multi-stage program acts as both meta and object program. The meta program generates specialized code for an object program once some critical information has become available at run-time. Such optimization could not be made at compile-time, hence the generation of the object program dynamically improves the performance of a single-stage program.

Numerical analysis is a field where code specialization is practically important. Mathematically many numerical algorithms can be specified at a very high level of abstraction. The algorithms can be similarly implemented at a very high level but this abstraction comes with a performance penalty, where performance is typically a paramount quality of the software. Meta programming systems \([8, 9]\) can be used to generate code from the abstract algorithm that is specialized for the concrete context in which it is used, e.g., to specialize the general LU decomposition algorithm for tridiagonal matrices.

Another computationally intensive application area which can benefit from meta programming is ray-tracing. A typical ray-tracer performs these steps: read a scene description file, build a data structure to representing the scene, and analyse this data structure in the computation of each pixel. The performance of the ray-tracer may be improved if after reading the scene description file it generates specialized code to
render that particular scene, without reference to a controlling data structure. This will eliminate at least any run-time spent traversing the data structure but will also enable optimizations in the pixel computation as complete information about the object to render is available.

2.1.1 Syntax-based meta programming

Thus two distinct use cases of meta programming have been identified and they may be labelled as syntax-based and semantics-based code generation. The former is syntax-based because it concerns mass producing source code that cannot be written abstractly. The latter is semantics-based because it involves altering how a program executes in order to optimize its performance. As a rough guide, code generation occurs at compile-time in syntax-based systems and run-time in semantics-based system. There are exceptions to this rule though: e.g., parser generators are typically run at compile-time but they are an example of semantics-based meta programming because the parser program could be written without meta programming and code generation exists only to improve the performance of the parser and to generate essential new source code.

In the case of syntax-based meta programming, the programmer wants to write a program that automates the very task of programming. In software development this need arises when the writing of code becomes repetitive and mechanical and hence amenable to automation. This situation could be viewed as a failure of the inherent abstraction mechanism of the underlying programming language. For example, say that a programming language only supports single-parameter functions and the repetitive code the programmer wants to create is a series of functions to increment an integer by one, two, three, etc. Here the abstraction capabilities afforded by the programming language have failed to meet to the demands of the programmer. A language with greater abstraction capabilities, viz. multi-parameter functions, could replace the series of auto-generated functions with a single function that adds two integers.

This is not to say that there is something wrong with the programming languages that benefit from syntax-based meta programming. A programmer will eventually
reach the abstraction limit of any programming language no matter how expressive or powerful it is purported to be. For example, even in Haskell it is not possible to abstract over the size of a tuple.

Matters of abstraction ability aside, syntax-based meta programming is still needed when considering aspects of programming outside of the programming language itself. A number of meta-programming systems that would be classified as syntax-based here, could be more accurately called pre-compile-time program generators. These systems create a swath of source code that the programmer, at development-time, either consumes directly or modifies or extends—the objective of this scheme being to facilitate and expedite the coding process.

In syntax-based meta programming the programmer knows precisely what the source code would look like if created manually, therefore the meta program is an exercise of automatically writing this source code. Hence the mind-set of the programmer is: this is the code I would write, and this is a program that writes it top-to-bottom, as though it were being written by hand.

2.1.2 Semantics-based meta programming

Programming in a semantics-based meta-programming system appeals to a different mentality. As the code generation in semantics-based meta programming typically occurs at run-time there is no opportunity for the programmer to inspect, adjust, or integrate the code generated at run-time into other code. Moreover the code generated by semantics-based meta programming is often large and complex and unintelligible to human readers. For example, imagine the specialized code for a ray-tracer that renders a complex scene. For effective semantics-based meta programming, the programmer must be freed from the need to visualize the output of code generation as a piece of source code that could have been written by a programmer given enough time and patience for monotonous, and instead approach meta programming at a more fundamental level.

Programs are abstract entities which describe computational intent. Depending on the reader’s background in programming languages, the evaluation of programs can
be modelled either by the lambda calculus or a finite state machine. In effect, any program, no matter how elaborate the programming language it which it was written, can be theoretically reduced to either of these computational models. In both these models the evaluation of a program consists of a series of discrete and atomic operations: lambda reductions in the lambda calculus and state transitions in a finite state machine. Hence the speed or performance of program execution depends on how quickly the hardware machine used to execute the program can perform its emulation to these abstract steps.

To execute programs faster one option is to use faster hardware. This solution works in practice so long as processors are steadily growing at an exponential rate, but this approach cannot be relied on forever and processors speed growth has already begun to level out and approach the limit of current technology.

The only other option to execute programs faster is to alter how the hardware machine emulates the operation of the abstract machine. Recall that the operation of the abstract machine (be it lambda calculus or a finite state machine) is a sequence of discrete steps. The performance characteristics of the program can be altered if these steps are executed ‘not in sequence’.

There are two ways to ‘un-sequential-ize’ the evaluation of a program: to execute more than one step at the same time, or to execute steps out-of-order. The former solution is commonly known as parallel or concurrent programming. The latter is known by many names: staged computation, multi-stage programming, partial evaluation and semantics-based meta programming, etc.

The task of a semantics-based meta programmer is determining what parts of the algorithm can be pre-executed so as to simplify the remaining code. In other words,
rather than executing a program from top to bottom in an imperative language, or start state to finish state in a finite state machine, or innermost to outermost for call-by-value lambda calculus, or outermost to innermost for call-by-name lambda calculus, parts in the middle of the program are executed first, resulting in code simplification, and then the rest of the program can be executed in-order.

Executing some steps out-of-order will reduce the total number of steps needed to be executed. For example a ray-tracer will typically have a loop over pixels and then a nested loop over scene objects. If the loop over scene objects were done first (traverse the scene graph, perform algebraic simplifications based on known object data) then this reduces the number of steps performed inside the outer loop.

Code generation enters into out-of-order evaluation because a copy of the program needs to be produced after some of its parts have been executed ahead-of-time. This is because in most systems the representation of program code is immutable. If the code representation was not immutable then self-modifying code would be a possible way to realize out-of-order evaluation.

Semantics-based meta programming is the complementary opposite to parallel programming. Both concepts alter the order of execution of a program to improve its efficiency but do so in contrasting ways. Although both methods strive to improve performance they each introduce their own performance overhead, which limits the indefinite application of these techniques. These overheads always need to be considered when evaluating net performance. In semantics-based meta programming the overhead comes from having to generate code at run-time; in parallel programming it comes from creating threads of execution and managing synchronization.

Ideally it should be possible for any sequential program to have its steps automatically reordered for optimal performance, but this goal is difficult in both cases. Partial evaluation \[28, 20\] is an attempt at creating this automatic process for semantics-based meta programming and there has been limited success. A means for automatically parallelizing a program remains elusive.

The problem with manual implementation of both these approaches is the lack of programming languages that can implement them in a manner that is clear, expres-

The lack of adequate programming tools to apply these techniques means that they are not applied as pervasively as their benefit might warrant. In today’s mainstream programming environments it is very difficult to write a parallel or staged program and get it correct. The techniques are most applicable to applications that are computationally intensive, in particular scientific computing. Parallel programming has been most developed in this field. The most common applications of semantics-based program generation are compilers, including just-in-time compilers.

2.1.3 Summary

Simply put: meta programming is useful because the meta program improves the expressiveness of an object program and an object program improves the efficiency of meta program.

Syntax-based meta programming improves the efficiency of the programmer, whereas semantics-based meta programming improves the efficiency of the program.

2.2 Literature review

2.2.1 Compile-time meta programming

Compile-time meta programming concerns itself with the generation of program code at compile-time and usually does not allow the analysis of code. Examples of compile-time meta-programming languages are the C pre-processor, C++ templates and Template Meta Haskell [40]. Functions or procedures in compile-time meta programming are sometimes referred to as macros. A compile-time meta program is executed at the compile-time of some base language program which it creates. Basically the macro is executed by expanding the meta language code into code of the
base language. The result is a pure base language program that is then compiled as per normal.

Compile-time meta programming languages have two levels of execution: compile-time and run-time. The meta language is usually un-typed as type checking is often regarded as unnecessary because if the macro generates invalid code then this error will be identified in the post-generation, base language compilation phase. This is not always a valid justification though as it may be difficult for the programmer to trace a compile error in generated code back to the source of the error in a macro.

2.2.2 Partial evaluation

A computer program can be seen as an abstraction that reads input, processes it in some way, and writes output. Many programs have multiple inputs or an input that can be divided into many distinct parts. Ordinary program execution requires all of a program’s inputs to be available before computation can begin. However if only part of the input is available it may be possible to execute part of the program. This is the key principle of partial evaluation.

A program’s input is divided into two parts: a static part that is known and a dynamic part that is yet to be known. Given a specification of what program inputs are static, an external program known as a binding-time analyser analyses the main program to identify what parts of its code solely depend upon the static input. Another program, known as a partial evaluator, takes the results of binding-time analysis as well as the actual values of the static input and evaluates the parts of the main program dependent on the static input. The output of the partial evaluator is a specialized version of the original program called a residual program. The residual program takes as input the dynamic part of the original program’s input and produces the same output as the original program. The partial evaluation process has separated the execution of the original program into two discrete stages. The first stage is the execution of the partial evaluator with static input and the second is the execution of the residual program with dynamic input.
Partial evaluation [28, 20, 18] is a technique for program optimization. The residual program is seen as the original program optimized for computing with a given static input. It may be that executing the partial evaluator and the residual program is faster than executing the original program, hence the partial evaluation process is a direct optimization of the original program. In other scenario the original program may be executed many times where part of its input remains constant (static) and the other part varies for each execution (dynamic). In this situation although running the partial evaluator and the residual may be slower than running the original program, running the residual program alone is faster than running the original program. So if the partial evaluator was run once and the residual program run many times, after the time running the partial evaluator has been amortised there will be a net gain in running the residual program over the original program. Hence the performance of many runs of the original program has been optimized.

Partial evaluators exist for many languages such as JSpec [8] for Java and Tempo [10] for C.

Partial evaluators can be seen as a generalization of compilers. A compiler is a program that translates a program from one language into another, where typically the source language is not executable or the destination language has better execution performance. Related to a compiler is an interpreter: a program that executes a source program by emulating the operational semantics of a source language. Any language can be implemented as a compiler or interpreter with a trade-off that compilers typically give better performance but interpreters are easier to write.

If a partial evaluator is used on an interpreter with the source program (i.e., the one to be interpreted) as static input then the residual program has the same operational effect as the source program, it accepts the same input as the source program and is written in the same language as the interpreter. Hence the partial evaluation of the interpreter has led to a compilation of the source language into the interpreter’s language. This is known as the first Futamura projection [28] of the interpreter. The second Futamura projection can be achieved if the partial evaluator supports self-application. The partial evaluation of applying the partial evaluator to the interpreter
yields a residual program that accepts a source program as input and produces a compiled source program as output: in other words, it has generated a compiler.

The output from the first step of partial evaluation, binding-time analysis, is a source program that has been annotated with information about how the program could partially evaluate if certain input was made static. This annotated output can be seen as a language in its own right as it contains all the source language plus constructs to delay computations that depend on dynamic input. The annotation language is the most important component of a partial evaluation system because it controls what partial evaluation can be expressed and influences the algorithm for binding-time analysis.

The annotation language separates the source program into different levels of execution. Typically there are two levels: a first level that executes immediately because it only depends on static input and a second level whose execution is delayed because it depends on dynamic input. More advanced partial evaluation systems may partition input into more than two classes (static and dynamic) so their annotated programs will define many levels of execution.

Hence annotation languages are often referred to as two-level or multi-level languages. Some examples of two-level languages [20, 28] are based on the un-typed lambda calculus and were used to design and reason about early partial evaluation systems. Multi-level languages [18] were introduced later and once again used to reason about partial evaluation of the un-typed lambda calculus.

2.2.3 Multi-stage languages

Multi-level languages themselves were never presented as user-programmable languages but were instead always used as calculi for formalizing partial evaluation algorithms or as intermediate languages, internal to the partial evaluator. Multi-stage languages [46, 43, 44] are an idea that grew out of multi-level languages with the intention of making the level annotations accessible to the programmer. Multi-stage languages serve the same purpose as multi-level languages—to divide a program into
levels of evaluation—but are designed to be usable, stand-alone programming languages written directly by the programmer.

MetaML [46] pioneered the application of multi-stage languages. MetaML extends a purely-functional subset of ML with staging annotations for controlling the order of execution of a program. Furthermore in doing so it preserves the static type safety of ML languages: meaning that a type-correct program cannot ‘go wrong’ at any level. This is a particularly challenging property to guarantee because it means programs produced at run-time (i.e., residual programs) must be proved type-correct even though they do not exist in complete syntactic form when the type checker is run. MetaML is implemented as an interpreter written in SML.

In multi-stage programming, a program outputting another program is referred to as a meta program; the program being produced is referred to as an object program. The key property that the multi-stage type system brings to a language is that every type-correct meta program can only generate type-correct object programs.

MetaOCaml [7] is a newer language that has evolved from MetaML. It differs from MetaML in that it extends OCaml and allows the use of ML’s side-effecting statements. It is implemented as modified OCaml byte-code and native code compilers.

Encompassing side-effects into a multi-stage language weakens the static type system as it is possible, by exploiting assignment, to construct a type-correct program that ‘goes wrong’. This well-known issue known as scope extrusion is an open research problem in the area of multi-stage languages. The current solution in MetaOCaml is to admit a small amount of dynamic type checking. This issue and its relation to this work is discussed more in Section 3.3.1., but this work does not make any contribution towards solving this problem.

The ultimate goal of multi-stage programming, as is the ultimate goal of partial evaluation, is to improve the run-time performance of a program by separating its execution into discrete levels. Moreover partial evaluation and multi-stage programming share a common underlying theory. However they differ in their methods. Partial evaluation seeks to use binding-time analysis to automatically divide an arbitrary program into levels. The binding-time analyser is a complex program to write and
consequently may be difficult or impossible for it to produce a levelling of a program that is optimal or desired—and this will translate into reduced potential performance of the residual program. Multi-stage programming on the other hand puts the task of binding-time analysis in the hands of the programmer. Instead of a program that will automatically stage what the programmer writes, a multi-stage programmer specifies the stages explicitly as part of the program writing process (discussed further in Section 3.2.5). This enables the programmer to control the precise partial evaluation behaviour desired for a particular program. Of course this comes at the cost of the programmer needing to do more work when programming. Nevertheless building a universal and powerful partial evaluator has been elusive for many industrial languages and multi-stage programming has enjoyed many practical applications by enabling the manual staging of an industrial language.

The language Metaphor was inspired from the multi-stage languages MetaML [46, 44] and MetaOCaml [7]. Metaphor uses the multi-stage type system that was developed in MetaML and also used in MetaOCaml. Metaphor and MetaOCaml are similar in that they both try to apply the multi-stage language theory developed in MetaML to compiled languages with a byte-code target. Their chief difference lies in the imperative versus functional nature of their base languages. An imperative setting does not alter the essence of multi-stage programming because in Metaphor code objects are still functional in the sense that they are parametrically typed, have immutable values and can only be built via composition.

Nevertheless Metaphor shares similar research goals to MetaOCaml: using a type system to statically guarantee the semantic correctness of program generators, and making multi-stage programming more accessible to mainstream programmers.

2.2.4 LISP

The programming language LISP (and related languages such as Scheme) has constructs for deferring and forcing the evaluation of code. These constructs allow the programmer to write meta programs, called macros, that generate LISP code at runtime. The interpretive nature of the LISP environment and its uniform treatment of code and data make it conducive to writing programs that generate programs.
LISP defines three basic constructs for controlling the evaluation of programs that are also used in multi-stage languages. The constructs are: a quotation construct that quotes a piece of program code and defers its execution, a splice or unquote construct that resumes evaluation of an expression to be included in a quotation and an ‘eval’ construct that evaluates a deferred code expression.

LISP macros are quite similar to multi-stage programming but differ because LISP is not a statically typed language and therefore has no notion of multi-stage type safety nor the guarantee that every type-correct meta program only produces type correct object programs. Furthermore LISP macros are not statically scoped, meaning that quoted code can introduce free variables, unquoting can capture variables and eval can attempt to evaluate code with unbound variables. Scheme introduced a safer approach to macro writing that alleviated some of these scoping problems.

### 2.2.5 Object-oriented meta-programming languages

Several meta-programming language extensions exist for Java such as Jumbo [29], for run-time code generation; Meta-AspectJ [50], for source code generation; and JTS [3], for domain-specific language implementation. These languages are similar to Metaphor in that they facilitate the task of generating code in an imperative, object-oriented language. Moreover, they do this by treating code as first-class values—which can be passed and returned to and from methods—and by providing a quasi-quotation mechanism for constructing and combining code values. Jumbo is most like Metaphor as it is designed for run-time code generation and is not implemented by invoking the Java compiler to compile generated code. The other languages and also systems like OpenJava [47] are designed for the ‘offline’ generation of source code that is then fed to the standard Java compiler. There are, however, significant differences between Metaphor and these languages in terms of what kind of code they can generate and what level of safety they provide in the generation process.

The languages cited above allow the generation of whole class declarations as opposed to Metaphor which only generates statements and expressions (however expressions in Metaphor include in-line (or anonymous) method declarations). As these systems work on the full language syntax, they find it necessary to use a different kind
of quote and escape operators for each syntactic category (e.g., statements, classes, methods), although these are usually inferred in Meta-AspectJ and are not specified by the programmer. The need for different kinds of quote and escape operators was further avoided in Metaphor by our design decision to unify statements and expression (i.e., treat statements as expressions of type void; see Section 3.3.2) and by the multi-stage type system that identifies the use of cross-stage persisted values (see Section 3.2.3).

None of the Java meta-programming languages mentioned perform full, static type checking of generated code as is done in Metaphor and other multi-stage languages. In other words, they do not have the multi-stage typing property that every type-correct meta program only generates type-correct object programs. Jumbo performs a run-time type-checking phase on generated code immediately before it is compiled, JTS relies on the Java compiler to do type checking and Meta-AspectJ uses a type system that will statically prevent syntax errors in generated code.

Aside from type safety, these languages also differ from Metaphor in that they are only two-level, do not support cross-stage persistence (except on primitive values) and—with the exception of JTS—have no support for preventing inadvertent variable capture (see Section 3.2.4). These languages also have an ability, known as name abstraction, to reify names of types, variables, etc. as first-class values which can be used in code generation, whereas Metaphor does not support this feature.

A summary of the differences between these meta-programming languages is below. Many of these differences arise from a fundamental difference these languages take in their approach to meta programming. Many imperative meta-programming languages are designed as a way for writing a program to write code. In this scenario, requirements such as expressive code generation, printing generated code and fine control over syntax are important. For this reason these languages tend not to have a fully-static type system, lack cross-stage persistence (CSP) and support name abstraction. Conversely, multi-stage languages are designed as manual partial evaluation systems, or in other words, a means of controlling the order of evaluation of a program. In this scenario, static typing and cross-stage persistence are natural requirements implied by the staging of a program (i.e., the introduction of staging should not break
type safety nor restrict access to values). Also, name abstraction is not really relevant in the multi-stage approach as specific names are not important because programs are considered equivalent with respect to renaming. In other words, just because a program is staged does not mean that the programmer wants to forget about type safety or restrict access to values because of the presence of a stage.

<table>
<thead>
<tr>
<th>Feature</th>
<th>JTS</th>
<th>Jumbo</th>
<th>Meta-AspectJ</th>
<th>Meta-OCaml</th>
<th>Metaphor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative language</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Run-time code gen.</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Code unit</td>
<td>class</td>
<td>class</td>
<td>class</td>
<td>lambda expr.</td>
<td>stmt./expr.</td>
</tr>
<tr>
<td>Static type check</td>
<td>no</td>
<td>no</td>
<td>syntactic structure</td>
<td>mostly</td>
<td>mostly</td>
</tr>
<tr>
<td>Dynamic type check</td>
<td>uses javac</td>
<td>yes</td>
<td>yes</td>
<td>imperative effects</td>
<td>imperative effects</td>
</tr>
<tr>
<td>Stages</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>multi</td>
<td>multi</td>
</tr>
<tr>
<td>CSP</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Name abstraction</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Name capture</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

2.2.6 Higher-level imperative code generation

A number of staged, imperative programming systems exist but differ from Metaphor in their balance of type safety versus expressiveness. ‘C [14] is a fast run-time code generation extension to C. It inherits C’s weak type system but allows for more expressive code generation than can be achieved with the static typing of a multi-stage language (e.g., functions with a dynamically determined number of parameters). DynJava [36] borrows its syntactic design from ‘C but introduces a static type
system for code fragments that ensures the correctness of generated code. DynJava includes the names of free variables in the types of code fragments which, although may be effective for preventing the execution of code with unbound variables, places serious limitations on the abstraction-building power of the language. For instance, the developer of a library must choose concrete names for variables, which are then encoded in the type of the public interface, but may conflict with variables names used by its caller. The performance data reported for DynJava show marginal speed-ups if any. Neither ‘C nor DynJava are multi-stage or support cross-stage persistence of more than primitive types.

Another example is a run-time code generation extension [25] to Cyclone [27]. This extension has statically-type-safe run-time code generation but rather limited expressivity: code values cannot be passed or returned from functions (i.e., lambda abstracted), there is no explicit ‘run’ operator and is not multi-stage.

CodeBricks [1] is a software library designed to facilitate code composition in the CLR. Although CodeBricks is strictly speaking a library not a language extension it is quiet different to conventional run-time code generation APIs as discussed in Section 2.2.9. Rather than dealing with byte-code generation directly the library provides a set of abstractions that create functions and partially apply them. At the low-level what is actually happening is the merging and specialization of existing compiler-generated byte-code. No notion of byte-code manipulation appears in the high-level interface used by the programmer. CodeBricks makes no attempt to statically control the correctness of generated code; therefore code generation errors are likely to be a problem. CodeBricks is at least as expressive as a multi-stage language, it just lacks the type safety.

2.2.7 Generic programming

Generic programming involves writing polytypic functions that implement algorithms that are induced by the structure of a type. Hinze and Peyton Jones [24] present a generic programming extension to Haskell that allows the definition of polytypic functions to produce implementations of Haskell type classes. The code of the type class implementation is expanded at compile-time.
Cheney and Hinze [22] propose a method to implement polytypic functions in Haskell with only minimal language extension. The representation types of this system are analogous to the Type type of Metaphor. However this system does not involve the generation of code.

Template Meta-Haskell [40] is a compile-time meta-programming language extension to Haskell. It can be used to generate code for polytypic functions at compile-time.

The reflection and staging constructs of Metaphor enable the definition of polytypic functions that are type-checked at compile-time, but have code generated for them at run-time (see Section 4.3).

### 2.2.8 Object-oriented reflection

Type-directed Java [49] is a thorough design for adding type-safe type analysis constructs to Java. These constructs allow the programmer to write programs that use the Java reflection API with the static guarantee that there will be no reflection-based errors at run-time. Such a feature is useful for programs that need to analyse types to drive their logic, e.g., polytypic programs such as serializers. Many constructs in Type-directed Java mirror Metaphor constructs used to achieve the same purpose. Type-directed Java employs an equivalent of Metaphor’s type-if statement (Section 4.5.2) and member-iterating statement (Section 4.5.3). The future work of Type-directed Java also suggests a construct similar to Metaphor’s typecast block (Section 4.6.2). Type-directed Java does not deal with code generation in any way. Type-directed Java also has not been implemented. Presumably it would be implemented by translating all type analysis constructs to their operational equivalent in the dynamically typed Java reflection API. Implementing various type analysis constructs in Metaphor encountered intractable implementation problems in the CLR (see Section 4.5.4). It is anticipated that high-level type analysis constructs in Java would be easier to compile because of the erasure implementation model used by generics in the JVM.
Fähndrich [15] describes a language for compile-time reflection. The language is an extension to Spec# [2], which in turn is an extension of C# that enables greater formal specification of program contracts. Compile-time reflection is designed as a way to automatically augment a class with new members derived from existing ones. For example, adding a property that returns a collection of all the class’s methods as delegates. The code generating transformation is specified as a template separate from the class that it applies to. This enables the transform to be applied to any class rather than just one. The language avoids direct interaction with the reflection API since only one class is under analysis at once and members are bound to meta variables by the specification of the template. The language is statically type-safe: code generation errors are reported by the compiler prior to template expansion and all generated code is guaranteed to be type-correct.

2.2.9 Run-time code generation APIs

Run-time code generation (RTCG) libraries represent a program as an abstract syntax tree or other object model. This offers a higher level of abstraction for manipulating code compared to directly reading or writing concrete syntax as strings or bytes. These libraries use their object model to enforce the syntactic structure of the code being generated. In theory it should not be possible to use the object model to construct syntactically invalid code, thus reducing programming errors. The type system of the language the object model is written in can effectively check at compile-time the syntax of all code that could be generated. Library based approaches to program reflection are quite common on modern object-oriented platforms such as the Common Language Runtime (CLR) and the Java Virtual Machine (JVM). Examples of such libraries include Reflection.Emit, part of the base class library of the CLR and designed as an extension to the CLR’s reflection API; BCEL for Java; and AbsIL and PERWAPI also for the CLR. These libraries operate on the byte-code language of their respective platforms. Consequently they tend to require a lot of code to use—so much so that the code for using the library overwhelms the code it wants to generate. This can make these libraries tedious to use. Furthermore these libraries offer no type safety support for the code they generate, hence making errors in run-time generated code a common problem.
These APIs are the current, dominant approach for run-time code generation on the JVM and CLR platforms. These libraries are tedious and verbose to program in and offer limited correctness guarantees about generated code.
3 Imperative multi-stage programming

The aim of this research is to investigate the interaction between multi-stage programming and object-oriented type introspection to enable the creation of staged, polytypic programs. To study the effect of object-oriented type introspection on multi-stage programming, it is necessary to first create a multi-stage language based on an object-oriented one. This chapter introduces Metaphor as a new multi-stage programming language extension to a C#/Java-like language. It discusses the impact an object-oriented language and virtual machine has on the design and implementation of a multi-stage language as compared to existing functional multi-stage languages. The foundation of the language presented in this chapter will be extended in the succeeding chapter.

The contributions of this chapter are

- the design and implementation of a multi-stage extension to a C# or Java-like base language, and

- the raising of the level of abstraction of current practices of run-time code generation in object-oriented virtual machines.

The design of Metaphor was heavily influenced by the multi-stage languages MetaML [46] and MetaOCaml [7]. Metaphor differs from other multi-stage languages because it is an extension of an imperative, object-oriented language and is implemented on a large virtual machine platform of similar nature. These differences raise issues for multi-stage language design and implementation that have not been addressed before. An imperative multi-stage language must handle pervasive mutable variables and the implementation must utilize the services of a powerful run-time system—sometimes in novel ways. These issues are discussed in this chapter.
3.1 Motivation

The Java Virtual Machine (JVM) and the Common Language Runtime (CLR) are two popular platforms for modern software development—both of which are imperative and object-oriented. Developers working on these platforms sometimes need to write programs that generate code. Programmers wanting to write a code generating application on these platforms have a number of challenges: the level of code generated (source, intermediate or native), the representation of the code (strings or object model), how to detect/prevent errors in code-generating code and how to provide external interfaces to code generation routines for other applications to build upon. Ideally, what is desired is a high-level, type-safe approach for generating dynamic code in modern object-oriented software development platforms—one that frees the programmer from dealing with the data representation of code and provides an abstraction-building capability.

MetaOCaml is an excellent tool that enables OCaml programmers to write code-generating programs in a way that is simple, expressive and statically typed. The staging of programs in MetaOCaml is done with the intent of increasing the run-time performance of the program. However generating highly efficient code in a functional language is not always possible. This has lead developers and users of MetaOCaml to write staged programs that generate C code, instead OCaml, via a process known as off-shoring [11]. Another useful approach would be a code generation system that provided the type safety of MetaOCaml and the performance of an imperative language. Currently no multi-stage language extension exists for an imperative language. In a homogeneous, imperative, multi-stage language there is no need to marshal data values between the meta and object programs, as is necessary with off-shoring. As discussed in Section 2.2.6, many imperative meta-programming languages either have weak type systems that provide few static guarantees or are overly restrictive in the code they can generate.

To address these deficiencies I have designed and implemented the programming language Metaphor. The language improves the level of abstraction of object-oriented meta programming and expands the applicability of multi-stage programming.
3.2 Language overview

Metaphor is an object-oriented language in the style of C# or Java that supports multi-stage programming. The base language includes common object-oriented constructs such as classes, methods, and fields, as well as the general imperative programming constructs of assignment and control flow.

3.2.1 Staging constructs

Metaphor employs the three classical constructs from multi-stage programming: brackets, escape and run.

3.2.1.1 Brackets

The brackets `<|e|>` quote a fragment of code to construct a code object. This can be viewed as delaying the execution of e. For example, whereas in a running program the code `1+1` will evaluate to `2`, the code `<|1+1|>` will evaluate to `<|1+1|>` (i.e., not undergo any evaluation) as the evaluation of the sum has been deferred by the brackets.

The result of the expression `<|1+1|>` is a code object that represents the code `1+1`. Code objects are first-class data values in the language and as such can be stored in a data structure or local variable, passed and returned to and from functions, and used inside brackets (i.e., nested quotations). Naturally code objects need a type: code objects have the parametric or generic type `Code<T>` where T is the type of the expression inside the brackets. The following table lists some code expressions and their corresponding types.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt;</td>
<td>1+1</td>
</tr>
<tr>
<td>`&lt;</td>
<td>Console.ReadLine()</td>
</tr>
<tr>
<td>`&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
3.2.1.2 Escape

Whereas brackets are used to delay computation, the escape operator, written \(-e\), is used to resume computation that has been delayed by brackets. Where an escape operator appears, which must be inside brackets, its operand is evaluated immediately instead of being delayed by the brackets. The operand must evaluate to a code object and hence must have a code type. The code object produced by evaluating the operand is spliced into the code object being built by the surrounding brackets.

The table below shows some code expressions and how the escapes within them are evaluated.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code&lt;double&gt; x =</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid 1 + 1 \mid&gt;);</td>
<td></td>
</tr>
<tr>
<td>Code&lt;double&gt; y =</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid 2 * \sim x \mid&gt;);</td>
<td>(y \rightarrow \mid 2 * (1 + 1) \mid&gt;)</td>
</tr>
<tr>
<td>Code&lt;String&gt; x =</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid \text{Console.ReadLine()} \mid&gt;);</td>
<td></td>
</tr>
<tr>
<td>Code&lt;string&gt; y =</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid \text{string.Format(&quot;Hello {0}&quot;, \sim x)} \mid&gt;);</td>
<td></td>
</tr>
<tr>
<td>Code&lt;Foo&gt; foo(bool b) { return b ?</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid \text{new Foo()} \mid&gt;);</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid \text{null} \mid&gt;);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Code&lt;Bar&gt; y =</td>
<td></td>
</tr>
<tr>
<td>(&lt;\mid \sim \text{foo(true)}.bar \mid&gt;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From a typing point of view the escape operator removes one level of code typing from its operand. For example if the type of the expression used as an operand to escape is Code<int> then the type of the whole escape expression would be int. It is a type error to use escape outside of brackets since there would be no code object into which to splice.

3.2.1.3 Run

Finally the run operation is exposed as a method on code objects, \(e\.Run()\). Invoking the run method will cause the code represented by the code object to be executed.
and the result of execution returned. The following table shows the effects of running various code objects.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>1 + 1)</td>
</tr>
<tr>
<td>(</td>
<td>1.0 / 0.0)</td>
</tr>
<tr>
<td>(</td>
<td>\lfloor 1 + 1 \rfloor)</td>
</tr>
</tbody>
</table>

### 3.2.2 Staging example

These three constructs are sufficient to control the order of execution within a program and thereby enable programmers to write programs that dynamically eliminate unneeded abstractions for a given context. To see how this works in practice let us consider a simple scenario: code that raises floating-point numbers to positive integer powers. Mathematically this operation is defined as

\[
\begin{align*}
    x^0 &= 1 \\
    x^{n+1} &= x \cdot x^n
\end{align*}
\]

#### 3.2.2.1 Specific functions

If in a particular context a program needs to raise numbers to the power of 3 then a programmer may write a function like `power3` below. If in another context the program needs to raise numbers to the power of 4 then the programmer would be forced to write another function, `power4` also below. Here a specialized version of a more general function needs to be written for every context.

```c
float power3(float x) {
    return x*x*x;
}

float power4(float x) {
    return x*x*x*x;
}
```
3.2.2.2 General function

Having realized that the program needs to compute powers with a number of different exponents, a sensible programmer would generalize these specific power functions into one that works with any exponent, shown below.

```c
float power(float x, int n) {
    float result = 1.0;
    while(n > 0) {
        result = result*x;
        n = n - 1;
    }
    return result;
}
```

This modification is a great improvement to the modularity and comprehensibility of the program’s code. It has taken a collection of related functions and replaced them with a single, generalized function that does all the work of the original functions: this is the very process of computational abstraction. The ability to abstract is essential to the construction of software systems. Only through abstraction is it possible for programmers to build larger and more complex software. Programmers should always strive for greater abstraction as it leads to higher quality and more reusable software. The power example may be trivial but it epitomizes the purpose and benefit of abstraction where it occurs at any level of complexity in a software system. Abstraction—

- reduces the amount of code, thereby reducing the number of places coding errors can occur and improving the readability and maintainability of the code, and

- provides a single point of interface from which the encompassed functionality can be used in building higher-level abstractions.

The one drawback of abstraction is that it incurs a performance cost in the execution of a program. In the power example this can be seen by the introduction of the loop in the generalized version. The loop requires branching, condition testing and counter management, none of which are present in the specialized versions and all of which increase program execution time. If the function is being called infrequently
or with varying exponents this cost may be considered acceptable. However, if the function is called multiple times with the same exponent this cost can have a significant effect on the overall performance of the program. So although abstraction leads to great gains in software development it can sometimes have serious drawbacks when the software is actually run.

3.2.2.3 Staged function

Multi-stage languages solve this problem by allowing the programmer to impose an order in which the abstractions in a program are applied. In the case of the power example the loop is applied first and the multiplications are evaluated symbolically. In compiler terminology this process is known as loop unrolling, but in a multi-stage language this optimization is directed by the programmer not the compiler. Below is the code for the staged version of the power function. It is exactly the same code used for the general power function except for the addition of staging constructs that delay the evaluation of the multiplications. As the loop in this function executes, rather than performing the $n$ multiplications it builds a code object that will perform the $n$ multiplications.

```c
Code<float> power(Code<float> x, int n) {
  Code<float> result = |1.0|;
  while(n > 0) {
    result = |result * x|;
    n = n - 1;
  }
  return result;
}
```

The invocation `power(|x|, 3)` would return the code object `|1.0*x*x*x|`. This is exactly the code (except for the unnecessary multiplication by one which can be eliminated by a slight modification to the staged power function) that was used in the original specialized `power3` function.

The staged power function has preserved the generality of the general power function (i.e., can compute for any value of $n$) but has altered the performance cost of the general power function by separating the execution of the loop and the multiplications into discrete stages. The loop stage executes first and produces the code for the second stage of multiplications. The code that executes in the second stage is the
same as the code in the original power functions specialized for a particular exponent, but rather than being written by a programmer this code has been computed by a program.

Say that a program needs to compute the cube of the numbers in a very large array. This can be achieved abstractly and efficiently by employing the staged power function to generate a specialized cube function and then calling this function when iterating over the array. This is shown in the code below. Delegates and anonymous functions are standard features of C# and are discussed in more detail with respect to multi-stage programming in Section 3.3.2. Func is the type of functions that have a float parameter and return a float. The code for the specialized power function is built between the quotation brackets. Invoking the Run method causes the code object representation of the function to be realized as callable, executable code and a reference to this is stored in the variable power3. power3 can be invoked from inside the for-loop to compute the cube of its argument without incurring the computational overheads of the general power function.

```csharp
delegate float Func(float x);

Func power3 = <|delegate (float x) {
   return ~power(<|x|>, 3);
}|>.Run();

float[] xs = ...;
for(int i=0; i < xs.Length; i++)
   xs[i] = power3(xs[i]);
```

This example could be further optimized by generating code for the for-loop and inlining the code for power3, thus eliminating the overhead of a function call.

```csharp
delegate void FuncArray(float[] x);

FuncArray forall_power3 = <|delegate (float[] xs) {
   for(int i=0; i < xs.Length; i++)
      xs[i] = ~power(<|xs[i]|>, 3);
}|>.Run();

float[] xs = ...;
forall_power3(xs);
```
3.2.3 Multi-stage type system

The multi-stage type system is designed to prevent errors in code generation by tracking the use of staging constructs. Type checking only occurs once at the compile-time of the source program. No type checking is needed at run-time on generated code. The type checker checks code inside brackets in a similar way as if it appeared outside brackets, but also performs additional checks to ensure that the type of spliced code is compatible with the environment it is spliced into and that variables are correctly scoped across stages. Below is an example of an incorrectly typed code splice.

```csharp
Code<string> x = <"hello">;
Code<int> y = <|Math.Abs(~x)|>; // type error
```

In this code the `Abs` function expects an `int` but the code spliced in from `x` will provide a `string`. Hence this is a typing error and will be detected by the compiler.

Every expression and statement in a program occurs at a particular `level`. The level is a non-negative integer equal to the number of brackets around the code minus the number of escapes around the code. In a running program, code at level 0 is executed and code at level 1 and higher is delayed. The `Run` method makes a transition between stages by executing one level of delayed computation in a code object.

Every variable used in a program has an associated declaration level (the level at which it is declared) and use level (the level at which it is used). If the use level is greater than the declaration level then the variable will already have a run-time value when the code object it is used in is constructed. Therefore the value of the variable can be substituted for the variable reference in the code being generated. For example,

```csharp
int x = 2; // x decl at level 0
Code<int> y = <|x + 1|>; // x used at level 1
```

When this code executes the value stored in `y` will be the code object `<|2+1|>` since the variable `x` has been replaced by its value. This process is known as cross-stage persistence.

If the use level is equal to the declaration level then the variable is used as a reference to its declaration, which is how variables normally work in programming languages.
If the use level is less than the declaration level a compile-time error occurs because it is semantically invalid to use a variable before the variable is declared. In the below code the variable \(x\) is declared at level 1 and used at level 0 but the variable does not exist at level 0 and so is effectively out of scope.

\[
\begin{align*}
\text{Code<int> } & \ x = \ldots; \ // \ x \ \text{decl at level 1} \\
& \ \ \ \text{int y} = \neg x; \ // \ \text{error: x used at level 0}
\end{align*}
\]

3.2.4 Automatic variable renaming

A common problem in meta-programming systems is inadvertent variable capture where two or more variables with the same name are used in the one context causing a conflict or shadowing of the name. Multi-stage languages avoid this problem by automatically renaming all variables as the code is generated. In the example below, two distinct variables in different lexical scopes are called \(x\) and both end up combined in the dynamic scope of the code that is generated. To avoid a collision of the variable name \(x\) both variables are renamed to have distinct names (\(x_0\) and \(x_1\)) in the generated code.

\[
\begin{align*}
\text{Code<int> } & \ f(\text{Code<int> } y) \ {\text{\{}} \\
& \ \ \ \text{return } <\text{int x} = 7; \ -y + x|>
\end{align*}
\]

\[
<\text{int x} = 5; \ -f(<\text{x}|>)|>
\]

// evaluates to:
// \<\text{int x0} = 5; \text{int x1} = 7; \text{x0 + x1}|>

3.2.5 Programming methodology

In practice, programmers typically write multi-stage programs by taking an unstaged program and modifying it with staging constructs. Because multi-stage languages have their origins as intermediate languages for partial evaluators, the process of introducing staging constructs into a program is related to binding-time analysis from partial evaluation.

Binding-time analysis is a phase of partial evaluation that takes a source program and set of inputs to that program which are marked static. The actual values of the input
are not known, only the fact some inputs will have static values available and others not. A binding-time analyser uses this information to determine what parts of the source program can be completely evaluated with only the static input available. The result of binding-time analysis is the source program marked with annotations that divide the code into two stages: code that can be evaluated with only static input and code that depends on some dynamic input. These annotations used by a binding-time analyser are equivalent to the staging constructs of multi-stage programming in that they delay (brackets) and resume (escape) evaluation of a program.

The process of a programmer writing a staged program can be viewed as a manual form of binding-time analysis. (Moreover multi-stage programming in general is often thought of as programmer-directed partial evaluation.) The programmer starts with an un-staged function, say

\[
T \ f(A \ a, \ B \ b, \ C \ c) \ { \ ... \ }
\]

and chooses one or more input parameters to be static The remaining parameters (viz. dynamic parameters) and the return parameter are annotated with a \texttt{Code} type to indicate that their values will be delayed (i.e., more delayed than the static parameters’ values). Hence, say, the first two parameters were decided to be static, then the function signature for the staged version would be

\[
\text{Code}\langle T \rangle \ fs(A \ a, \ B \ b, \ \text{Code}\langle C \rangle \ c) \ { \ ... \ }
\]

Making this change will bring about a multitude of type errors in the body of the function as the type of a parameter has changed. These type errors are good starting points for the programmer to identify what parts of the function cannot be completely evaluated with only the static parameters and hence stage the program by analogy to binding-time analysis.

As a concrete example consider the process of staging the power function from Section 3.2.2. The un-staged and staged versions of this function are repeated below. After writing the un-staged version the first step is to decide which of its two parameters shall become static. At this step the programmer must exercise his intuition as to what choice of parameters would bring about a sensible staging of the function and what the likely optimization effect would be. For the power function we know that
choosing the exponent parameter to be static will yield the ability to unroll the while loop and hence produce an effective optimization. If the base parameter were chosen then very little partial evaluation (only the insertion of a literal value) could be performed and hence would not be an effective optimization of the function. More complex algorithms than power require the programmer to have a deeper understanding of how the algorithm operates and what its performance bottlenecks are in order to yield a judicious choice of static parameter selection.

<table>
<thead>
<tr>
<th>Un-staged</th>
<th>Staged</th>
</tr>
</thead>
<tbody>
<tr>
<td>float power(float x, int n)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>float result = 1.0;</td>
<td></td>
</tr>
<tr>
<td>while(n &gt; 0) {</td>
<td></td>
</tr>
<tr>
<td>result = result*x;</td>
<td></td>
</tr>
<tr>
<td>n = n - 1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return result;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>Code&lt;float&gt; power(Code&lt;float&gt; x, int n)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>Code&lt;float&gt; result = &lt;</td>
<td>1.0</td>
</tr>
<tr>
<td>while(n &gt; 0) {</td>
<td></td>
</tr>
<tr>
<td>result = &lt;</td>
<td>*result * ~x</td>
</tr>
<tr>
<td>n = n - 1;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return result;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Once the static/dynamic parameter split is decided upon, the types of parameters and return value are modified and staging annotations (brackets and escapes) are applied to the body of the function. In this example the staging constructs are truly annotations because the un-staged code can be recovered simply by eliding the staging constructs from the staged version.

This pure annotation approach is always helpful as a general guiding principle for creating staged programs, but it sometimes breaks down in the finer points of a complex application. Programs are staged as a means to provide run-time optimization through run-time code generation. In many scenarios the precise code that needs to be generated at run-time is specified and the staging process must be crafted to yield this exact result. Often this involves including additional logic in the staged version that while it did not occur in the un-staged version would be semantically benign if it did. For example in an un-staged program, whenever multiplying two numbers a programmer might consider first testing if the first number is zero and thereby avoiding the multiplication if it is. In practice this may not be done very often because it is assumed that the cost of a if-test on every multiplication will not be amortised by
the avoidance of some multiplications. However for a staged program this is not so: since the if-test is performed statically there is always either a non-zero multiplication in the dynamic program or no multiplication at all.

This distinction can be seen in the code below. A programmer would seldom write the un-staged case but would frequently write the staged case. Note though that the staged version is an annotation of the un-staged version.

```
<table>
<thead>
<tr>
<th>Un-staged</th>
<th>Staged</th>
</tr>
</thead>
<tbody>
<tr>
<td>float mul(float x, float y)</td>
<td>Code&lt;float&gt; mul(float x, Code&lt;float&gt; y)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>if(x == 0.0) return 0.0;</td>
<td>if(x == 0.0) return &lt;</td>
</tr>
<tr>
<td>else return x * y;</td>
<td>else return &lt;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
```

Consider another example to illustrate the difference between the economies of scale of un-staged and staged programming. An un-staged function that calculates the sum-product of two arrays would simply be a loop that sums the pair-wise products of elements from the two arrays. A staged version where the first array is statically known could, by analysing the static array, eliminate multiplications because of the presence of zeros and ones, and factor the order of the sum to avoid repeated multiplication by the same number. For example say that the value of the static array is \([0, 2, 1, -2]\) and the dynamic array is \(y\), then the code produced by a staged sum-product function could be \(<|2 * (y[1] - y[3]) + y[2]|>\), which only involves one multiplication. Performing this kind of analysis in the un-staged version is possible and would not affect the outcome of the function, but it would never be advantageous performance-wise. However performing this analysis in the staged version would be worthwhile assuming the static array often has a value amenable to the resultant optimization of this analysis.

### 3.2.6 Testing and debugging

The testing and debugging of multi-stage programs is an interesting and challenging problem. In a traditional debugging session, the source code of a program exists before the program is run and the programmer would usually have some familiarity
with this source code since it was hand-written. Neither of these assumptions hold in a multi-stage context because new code is produced at run-time as a product of the source program's computation.

For the most part the theoretical properties of multi-stage languages go a long way to reduce the amount of debugging required to give rise to a correctly working program. Firstly, the multi-stage type system guarantees that the generated program has no syntactic or semantic errors, leaving only logic errors as a possibility. This is a great advancement over many meta-programming systems where the programmer must first contend with semantic errors in the code before debugging can start in earnest. Secondly, multi-stage programs can be created by annotating already correct un-staged programs, therefore reducing the introduction of logic errors by reusing an existing, sound program structure.

The practical problem of how to debug multi-stage programs is beyond the scope of this work and so not addressed. We could however imagine how multi-stage debugging might work. The typical approach taken to debugging in the CLR and JVM environments is that of an interactive debugger integrated with a visual development environment such as Visual Studio or Eclipse. The debugging of the generation stage of a multi-stage program easily fits into this model as at this stage the generated code is still just data and has not become ‘new’ code yet. So a debugger aware of the syntax and semantics of the multi-stage program could trace the flow of execution in and out of the brackets in the meta program. At the implementation level code objects are abstract syntax trees and it would be straightforward to write a function that unparses the tree into textual source code. This function would be used by the debugger to pretty-print the values of code objects for the user at run-time.

The debugging of the generated stage (i.e., object program) is slightly more technically complicated. The underlying development environment would need to support the dynamic loading of source files and symbol data. In debug mode, whenever a code object is run the un-parser would be used to generate textual source code from the abstract syntax tree. The generated source code would be saved to a file on disk and the un-parser would keep track of what fragment of the text file is associated with each node. Then the regular code generator would be used to produce a
binary file on disk. Finally a third phase, which may be incorporated into a special
debug mode of the code generator, would emit the symbol file that correlates code
between the generated binary and source files. The debugger can then dynamically
load the source, binary and symbol files and integrate them into the active debug-
ging session.

The ability to use the un-parser to save generated source code to disk is useful as a
debugging tool in its own right: once saved the programmer can analyse the pro-
gram for bugs off-line or compile-execute-debug the code separate from its generat-
ing meta program.

For the purposes of developing programs for this work, the debugging of multi-stage
programs was undertaken in a very basic approach. Firstly, so-called ‘printf’ debug-
ging can be used to trace the execution of a generated program. Secondly, the gener-
ated code can be saved to disk in compiled form and a disassembler used to view and
analyse the code to discover bugs. The intermediate language of the CLR (which the
disassembler shows) is quite high-level and tolerable for human reading. Addition-
ally since staged programs are written with a specified target in mind for the gener-
ated code, inspecting the code in either source or disassembled form is invaluable for
verifying the correctness of staging.

### 3.3 Issues

When applying a multi-stage language design to C#, a number of issues that do not
occur in a functional setting arose because of the imperative and object-oriented na-
ture of C# and from characteristics of the underlying CLR virtual machine. Many of
these issues have been published in [35].

#### 3.3.1 Scope extrusion

*Scope extrusion* is a well-known problem in multi-stage programming concerning the
use of free variables in code values. The problem can cause well-typed staged pro-
grams to ‘go wrong’ and is hence an issue for the static typing of any multi-stage lan-
guage. Ideally the type system should statically prevent all erroneous use of scope extrusion, but as we shall see this may not be possible or feasible and so a weaker solution needs to be adopted. Although Metaphor does not offer any new solutions to scope extrusion, we nonetheless find it useful to discuss the problem here.

Multi-stage languages allow the programmer to construct code values containing variables bound outside of the code value itself, and hence from the context of the code value the variable appears free. In the following MetaOCaml code, the variable \( x \) appears free in the code value passed to \( f \).

\[
\langle\text{fun } x \rightarrow .\,~(f \langle x \rangle.)\rangle.
\]

This ability is essential to the power of multi-stage programming as it enables function application to be staged.

Constructing code values with free variables, known as open code, is not a problem in itself. A problem arises however when an attempt is made to run an open code value: there are no operational semantics to evaluate a program with a free variable. This unsafety of the multi-stage type system was identified with the following MetaOCaml code that runs open code.

\[
\langle\text{fun } x \rightarrow .\,~(.!\langle x \rangle.)\rangle.
\]

To overcome this problem, MetaOCaml introduced the environment classifier type system \([45]\) to ensure the safety of the run construct. In this type system a superfluous type variable is added to the code type—similar to the superfluous type variable used in Haskell’s state monad—which restricts how code values can be run. Run, similar to \( \text{runST} \) in Haskell, would have the rank-2 type \( \forall a. (\forall s. \text{Code } s a) \rightarrow \forall s. a. \)

An inference algorithm exists for this type system and is implemented in the current release of MetaOCaml \([6]\). The algorithm is an extension to the standard type inference algorithm used in OCaml. The programmer never needs to specify the classifier parameter as it is inferred.

---

1 In MetaOCaml brackets are written \( \langle \ldots \rangle \), escape is written \( .\,~ \) and run is written \( .\,~!\).
Scope extrusion is a typing problem that occurs in a multi-stage language (i.e., one with symbolic evaluation or ‘evaluation under the lambda’) with mutable values. The problem is exemplified by the code

```plaintext
let r = ref .<0>. in
     .<fun x -> ~(r := .<x>.; !r)>.
```

or in Metaphor syntax

```plaintext
Code<int> r = <|0|>;  
    <|delegate (int x) { 
       return ~(r = <|x|>); 
    }|>
```

When constructing the code for the anonymous function, a code value containing the formal parameter of this function is stored in a location outside the scope of the function. Once this variable has extruded its defining scope any use of it (i.e., with run or escape) is invalid.

This problem seems strongly related to the safety of run. Both problems can be generalized into the same code pattern

```plaintext
     .<fun x -> ~(e .<x>.)>.
```

where \( e \) is an expression that performs a stateful operation with its argument \( .<x>.. \)
A stateful operation could be executing the code value, storing the code value in a memory location, or engaging the code value in an I/O operation. Furthermore the reason why this code pattern is dangerous while other code patterns that perform stateful operations on code values are not is because this stateful operation works on a code value with free variables—so called open code.

It is not valid to execute code that has free variables hence running an open code value is always invalid. Once an open code value has extruded its scope it is no longer possible for that value to be spliced in such a way as to capture the free variable and create a closed code value again. Therefore there is nothing useful that can be done with such a value: it cannot be run because it is open and it cannot be closed.

Since the problems of scope extrusion and run safety both arise from the existence of open code and what operations are valid on it, it would seem logical that both prob-
lems share a common solution. Run safety has been solved in MetaOCaml however its solution does not cover scope extrusion. This code will store a code value of the formal parameter \( x \) in the reference cell \( r \).

\[
\text{let } v = \\
\text{ let } r = \text{ref } .<0>. \text{in} \\
\text{ignore } .\langle\text{fun } x \to .-(r:= .<x>.; !r>).. !r}
\]

Calcagno et al. [5] suggest a type system to solve the scope extrusion problem. This type system has not been implemented in MetaOCaml and it is difficult to apply the type system to Metaphor chiefly because references can and should be used sparingly in an ML language but all variables are references in a C#/Java language. A monadic approach to multi-stage programming [42] may offer some insights into how to safely type scope extrusion in an imperative multi-stage language, but as yet no results have been published.

The current release of MetaOCaml deals with the scope extrusion problem by checking a code object for free variables when it is run. If a free variable is found an exception is raised and the code object is not run. The same approach is adopted in Metaphor.

Scope extrusion is an active research problem in multi-stage programming that affects both imperative and impure functional languages alike. It is a difficult problem to tackle and although no contribution is made towards it by this work, the problem is largely orthogonal to the issues dealt with by this work.

### 3.3.2 Syntactic issues

Functional languages typically have a simple syntactic structure comprising of a single category, the expression, for all evaluable code, which includes lambda expressions for defining functions. On the other hand, languages like C# and Java have a complex syntactic structure for code divided into expressions, statements, methods and classes. Metaphor inherits from these languages their complex syntax, but in order to avoid separate staging operators and types for each syntactic category, the Metaphor language makes a few simplifications. Expressions and statements are unified into a single category whereby statements become \texttt{void} expressions (cf. the use of the
unit type for imperative operations in ML languages). This leads to programs that are syntactically perverse (e.g., a while loop in a function call’s argument list) but which nevertheless have fairly intuitive semantics.

The `void` type is not a first class type in C# because it can not be used to declare a variable or to instantiate a generic type parameter. Therefore the code of void type cannot be written as an instantiation of the generic `Code` type but rather as the non-generic type `CodeVoid`, which is intuitively equivalent to `Code< void >`.

The result of an expression can be coerced to `void` by suffixing a semicolon at the end of the expression (cf. the `ignore` function from OCaml). For example, the below code shows how a method invocation of a value-returning function can be interpreted either as returning a value or not.

```csharp
Code<double> expr = <|Math.Abs(~x)|>;
CodeVoid stmt = <|Math.Abs(~x);|>;
```

The ability to generate code for a function is essential to the usefulness of multi-stage programming and this ability is achieved rather elegantly in a functional language through the use of higher-order functions. Metaphor achieves the same by borrowing from C# the features of delegates and anonymous functions.

A delegate type is the object-oriented equivalent to a named function type; delegate objects are instances of delegate types and are a special kind of immutable object that refers to a function. Delegate types are declared by specifying the `delegate` keyword followed by a method signature where the name of the method is the name of the delegate type.

Delegate objects are created either by pointing them to an existing method or as a result of an anonymous function declaration. An anonymous function is like a lambda expression and is defined by specifying the `delegate` keyword, a list of function parameters and a function body.

```csharp
delegate int Func(int x);

Func f = delegate (int x) { return x + 1; }
```
The first line above declares a new delegate type `Func` for functions that take an `int` parameter and return an `int` result. The second line defines an anonymous function. The delegate type of the anonymous function is inferred from its context, and the return type of the function is then taken from the delegate type. An anonymous function is a kind of expression and hence can be quoted by staging brackets thus allowing the staging of function definitions whilst eliminating syntax required to quote method definitions.

A goal of Metaphor was to keep the staging syntax simple by avoiding separate staging constructs for statements, expressions and methods. This was achieved through the unification, for the purposes of quotation, of the traditional syntactic categories statement and expression, and by the use of anonymous function definitions.

### 3.3.3 Assignment

Not all expressions can be used on the left side of an assignment. Only particular expression (variables, object fields and array elements) are assignable. Therefore if the escape operator can be used on the left side of an assignment then the type system must perform additional checking to ensure that the code being spliced in is assignable. Concretely the type system must disallow code such as

```plaintext
Code<int> x = <|0|>;
CodeVoid y = <|~x = 1;|>;
```

but allow code such as

```plaintext
int[] a = new int[] { 0 };
Code<int> x = <|a[0]|>;
CodeVoid y = <|~x = 1;|>;
```

Unlike functional languages, imperative languages do not distinguish at the level of the type system between values that are immutable and locations that are assignable. This works for strongly-typed imperative languages because locations are not first-class in the sense that they can be used only with limited forms of abstraction (viz. cannot be nested, cannot be returned from a function and, in Java, cannot be passed to a function). It is this fact that makes the handling of locations in a multi-stage C#/Java language different from ML-style references in MetaOCaml.
Imperative language compilers internally differentiate between \textit{l-values} which are assignable locations and \textit{r-values} which are not. By making this distinction explicit in the types of code objects, it is possible for the type system to enforce the safety of left-side assignment splicing.

The type \texttt{CodeRef<T>} is a subtype of the regular code type \texttt{Code<T>}, meaning that it can be used anywhere a code object can be used. Expressions that are capable of assignment (local variables, method parameters, object fields and array elements) are given the assignable code type when they are quoted in brackets. The escape operator expects an assignable code object when it is used as the left-side of an assignment or as a pass-by-reference actual parameter.

With this type system both code fragments at the beginning of this subsection would be rejected, but the latter can be rewritten like so to remedy the problem.

\begin{verbatim}
int[] a = new int[] { 0 };
CodeRef<int> x = <a[0]>;
CodeVoid y = <x = 1;>;
\end{verbatim}

### 3.3.4 Cross-stage persistence semantics

Cross-stage persistence (CSP) is a novel feature of multi-stage languages that allows expressions to be evaluated at an earlier stage and embedded as a constant value in the current stage.

An interesting semantics question arises when applying the implicit cross-stage persistence of variables in an imperative language. Intuitively all variables in an imperative language are like mutable references in an impure functional language. However when these variables are cross-stage persisted should they be persisted by value or reference?

In practice both semantics are useful in different scenarios: value cross-stage persistence to introduce literals into compiled code as a form of code optimization, and reference cross-stage persistence to allow generated code to share data. The difference between the two semantics is expressed in the following pieces of code. Should the Metaphor code be semantically analogous to the first MetaOCaml code fragment
which embeds the variable by reference or the second MetaOCaml code fragment which embeds the variable by value?

```csharp
// Metaphor CSP of variable
bool x = false;
Code<bool> y = <\x>;:

// MetaOCaml CSP of reference by reference
let x = ref false
let y = .<!x>.:

// MetaOCaml CSP of reference by value
let x = ref false
let y = let x' = !x in .<x'>.:
```

It is possible to express both semantics easily in MetaOCaml because it has an explicit dereference operation (i.e., !) which allows the programmer to precisely control at what level the variable is dereferenced (level 1 for reference cross-stage persistence or level 0 for value cross-stage persistence). In Metaphor there is no syntax to specify the dereference operation hence there is no way to express both semantics.

To overcome this problem, Metaphor chooses to implement value cross-stage persistence because (1) it is the most common semantics desired in practice, (2) producing literals in compiled code is an important optimization in a staged program, and (3) reference cross-stage persistence can be achieved on top of value cross-stage persistence but not vice versa. Reference semantics can be done by introducing a layer of indirection to the value using, say, a wrapper class or a singleton array. (Arrays in Metaphor, as in C#, are always treated as reference objects.) For example, below is how reference cross-stage persistence can be written by the programmer.

```csharp
bool[] x = new bool[] { false };
Code<bool> y = <\x[0]>:
x[0] = true;
bool z = y.Run(); // z == true
```

A singleton array is used to indirect access to the location. The object reference to the array itself is cross-stage persisted in the generated code rather than the array element. Hence modifications to the array element will be observed when the code is run.
With value CSP it is not valid to use a cross-stage persisted variable as an *l-value* (e.g., the target of an assignment) because this would be equivalent to assignment to a literal. Therefore when type checking assignment statements (at levels greater than 0) the Metaphor compiler will reject this use of cross-stage persisted constants, as in the following code.

```csharp
bool x = false;
Code<void> y = <x = true;>; // error: cannot assign to x
```

### 3.3.5 Code persistence

In a typical multi-stage program a meta program will generate an object program and immediately run it. However in some scenarios it may be desirable to separate the execution of an object program from its generating meta program. Traditional compilers operate by saving a compiled program to disk rather than executing the program immediately, and in mobile code scenarios programs need to be sent across a network and executed on a remote machine. These situations require that code objects be persistent instead of purely in-memory objects. Given the depth and convenience of support for serialization and remote object access in object-oriented virtual machines (CLR and JVM) it seems natural to exploit these features for use in a multi-stage language to provide code persistence and an expanded operation of cross-stage persistence.

A code object in Metaphor can be persisted in one of two ways: in its un-compiled form as an AST or in a compiled form as an executable program file. Un-compiled code can be persisted using the standard serialization functionality provided by the CLR. Once persisted in this manner the original code object can be restored via deserialization. Compiled code can be persisted by using the `Save` method on code objects (like the `Run` method) that compiles the code object and saves an executable program file to disk. This method takes a string parameter for the name of the file to save to. For example, the following program will save the compiled object to the stand-alone program file `HelloWorld.exe`.

```csharp
CodeVoid helloWorld = <Console.WriteLine("Hello world!");>; helloWorld.Save("HelloWorld.exe");
```
Cross-stage persistence continues to work even when the meta and object programs execute in different processes or on different machines, albeit with some limitations. Metaphor is the only multi-stage language with this feature, which it calls *out-of-process* cross-stage persistence, and its implementation is discussed in the next section. Out-of-process cross-stage persistence is useful in mobile agent-like staged programs whereby the cross-stage persisted object behaves like a network end-point in the computation, similar to the locations from the nomadic data collector example in MetaKlaim [16].

Metaphor’s ability to persist code objects complicates its implementation of cross-stage persistence. A persisted object program may execute in a different process (or possibly machine) to its meta program, and its meta program may or may not still be running. In order to maintain cross-stage persistence, all cross-stage persisted objects in a saved object program need to be externalized from their actual instances in the meta program.

The problem of inter-process communication of objects is by no means specific to multi-stage programming, and the CLR (and also the JVM) already provides considerable support for solving this problem. In the CLR, objects are marshalled between processes either through serialization (the object is serialized in the source process and deserialized in the destination process) or remote object access (the destination process accesses the object in the source process via some inter-process communication mechanism, such as TCP sockets). The programmer specifies attributes on a class definition to tell the CLR if objects of that class can be marshalled and what marshalling method to use. The marshalling of objects is then performed by the CLR transparent to the programmer.

Code objects are saved in un-compiled form by using the CLR to serialize a code object. This process will automatically pick up and marshal any cross-stage persisted objects that may occur in the code object. When a code object is compiled and saved to disk, the result of marshalling all the cross-stage persisted objects is stored in the program file as an embedded resource. Initialization code that will un-marshal the objects when the saved program is executed is also generated and added to the program file. Cross-staged persisted objects are stored and accessed in global variables of
the object program. The following program compiles and saves a code object with a
cross-stage persisted object, Foo, which is to be marshalled by serialization.

```csharp
[Serializable] class Foo {
    void Bar();
}

static void Main() {
    Foo foo = new Foo();
    <|foo.Bar();|>.Save("tmp.exe");
}
```

The compiled program saved from the above code will be similar to the below code. The static constructor for the class Program deserializes the cross-stage persisted object and stores it in the static field csp001. The Run method contains the compiled code from the code object that was saved.

```csharp
class Program {
    static Program() {
        csp001 = /*deserialize CSP object*/;
    }

    static Foo csp001;

    static int Run() {
        return csp001.Bar();
    }
}
```

Cross-staged persisted objects that cannot be marshalled (i.e., neither by serialization
nor remote object access) will cause a run-time exception to be thrown when the
code object is saved. Note that a code object can still be Run if it contains non-mar-
shallable objects as this would use in-process cross-stage persistence.

By exploiting standard features (serialization and remote object access) found in ob-
ject-oriented virtual machines, Metaphor is able to enhance the potential applica-
tions of multi-stage programming.

### 3.3.6 Staged array access

A prominent application area of multi-stage programming is numerical algorithms.
These algorithms typically involve processing a large collection of numbers stored in
an array. One way to stage these programs is to fix the size of the array and specialize the code for that array size. For example a popular application in MetaOCaml is to specialize a Fourier transform function for a particular input size [30]. The simplest example of this basic idea is to stage the summation of an array. The code for the unstaged sum function is given below.

```csharp
double sum(double[] x)
{
    double result = 0.0;
    for(int i = 0; i < x.Length; i++)
        result += x[i];
    return result;
}
```

Staging this function with respect to the size of the input array looks like this.

```csharp
delegate B Func<A,B>(A x);

Code<Func<double[][],double>> sum(int n)
{
    return <delegate (double[] x) {
        if(x.Length >= n)
            return ~sum(n, <|x|>);
        else
            throw new IndexOutOfRangeException();
    }|>;
}

Code<double> sum(int n, Code<double[]> x)
{
    Code<double> result = <|0.0|>;
    for(int i = 0; i < n; i++)
        result = <|~result + (~x)[i]|>;
    return result;
}
```

In the staged version of the function, the for-loop is unrolled and all array accesses are made with literal indices. The code generated for $n = 3$ is shown below.

```csharp
delegate (double[] x) {
    if(x.Length >= 3) return 0.0+x[0]+x[1]+x[2];
    else throw new IndexOutOfRangeException();
}
```

The code generated above is the optimal solution to the problem in terms of source code but an inefficiency will exist when this code is JIT-compiled. The CLR performs
run-time bounds checks on array element accesses. In the generated code above, the if-condition explicitly performs a bounds check for all array element accesses inside the block. Therefore it is not necessary for the CLR to perform bounds checks on every access to the array in this block. Eliminating this bounds checking is important part of staging numerical algorithms.

To enable this ability in Metaphor there needs to be a way for the programmer to control at what stage bounds checking occurs. When bounds checking has been performed at an earlier stage the compiler can emit byte-code that skips the run-time bounds check. To support this, Metaphor provides a primitive class called StagedArray. The interface for this class is given below.

```csharp
delegate CodeVoid StagedArrayCont<T>(StagedArray<T> sa);
delegate Code<U> StagedArrayCont<T,U>(StagedArray<T> sa);

class StagedArray<T> : Code<T[]>
{
    static CodeVoid Insert(int n, StagedArrayCont<T> k);
    static Code<U> Insert<U>(int n, StagedArrayCont<T,U> k);

    int GetLength();

    CodeRef<T> GetElement(int i);

    CodeVoid FromArray(Code<T[]> array); 
}
```

The StagedArray class is a subtype of the code type meaning that staged array objects can be used anywhere that an appropriately typed code object could be used. Staged array objects are created using the static Insert methods. These methods take the fixed size of the array and a continuation that will build code using a staged array object. The GetElement method returns a code object to access an array element at a statically known index. This method performs bounds checking ahead of time: if its parameter, i, is larger than the size of the array then an exception is thrown. No bounds checking is done when the code object returned from this method is run. The FromArray method associates an (non-fixed size) array with this staged array. This method generates code that will (dynamically) check that the incoming array is at least as large as the specified fixed size of this staged array. If the check fails then an exception is thrown when the generated code is run.
The use of continuation-passing style (CPS) is analogous to the use of CPS to stage let binding in MetaOCaml [42].

Here is the staged sum example implemented using staged arrays.

```csharp
Code<Func<double[], double>> sum(int n)
{
    return <|delegate (double[] x) {
        return ~StagedArray<double>.Insert<double>(n,
            delegate (StagedArray<double> sa) {
            return <|~sa.FromArray(<|x|>); ~sum(sa)|>;
            });
        }|>;
}

Code<double> sum(StagedArray<double> x)
{
    Code<double> result = <|0.0|>
    for(int i = 0; i < x.GetLength(); i++)
        result = <|~result + ~x.GetElement(i)|>
    return result;
}
```

When staging array operations, the programmer must decide at what stage bounds checking shall occur: static checking or dynamic checking. Metaphor enables this control by introducing a single primitive class and through use of CPS. This improves the efficiency of generated code and the effect of staging.

### 3.3.7 Memory management

The CLR is a run-time environment with automatic memory management: i.e., garbage collection. The implementation of Metaphor must interact with this system as it dynamically generates code and cross-stage persists objects.

#### 3.3.7.1 Code

JIT compilation allows run-time code generators to avoid the difficulty of native code generation, but still enjoy the performance of native code execution. However, JIT compilers are often developed without multi-stage programming in mind and therefore they treat code as being permanent, static and constant. In a multi-stage environment, code is the product of run-time computation and must be handled
like regular data values by a run-time system. Code produced by a meta program must become eligible for garbage collection when it can no longer be used, otherwise a meta program that is continually generating code will run out of memory. In a multi-stage run-time system, dynamic code typically exists in three forms: as an AST of the multi-stage language, as byte-code generated by the language’s run construct, and as native code produced by the JIT compiler. Garbage collection of code in the first form is handled by the standard garbage collector of the run-time environment. Garbage collection of code in the third and possibly second form, depending on the implementation of the JIT, requires a run-time system aware of the dynamic nature of code.

The JIT compilers for OCaml [41] and the JVM cannot completely handle the garbage collection requirements of multi-stage languages. However the CLR does support a lightweight mode of run-time code generation (RTCG) that allows code to be garbage collected when it is no longer needed. The Metaphor compiler exploits this feature to create a fully garbage-collected, multi-stage run-time environment. The lightweight RTCG mode is approximately 6 times faster at generating code than the regular RTCG, however the code generated typically runs about 20% slower because the JIT compilation is done without optimizations. Nevertheless the lightweight mode will result in an improvement of the break-even point (see Section 3.4.2) for most applications.

3.3.7.2 Cross-stage persisted objects
The implementation of cross-stage persistence for primitive values (numbers, true/false, strings, null) is trivial as there are specific CIL instructions for loading these values. However cross-stage persisting a whole object is more complicated as an object cannot be embedded in CIL, unlike primitive values. Instead the compiled code must access the object at its location on the heap.

In a memory managed environment like the CLR it is not possible to emit a constant memory address for the object in the compiled code because the runtime’s garbage collector may relocate the object in memory. Pinning is a technique that can be used to tell the garbage collector not to relocate a particular object. Pinned objects could
be used to implement cross-stage persistence, but the CLR only allows pinning of an arbitrary object for the lifetime of a call stack frame. Therefore because in Metaphor it is possible to return a code object from a function this solution is not sufficient.

The solution adopted by Metaphor is to treat cross-stage persisted objects in a similar way to captured variables in a closure. When a code object is compiled its cross-staged persisted objects are identified and added to the closure environment of the compiled function generated for the code object. The run operation is responsible for constructing the closure with the cross-staged persisted objects needed by the generated code. To illustrate how this works consider the two semantically equivalent code fragments below. The former runs a code object containing a cross-stage persisted object. The latter forms a closure containing a captured variable. The run operation in the former code fragment will compile the code object similar to how the latter fragment is compiled, whereby the cross-staged persisted object is passed to the generated code via a closure environment.

```csharp
object obj = ...;
Func f = <|delegate () { return obj; }|>.Run();

object obj = ...;
Func f = delegate () { return obj; };
```

Passing cross-stage persisted objects through closures is superior to accessing them from a global store of objects as it does not introduce any complications to the garbage collection process. If a global store was used then special care would need to be taken to ensure cross-stage persisted objects are collected properly. With the closure approach, however, references to cross-stage persisted objects are released automatically when dynamically generated code is garbage collected.

### 3.4 Results

So far this section has illustrated the code for staging the power function in Metaphor. Metaphor is a complete language and as such can be used to stage a wide range of algorithms. To be a suitable candidate for staging, an algorithm must undergo a reduction in computational complexity when specialized with respect to a fixed in-
put (e.g., staging the power function removes a loop). Dynamic programming algorithms are a class of algorithm known to be suitable for staging [42].

Section 3.4.1 describes how dynamic programming problems can be staged using Metaphor. The implementations of these algorithms demonstrate the correctness and versatility of Metaphor as a multi-stage programming language. Section 3.4.2 discusses the performance characteristics of using Metaphor as a system for realizing run-time optimization. The source code for the examples presented in this section, as well as many more examples, can be read, compiled and executed as part of the Metaphor compiler package available for download on the Metaphor web site [32].

3.4.1 Example applications

Dynamic programming is a technique used to efficiently implement divide-and-conquer style solutions: i.e., where the solution to a problem can be expressed as a function of the solutions of sub-problems. Problems suitable for dynamic programming implementations are usually NP problems. Dynamic programming works by memoizing the results of intermediate calculations and then reusing them later in the computation instead of re-calculating them.

A typical example of a dynamic programming application is the 0/1 knapsack problem. In this problem there are \( n \) items, each item has a cost, \( c_i \), and a value, \( v_i \). The knapsack problem asks: what is the optimal subset of the \( n \) items such that the sum of their costs is less than \( C \) and the sum of their values is maximized? The code for the un-staged solution is given below.

```java
class Knapsack {
    int[] c, v;
    int[,] memo;

    int Solve(int C) {
        memo = new int[c.Length, C];
        return f(c.Length-1, C-1);
    }

    int f(int i, int j) {
        if (i < 0 || j < 0) return 0;
        if(memo[i,j] != 0) return memo[i,j];
```

int j1 = j - c[i];
int value = f(i-1, j);
if(j1 > 0) value = Math.Max(value, v[i] + f(i-1, j1));

memo[i,j] = value:
return value;
}
}

The client of this code would call the method Solve which sets up the memo table and forwards the call onto the internal method f. The two-dimensional array memo is used to memoize the results of the calls to f (line 18). The function checks, on line 13, if it has been called before with the same arguments. If it has then it returns the memoized value and avoids re-calculation.

To stage this function the input parameters c and C will be taken to be static. The generated code will not use a data structure, such as an array, as a memo table, but instead will memoize values directly into local variables. This optimization eliminates hit-testing (testing if a value exists in the memo table) and data structure lookup done on line 13 of the un-staged version. Furthermore the staged version will also eliminate the subtraction on line 15 and the if-test on line 17 as all this computation can be completely evaluated with the static input of the function. The code for the staged version is shown below.

class StagedKnapsack {
    int[] c;
    Code<int[]> v;
    Code<int>[][] memo;

    Code<int> Solve(int C) {
        memo = new Code<int>[c.Length, C];
        return f(c.Length-1, C-1);
    }

    Code<int> f(int i, int j) {
        if (i < 0 || j < 0) return <|0|>:
        if(memo[i,j] != null) return memo[i,j]:

        int j1 = j - c[i];
        Code<int> value = f(i-1, j):
        if(j1 >= 0)
            value = <|Math.Max(~value, (~v)[i] + ~f(i-1, j1))|>:
        value = <||
        int tmp = ~value:
    }
}
As described in Section 3.2.5 this code was created by annotation of the un-staged code. When using memoization in a staged program one must be careful, as examined in [42], to memoized the bound location of the memoized code object and not the code object itself. In MetaOCaml this concern is handled by programming in a monadic, continuation-passing style. Although an analogous solution could be constructed using Metaphor, concepts such as continuation passing and monads are alien to imperative, object-oriented programming. Instead, in Metaphor we abuse the pervasiveness of mutable state and the weakness of the multi-stage type system (Section 3.3.1) to implement code memoization in a direct, imperative style. Two caveats follow from this:

1. If one does not know what one is doing then it is possible to write a generator that produces invalid code (i.e., code containing unbound variables). However some consolation is that it is easier to know what one is doing than if continuation passing and monads were being employed.

2. Although the code presented in this example is correct (i.e., will not generate invalid code) the code may be rejected by a type checker of some future solution to the scope extrusion problem (Section 3.3.1). In other words, a futuristic type system invented to address scope extrusion may reject, if it is too conservative, the technique used here to implement code memoization.

Code memoization is performed on lines 20–22 of the above code. The code object to memoize is stored in \texttt{value}. This code is bound to a new local variable, \texttt{tmp}, a reference to \texttt{tmp} is inserted into the memo table, and \texttt{tmp}, the value of the memoized code, is returned as the result of the expression. A naive staged implementation may be tempted to use the following code as a replacement to lines 20–22 as this is the code used to perform memoization in the un-staged version.

\begin{verbatim}
20   -(memo[i,j] = <|tmp|>; tmp|); return value;
\end{verbatim}
However what is stored in the memo table in this case is the entire code expression contained in \texttt{value}, not a local variable reference. Hence whenever this is retrieved from the memo table a copy of this entire expression will be inserted in the generated code. This results in an exponential amount of code being generated and effectively means that the generated program recomputes sub-problems instead of memoizing their result.

Although the solution previously presented for the staged knapsack algorithm is completely valid, looking at the actual code it generates reveals that it is rather poor in a number of ways. The generated code contains quite a lot of inefficient statements similar to the ones listed below.

\begin{verbatim}
1 int tmp1 = 0;
2 int tmp2 = tmp1;
3 int tmp3 = v[0];
4 int tmp4 = Math.Max(0, v[1] + tmp2)
5 int tmp5 = Math.Max(tmp3, v[2] + 0)
\end{verbatim}

Lines 1–3 do not require the declaration of a new variable as their right-hand sides can be in-lined wherever the variable is used. Since all the values in this algorithm are non-negative, the call to \texttt{Max} on line 4 is redundant and can be reduced to just its second argument. The addition of zero in line 5 is also redundant and can be eliminated.

This is another instance of what was discussed in Section 3.2.5: sometimes the staged version of a function needs to be more complicated than its un-staged counterpart in order to fine-tune the code that it generates. A modified staged implementation is shown below that optimizes the above examples of inefficient code generation.

```csharp
class StagedKnapsack {
    int[] c;
    Code<int[]> v;
    Code<int> memo;

    Code<int> Solve(int C) {
        memo = new Code<int>[c.Length, C];
        Code<int> r = f(c.Length-1, C-1);
        if(r == null) return <0>;
        else return r;
    }
}
```
```csharp
Code<int> f(int i, int j) {
    if (i < 0 || j < 0) return null;

    if(memo[i,j] != null) return memo[i,j];
    int j1 = j - c[i];
    Code<int> value = f(i-1, j);
    if(j1 > 0) {
        Code<int> tmp = f(i-1, j1);
        if(value == null) {
            if(tmp == null) {
                value = <|(~v)[i]|>;
                memo[i,j] = value;
            } else {
                value = <|~value| + ~tmp|>
                value = <|int tmp = ~value; (memo[i,j] = <|tmp|>;) tmp|>
            }
        } else {
            if(tmp == null)
                value = <|Math.Max(0, ~v[i] + 0)|>
            else value = <|Math.Max(0, ~v[i] + ~tmp)|>
            value = <|int tmp = ~value; (memo[i,j] = <|tmp|>;) tmp|>
        }
        return value;
    }
}
```

This code exploits the fact that in Metaphor code objects are objects and hence can have null values. The function f returns null instead of the code literal <|0|> (line 14) so that its caller can detect when a zero literal is generated and optimize its code generation accordingly. The table below displays where in the code various optimizations are performed: i.e., on this line number, instead of generating the original code the optimized code is generated.

<table>
<thead>
<tr>
<th>Line no.</th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Math.Max(0, v[i] + 0)</td>
<td>v[i]</td>
</tr>
<tr>
<td>28</td>
<td>Math.Max(0, v[i] + tmp)</td>
<td>v[i] + tmp</td>
</tr>
<tr>
<td>36</td>
<td>Math.Max(tmp, v[i] + 0)</td>
<td>Math.Max(tmp, v[i])</td>
</tr>
</tbody>
</table>
Additionally on line 25, instead of creating a new temporary variable to alias the array element, the array element itself is stored as the memoized function value. Line 37 generates the code when no optimizations are applicable.

In Section 1 it was stated that the *raison d'être* of multi-stage programming is to allow programmers to write more abstract and general programs but eliminate the performance cost greater abstraction entails. Once a basic version of a staged algorithm, such as knapsack, has been written it is often very easy to generalize the algorithm without impacting at all on the performance of generated code. For example, instead of being fixed to work with integers this knapsack implementation could be generalized to work over any two types for the costs and values. Maximum and addition binary operations need to be performed on values so functions that build code to perform these operations would be given to the generic staged knapsack class. For example,

```csharp
delegate T BinOp<T>(T x, T y);

class StagedKnapsack<C,V> {
    Code<V>[] v;
    BinOp<Code<V>> max, add;
    // …
}
```

In the case where the type of values was `int`, the following ‘add’ function would be given to the algorithm and exactly the same generated code as in the non-generic case (with concrete integer addition) would result despite the increase generality of the generator.

```csharp
add = delegate (Code<int> x, Code<int> y) {
    return <int>[-x+y]:
};
```

Furthermore if the type of costs was not integers then a hash-table is needed as the memo table instead of an array. Access to a hash-table is much slower than an array, so the performance of the un-staged version would suffer greatly as a consequence of this generalization. However the generated code in the staged version would not suffer any performance penalty because the hash-table is only used at generation-time, not run-time.
3.4.2 Performance data

The goal of multi-stage programming is to eliminate the overhead of abstraction and so improve the performance of a program. There are various quantities that can be measured to assess the performance improvement of staging.

The simplest quantity is the speedup, which is the ratio between the run times of the un-staged and staged versions of the program. A staged program ought to run faster than its un-staged counterpart, therefore the speedup should be greater than one.

The speedup statistic, however, does not tell the whole story of the effect of staging because it neglects to take into account that the generation of the specialized program requires computation time. In some cases this time may be significant and outweigh the speedup improvement of the specialized program. A better statistic to measure the successfulness of staging is the break-even point. This is a value, $n$, such that running the un-staged program $n$ times takes the same amount of time as generating the specialized program and running it $n$ times. Informally it is the number of times the specialized version needs to be run to amortise the cost of generation and to start achieving a net performance gain. The break-even point, $n$, is related to the un-staged run-time, $U$, the staged run-time, $S$, and the generation time, $G$, by the following formula.

$$n = \frac{G}{U - S}$$

In Metaphor the generation time includes three distinct activities.

1. The construction of the code object. In the implementation, code objects are represented as regular objects that form an AST. Constructing a code object involves allocating a new object, calling its constructor and performing basic checks on its parameters (e.g., checking for null parameters).

2. The traversal of the code object AST and generation of CIL code. This is where the code object is compiled to CIL, the intermediate language of the CLR. The implementation of Metaphor uses the standard CLR library Reflection.Emit to produce the CIL and therefore the speed of this phase is
largely dependent on the performance of this library. Also occurring in this phase is variable renaming and checking of out of scope variables.

3. The JIT compilation of CIL into native machine code. This phase is performed by internal mechanisms of the CLR and the Metaphor compiler nor run-time system has no influence over its speed.

The table below summarizes the performance characteristics of various staged algorithms in Metaphor. The tests were performed on an Intel Pentium 4 processor at 3 GHz clockspeed running Microsoft Windows XP and CLR 2.0. The table lists the un-staged execution time ($U$), the staged execution time ($S$), the time to generate, compile and JIT-compile the staged program ($G$), the speedup ($s$) and the break even point ($n$). Each stage of the process (un-staged time, staged time, and generation time) was run continuously for 3 seconds and an average time found. All values are rounded to two significant digits. The example programs are as follows.

- **Modular power.** Modular exponentiation of integers with exponent $22013$. Exponent and modulus are known statically.

- **Polynomial eval.** Evaluation of a 16-degree even polynomial. Coefficients are known statically.

- **Dot product.** Dot product of vectors of length 30 with 50% zero elements. The first vector is known statically.

- **Binary search.** A generic binary search on an integer array of length 31. The array is known statically.

- **Knapsack.** The 0/1 knapsack problem with 30 items. The array costs and the cost limit are statically known.

- **Matrix mult. order.** The optimal matrix multiplication order problem for 18 matrices. The number of matrices is statically known.

- **LCS.** The least common subsequence problem for sequences of length 25 and 34. The lengths of the sequences are statically known.
<table>
<thead>
<tr>
<th>Program</th>
<th>U (µs)</th>
<th>S (µs)</th>
<th>G (µs)</th>
<th>s</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular power</td>
<td>0.39</td>
<td>0.33</td>
<td>600</td>
<td>1.1</td>
<td>12000</td>
</tr>
<tr>
<td>Polynomial eval.</td>
<td>0.081</td>
<td>0.0033</td>
<td>510</td>
<td>2.4</td>
<td>11000</td>
</tr>
<tr>
<td>Dot product</td>
<td>0.099</td>
<td>0.049</td>
<td>1000</td>
<td>2.0</td>
<td>20000</td>
</tr>
<tr>
<td>Binary search</td>
<td>0.097</td>
<td>0.0057</td>
<td>1300</td>
<td>17</td>
<td>14000</td>
</tr>
<tr>
<td>Knapsack</td>
<td>57</td>
<td>7.9</td>
<td>120000</td>
<td>7.2</td>
<td>2400</td>
</tr>
<tr>
<td>Matrix mult. order</td>
<td>36</td>
<td>12</td>
<td>140000</td>
<td>3.0</td>
<td>5800</td>
</tr>
<tr>
<td>LCS</td>
<td>43</td>
<td>17</td>
<td>230000</td>
<td>2.5</td>
<td>8700</td>
</tr>
</tbody>
</table>

Table 1. Metaphor performance results

The speedup result really has nothing to do with the Metaphor system. The speedup can be found by comparing the run time of the un-staged version with the run time of a hand-written staged version created without any use of Metaphor. Speedup therefore indicates how amenable an algorithm is to specialization on the CLR, regardless of how specialization is achieved.

The binary search example performs a generic binary search in the sense that it operates over arrays of any type and must be passed a comparison function to compare values of that type. The staged version is able to in-line this function call. The curiously high speedup for this example is caused by the elimination of the comparison function call.

The break-even point is the chief indicator of the practical usefulness of Metaphor. Although staging might bring about a huge speedup to a program, if it takes too long to generate the specialized code then the effect of the speedup is lost. This trade-off is indicated by the break-even point. No amount of engineering in the Metaphor system will alter the speedup value, therefore the break-even point is solely dependent on generation time. Any improvement in generation immediately translates into a similar improvement of break-even point.

The actual implementation of Metaphor is a prototype compiler. Work on the compiler focused on producing a working implementation of the design rather than an implementation with utmost efficiency. Code generation in the Metaphor compiler
is not optimal hence improvements could be made in the implementation of phases 1 and 2 above that would improve the efficiency of run-time code generation.

Performance figures for MetaOCaml using similar algorithms on similar hardware have been published \cite{11} and are summarized below.

<table>
<thead>
<tr>
<th>Program</th>
<th>$U$ (ms)</th>
<th>byte-code</th>
<th>native</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$S$ (ms)</td>
<td>$G$ (ms)</td>
</tr>
<tr>
<td>Knapsack</td>
<td>5.2</td>
<td>0.097</td>
<td>36</td>
</tr>
<tr>
<td>Matrix mult. order</td>
<td>3.6</td>
<td>0.14</td>
<td>66</td>
</tr>
<tr>
<td>LCS</td>
<td>5.5</td>
<td>0.12</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 2. MetaOCaml performance results

MetaOCaml programs compile to OCaml byte-code which is executed via interpretation. MetaOCaml offers two options for executing generated code: compilation to OCaml byte-code for subsequent interpretation, or compilation to C code for subsequent compilation by a standard C compiler and native code execution. Separate staged run times and generation times are given in the above table for each compilation strategy. Note that the unit of time is changed from microseconds to milliseconds.

Comparing the Metaphor results with these MetaOCaml results reveals that the CLR is approximately 100 times faster than the OCaml byte-code interpreter at running the unstaged program and 10 times faster at running the specialized program. This is to be expected since the CLR is JIT-compiled with aggressive optimizations whilst OCaml byte-code is interpreted. Removing a byte-code instruction from a JIT-compiled program does not translate to a proportional reduction in execution time as it would in an interpreted program, which could explain the approximately 10 times smaller speed-up shown in the CLR. Execution times of C-compiled MetaOCaml code are about twice as fast as CLR execution times. Time to compile to native code is about 50 times faster in Metaphor/CLR than it is in MetaOCaml, but this is to be expected since invoking an out-of-process C compiler is a massively slower operation than the in-process support for JIT-compiled run-time code generation provided by the CLR.
3.5 Conclusion

This chapter has given a basic overview of the Metaphor language and multi-stage programming. Specifically this chapter has highlighted some of the issues a language designer faced in applying multi-stage programming concepts from their previously exclusive domain of functional languages to an imperative language. Problems arose from the differing syntactic structure of C#, the pervasiveness of imperative references, and limitations and bonuses provided by the underlying virtual machine. All problems were solved through the tweaking of syntax or types, or by exploiting additional functionality provided by the CLR. Metaphor is a successful example of an imperative, object-oriented, multi-stage language.

In bringing multi-stage programming to a C#/Java-like language, Metaphor offers a new approach to object-oriented meta programming and run-time code generation. Existing approaches in this area are run-time code generation libraries, and static and dynamic meta-programming languages. Multi-stage programming can be considered a better approach because of its type safety, abstraction-building power and sound formal basis. Metaphor also enhances the universality of multi-stage programming as a programming discipline by applying it to a different—sometimes fundamentally—software development environment and introducing it to a wider audience of programmers.

With a sound understanding of a basic, imperative, multi-stage language in place it is now possible to explore the relationship between staging and type introspection in an object-oriented setting, which is the topic of the next chapter.
4 Polytypic multi-stage programming

This chapter details the central tenet of this work: that object-oriented-style type reflection can be combined with multi-stage programming in a type-safe manner to enable staged, polytypic programming.

The contributions of this chapter are

- an extension of multi-stage programming to polytypic programming, and
- a statically-typed type reflection system.

Part of this work has been published in [33].

4.1 Motivation

The popular mainstream languages C# and Java whilst being statically typed languages also employ a large degree of dynamic typing in their run-time systems. All objects in the CLR carry around type information with them at run-time. (Not all objects in the JVM carry accurate type information because of the erasure of generic types.) Given any object it is possible at run-time to query the type of that object. The type of the object is described at run-time by another object. A program can use this type object to query information about the type such as its name and its members (fields and methods). Members are similarly reified as objects at run-time that can be used to dynamically access the members on objects. The collection of classes and methods in the virtual machine that allow this functionality are loosely referred to as the reflection API.

Programs are written to use the reflection API typically because they want to do something that cannot otherwise be done because of limitations of the static type system. The reflection API complicates source code for the programmer and has poor performance at run-time so its use is discouraged outside of this absolutely essential
scenario. Two broad classes of applications that fall into this scenario are polytypic programs and domain-specific embedded languages.

Programs or algorithms that are induced by the structure of types are known as *polytypic* programs [26, 21]. Examples of polytypic programs include serializers, equality comparison and deep-copying. Polytypic programs in object-oriented languages use the reflection API to recursively enumerate the fields of classes.

A domain-specific embedded language (DSEL) is a little language represented as a data structure inside a general-purpose host language. The DSEL is tailored for expressing solutions in a particular problem domain. To execute a DSEL program one typically needs to interpret its data structure. If the DSEL can fully interoperate with types from the host language then the interpretation must necessarily use the reflection API.

Both of these applications could potentially reap the benefits of staging if there was a way to stage the reflection API. The programs can be divided into two stages: the first stage uses the reflection API to initially analyse the types, and the second stage performs the remainder of the work without using the reflection API. As is typical in multi-stage programming, the first stage executes once while the second stage executes many times, for example, by serializing many objects of the same type or by running a DSEL program more than once. Thus the performance overhead of the reflection API is eliminated (isolated to the first stage).

### 4.2 Introduction

The aim of this chapter is to incorporate a reflection API similar to C#’s or Java’s into the operational semantics of a multi-stage programming language. In doing so an effort must be made to preserve, as far as possible, the static guarantee of multi-stage programming: that a type-correct program can only generate type-correct programs. Chapter 3 discussed the design of an object-oriented, multi-stage language called Metaphor, which will be extended to support this aim.
4.2.1 The Reflection API

The CLR and JVM frameworks include a reflection API that allows a program to discover information about its types at run-time. Both APIs are similar in their essential structure, so although this chapter concretely discusses reflection in terms of the CLR’s API, the results equally apply to the JVM’s.

Using reflection, types can be reified as objects at run-time. Such an object has the type `Type` and provides information about the type it describes (e.g., name, base class, list of defined fields and methods). Querying a type object about its members (fields or methods) returns another object of type `Field` or `Method` to describe the member. Member objects then provide methods for dynamically accessing a field or invoking a method on a given target object.

The C# code below shows how to use reflection to dynamically access the field `i` on an object of type `A`. `typeof(A)` creates a type object describing the type `A`. `GetField` returns a field object for the field named `i` on `A`. Finally, `GetValue` gets the value of the field on a new instance of `A`.

```
class A { int i; int j; }
Type t = typeof(A);
Field f = t.GetField("i");
int val = (int) f.GetValue(new A());
```

The reflection operation `GetValue` is dynamically typed because it passes data as the type `object`—which holds no static information about the types of the actual objects being passed. This design choice was made by the designers of the reflection API because there is no way to statically type this operation in the standard type system of C#.

The type checking in this code can be considered to occur in two phases, see figure below. The first phase occurs at compile-time when the source file is compiled by the C# compiler. This performs some static checking on the program such as, ensuring the type `A` exists, `Type` has a `GetField` method and `Field` has a `GetValue` method. The second phase occurs at run-time when the program is actually executing. It is in this phase that the essential function of the reflection API is verified through dynamic tests such as testing the type `A` has a field `i` of type `int` and the expression
new A() returns an object that is a subtype of A. If one of these conditions fails then a run-time exception is thrown. In this example, because of its simplicity, it would be possible to statically verify these conditions. Projects such as Generic Haskell [22] show that it is possible for a static type system to constrain dynamic access to data structures, hence eliminating the possibility of all reflection-induced run-time errors; however such approaches are likely infeasible to employ in C#.

4.2.2 Staging reflection

Let us consider a staged version of the above code. In the staged version the field is known ahead of time (stage 0) but the instance object is not (stage 1). Therefore we want to generate code for a field access on an object where the field to access is determined dynamically. This is precisely the task a compiler performs at compile-time: by analysing source code the compiler has full information about the field to access, but the object to work on does not exist yet as the program has not been run. The difference here however arises in that the programmer wishes to write a program that will do the same at run-time. The below code fragment is a first attempt at how this could be done.

```csharp
Type t = typeof(A);
Field f = t.GetField("i");
Code<int> val = (Code<int>) f.GetValue(<|new A()|>);
```

The first two lines that construct the reflection objects are the same as from the un-staged case. The last line performs the staged field access using an overload of the GetValue method that accepts and returns code objects instead of value objects. The method signature for this overload is

```csharp
class Field {
    Code<object> GetValue(Code<object> codeObj);
}
```
The un-staged GetValue method relied on sub-typing to provide polymorphism: i.e., the object parameter and return types allowed any type of object to be passed and returned. It is assumed that the code type is co-variant so that the staged GetValue method can continue to exploit polymorphism through sub-typing. A detailed discussion on the ramifications of code type co-variance and polymorphism of code objects is given in Section 4.3.5.

When the above example code executes the variable val will contain the code object \( \langle |new A().i| \rangle \). As can be seen the use of the reflection API has been eliminated in the generated code. Instead of using reflection to access the field, the field is directly accessed in the code object. Hence when this code object is Run it will be operationally equivalent to accessing the field with statically compiled code. This is precisely the motivation for integrating the reflection and staging abilities of the language.

### 4.2.3 Type checking reflection

Now consider the question: when are the conditions of use of the reflection API tested? In the staged version there are three possible phases in which type checking may occur.

As before there is compile-time which is when the standard language constructs of C# and the multi-stage constructs of Metaphor are verified. Next there is generation-time which is the first stage of execution of the staged program during which later stage code is generated. Finally there is the run-time of the generated code, or the second stage of execution of the staged program.

Clearly the earlier the checking can be done, the greater the static type safety of the program and therefore the more chance that program errors will be identified before they occur. Consequently checking at run-time is the worst option. With this option ill-typed code can be generated and the error not exposed until the code object is Run. The program below generates (erroneous) code for accessing the field on a new
instance of B, where B is not a subtype of A. In the un-staged case this error would be exposed on the invocation of GetValue. In the staged case if run-time checking were employed, this error would not be exposed until the Run method was called on val or some other code object that incorporated val. In general, the invocation of Run would be at a completely different location in the program and hence difficult to correlate the error with its true cause. The Run method would detect this error either through an explicit type-checking phase that is part of the code object compilation process, or by produced invalid byte-code that fails to execute.

```csharp
Code<int> val = (Code<int>) f.GetValue(<|new B()|>);
// ...
val.Run(); // will throw exception
```

Checking at generation-time will mean that although reflection errors may still occur during the execution of the program (i.e., not at compile-time), the error will occur at a stage before the code is executed. A requirement of this approach is that code objects must carry run-time type information, which is a reasonable requirement in an object-oriented language as code objects ought to be self-describing and capable of undergoing reflection themselves. In this approach the GetValue method would examine its code object argument and ensure that its dynamic type is compatible with the field to be accessed. Similarly down-casting objects to code types (such as the cast to Code<int> in this running example) will also involve an inspection of the dynamic type to ensure it is compatible with the target code type. If the types are incompatible then a dynamic typing exception will be thrown. So, in the previous example of bad reflection the exception would be thrown in the first line (instead of the last). GetValue would detect that the dynamic type of its argument, Code<B>, is not compatible with the declaring type, A, of the field referred to by f. It would therefore throw an exception, which is analogous to what the un-staged version of GetValue would do in the un-staged case.

```csharp
// first line throws exception
Code<int> val = (Code<int>) f.GetValue(<|new B()|>);
// ...
val.Run();
```

Checking at compile-time means that the compiler statically checks all uses of reflection and no reflection-based errors are possible when the program executes. This is
the most desirable solution as it gives the highest degree of static type safety. With this option the bad reflection code shown above would not even compile. In a multi-stage environment it is desirable to prevent this error at compile-time to ensure the static safety of multi-stage programming. By preventing this error at an earlier stage makes staged programs more reliable and easier to debug. However it is not obvious how such a type-checking algorithm could be devised to work with the syntax of the above code as written. To achieve this level of type safety it is necessary to extend the language with richer types and/or new constructs.

4.2.4 Plan

The remainder of this chapter discusses different approaches for integrating multi-stage programming with the reflection API and what degree of type safety is reached in the process.

Section 4.3 gives a motivating example of a polytypic algorithm that will guide and evaluate the development of various staged, reflective programming designs presented in subsequent subsections.

Sections 4.4, 4.5 and 4.6 investigate three different designs for staged reflective programming, each with a different degree of type safety. In Section 4.4, type checking of reflection operations occurs at generation-time. Section 4.5 introduces a new static type system for reflection that enables reflection operations to be checked at compile-time. Finally Section 4.6 combines results from the previous two subsections to develop a hybrid system where some type checking is done at compile-time and some at generation-time. The hybrid system is a practical solution that strikes a balance between type system complexity and programming simplicity.

4.3 Polytypic algorithms

To guide the development of the staged reflection capabilities of Metaphor we will use object serialization as the archetypal example of a polytypic program which uses reflection. Object serialization has three key elements of polytypic programming: ac-
cessing fields indirectly through reflection, enumerating the fields of a type, and conditional logic controlled by type comparisons.

### 4.3.1 Basic serialization

Serialization is the process of taking an object graph and converting it into a stream of bits. An object is typically serialized by recursively serializing each of the object’s fields. Serialization is a fairly regular algorithm and it is possible, with the help of the reflection API, to write a polytypic program that will serialize any type.

```csharp
void serialize(object obj, BinaryWriter b)
{
    Type typ = obj.GetType();
    if(typ == typeof(int)) b.Write((int)obj);
    else if(type == typeof(float)) b.Write((float)obj);
    // ...
    else
        foreach(Field fld in typ.GetFields())
            serialize(fld.GetValue(obj), b);
}
```

The method `serialize` takes an object to serialize and a `BinaryWriter` to write its serialized output to. A `BinaryWriter` object contains methods for writing primitive values (integers, floating-point numbers, etc.) to a binary stream. This method works by using reflection to discover the type of the object and testing whether it is a primitive type which can be directly written with the `BinaryWriter`. If it is not a primitive type then the algorithm uses reflection to access every field on the object and recursively invoke itself.

The three key requirements of expressing a serialization algorithm can be seen in this code: `GetValue` to dynamically access a field of an object, the `foreach`-loop and the `GetField` method to enumerate the fields of an object/type and the comparison of `Type` objects to control recursion.

The simple algorithm presented here neglects to handle many practical aspects of object-oriented programming such as cyclic object graphs, null references and subtyping. This serializer algorithm is capable of handling what the CLR terms as *value types*. Unlike object or reference types, value types are accessed by value not reference.
Because the serializer works by recursively serializing all the fields of a type, the fields of the value type must also be value types. This serialization scenario is very simple because neither the type nor the object can be recursive. This restriction may seem severe but this simple algorithm is sufficient to implement XML serialization as the strict tree structure of XML precludes type or object recursion.

The staged version of this algorithm will accept the type of object to serialize as static input and the object itself as dynamic input. The generated code must not use any dynamic actions such as reflection or run-time type tests/casts. The validity of a serializer generator is assessed like so. For the types

```java
class Foo {
    int x;
    Bar y;
}

class Bar {
    int z;
}
```

the generator must produce the code

```java
void serialize(Foo obj, BinaryWriter b) {
    b.Write(obj.x);
    b.Write(obj.y.z);
}
```

Staging the serialization algorithm like this has the inadvertent side-effect of determining the object’s type at generation-time rather than run-time: e.g., if the field `y` of `Foo` contained an object that was an instance of some subclass of `Bar` that defined additional fields, then these additional fields would not be serialized by this staged serializer. This discrepancy between the un-staged and staged versions will be ignored for now but will be re-visited later in the Section 4.7.3.

### 4.3.2 Recursive serialization

A more advanced serializer supporting objects and reference types will now be considered. Because objects are accessed indirectly through a pointer or reference, it is possible for reference types to be recursive, e.g.,
class Foo {
    Foo foo;
}

and for objects themselves to be recursive, e.g.,

    Foo foo = new Foo();
    foo.foo = foo;

This presents three challenges to a serialization algorithm:

1. An object reference may be null and not refer to an object at all.

2. The graph of objects may be cyclic and hence could lead to infinite recursion in the serializer.

3. The same object may appear multiple times in an object graph and so its identity needs to be preserved.

The code for the un-staged, recursive serialization algorithm

    void serialize(BinaryWriter b, object obj, List<object> visited) {
        Type typ = obj.GetType();
        if(typ == typeof(int)) b.Write((int)obj);
        // else more primitive type cases
        // else value type case
        else if(obj == null) b.Write(0);
        else {
            int index = visited.IndexOf(obj);
            if(index != -1) b.Write(index + 1);
            else {
                b.Write(-1);
                visited.Add(obj);
                foreach(Field f in typ.GetFields())
                    serialize(b, f.GetValue(obj), visited);
            }
        }
    }

These problems are overcome in an un-staged serializer by maintaining a collection of 'visited' objects during the serialization process. In this approach primitive and value types are serialized as before. Reference types are serialized by:
1. If the reference is null output zero.

2. If the object is in the visited collection then output the index of the object in this collection.

3. Otherwise output –1, add the object to the visited collection, and serialize the object.

When this algorithm is staged an interesting thing occurs: the recursion in the un-staged version is split between both stages of the staged version. The first stage operates on types and is recursive because the types are recursive. The second stage operates on objects and is recursive because the objects are recursive. The second stage recursion (i.e., recursion in the objects) can be dealt with in the same manner as in the un-staged case by maintaining a collection of visited objects. In a sense this part of the un-staged implementation is simply translated into the second staged of the staged implementation.

The first stage recursion (i.e., recursion in types) is more subtle to handle. The type `Foo`, below, is recursive because it directly references itself and also because it indirectly references itself through the type `Bar`. If a staged serializer, like the one hypothesized in the previous subsection, that generates one monolithic function were applied to this type, the generator would not terminate because of this recursion.

```csharp
class Foo {
    Foo foo;
    Bar bar;
}

class Bar {
    Foo foo;
}
```

To generate a serializer for this type, a separate serialization function must be generated for each type, and the generator should not regenerate the same function for the same type. A correct serializer generator should produce the code

```csharp
void serializeFoo(BinaryWriter b, Foo obj, List<object> visited) {
    if(obj == null) b.Write(0);
    else {
```
for these types. The recursion between the types corresponds to the recursion between the two generated functions.

4.3.3 Recursive function generation

In MetaOCaml this kind of multi-stage recursion can be expressed using a staged fixed-point combinator [42]. Metaphor takes an approach more congruous to its imperative nature. Firstly, anonymous functions do not need to be anonymous: they can be named so to support recursion without a fixed-point combinator. Secondly, the fact that functions are not first-class values in an object-oriented language is exploited to allow functions to be safely scope extruded (see Section 3.3.1).

The code for writing a recursive-type staged serializer in Metaphor encounters similar problems to staged dynamic programming (see Section 3.4.1). To prevent (infinite) code explosion it is necessary to memoize serialization functions when they are first generated and reuse them later on if they are needed. To do this a lookup table is needed that maps types to the code objects of their generated serializers. What this
lookup table (or memo table or map function) looks like depends on the implementation of the serializer and specifically on the type given to the code objects. Different implementation approaches have different requirements for the lookup table, so further details on memoization are deferred to implementation-specific areas (Sections 4.4.2 and 4.6.2).

Another problem with staging this recursive algorithm is the need to generate recursive functions. The functions that serialize specific types must be generated all at once; they cannot be generated incrementally because they are (in the general case) mutually recursive. In C# it is not possible to write recursive anonymous methods, nor in the Metaphor presented in Section 3 was it possible to generate recursive functions. Recursive anonymous methods can be emulated in C# using delegates, for example

```csharp
Func<int, int> f = null;
f = delegate (int n) {
    return n == 0 ? 1 : n * f(n - 1);
};
```

Here the recursive call is made indirectly through the delegate `f`, which at the CLR level is a different calling mechanism than if the function was written recursively as a regular class method. Delegate calls are slower than direct method calls because of the way the CLR is engineered. The recursive, staged serializer could use the same technique but would entail unnecessary work being done in the generated code. The generated serializers would deviate from the desired, target code given in Section 4.3.2 by the need to retrieve the appropriate delegate from the lookup table and make the recursive call through that delegate. Because algorithms are usually staged to gain maximum performance, this performance overhead in the generated serializers using this technique for recursive function generation is considered unacceptable.

To solve the problem it is necessary to generate functions that are directly (without delegates) mutually recursive. This solution is enabled through the introduction of a named anonymous methods, or perhaps a better term is named in-line function expressions. An optional name of a function is specified between the delegate keyword and parameter list (in the example below, `f`). Naming anonymous methods chief-
ly enables the ability to express recursion, e.g., the factorial function written with a named, anonymous method.

```csharp
Func<int, int> factorial = delegate f(int n)
{
    if (n == 0) return 1;
    else return n * f(n - 1);
};
```

In the staged case the name of the function can appear in an escape/quote pattern just like one of its parameters could. For example

```csharp
class C
{
    static Code<Func<int, int>> factorial = null;

    static Code<Func<int, int>> f1()
    {
        return <|delegate f(int n) {
            -(factorial = <|f|>);
            return n == 0 ? 1 : (~f2())(n);
        }|>;
    }

    static Code<Func<int, int>> f2()
    {
        return <|delegate g(int x) {
            return x * (~factorial)(x - 1);
        }|>;
    }
}
```

The function `f1` generates code for a factorial function. The first line inside the factorial function assigns (at stage 0) a code object for the function being generated (`f`) to the global variable `factorial`. Code responsible for generating the body of `f` can use this code object to refer to `f` and invoke it in a recursive manner. In this example the function `f2` generates a function, `g`, that is mutually recursive with `f` and where the recursive link to `f` is created through the code object stored in `factorial`. When invoked `f1` will generate these two functions.

```csharp
delegate f(int n) {
    return n == 0 ? 1 : g(n);
}
delegate g(int x) {
    return x * f(x - 1);
}
```
The function \( f \) is the principal value of the code object returned by \( f1 \) and \( g \) is only internally referenced inside this code object.

The problems of run unsafety and scope extrusion (see Section 3.3.1) have an impact on the quotation of named anonymous functions. The problem with run unsafety in this scenario becomes

```c
<|delegate f() {
   ~{|f|.Run();}
}|>
```

This code is trying to compile the function being generated before its generation is complete. This is clearly an error but is not prevented by the Metaphor type system as it does not enforce the safety of \( \text{Run} \).

When an anonymous function name is quoted it is not considered a free variable as if the name of a local variable or method parameter were quoted. The code object \(<|f|>\) becomes the function definition itself rather than a variable that refers to the function. This code object can be spliced into many places but only one compiled function will be produced when code is \( \text{Run} \) as all splice sites will share the same implementation. This behaviour is possible because the anonymous function name is like a method name in an object-oriented program. Method names are not first-class values—they are not mutable objects—therefore function names can be treated differently to variable names in the semantics of staging.

The code object \(<|f|>\) can still be open (i.e., contain free variables) if the function, \( f \), being quoted captures variables from an outer scope. (The variables must be from the same level as the function so as to form a true variable capturing closure instead of simple cross-stage persistence.) When this is the case a scope extrusion error could possibly occur, and so the programmer must be mindful of this since scope extrusion is not prevented by the type system. However because of the nature of imperative programming, although theoretically possible, mutually-recursive, variable capturing functions almost never occur in practice. They certainly never occur in any code fragment or sample application considered in this work. Therefore since generated functions never capture free variables their code objects are closed and hence do not run the risk of scope extrusion.
4.3.4 Polytypic method signatures

Polytypic methods like serialize can accept a code object of any underlying type as an actual parameter. The question is: how should this kind of polymorphism be encoded in the signature of the method? There are three candidates for the method signature:

1. `CodeVoid serialize(Code<object> obj):`


3. `CodeVoid serialize<T>(Code<T> obj):`

To emphasize the differences between these method signatures more precisely, they can be written as types is a pseudo, typed, functional language.


2. `(∃T.Code T) → CodeVoid`

3. `∀T.(Code T → CodeVoid)`

The type `object` (the super-type of all types) is equivalent to the existential type `∃T.T`, (i.e., the sum of all types). The syntax for the type `Code<?>` is borrowed from Java and intuitively specifies that the underlying type is unknown. In the functional type language, an existential type is used to express the hidden type. Finally the last candidate uses generics or parametric polymorphism.

A trend evident in these three signatures is the progression of the quantifier (∃ or ∀) further and further away from the site where the quantified variable is used. In the first signature the quantifier occurs in the same place as the quantified variable. In the second signature the quantifier is moved outside the `Code` type constructor and in the third it is moved outside the function arrow.

This typing issue has an equivalence in un-staged programming too. The un-staged serializer could be written with either signature below, which are variations on signatures 1 and 3 above (with the elimination of the `Code` type constructor, signatures 1 and 2 collapse into the one signature).
void serialize(object obj):
void serialize<T>(T obj):

Previously the first signature was chosen in the serializer code example. The second signature is rejected for good reason. Consider writing the (un-staged) serializer method with this signature.

void serialize<T>(T obj, BinaryWriter b)
{
    Type typ = obj.GetType();
    if (typ == typeof(int)) b.Write((int)obj);
    else if(type == typeof(float)) b.Write((float)obj);
    // ...
    else
    foreach(Field fld in typ.GetFields())
        serialize<?>(fld.GetValue(obj), b); // error here
}

Firstly this signature does not make the method any easier or safer to implement as the added type parameter is practically useless. Secondly, and what is far more problematic, there is no way to statically make the recursive call to serialize. Like all generic method calls, the call site requires a type argument to be specified. The proper type to pass as the argument is the type of the field being accessed, but this type never manifests itself in the program text and therefore the call cannot be made. The only way around this problem is to use reflection to invoke the method but this is clearly undesirable.

Another scenario where this polymorphism dilemma arises in un-staged programming is in a method that consumes a generic type but does not care about its type arguments. An example is a method that computes the size of a data structure, e.g., the length of a list.

int length(List<object> list);
int length(List<?>> list);
int length<T>(List<T> list):

The first signature is practically useless since neither Java nor C# support co-variance of generic types. Without co-variance List<string> is not a sub-type of List<object>, therefore the length method is not polymorphic over all lists. Signature 2 is how the method would be written in Java since it supports wildcard parameters and
superfluous type parameters are discouraged. Signature 3 is how the method would be written in C# as it lacks wildcards. Wildcards are simple to implement in Java since generics is implemented via erasure [4] so all generic types effectively have wildcards at the byte-code level. Wildcards would be harder to implement in C# because the CLR preserves full information about generic types at run-time.

Returning to the problem of the method signature of the staged serializer method, the first signature is only usable if the code type is co-variant. A generic type C is co-variant if C<T> is a subtype of C<U> when T is a subtype of U. Similarly contra-vari ance means that C<T> is a subtype of C<U> when U is a subtype of T. Neither Java nor C# support the notions of co- and contra-variance in their type systems. (CIL, the intermediate language of the CLR, does but this feature is not exploited by C#.) Metaphor does not support co- and contra-variance in general either, but it does allow co-variance on the code type as this is a primitive type of the language. It is possible to support code type co-variance in Metaphor because code objects are immutable: i.e., there are no methods that mutate the state of a code object. Mutability is the reason why co-variance does not hold on other generic types such as List (e.g., up-casting a List<int> to List<object> allows a string to be added to the list). It was decided to allow code type co-variance in Metaphor because it is necessary to make staging as orthogonal an extension as possible. A caller of an un-staged method may exploit parameter sub-typing when making its call. Therefore if this code were staged then code type co-variance is necessary for the use of sub-typing in the underlying algorithm. Further supporting code type co-variance is non-intrusive in that it does not change the meaning of any program written without it. Hence with Metaphor’s support of code type co-variance signature 1 is a viable candidate for a serializer method.

Metaphor does not support wildcard type parameters so it would seem signature 2 is non-viable. However with co-variance, since every type that is a sub-type of Code<object> is also a sub-type of Code<?,> the two types are equivalent; hence signatures 1 and 2 are actually the same. However since not all types in Metaphor are co-variant this correspondence only goes so far (e.g., Code<List<object>> is not equivalent to Code<List<?,>> as Code<List<int>> is a sub-type of the latter but not the former).
Nevertheless by the simplicity of the serializer example, signature 2 can be permanently discounted. A more complex example (i.e., one where the type parameter occurs inside an invariant type constructor) arises in the development of a static type system for the reflection API. Section 4.5.1 discusses the shortcomings of wildcards as a solution to this problem and proposes existential types as a solution which are a generalization of wildcards.

Signature 3 does not require any additional language features (other than generics).

4.3.5 Code type co-variance

The following Metaphor code exploits code type co-variance.

```csharp
Code<string> x = <|"hello"|>; serialize(x);
```

Here the type `Code<string>` is a subtype of `Code<object>`, hence the value of `x` can be passed as an actual parameter of `serialize` (using signature 1). If `x` was the code of a value type (instead of a reference type, e.g., `Code<int>`) then an implicit staged boxing operation needs to be performed at the call site to `serialize`. In C# using a value type in a method call or assignment where the expected type is `object` causes the compiler to insert an implicit boxing operation. This implicit boxing needs to be preserved in the staged sense also. If the code of a value type is passed or assigned where the type `Code<object>` is expected, then the compiler implicitly emits a staged boxing operation whereby the code object will be modified to indicate that it should compile to a boxed value type.

In the following code `x` is the code for the integer literal 1. If this code object were compiled (e.g., `x.Run()`) it would result in byte-code that loads the unboxed integer 1. At the call site of `f` a new code object is implicitly created and passed to `f`, instead of `x`. This code object will compile to a boxed integer. Therefore when the code object into which `y` is spliced is compiled, a boxed integer will be assigned to `z`, not an unboxed one.

```csharp
CodeVoid f(Code<object> y) {
```
Therefore just as subtyping obligates an un-staged compiler to insert implicit boxing operations, so to does the co-variance subtyping of the Code type force a staged compiler to insert implicit staged boxing operations.

### 4.4 Dynamically-typed approach

This section will explore writing staged reflective programs where the type checking of reflective operations occurs dynamically (i.e., at generation-time).

Key features of this approach are

- Code objects carry type information about the code they contain. This type information is used by run-time checks that occur whenever a code object is down-cast.

- The types from the reflection API are extended with these methods.

```csharp
class Type {
    Code<object> Cast(Code<object> obj);
}
class Field {
    Code<object> GetValue(Code<object> obj);
    CodeVoid SetValue(Code<object> obj, Code<object> val);
}
class Method {
    Code<object> Invoke(Code<object> obj, Code<object>[] args);
    CodeVoid InvokeVoid(Code<object> obj, Code<object>[] args);
}
class Constructor {
    Code<object> Invoke(Code<object>[] args);
}
```
The **Cast** method builds a code object for a type cast. At run-time it checks that the dynamic type of `obj` can be cast to the type represented by the method call's receiver. Typically this will be a down-cast in which case the return value will be a code object that type casts `obj`. If an up-cast is inferred then the function behaves as the identity. If the two types are unrelated in the class hierarchy then a run-time type exception is thrown.

The **GetValue** method, as was previously discussed, builds a code object to access a field. If the field described by the receiver is not compatible with the dynamic type of `obj` then a run-time type exception is thrown. The **SetValue** method builds a code object that assigns a value to a field. Similarly if the field does not exist on the dynamic type of `obj` or if the dynamic type of `val` is not compatible with the field type, then a run-time exception is thrown. It is not possible to unify these two methods into a single method that returns an assignable code type, such as

```
CodeRef<object> Value(Code<object> obj);
```

With such a method, a field could be assigned to by splicing the returned `CodeRef` object as the right-hand side of an assignment expression. This however is invalid:

```csharp
class C {
    int f;
}
CodeRef<object> cf = fieldof(C.f).Value(<|new C()|>);
CodeVoid wrong = <|~cf = "x"; |>
```

This code uses the errant `Value` method to construct `cf`, an assignable code object that accesses the field `f` on a new instance of `C`: i.e., `<|new C().f|>`. This code object has `object` as its underlying type even though the type of the field is `int`, thus it is possible to assign to it any object, such as a string. This is a dynamic typing error since `f` is an integer field. However the building of the assignment code object will not check for this error because all code quotation is statically type-checked. This problem is avoided by having separate get and set methods to dynamically stage field access. Separate methods allow dynamic type-checking to be performed in only re-

---

1 The expression `fieldof(C.f)` gets the reflection field object for the field `f` in class `C`. Cf. the `typeof` operator from standard C#.
flection classes and when it is needed, instead of becoming part of the statically-typed quotation mechanism.

The Invoke and InvokeVoid methods build code objects that invoke a method. The InvokeVoid method can be called if and only if the receiver object describes a void-returning method. These methods check that the method described by the receiver is compatible with the dynamic type of obj. Additionally they also check compatibility of the actual parameters, args. For pass-by-value parameters the underlying type of the code object must be a subtype of the formal parameter type. For pass-by-reference parameters the code object must have a CodeRef type and its underlying type must be equal to the formal parameter type.

The Invoke method on the Constructor class works similarly as on Method and is used to create new instances of classes.

Finally local variables with dynamically determined types can be declared using the Local method on Type.

```csharp
delegate CodeVoid LocalSet(Code<object> val);
delagate CodeVoid LocalCont(Code<object> get, LocalSet set);

class Type {
    CodeVoid Local(LocalCont k);
}
```

The Local method builds code that will declare a local variable of the type described by the receiver object. The method is passed a continuation (similar to the continuation-passing style used in Section 3.3.6) that will build a code object that uses the declared local variable. The continuation accepts two arguments: a code object used to access the local variable and a function used to assign to the local variable. Similar to the case of staging reflective field access, the continuation cannot accept a code object of type CodeRef to refer to the local variable as the CodeRef type is invariant. The set function passed to the continuation is responsible for conducting run-time tests to ensure the validity of the assignment. As before this limits run-time type tests to the staged reflection API and avoids the need for dynamic type checking by the quotation operator. If the run-time test fails in set then an exception is thrown.
which, if not handled by the continuation function, would usually propagate to the caller of Local.

Using this approach to staged reflection it is not possible to use a reflection type object to declare the type of a method formal parameter. Furthermore this approach fails to make use of the CodeRef type for assignable code objects. This prevents reflective programs from consuming external interfaces that involve the CodeRef type. These limitations and the tediousness of introducing local variables are seen as technical shortcomings of the generation-time checking approach to staged reflection.

4.4.1 Basic serialization

As discussed in Section 4.3.1 there are two conceivable signatures for a staged serialization method.

```csharp
CodeVoid serialize(Code<object> obj);

CodeVoid serialize<T>(Code<T> obj);
```

The second signature is rejected for the same reason it was rejected for the un-staged serializer in Section 4.3.4: it is not possible to make the recursive call without reflection.

A first attempt at writing a staged serializer using dynamically-typed reflection is

```csharp
CodeVoid serialize(Code<object> obj, Code<BinaryWriter> b)
{
    Type typ = obj.GetType().GetGenericArguments()[0];
    if(typ == typeof(int))
        return <|--b.Write(~(Code<int>)obj);|>:
    // ...
    else
    {
        CodeVoid result = < | ; | >;
        foreach(Field fld in typ.GetFields())
            result := serialize(fld.GetValue(obj), b);
        return result;
    }
}
```

The code was written by annotating the un-staged serializer from Section 4.3.1 using the technique described in Section 3.2.5. The run-time type of `obj` will be `Code<T>`
for some type $T$ (a subtype of object). The first line of the function extracts the type argument of the run-time type of obj (i.e., the $T$) and stores it in the variable typ. This type object is then used to drive the algorithm as in the un-staged case. The operator $:=\,$ is used to append code to a code object of statements (i.e., $l := r$ is syntactic sugar for $l = <\neg l; \neg r; >$). Using the following anonymous function expression this method will produce the desired target code stated in Section 4.3.1.

```csharp
delegate void Serializer<T>(T obj, BinaryWriter b);

Code<Serializer<Foo>> s = <|delegate (Foo obj, BinaryWriter b)
{
    ~serialize(<|obj|>, <|b|>);
}|>

// s ->
//    delegate (Foo obj, BinaryWriter b)
// {
//        b.Write(obj.x);
//        b.Write(obj.y.z);
//    }
```

The weakness of this approach is its use of dynamically-typed reflection. The recursive call to GetValue is valid because typ is the inner type of obj, fld is a field of typ and fld is being accessed on obj. However the compiler is unable to make this inference, making the programmer’s code more susceptible to run-time errors. A type system that allows static checking of reflection operations will reduce the frequency of programming errors in reflective code.

### 4.4.2 Recursive serialization

This section reports on the implementation of a staged serializer capable of handling recursive types using the dynamically-typed approach to reflection.

#### 4.4.2.1 The ‘typewith’ construct

The algorithm for a staged serializer supporting recursive types (see Section 4.3.1) requires that each type has its own serializer function, and that this function accepts a parameter with the exact static type of the object to be serialized. In other words, the function generated to serialize a type $\text{Foo}$ must accept the object to serialize through a parameter of type $\text{Foo}$ and not of type $\text{object}$. 
As previously mentioned in Section 4.4.1, a limitation of the dynamically-typed approach to reflection as presented so far is its inability to use reflection objects in the formal parameter types of dynamically generated functions. In other words, it is not possible to generate a function where the types of its formal parameters are determined at run-time. This is a direct obstacle to the requirements of a staged recursive serializer.

Thus to overcome this problem it is necessary to add a new construct to the language to enable generation of static parameter types from reflection type objects. This construct is called typewith and effectively un-reifies a reflection type object into an actual type. The typewith construct is useful in situations where a type is identified at run-time via reflection and this type needs to be used like a static type when building code.

```csharp
delegate void Action<T>(T x);
Type typ = ...;
typewith<T>(typ) {
    Code<Action<T>> code = <[delegate (T obj) {
        Console.WriteLine(obj);
    }]>;
}
```

In the typewith statement, the expression within round brackets must evaluate to a Type object. The type represented by this object is then bound to the type variable declared in the angle brackets. This type variable is only in scope for the block of the typewith. The type variable T is cross-stage persisted in the code object, thereby constructing a function with a static parameter type of whatever type typ refers to at run-time.

A problem arises if this code objects needs to be assigned to a local variable outside of the typewith or returned from the enclosing method because the type variable T has limited scope. Since ordinary C#/Metaphor types (including delegate types) are not co-variant, the type Code<Action<T>> is not compatible with Code<Action<object>>. The closest compatible super-type is Code<object>. Any code that consumes this value must cast it back to the Action delegate type, which is rather tedious and begins to defeat the purpose of a type system—be it static or dynamic.
The `typewith` construct also eliminates the need to use the cumbersome continuation-passing style for declaring local variables of a dynamic type. Similarly it can also be used as a simpler, alternative approach for object creation and type casting in the staged, reflective setting. The following code fragments generate the same code object in `result`: the former solely using the staged reflection API and the latter using the `typewith` construct. The latter fragment is much easier to comprehend.

```csharp
Type typ = ...;
Constructor ctor = typ.GetDefaultConstructor();
CodeVoid result = typ.Local(
    delegate (Code<object> get, LocalSet set) {
        return <|
            -set(ctor.Invoke());
            -set(type.Cast(<|(~get).Clone();|>));
    });

Type typ = ...;
CodeVoid result;
typewith<T>(typ) {
    result = <|
        T x = new T();
        x = (T) x.Clone();
    |>
}
```

Similar ‘with’ constructs could be used to un-reify field and method reflection objects too. Not only would such a construct bind the field object to a ‘field variable’ (or a method object to a ‘method variable’) but would also bind the member’s declaring type and the field’s type or method’s return and parameter types to new type variables. The design of these constructs is straightforward and omitted here. In practice however these constructs are not as useful as they may seem as every usage of a member is already adequately covered by methods on the member’s reflection object and the construct tends to introduce excessive type casting with little gain in type safety.

Adding a construct that un-reifies reflection type objects into actual types simplifies the syntax of staged reflective programming and enables the formal parameter types in generated functions to be determined at generation-time. This latter functionality is essential to the implementation of a staged, recursive-type serializer.
4.4.2.2 Memoizing data structure

Recall from Section 4.3.3 that the first step in implementing a staged recursive-type serializer is to design a data structure for memoizing the generated serializer functions.

The signature of generated serializer functions must be described by a delegate type. Two possible signatures for serializer functions are

```csharp
delegate void Serializer(
    object obj, BinaryWriter b, List<object> visited);

delegate void Serializer<T>(
    T obj, BinaryWriter b, List<object> visited);
```

The former signature accepts the object to be serialized as a dynamically-typed object whereas the latter accepts the object as a parameterized static type. The former delegate type cannot be used because it is not a super-type of all serialization functions—e.g., a serializer for the type `Foo` cannot be assigned to a delegate of this type because the sub-typing of delegate types is contra-variant in their parameter types but `Foo` is a sub-type of `object`. Hence the latter is the valid choice. Therefore the memo data structure accepts types as the keys to lookup and returns objects of type `Code<Serializer<T>>`. Of course the type used as the key becomes the value of $T$ in the return type. This suggests that a type-indexed data structure [23] is appropriate for the job.

Object-oriented languages, rather coincidentally, can support type-indexed data structures through static members on generic classes. A separate instance of a static field exists for every distinct instantiation of its declaring generic type. Hence a type-indexed map can be implemented simply by

```csharp
class SerializerMap<T>
{
    static Code<Serializer<T>> Value;
}
```

The expression `SerializerMap<T>.Value` will return the object that has been associated with the type $T$ (or null if no object has been associated). Assigning to the same expression associates an object with the type. Note that this is different from a
usual map (or dictionary) data structure because (1) the key is a type not a value, and (2) the type of the associated value depends on the key.

The drawback of this approach is that since it relies on static fields, separate instances of the map cannot be created. Instead there is always a single, shared, global map. This will be too restrictive for many applications. A better approach for creating a lookup table for the staged serializer is to use a dynamically-typed instance map, such as the standard Dictionary collection type. However without the type indexing something must be done with the type variable, $T$, in the result type as it is no longer bound. All that may be done is round it up to its nearest non-generic super-type, which is object. Hence the type of the memo data structure is Dictionary<Type, object>. Using object here is unfortunate as it requires type casting to use values extracted from the collection. Nonetheless this memo data structure will be used in following all examples of staged recursive serializers.

4.4.2.3 Serializer implementation

With the new typewith extension and suitable memo data structure it is now possible to write a staged, recursive-type serializer.

```csharp
class Persistence
{
    Dictionary<Type, object> map;

    Code Void Serialize(Code<object> obj,
                        Code<BinaryWriter> b,
                        Code<List<object>> visited)
    {
        Type typ = obj.GetType().GetGenericArguments()[0];
        if (typ == typeof(int))
            return <(~b).Write(~(Code<int>)obj); | >;

        // else more primitive type cases
        else if (typ.IsValueType)
            return SerializeFields(obj, b, visited);

        else typewith<T>(typ)
        {
            if (!map.Contains(typ))
            {
                Code<Serializer<T>> serializer =
                < |delegate s(T obj.
```
BinaryWriter b.
List<object> visited) {
    -(map.Add(typeof(T), s));
    if(obj == null) b.Write(0);
    else {
        int index = visited.IndexOf(obj);
        if(index != -1) b.Write(index + 1);
        else {
            b.Write(-1);
            visited.Add(obj);
            ~SerializeFields(obj, b, visited);
        }
    }
}

return Code<Serializer<T>>(~(Code<T>)obj, b, visited);
}

CodeVoid SerializeFields(Code<object> obj, Code<BinaryWriter> b, Code<List<object>> lookup) {
    Type typ = obj.GetType().GetGenericArguments()[0];
    CodeVoid result = ;
    foreach(Field f in typ.GetFields())
        result := Serialize(f.GetValue(obj), b, visited);
    return result;
}

Line 3 declares the data structure used to memoize generated serializer functions. The
Serialize method on line 5 is the primary function for generating a serializer. It
returns a code object that will serialize its code object parameter obj. As part of the
generation process zero or more staged functions may be generated for other types of
objects that occur as fields in the object graph.

Similar to the staged serializer for simple types, lines 9–12 extract the underlying type
of the code object and perform serialization for primitive types. On lines 14–15, if the
type to be serialized is a value type then the simple algorithm can be used as there is
no recursion to handle. The method SerializeFields simply iterates through the
type’s fields accumulating the code to serialize each.
Line 16 un-reifies the type to serialize as the type variable T. Line 18 checks if a serializer for this type has already been generated and stored in the memo table. If it has not then line 21 begins the generation of a new serializer function. When generating this function, a reference to the function itself is added to map (on line 24) before the function has been completely generated using the technique discussed in Section 4.3.3. SerializeFields may recursively invoke Serialize. Therefore it is essential that <s> is added to map prior to the call to SerializeFields so that if SerializeFields recursively invokes Serialize the memoized code is there awaiting retrieval.

The remainder of this code to line 37 is the basic algorithm for serializing objects. Lines 39–40 return a code object consisting of an invocation of the generated serializer function with the arguments that were passed as code objects to Serialize.

Typically a wrapper function would be created at the top level to expose this code by setting up a single callable function that will serialize objects of a specified type. For example,

```csharp
delegate void MySerializer(object obj, BinaryWriter b);

MySerializer MakeSerializer(Type typ)
{
    typewith<T>(typ)
    return <|delegate (object any, BinaryWriter b) {
        T obj = (T)any;
        List<object> visited = new List<object>();
        ~Persistence.Serialize(<|obj|>, <|b|>, <|visited|>);
    }|>.Run();
}
```

In this code the MakeSerializer method accepts a reflection type object that specifies what type to generate a serializer for. It uses the Serialize method above to generate the code and returns a delegate to the compiled function.

### 4.4.3 Revisiting the parametric signature

At the beginning of Section 4.4.1 it was stated that the signature

```csharp
    CodeVoid serialize(Code<object> obj);
```
must be chosen to write a basic (non-recursive) staged serializer over the signature

\[
\text{CodeVoid serialize}\langle T\rangle(\text{Code}\langle T\rangle \text{ obj});
\]

because with the latter it is impossible to supply a type parameter in the recursive call
to the method. However with the introduction of the `typewith` statement this is no
longer the case as the field’s type’s reflection object could be un-reified and passed as
the type parameter to a parametric `serialize` method.

```csharp
CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
    if(typeof(T) == typeof(int))
        return <\[\!<b\>.Write(<\![\!(Code<int>)\!obj]\!]>)\];
    // ...
    else
    {
        CodeVoid result = <\[\!:\]>;
        foreach(Field fld in typeof(T).GetFields())
            typewith<F>(fld.FieldType)
                result := serialize<F>((Code<F>)fld.GetValue(obj), b);
        return result;
    }
}
```

An advantage of the approach is that the expression `typeof(T)` is a much simpler
way to extract the underlying type of the code object. Unfortunately though a type
cast is now needed in the recursive call site of `serialize` to give a parametric type to
the dynamically-typed code object returned from `GetValue`.

There is a serious problem hidden in the implementation of a language that accepts
the above code: the type variable bound by the `typewith` construct is not a first-class
type variable in the underlying CLR virtual machine. Unlike the JVM where gener-
ics are erased during compilation, the CLR preserves full type information about
generic programs in its byte-code language. The compilation process of Metaphor
would like to take advantage of CLR generics for interoperability with the standard
libraries and other CLR languages, and for the performance gains it allows.

However, the CLR only allows type variables to be bound at class and method defi-
nitions, and not at arbitrary points within a method body (as would be required by
Furthermore it has no mechanism to un-reify reflection type objects as type variables. So when code such as

```csharp
Type typ = ...;
typewith<T>(typ)
{
    List<T> list = new List<T>();
}
```

is used to instantiate generic types or call generic methods, these constructors or methods can only be called via reflection because of the lack of an actual CLR type variable.

```csharp
Type typ = ...;
object list =
typeof(List<>)
    .MakeGenericType(typ)
    .GetDefaultConstructor()
    .Invoke();
```

Hence the `typewith` construct is compiled using erasure (because the type of `list` is erased to `object`) and reflection (to dynamically invoke the `List` constructor). The use of reflection is, as is typical, slow and its use here is hidden from the programmer as it is only introduced by the compiler. This problem of the distinction between first-class type variables and un-reified type objects also arises in the statically-typed approach to staged reflection in Section 4.5.4. Further analysis and solutions of the problem are deferred until then.

### 4.4.4 Conclusion

This section has presented a design for integrating the reflection API with multi-stage programming, where the type checking of reflection operations is performed dynamically (i.e., at generation time). The approach involved extending the reflection API with methods that operate on code objects and the addition of a new language construct, `typewith`, which converts a type object to a syntactic type. This approach can express the serializer example in both its basic and recursive forms. Although a technical shortcoming of this approach is its inability to utilize the `CodeRef` type.
A dynamically-typed approach makes sense from a reflection API point of view because the standard reflection API in C#/Java is dynamically typed. This approach is also appealing from a language design perspective because minimal language extension is required: only one new construct and no type system changes.

However, from a multi-stage programming point of view it makes less sense since multi-stage languages employ a static type system that prevents code generation errors at compile-time and dynamically-typed reflection breaks this key property. In the next section it will be shown how a statically-checked reflection type system will smooth the integration between multi-stage programming and object-oriented reflection.

### 4.5 Statically-typed approach

Section 4.4 discussed a dynamically-typed approach to integrating the reflection API and multi-stage programming where type checking of reflection operations is performed at generation-time. Two problems with the dynamically-typed approach were

1. Run-time checking in base reflection operations means that programming errors, including code generation errors, do not occur until generation-time.

2. A considerable amount of type casting is involved on the interface between the dynamically-typed reflection API and the statically-typed base language and multi-stage extensions.

This section will consider a statically-typed approach to integrating the reflection API with multi-stage programming where the type checking of reflection operations occurs at compile-time. This static typing is advantageous for multi-stage programming because it enables code generation errors to be caught at compile-time.

To achieve this goal, the reflection API must be statically typed to bring it in line with the static type system of multi-stage languages. A static type system for the reflection
API will address both of the problems of the dynamic approach above: (1) because the static system will prevent more programming errors at compile-time, and (2) because of the absence of dynamically-typed values that need to be type cast.

4.5.1 Parametric reflection types

To solve the dynamic typing problems, richer types are given to reflection objects so that the type checker has more information to work with when analysing the validity of reflection operations. The types `Type`, `Field` and `Method` are parameterized to contain information about the type and members they represent. Type objects are described by the type `Type<T>` where `T` is the type the object represents. This type is invariant in its parameter (i.e., `Type<A>` is a subtype of `Type<B>` iff `A = B`) and there is only one object of type `Type<T>`. This parameterization is similar to the parameterization of the `Class` type in Java.

Field objects have the type `Field<A,B>` where `A` is the type the field is defined on and `B` is the type of the field itself.

Methods are more problematic than fields because they have a variable number of parameters (arity), possibly return `void` and possibly have pass-by-reference (by-ref) parameters. The arity problem can be handled by using a series of `Method` types, one for each arity: e.g., `Method<A,B>`, `Method<A,B,C_1>`, `Method<A,B,C_1,C_2>`, etc. where `A` is the type the method is defined on, `B` is the return type of the method and the `C_i` are the parameter types of the method. (As in C#, a generic type name such as `Method` can be overloaded on its type parameter arity.) However since `void` and by-ref types are not first-class types in C#, explicit variants of the `Method` type need to be created to cater for each of these cases. A naming scheme for this family of `Method` types is given by the EBNF, where `n` is a natural number:

\[
\text{Method}(\text{Void})?(_n)^*\langle A(.B)^*\rangle
\]

The string ‘Void’ in the type name indicates that the method returns `void`. The numbers `(n)` in the type name indicate what parameter positions are by-ref. Some examples are
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MethodVoid&lt;A&gt;</td>
<td>Defined on A, returns void, no parameters.</td>
</tr>
<tr>
<td>Method_1&lt;A,B,C1&gt;</td>
<td>Defined on A, returns B, one by-ref parameter of type C1.</td>
</tr>
<tr>
<td>MethodVoid_1_3&lt;A,C1,C2,C3&gt;</td>
<td>Defined on A, returns void, three parameters of types C1, C2, C3, the first and third parameters are by-ref.</td>
</tr>
</tbody>
</table>

This set of types is quite cumbersome and the presence of by-ref parameters means there is an exponential number of Method types for methods with up to $n$ parameters. However methods with by-ref parameters are fairly rare and in practice it would be reasonable to have methods with by-ref parameters only up to an arity of, say, 3. Nevertheless dealing with methods in this approach to statically-type-safe reflection is extremely tedious, and for this reason further discussion in this section will neglect the treatment of methods. Methods will be handled with less tedium in Section 4.5.3 using a different approach to achieving static type safety.

To avoid the bother with methods repeating itself for constructors it is assumed that all types have a default constructor. It also would be possible to define static versions of the reflection member types (e.g., StaticField<B>) to refer to static members, but this is omitted.

Using this richer typing of reflection objects, the primitive, staged reflection operations presented in the beginning of Section 4.4 can now be re-expressed.

```csharp
class Field<A,B> {
    CodeRef<B> GetValue(Code<A> obj);
}

class Type<A> {
    Code<A> InvokeDefaultCtor();
    Code<A> Cast<B>(Code<B> obj);
    CodeVoid DeclareLocal(DeclareLocalCont<A> k);
}

delegate CodeVoid DeclareLocalCont<A>(CodeRef<A> var);
```

In the dynamically-typed staged reflection approach it was not possible to use the CodeRef type because, being dynamic, values were described using the super-type
object and the CodeRef type is necessarily invariant. However with the statically-typed parameterization of reflection types, full type information about the code objects these methods construct is precisely known through type variables. Therefore the CodeRef type can be employed which simplifies the interface to the reflection system (by eliminating separate ‘set’ methods to assign to fields or local variables) and increases the cohesiveness of the language (by allowing the CodeRef type to interoperate with the reflection API).

With strong typing of reflection operations in place, it seems appropriate to incorporate them into the language’s syntax to give more readable code since these reflection operations are simply generalized versions of fundamental object-oriented operations.

Reflection operations that involve types are automatically integrated into the syntax, as they were for the typewith statement in Section 4.4.1, since the type variable from the Type type can be used to create new objects, cast objects and declare local variables and function parameters.

The reflective field access operation is worked into the language’s syntax by introducing a staged dot operator. The staged dot operator is a generalization of the regular dot operator for field access where the right-side of the dot can be an arbitrary expression which evaluates to a reflection field object instead of the name of a field. This new operator is written \( \_ \_ \) by analogy with the escape operator \( \_ \_ \) since it allows splicing a field into generated code. So instead of being limited to writing field access as \( e.f \) where \( e \) is an expression and \( f \) is the name of a known field, the programmer can write \( e_1.\_e_2 \) where \( e_2 \) is an arbitrary expression that evaluates to a reflection field object. Like the escape operator, the staged dot operator must appear under brackets.

The static semantics for the staged dot operator are
<table>
<thead>
<tr>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$ is an expression of type $C$ at level $n+1$</td>
<td>$e_1 \cdot e_2$ is an expression of type $B$ at level $n+1$</td>
</tr>
<tr>
<td>$e_2$ is an expression of type $\text{Field}&lt;A,B&gt;$ at level $n$</td>
<td></td>
</tr>
<tr>
<td>$C$ is a subtype of $A$</td>
<td></td>
</tr>
</tbody>
</table>

The operational semantics of the staged dot operator are equivalent to $\text{Field}$’s staged $\text{GetValue}$ method as the former is simply syntactic sugar for the latter. The code below obtains a reflection field object for the field $i$ on $A$. It then uses this field object with the staged dot operator to build a code object that accesses the field on a new object. Note there is no use of reflection in the generated code object.

```java
class A { int i; int j; }
Field<A,int> f = fieldof(A.i);
Code<int> val = <|new A().~f|>:
// val evaluates to <|new A().i|>
```

### 4.5.2 Encompassing more of reflection

The type $\text{Type}<T>$ requires a $\text{GetFields}$ method to return all the methods of the type. This method is needed, for example, to enumerate all the fields of a type when implementing a serializer. A first approximation of this method is

```java
class Type<A> {
    Field<A,B>[] GetFieldsWithType<B>();
}
```

This method returns an array of field objects. Each field object describes a field on the type $A$ and having a field type $B$, which is the type argument the caller has passed to the method. The call $\text{GetFieldsWithType<int>()}$ would return all the $\text{int}$ fields of the type. However, to be useful for the serializer this method must be generalized to return all field types, not just a particular one.

Typing the generalized version of this method is problematic as the method returns a heterogeneous collection of field objects—i.e., although all the field objects are de-
fined on the same type, they each have different field types. Intuitively the signature of this method could be written

```java
class Type<A> {
    Field<A,?>[] GetFields();
}
```

The ? indicates missing type information and is equivalent to the wildcard type parameters in Java. The use of wildcard type parameters introduces a form of existential typing into the language. The return type of this method is semantically equivalent to the existential type `(∃B.Field<A,B>)[]`.

However the use of ? as a shorthand for existential typing is ambiguous. As written the type `Field<A,?>[]` could refer to any of these three types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Field&lt;A,∃B.B&gt;[]</code></td>
<td>Equivalent to <code>Field&lt;A,object&gt;[]</code>.</td>
</tr>
<tr>
<td><code>(∃B.Field&lt;A,B&gt;[]</code></td>
<td>Heterogeneous array of field types.</td>
</tr>
<tr>
<td><code>∃B.Field&lt;A,B&gt;[]</code></td>
<td>Homogeneous array of an unknown field type.</td>
</tr>
</tbody>
</table>

The ambiguity is solved in Java by a rule that states that all wildcard type parameters are quantified at the outermost level in their corresponding existential type. Therefore the return type of `GetFields` would be interpreted as `∃B.Field<A,B>[]` using the Java model, which is not the intended type.

To resolve this ambiguity in Metaphor, explicit existential quantification is used instead of the Java wildcard shorthand. (In actual source code the existential quantifier, ∃, is written as the keyword `exists`.) The method signature of the `GetFields` method becomes

```java
class Type<A> {
    (exists B.Field<A,B>[] GetFields();
}
```

Essential elements of any language with an existential type system [37] are constructs for the introduction and elimination of existentially-typed values. The introduction construct, commonly known as `pack`, takes a value and renders it with a given existential type. This operation can be naturally accomplished in an object-oriented lan-
guage through the existing process of *up-casting*. In an object-oriented setting, existential typing is essentially a generalized version of inheritance. An existential type expresses a set of types whereby the set itself acts as a super-type of any type in that set, e.g., `List<int>` is a subtype of `∃T.List<T>`.

The opposite of introduction is elimination and this construct is commonly known as *open*. The opposite of up-casting is down-casting so it would be reasonable to expect that elimination can be expressed as down-casting. However, this is only partly the case. Java uses down-casting for a form of elimination. In Java, a value of type `List<?>` can be downcast to a `List<int>`. As with all down-casting a run-time type test is performed and an exception is thrown if the types are not compatible. However down-casting is not the same as open because it asserts the quantified type variable to be a particular type instead of binding it to a new type variable.

This behaviour is perhaps not a practical problem for typical Java programs, which is why Java can suffice with this down-casting approach. However down-casting is not adequate to bring about the static type safety of staged reflection. Consider generating code to access a field where the field to access is chosen from the array returned by `GetFields`. Indexing into the array yields a value of type `∃B.Field<A,B>` but the staged field access operator, `~`, requires a value of the non-existential type `Field<A,B>`. Hence the array element needs to be properly opened and the abstracted type bound to a new type variable.

In a way ‘open’ is the statically-typed equivalent of `typewith` from the dynamically typed approach: both constructs bind new type variables to hold type information that was abstracted or hidden. An open construct typically has three parts: an expression of an existential type, a variable to bind the ‘opened’ existential value, and the (non-existential) type of the opened value. In Metaphor the open construct is called *typecast* because it is seen as a generalization of down-casting that binds new type variables.

The remainder of this subsection introduces three new syntactic elements: *typecast*, *typeif* and *foreach*. These constructs enable the programmer to more naturally work with existentially-typed values.
4.5.2.1 The ‘typecast’ construct

The typecast construct can also be seen a generalization of typewith. Since the Type type is now parametric, its type parameter already serves as the un-reified representation of the type object. Therefore when the type parameter is not existentially abstracted (e.g., in the type Type<T>) then a typewith construct is redundant since T is already the un-reified type that typewith would produce. If however the type parameter is abstracted (e.g., exists T.Type<T>) then the expression typewith<T>(e) can be expressed as typecast<T>(x = e as Type<T>). Therefore the former typewith construct is superseded by typecast.

The abstract syntax of typecast and example concrete syntax are

\[
\text{stmt ::= ... | } \quad \text{typecast<type-var>(var = expr as type) block}
\]

\[
\text{typecast<T>(fld = flds[i] as Field<A,T}) \{ \ldots \}}
\]

Similar to a class or method definition, the typecast statement defines a new type variable, type-var. This type variable will be bound to the type that was abstracted through existential quantification. Inside the typecast statement’s parentheses is much like an as type test expression from C#. The expr must have an existential type. The type must be the non-existentially-quantified version of the expression’s type where the newly defined type variable is substituted for occurrences of the existentially-bound type variable. In the above example flds[i] has the type \(\exists B.Field\langle A,B\rangle\). Removing the existential quantification and replacing B with T, the type variable defined by the typecast statement, gives Field\langle A,T\rangle which is the type given to the variable fld. The type variable and variable defined by the open statement are in scope within block.

Now with existentially quantified types, a language construct to open them, and a statically-typed reflection API, an attempt can be made to write a version of the serializer algorithm that is statically type safe (i.e., will not produce any dynamic typing errors at run-time).

```csharp
CodeVoid serialize(Code<object> obj, Code<BinaryWriter> b)
{
    exists T.Type<T> typ =
```
obj.GetType().GetGenericArguments()[0];
if(tytpl == typeof(int))
    return <|(~b).Write(~(Code<int>)obj)|>;
else typecast<T>(t = typ as Type<T>)
{
   CodeVoid result = <|
   foreach(exists X.Field<T,X>f in typeof(T).GetFields())
     typecast<F>(fld = f as Field<T,F>)
     result := serialize(<|~(Code<T>)obj.~fld|>, b);
   return result;
}

This version uses the non-parametric method signature (see Section 4.3.4). Writing
the serializer like so solves one of the identified problems of the dynamic typing ap-
proach: the expression that performs the reflection operation to generate code (i.e.,
the .~ operator) is statically checked. In other words this operation does not need
to check at generation-time that the field access is valid because the type system can
verify this at compile time. However the usefulness of the static check is significantly
undermined by the need to type cast the left operand of .~. This problem is the oth-
er identified problem of the dynamic typing approach: numerous type casts in the
code. This version of the serializer program fails to adequately address this problem
because cumbersome type casts are required in both the base (for int type) and re-
cursive cases of the algorithm.

Luckily the need for type casts evaporates simply by writing the function in paramet-
ric style (i.e., using the parametric signature alternative from Section 4.3.4).

CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
   Code<int> intObj = obj as Code<int>;
   if(intObj != null)
      return <|(~b).Write(~intObj)|>;
   else
   {
      CodeVoid result = <|
      foreach(exists X.Field<T,X>f in typeof(T).GetFields())
        typecast<F>(fld = f as Field<T,F>)
        result := serialize<F>(<|~obj.~fld|>, b);
      return result;
   }
The recursive case can now be written without a type cast and the reflective code generation portion is truly statically typed: \( \text{obj has type } \text{Code}<T> \), \( \text{fld has type Field}<T,F> \), therefore \( \neg \text{obj.} \neg \text{fld has type } F \). Furthermore re-writing in parametric style has eliminated one typecast block in the else-branch of the recursive case.

In the base case, the type cast in the quoted expression is removed. A type test is still used but it serves a double purpose: reducing \( \text{obj} \) to the more specific type \( \text{Code}<T> \) and also testing if \( T \) equals \( \text{int} \). This is an improvement in terms of code clarity from the previous version which used type object equality to compare types. Although this code still involves a type cast of sorts, it is still considered an acceptable solution to the statically-typed, staged serializer objective since type casting is the conventional way to implement type conditional logic in an object-oriented language.

4.5.2.2 The ‘typeif’ construct

If a truly statically-typed solution is sought then the type cast can be replaced with the introduction of a type case \([48]\) construct into the language. In Metaphor the construct is called typeif by analogy with an if-else-statement rather than with the more functional-esque case analysis expression. The equivalent of a type case can still be achieved by building a typeif-else ladder. The typeif statement tests whether a type variable is equal to a particular type. If equality holds then in the following block every occurrence of the type variable in the type of an expression is substituted with its asserted literal type. The abstract syntax for this construct is

\[
\text{typeif}(\text{type-var is type}) \, \text{block (else block))?}
\]

The base case for the serialize method using typeif becomes

```java
CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
    typeif(T is int)
    return <|(~b).Write(~obj):|>
    else ...
}
```

Now no type cast is involved but it comes at the cost of an additional language construct. If the type variable \( T \) is bound to \( \text{int} \) then the first block of the typeif will
execute. Within this block all occurrences of $T$ are substituted with $\text{int}$, hence the type of $\text{obj}$ becomes $\text{Code<int>}$, thereby avoiding the need for a type cast.

The `typeof` construct also has uses in general (viz. not necessarily using staging or reflection) object-oriented programming as it can be used in place of type casting for any type conditional logic. Moreover the construct can be extended to do more than mere type equality, it can assert arbitrary characteristics of a type. For example, Java and the CLR allow type variables to be constrained: i.e., be required to implement a certain interface or have a no-argument, public constructor. The `typeof` statement could test whether a type variable satisfies these constraints. The statement could also do pattern matching of types, so instead of just comparing a type variable against concrete types like $\text{List<int>}, \text{List<string>}$, etc. it can be compared against a type pattern like $\text{List<T>}$ which will introduce a new type variable, $T$, bound to the actual type parameter of $\text{List}$.

### 4.5.2.3 The ‘foreach’ construct

The pattern used in the recursive case of the `serialize` method for opening every item in a collection of existentially-typed values arises quite frequently. It is therefore convenient to have syntactic sugar that combines the for-loop and typecast statements. This sugaring is given by the following equivalence in abstract syntax.

$$
\begin{align*}
\text{foreach}<\text{type-var}>(\text{type var in expr}) \text{ block} & \equiv \\
\text{foreach}(\exists \text{X. type[type-var/X]} \ x \ \text{in expr}) & \\
\text{typecast}<\text{type-var}>(\text{var = x as type}) \text{ block}
\end{align*}
$$

The recursive case of the serializer can now be written more concisely, like so

```csharp
CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
    ...
    else
    {
        CodeVoid result = <|::|>; 
        foreach<F>(Field<T,F> fld in typeof(T).GetFields())
            result := serialize<F>(<|~obj.-fld|>, b);
        return result;
    }
}
```
4.5.3 Syntactic reflection

The type systems discussed so far work by retrofitting a static type system onto the existing mechanics of the CLR reflection API. The essential nature of the reflection API is that types and members (i.e., fields and methods) are reified as values in the evaluation of a program. As values they consequently need types. In the CLR they have simple types of Type, Field, etc. However to permit static type checking of the operations that use these values, their types need to carry more information (in Metaphor, Type<T>, Field<A,B>, etc.). However typically a program cannot provide all the static information needed by these types because the program is doing something that is essentially dynamic: hence its motivation for using reflection in the first place. So the type system then uses existential quantification to abstract away the unknown parts of the type, e.g., exists B.Field<T,B>. This type system is quite expressive and seems to be informally sound. However it comes at a high cost to the language implementer and language user. To the user existential types are very big and the open construct is introduced pervasively throughout the code. To the implementer existential types are difficult and inefficient to compile on the CLR (see Section 4.5.4), which then encourages placing arbitrary restrictions on the domain of existential quantification: should only reflection types be existentially-quantifiable? what about the code type? what about any CLR type? how are existentially-quantified type variables used with generics?

Another approach to type reflection [49] works by adding type reflecting constructs to the language directly rather than trying to directly tame the reflection API. In this approach types and members are not reified as values. The reification of types as values in the previous approach is the root of all its problems. Making a type a value puts it at the mercy of the full computational power of the language, which is why so much typing machinery is needed to ensure nothing goes astray. Arguably, for most applications, types and members do not need to be manipulated by so much computation. By restricting the reflection of types and members to the level of types it is possible to preserve most of the expressiveness of reflection while significantly reducing the complexity of the type system.
This subsection discusses how an alternative approach to statically type the reflection API [49], which does not use existential types, could have been applied to Metaphor. This approach introduces a new construct into the language for iterating over the members of a type. The `formember` statement is similar to the `foreach` statement and is used to iterate over the members in a type. The syntax is

```
embedded-statement ::= ... | formember ( static? member-pattern in type ) embedded-statement
```

```
member-pattern ::= 
  type-pattern identifier | matches field
  type-pattern identifier ( param-pattern-list ) | matches method
  void identifier ( param-pattern-list ) | matches void method

type-pattern ::= identifier

param-pattern ::= identifier | ref identifier | out identifier
```

The optional `static` modifier specifies whether to iterate over static or instance members of the type. The member pattern is used to identify whether the member is a field or a method, and, if a method, what is its arity and calling convention (i.e., whether it returns a value or not and if any of its parameters are passed by reference). All types in the pattern (field type, method return type, method parameter types) are bound to fresh type variables. The name of the member declared in the pattern, known as a member variable, and the type variables declared in the pattern are in scope for the body of the `formember` statement. The member variable can be used in the body of the statement as though it is a real member defined on the type specified in the head of the statement. Therefore it is valid on the right-side of the member access (dot) operator.

The staged serializer can be written using this approach like so

```csharp
CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
  if (T is int) return <(~b).Write(obj),>

  CodeVoid result = <>::
  formember(F fld in T)
    result := serialize<F>(<(~obj).fld>, b)
  return result;
}
```
The member variable and type variables can be reified into reflection objects through the `typeof`, `fieldof` and `methodof` operators. This process is not essential to the usefulness of the `formember` construct but is practical when interoperation with the reflection API is needed. For example, to get the name of the field as a string.

```csharp
List<string> names = new List<string>();
for (member(F fld in T)
    names.Add(fieldof(fld).Name);
```

The member pattern could be refined to also match types within the member (e.g., match all fields of type `int`). This would be syntactic sugar for matching all members and then using the `typeid` statement inside the `formember` statement to do the type matching.

```csharp
T obj;
for (member(F fld in T)
    typeid(F is int)
    {
        int val = obj.fld
    }
```

A good thing about the member pattern matching is that it provides natural way of handling reflection of methods, which have varying arity. This is a great improvement over the complex system of method types needed in the existential approach.

This construct will only match non-generic methods. Another refinement would be to match generic methods as well, however this is quite difficult. It is difficult because the pattern must somehow establish the relationship between the type parameters of the matched method and how they are used in the method's parameter types. The following code could match generic methods with one type parameter on the class `C`.

```csharp
class C {
    T m<T>();
}
for (member(R mth<_>() in C)
{
    R val = mth<int>();
    val = mth<string>(); // error
}
```
However type variable $R$ in the pattern does not depend on the method’s type parameter. Hence different instantiations of the generic method will have the same return type, $R$, even though, as is the case for the method in class C, the return type is controlled by instantiation of the generic method. In seems $R$ needs to be a type function, as opposed to a type variable, that accepts the method’s type parameters and returns the instantiated type for that part of the method signature. This is all rather complicated and since generic methods are much less commonly used than ordinary methods it is a reasonable comprise that the `formember` construct does not work with generic methods.

### 4.5.4 Implementation

The creation of a static type system for reflection in this chapter is predicated on the introduction of existential types. Existential types are a powerful language feature that accurately describe the reflection operations at hand and successfully create the desired static type environment. While a triumph for language design, existential typing is difficult to implement on the CLR platform.

Treating existential types as a first-class extension to the language means that the programmer may use them anywhere in a program (i.e., in more than just the specialized use in the reflection API) and the compiler must have a systematic and correct method for translating them into compiled form. Consider the compilation of the local variable declaration

```csharp
exists X.List<X> anyList = ...;
```

There is no equivalent to existential types in the CLR; without modification to the CLR it is not possible to represent existential types in CIL and therefore they must be erased. The erasure of an existential type is the nearest super-type that does not involve the quantified type variable. Some examples are given in the table below where `MyClass<A,B>` is a subtype of `MyBase<A>`.

<table>
<thead>
<tr>
<th>Existential type</th>
<th>Erased type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exists X.List&lt;X&gt;</code></td>
<td><code>object</code></td>
</tr>
<tr>
<td><code>exists X.MyClass&lt;T,X&gt;</code></td>
<td><code>MyBase&lt;T&gt;</code></td>
</tr>
<tr>
<td>Existential type</td>
<td>Erased type</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>exists X.MyClass&lt;X,T&gt;</td>
<td>object</td>
</tr>
</tbody>
</table>

Hence the local variable declaration will be compiled to

```java
object anyList = ...;
```

Now consider this use of the variable `anyList`:

```java
typecast<T>(list0fT = anyList as List<T>) {
    T item = list0fT[0];
}
```

One approach to compiling this code would produce the equivalent CIL to this C# code:

```java
Method getItem = anyList.GetType().GetMethod("get_Item");
object item = getItem.Invoke(anyList, new object[] { 0 });
```

In CIL the [...] indexing syntax is represented as a method named `get_Item`. The `get_Item` method cannot be invoked directly on `anyList` because it has type `object` and the method is defined on `List`. Nor can `anyList` be cast to `List` because the type argument of `List` needs to be known statically to perform a cast. Therefore the method can only be invoked through reflection. The first line discovers the runtime type of `anyList` and acquires a method object to the target method. The second line then invokes the method via reflection. In general the code emitted for the method lookup can be quite verbose when factoring in method overloading.

The obvious problem with this technique is performance. Although the source code contains a simple, statically-checkable method invocation, the compiler actually emits code for a reflective method lookup and invocation, giving the unwitting programmer unexpected inefficiency.

A sufficiently optimizing compiler may be able to mitigate the performance costs, e.g. by dynamically generating optimized code for a `typecast` block. However such an optimizer increases the complexity of the compiler in a way that is not directly needed for staged reflective programming: i.e., the code this optimizer works on is existentially-typed expressions that are not used in a staged context. Therefore such
an implementation burden was not worthwhile for this research. Moreover the effect of the optimizer is redundant as almost all optimization can be expressed in the multi-stage language by the programmer and need not be hard-coded into the compiler. For example the following code converts an existentially-typed list (anyList) into a list of strings (strList). The contents of the typecast have been staged so that instead of emitting reflection code the compiler will emit code that builds a code object and runs it. This code object is optimized to work with the current value of T and does not use reflection.

```csharp
exists X.List<X> anyList = ...;
List<string> strList = new List<string>();

typecast<T>(tList = anyList as List<T>)
  <|foreach(T elem in tList)
    strList.Add(elem.ToString());|>.Run();
```

However identifying and modifying every instance of typecast that will compile to inefficient code can be tricky and tedious for the programmer. Hence the real problem here is that with existential types it is too easy to write source code that compiles to inefficient code but too difficult to always write source code that compiles to efficient code.

The other major problem with implementing existential types is the second-class nature of type variables bound by the typecast construct. This problem was first described in Section 4.4.3. Their second-class nature becomes apparent when attempting to invoke a generic method or instantiate a generic class. This code creates a new instance of the generic list class using a typecast-bound type variable.

```csharp
typecast<T>(tList = anyList as List<T>)
{
    List<T> newList = new List<T>();
    newList.Add(tList[0]);
}
```

Because T is not a type variable at the implementation level, the List<T> constructor cannot be invoked directly. Instead the compiler must emit code that will use reflection to create the object, thus will produce the CIL equivalent to the C# code
Type T = anyList.GetType().GetGenericParameter(0);
Type ListT = typeof(List<>).MakeGenericType(T);
object newList = Activator.CreateInstance(ListT);

Method getItem = anyList.GetType().GetMethod("get_Item");
object item = getItem.Invoke(anyList, new object[] { 0 });

Method add = newList.GetType().GetMethod("Add");
add.Invoke(newList, new object[] { item });

Once again this code is inefficient and verbose, but what makes it worse is that in many scenarios the reflective invocation of a generic method is not essential to the functioning of the algorithm. Returning to the serializer example

CodeVoid serialize<T>(Code<T> obj, Code<BinaryWriter> b)
{
    typeif(T is int) return <\{b}.Write(~obj);>
    else
    {
        CodeVoid result = <|;|>; 
        foreach<F>(Field<T,F> fld in typeof(T).GetFields()) 
            result ::= serialize<F>(<\{~obj.~fld|>, b); 
        return result;
    }
}

The serialize method never uses its type parameter T in a way that is necessary or convenient to be represented as a type variable in CIL. Instead the type parameter could be compiled as regular parameter of type Type and in doing so avoid the need for reflection at the recursive call site of serialize.

The upshot of this is that if the language designer lets loose existential types over the entire language it creates havoc for the language implementation because

- existential types need to be erased when compiled to CIL,
- run-time reflection needs to be used to recover erased type information, and
- type variables bound by the existential open construct cannot be represented as type variables in CIL.
To relieve the implementation burden it is necessary to at least restrict and stylize how existential typing can be used in the language. This will be the topic of the following section.

4.5.5 Conclusion

This section has described the integration of the reflection API and multi-stage programming in a statically-typed manner. This approach has overcome two concerns of the dynamically-typed approach from Section 4.4: reflection errors are identified at compile-time hence avoiding code generation errors at generation-time, and many programs can be written without dynamic type tests and casts where they were needed in the dynamically-typed approach.

However this was achieved at a large cost to the simplicity of the language design and implementation. A static type system involving existential types was developed for the reflection API and an existential ‘open’ construct added to the language. This development had a number of ensuing effects:

- the existential type system affects the entire language and is not localized to the reflective or staged parts,

- the empirical usage of the ‘open’ construct is syntactically verbose but can be improved by introducing new constructs as syntactic sugar, and

- the implementation of a general-purpose existential type system is problematic on modern object-oriented virtual machines, or at least on ones which do not implement generics via erasure such as the CLR.

In the next section a pragmatic approach is taken to type safety by developing a hybrid dynamic/static approach for integrating the reflection API with multi-stage programming. This approach tries to adopt as much static typing as possible but without becoming entangled in the ensuing complexities wrought by the approach in this section.
4.6 Hybrid approach

The section presents a compromise solution to the problem of staged reflective programming. It combines the best features of the two approaches discussed so far: dynamic and static. From the dynamically-typed approach it takes the simplicity and efficiency of implementation. From the statically-typed approach it takes the safety and conciseness of programming.

4.6.1 Existential typing simplified

Section 4.5 concluded that although existential types provide a suitable framework for controlling the safety of staged, reflective operations, they are difficult and inefficient to implement. To preserve type safety and improve implementation, existential types must be limited in their utility in the language. Rather than introducing existential typing as a first-class language feature, a reflection API will be created that whilst modelled on principles of existential types never explicitly uses them. Two observations enable this development.

Firstly, although adding an existential quantifier to the language greatly improves the expressivity of the type system, it is for the most part non-essential. Despite its usefulness, a great many object-oriented programs can be adequately written today without the aid of an existential type system. Presumably this is because the cost of added language complexity and implementation outweighs the benefits it brings to programming. However in this work existential types were not introduced to make a general improvement to object-oriented programming, instead the purpose was to preserve multi-stage static type safety when incorporating object-oriented reflection. By restricting the application of existential types to just the parts of the reflection API required to fulfil this objective then a degree of static type can be attained without overly complicating the language.

Secondly, a limited form of existential typing can be expressed using inheritance and generics in Java/C# without modification. Given a generic type that inherits from a non-generic class

    class ExistsFoo { }
class Foo<T> : ExistsFoo { }

the type ExistsFoo is equivalent to the type $\exists T$. $\text{Foo}<T>$. The approach is limited because the generic type (1) must inherit from a specially defined existential base class, (2) cannot encode all abstraction variations on multi-parameter generic types, and (3) cannot represent high-order existential types such as $\exists T$. $\text{Foo}<\text{List}<T>>$.

4.6.2 Semi-static reflection type system

From the examples considered so far there are three places where existential types are necessary to bring about static typing to staged, reflective programs.

- **exists T.Type<T>**: To represent types that are not known statically. Objects of this type arise through the operations Type.Parse which creates a type object from a string and object.GetType which returns the run-time type of an object.

- **exists F.Field<T,F>**: To represent fields were the declaring type of the field is statically known but the field type is not. The Type<T>.GetFields method returns an array of these objects representing each field on the type T.

- **exists T.Code<T>**: To represent code objects with a statically-unknown underlying type. Objects of this type typically arise using the reflection API to generate code. In Section 4.3.5 it was commented that because of code type co-variance this type is equivalent to Code<object> and hence the latter type has always been used in preference.

A class hierarchy is created that encodes the specific forms of existential typing mentioned above.

```csharp
class Type { // exists T.Type<T>
    static Type Parse(string name);
    Field[] GetFields();
}

class Type<T> : Type {
    Field<T>[] GetFields();
}
```
class Field { // exists T,F.Field<T,F>
    CodeRef GetValue(Code obj);
}

class Field<T> : Field { // exists F.Field<T,F>
    CodeRef GetValue(Code<T> obj);
}

class Field<T,F> : Field<T> {
    CodeRef<F> GetValue(Code<T> obj);
}

class CodeVoid {
    void Run();
}

class Code { // exists T.Code<T>
    object Run();
}

class Code<T> : Code {
    T Run();
}

class CodeRef : Code { // exists T.CodeRef<T>
}

class CodeRef<T> : CodeRef, Code<T> {

}

Although the type \( \exists T.\text{Code}<T> \) is equivalent to \( \text{Code<object>} \) it is sometimes more intuitive to think of the type in its existential form. Therefore a base class, Code, is created.

The added type CodeRef is equivalent to \( \exists T.\text{CodeRef}<T> \), but since the CodeRef type is not co-variant this type is not equivalent to CodeRef<object>. Consequently a value of this type cannot be spliced until it is opened to give a more specific CodeRef<T> type.

The CodeVoid type is not part of the hierarchy as void can never be bound to a type variable.

The hierarchy for fields comprises three types giving a progression of increasing static type information. The first type, Field, is fully dynamic—neither the declaring type nor field type is known statically—and is equivalent to the field type from
the dynamically-typed approach. The last type, Field\(<T,F>\), is fully static and corresponds with the field type from the statically typed approach. The middle type, Field\(<T>\), conveys only the declaring type statically and is equivalent to the type exists F.Field\(<T,F>\) from the static approach.

The field types possess methods for building code objects to access fields. The Field\(<T,F>\) class has a single method for getting and setting a field that returns an assignable code object. The two other classes, Field and Field\(<T>\), exploit the new (implicitly existentially-quantified) CodeRef type as their return types, thereby using existential typing to solve the dynamically-typed CodeRef invariant problem from Section 4.4. The class Field uses the (un-parameterized) Code type to describe the receiver object as it is unknown for this class. The GetValue methods on Field\(<T,F>\) and Field\(<T>\) perform no run-time type tests, hence these are fully, statically checked operations. The GetValue method of Field, being fully dynamic, tests at run-time the type of the receiver object.

The class Type\(<T>\) has a method to return all the fields of \(T\) as an array of Field\(<T>\) objects. It inherits from its more dynamic counterpart Type which contains no type information about the type an object represents. This type has a Parse method for creating a type object from its name, and a GetFields method for listing all the fields on a type. The return type of this method is slightly inaccurate: in existential notation it should have the type \(\exists T.(\exists F.Field\(<T,F>\))[]\) but actually has the type \((\exists T.\exists F.Field\(<T,F>\))[]\). Its true type cannot be easily encoded in this class hierarchy, but since this is a small flaw and this method will be seldom used (as the version on Type\(<T>\) is more useful) it will be overlooked.

Once again up-casting manifests the pack existential type operation—the correspondence between pack and up-casting is explicit now that types are actual sub-types of their existentially quantified variants. The open existential type operation is modelled on a generalized version of type casting. Where regular type casting only allows down-casting an object to a mono-type, the generalized version can down-cast to a generic type pattern and bind new type variables as needed. This construct is called typecast and has the abstract syntax
The `typecast` statement declares zero or more type variables (between angle brackets). If no type variables are declared then the operation is performing a monotypic type cast, in which case the standard type casting syntax could also be used. The expression `expr` is evaluated and its run-time type compared with `type`, which is the target type the programmer wishes to cast the object to and may contain the type variables declared by this construct. If the evaluated object's run-time type is compatible with the target type then the object is bound to the variable `var`, which is given the type `type`, and the first block executes. Otherwise the else block executes if present. The type variables declared at the head of the typecast statement are only in scope for the first block.

The following code demonstrates the use of this construct by testing if `anyList` is a kind of list and if so converting all its items to strings.

```csharp
object anyList = new List<int>();
List<string> strList = new List<string>();
typecast<T>(tList = anyList as List<T>)
<|foreach(T item in tList)
    strList.Add(item.ToString());|>.Run()
else throw new Exception("Not a list.");
```

In order to prevent the implementation troubles caused by existential typing (see Section 4.5.4) a number of restrictions are placed on the type variables and value variable bound by `typecast`. At the same level as the statement, the type variable(s) can only be used in a type that is the subject of the `typeof` operator, or as a type argument (possibly indirectly) of one of the types `Code`, `Type` or `Field`. Hence a type variable cannot be used to declare local variables or passed as an argument to a generic method or generic class instantiation.

At the same level as the statement, the value variable cannot have members invoked on it or be the subject of a `foreach` statement.

At levels greater than the level the `typecast` appears there are no restrictions on the declared type or value variables.
4.6.3 Serialization

With this hybrid static/dynamic reflection type system defined, it is now possible to write this chapter’s motivating example: the serializer generator.

```csharp
CodeVoid serialize(Code obj, Code<BinaryWriter> b)
{
    typecast(intObj = obj as Code<int>)
    return <(-b).Write(intObj)>;
    // type test for other primitive types
    else
    {
        CodeVoid result = <;>
        typecast<T>(tObj = obj as Code<T>)
        foreach(Field<T> fld in typeof(T).GetFields())
            result := serialize(fld.GetValue(tObj), b);
        return result;
    }
}
```

This program cannot use the parametric method signature (see Section 4.3.4) because of the restriction that `typecast`-bound type variables cannot be passed to method type arguments at the same level as the typecast, hence making the recursive call to `serialize` impossible. This program adopts the non-parametric approach by using `Code` as the type of `obj`. The program uses `typecast` to test for primitive types—although a regular C#-style type cast would suffice in this instance, the `typecast` construct gives more concise syntax. The `typecast` construct is used again (this time necessarily) to convert `obj` to have a parametric code type so that it can be used with the static reflection type system. The `GetValue` method builds the code for the field access and does so without the need for run-time type checks. In the dynamic reflection type system this operation would require dynamic type testing and in the static system would require using the field’s type as a method type argument.

The recursive-type serializer can also be written clearly and concisely using this approach.

```csharp
delegate void Serializer<T>(T obj, BinaryWriter w);

class Persistence
{
    Dictionary<Type, Code> map;
}```
CodeVoid Serialize(Code obj,
Code<BinaryWriter> b,
Code<List<object>> visited)
{
  typecast(intObj = obj as Code<int>)
  return <(~b).Write(~obj);>
  //else more primitive type cases
  else typecast<T>(tObj = obj as Code<T>)
  {
    if(typeof(T).IsValueType)
      return SerializeFields(obj, b, visited);
    else
    {
      if(!map.Contains(typeof(T)))
      {
        Code<Serializer<T>> serializer =
        <|delegate s(T obj,
            BinaryWriter b,
            List<object> visited)
        {
          ~(map.Add(typeof(T), <|s|>));
          if(obj == null) b.Write(0);
          else
          {
            int index = visited.IndexOf(obj);
            if(index != -1) b.Write(index + 1);
            else
            {
              b.Write(-1);
              visited.Add(obj):
              ~SerializeFields(<|obj|>, <|b|>, <|visited|>);
            }
          }
        }|>
        typecast(serializer = map[typeof(T)]
          as Code<Serializer<T>>)
        return <(~serializer)(~obj, ~b, ~visited);>
      }
    }
  }
}

CodeVoid SerializeFields(Code obj,
Code<BinaryWriter> b,
Code<List<object>> lookup)
{
  CodeVoid result = <|;|>
  typecast<T>(tObj = obj as Code<T>)
  foreach(Field<T> fld in typeof(T).GetFields())
    result := Serialize(f.GetValue(obj), b, visited);
  return result;
The field `map` is used to store a data structure for memoizing generated code objects. Actual values stored in this dictionary will have type `Code<Serializer<T>>`, for various types T. Much of the code in `Serialize` is unchanged except for the use of `typecast` to perform type testing. A type cast is still required to down-cast the objects retrieved from `map` but this will always be inevitable without the support of type-indexed data structures.

```csharp
delegate void Serializer(object obj, BinaryWriter b);
static Code<Serializer> Make(Type typ) {
    typecast<T>(t = typ as Type<T>)
    return <|delegate (object anyObj, BinaryWriter b) {
        T obj = (T)anyObj;
        List<object> visited = new List<object>();
        ~Serialize(<|obj|>, <|b|>, <|visited|>);
    }|>;
}
```

The above code is used to generate a serializer for a type specified by a reflection type object. The generated function accepts the object to be serialized as an `object` and internally type casts it to its specific type. This is necessary if the function will be compiled and invoked by level 0 as there is no static type information at this level available about the type of object to serialize.

### 4.6.4 Conclusion

This section presented a design for staged reflective programming that was a combination of the designs presented in the previous Sections 4.4 and 4.5. This design introduces a new class hierarchy for the reflection API. It utilizes generic type inheritance to partially emulate the static existential type system (Section 4.5) but admits dynamic type checking (Section 4.4) in cases where this emulation breaks down. The hybrid approach does not introduce a heavyweight type system and only introduces one new language construct, which is simply a generalization of C#’s existing type cast construct.
In the hybrid approach type checking of staged reflection operations can occur at compile-time or generation-time, depending on how the programmer writes the program. If a meta program only uses staged reflection constructs that can be statically typed then any code that it generates will be type correct: hence conforming to the multi-stage type safety guarantee. It is expected that many useful staged reflective programs can be expressed using only the static components of this hybrid type-checking approach. Both the basic and recursive serializer algorithms from Section 4.3 have been expressed in this section using only statically-typed staged reflection operations. The same is also true for all of the example applications in the next section, with the exception of the domain-specific embedded language compiler which requires one dynamic type cast on a code object.

Because of the balance the hybrid approach achieves between language complexity (both design and implementation) and pragmatic usage when writing programs, it was chosen as the final approach adopted by the Metaphor compiler.

4.7 Results

This section describes a number of examples (other than the serializer example described thus far) that make use of the staged reflection capabilities of Metaphor. The approach taken to write these programs was to annotate their un-staged versions. This shows that the technique described in Section 3.2.5 is still effective for polytypic staged programming.

4.7.1 Deserializers

Presented here is the code for a deserializer generator that is the complement of the serializer generator presented in Section 4.6.3. It uses the simple (non-recursive type) algorithm. The staged function is modelled on an un-staged deserializer with method signature

\[
\text{void deserialize(ref } T \text{ obj, BinaryReader } b); 
\]

rather than the more usual signature
The former signature was chosen because it gives more efficient deserialization of value types as the latter signature produces code with unnecessary copying of temporary values. The code for the generator is

```
CodeVoid Deserialize(CodeRef obj, Code<BinaryReader> b)
{
    typecast(intObj = obj as CodeRef<int>)
    return |intObj = (~b).ReadInt32():|;
    else
    {
        CodeVoid result = |:|
        typecast<T>(tObj = obj as CodeRef<T>)
        foreach(Field<T> fld in typeof(T).GetFields())
            result := deserialize(field.GetValue(tObj), b);
        return result;
    }
}
```

The parameter `obj` has type `CodeRef` and refers to the location where the deserialized value should be stored.

The following function would be used to create a callable deserializer for a specified type.

```
delegate object Deserializer(BinaryReader b);

static Deserializer MakeDeserializer(Type typ)
{
    typecast<T>(t = typ as Type<T>)
    return delegate (BinaryReader b)
    {
        T obj = new T();
        |Deserializer(|obj|). |b|);
        return obj;
    }>.Run();
}
```

For the class declarations

```
struct Foo {
    int x;
    Bar y;
}
```
The invocation `MakeDeserializer(typeof(Foo))` would generate the code

```csharp
delegate (BinaryReader b) {
    Foo obj = new Foo();
    obj.x = b.ReadInt32();
    obj.y.z = b.ReadInt32();
    return obj;
}
```

If the alternate signature were used for the staged function (where the deserialized value is returned from the function instead of stored at a specified location) then the code generated would be

```csharp
delegate (BinaryReader b) {
    Foo obj = new Foo();
    obj.x = b.ReadInt32();
    Bar obj_1 = new Bar();
    obj_1.z = b.ReadInt32();
    obj.y = obj_1;
    return obj;
}
```

which involves the unnecessary creation of the temporary variable `obj_1`.

### 4.7.2 Staged polytypic functions

Serialization and deserialization are examples of polytypic algorithms as they are defined inductively on the structure of types. These examples are also members of a small number of 'pure' polytypic algorithms that perform some meaningful function on any defined type. Other members of this group are equality and cloning. Staged versions of these algorithms can also be written in Metaphor.

The equality algorithm compares objects of the same type, field by field, to determine their equality. A staged version of equality looks like this in Metaphor:

```csharp
Code<bool> Equals(Code x, Code y) {
    typecast(intx = x as Code<int>)
    typecast(inty = y as Code<int>)
```
return `<|~intx == ~inty|>`;
else throw new Exception("type mismatch");

typecast<T>(tx = x as Code<T>)
typecast(ty = y as Code<T>)
{
  Code<bool> result = `<|true|>`;
  foreach(Field<T> fld in typeof(T).GetFields())
    result = `<|~result && ~Equals(fld.GetValue(x), fld.GetValue(y))|>`;
  return result;
} else throw new Exception("type mismatch");

All pure polytypic algorithms have the same basic structure: a base case that performs a well-known operation for primitive types and a recursive case that applies the algorithm recursively for complex types. The `Equals` method is no different: in the base case the code objects `x` and `y` are tested to be integers and if successful compared using primitive integer comparison, otherwise a code object is built that equates each of the object’s fields in turn.

An unfortunate side-effect of the hybrid reflection type system is its inability to express the relationship between the variables `x` and `y`: i.e., that they are both code objects of the same unknown type. Because of this two `typecast`s (instead of one) are required to get the objects into a usable form: the first to open the implicit existential type and the second to assert both types are equal. The second `typecast` could possibly fail (e.g., in the call `Equals(<|1.0|>, <"a"|>)`) so error handling code is required.

This code would ideally be written with the static reflection type system using the `typeid` construct.

```csharp
Code<bool> Equals<T>(Code<T> x, Code<T> y)
{
  typeid(T is int) return `<|~x == ~y|>`;
else
{
  Code<bool> result = `<|true|>`;
  foreach<F>(Field<T,F> fld in typeof(T).GetFields())
    result = `<|~result && ~Equals(<|~x.-fld|>, <|~y.-fld|>)|>`;
  return result;
}
Here the universal quantification of the type $T$ on the method imposes that both $x$ and $y$ have the same type. The `typedef` construct can simultaneously convert both $x$ and $y$ to a target type.

### 4.7.3 Serialization with sub-typing

In the un-staged serializer the run-time or dynamic type of an object is available for the algorithm to use when serializing the object. However in the staged version only the static type is available, which may differ by a subtype to the dynamic type. Hence in the staged version objects may only be serialized up to their static type, which may be insufficient for some applications. In the code below, $B$ subclasses $A$ and adds a field $y$. If a serializer were generated on the type $A$ and used to serialize an instance of $B$, it would not serialize $B$’s field, $y$. This problem particularly arises when a class has a field of type $A$ but an object of type $B$ is assigned to it. To solve this problem the type of the object must be determined at run-time (stage 1) instead of at generation-time (stage 0).

```csharp
class A {
    int x;
}

class B : A {
    int y;
}

BinaryWriter b = ...;
MakeSerializer(typeof(A))(new B(), b);
```

In an object-oriented environment this problem would typically be dealt with by virtual methods. Hand-writing serializers for these classes would involve defining a virtual method on $A$ to perform serialization and overriding this method on $B$ to serialize any additional fields that $B$ defines.

```csharp
class A {
    int x:

    virtual void Serialize(BinaryWriter b) {
```
class B : A {
    int y;

    override void Serialize(BinaryWriter b)
    {
        base.Serialize(b);
        b.Write(y);
    }
}

In the statically-typed virtual machines of Java and the CLR it is not possible to define new members on a class at run-time. Staged serializers are not defined until run-time so it is not possible for them to dynamically add a new virtual method to an existing class, therefore another approach is required.

The approach adopted here is to use run-time type analysis to identify the type of an object being serialized. This type analysis requires the use of the reflection API (namely, the `GetType` method) but as far as the reflection API goes this is a fast method to call. Once an object’s type object is retrieved it is used to index into a map to get a delegate to a serialization function that will serialize the object using its dynamic type. If the type’s entry is not defined in the map then a new function is generated, compiled and stored there. This serializer generator uses memoization to store generated functions in their compiled form, rather than in code object form as was done previously. This is rather like a JIT compiler that compiles a method when it is first run and reuses the compiled code on subsequent runs. The purpose of memoization has also changed in this serializer. Previously, memoization was required to prevent infinite recursion and code explosion in the generator. In this serializer there is no recursion at the meta level (i.e., stage 0) since type analysis has been deferred to a later stage so to dynamically identify the types of objects. Instead memoization serves to cache and reuse compiled functions.

Although this algorithm uses the reflection API at run-time in an efficient manner it will nevertheless be much slower than the conventional object-oriented approach of
virtual method invocation. However with the restriction imposed by the underlying virtual machine, this is the most efficient way to implement dynamic serialization.

The code below shows a staged serializer capable of dynamic type identification.

delegate void Serializer(object obj,
    BinaryWriter b,
    List<object> visited);

class Persistence
{
    Dictionary<Type, Serializer> map =
        new Dictionary<Type, Serializer>();

    void Serialize(object obj,
        BinaryWriter b,
        List<object> visited)
    {
        if(obj == null) b.Write(0);
        else
        {
            int index = visited.IndexOf(obj);
            if(index != -1) b.Write(index + 1);
            else
            {
                b.Write(-1);
                visited.Add(obj);
                typecast<T>(t = obj.GetType() as Type<T>)
                {
                    b.Write(typeof(T).FullName);
                    if(!map.Contains(typeof(T)) map.Add(typeof(T),
                        delegate (object obj,
                            BinaryWriter b,
                            List<object> visited)
                        {
                            T tObj = (T)obj;
                            ~SerializeFields(<|tObj|>, <|b|>, <|visited|>);
                        }).Run());
                map[typeof(T)](obj, b, visited);
            }
        }
    }
}

CodeVoid Serialize(Code obj,
    Code<BinaryWriter> b,
    Code<List<object>> visited)
{
    typecast<intObj = obj as Code<int>>
    return <|(~b).Write(~obj);|>:
// else more primitive type cases
else typecast<T>(tObj = obj as Code<T>)
{
    if(typeof(T).IsValueType)
        return SerializeFields(obj, b, visited);
    else return Serialize(tObj, b, visited);
}

CodeVoid SerializeFields(Code obj,
    Code<BinaryWriter> b,
    Code<List<object>> visited)
{
    CodeVoid result = Serialize(obj, b, visited);
    foreach(Field<T> fld in typeof(T).GetFields())
        result :=
        Serialize(fld.GetValue(tObj), b, visited);
    return result;
}

The first Serialize method does not take any code parameters. Instead it can be
passed any object and it will identify its type and serialize it, generating code as nec-
essary. When serializing an object not contained in the visited set, the method calls
GetType to identify the run-time type of the object. The full type name of the object
is written to the output stream so that the deserializer will know what type to recre-
ate an instance of. If a compiled serialization function for the type has already been
memoized then it can be called immediately. Otherwise a new function is generated,
compiled and added to the memo table.

4.7.4 DSEL programming

This section gives an example of writing a compiler in Metaphor for a domain-spe-
cific language. The language is a minimal, functional, object-oriented language.

The abstract syntax for the DSEL is

\[ e \ ::= \ x \quad \text{variable} \]
\[ \mid \ e.f \quad \text{field access} \]
\[ \mid \ \text{new } T \quad \text{object creation} \]
\[ \mid \ (T) e \quad \text{type cast} \]
for types $T$, variable names $x$, field names $f$ and expressions $e$. Types are defined outside the language.

Assume there is a parser that parses the concrete syntax of this language and produces an abstract syntax tree (AST). The AST produced, along with methods on it that translate it into code objects are shown and discussed below.

```csharp
abstract class Expr
{
    abstract Code Compile(Dictionary<string, Code> env);
}
```

The abstract class `Expr` is the base class for all AST nodes and defines a virtual `Compile` method that takes an environment and returns a code object containing the translation of the expression into Metaphor code. The environment used is a dictionary that maps variable names to code objects. The type of the expression the AST node represents is not known statically, hence the use of the implicit existential code type. This return type can be viewed as a way of passing the inferred type of the expression back to the caller of `Compile`, as the caller can extract this type using type-cast.

The `Variable` class compiles a variable by looking-up its name in the environment. An error is raised if the variable's name is not found.

```csharp
class Variable : Expr
{
    string name;

    override Code Compile(Dictionary<string, Code> env)
    {
        if(env.Contains(name)) return env[name];
        else throw new Exception("Variable not found.");
    }
}
```

The `FieldAccess` class compiles a field access. Compiling the expression in the field access infers the type of object the field is being accessed on. This class uses type reflection to find a matching field by name on the inferred expression type (represented
by the type variable \( T \). If the field is found then the reflection field object is used to build code otherwise an error is raised.

```csharp
class FieldAccess : Expr
{
    Expr expr;
    string field;

    override Code Compile(Dictionary<string,Code> env)
    {
        typecast<T>(tExpr = expr.Compile() as Code<T>)
        foreach(Field<T> fld in typeof(T).GetFields())
            if(fld.Name == field) return fld.GetValue(tExpr);
        throw new Exception("Field not found.");
    }
}
```

The `CreateInstance` and `TypeCast` classes use the `Type.Parse` method to create a reflection type object from a string and with a `typecast` generate the returned code object.

```csharp
class CreateInstance : Expr
{
    string type;

    override Code Compile(Dictionary<string,Code> env)
    {
        typecast<T>(t = Type.Parse(type) as Type<T>)
        return new T();
    }
}
```

```csharp
class TypeCast : Expr
{
    string type;
    Expr expr;

    override Code Compile(Dictionary<string,Code> env)
    {
        typecast<T>(t = Type.Parse(type) as Type<T>)
        return (T)~expr.Compile();
    }
}
```

The `Function` class generates code for a function. Functions in the embedded language are single argument and may be recursive. The argument type and the return type and converted into type objects and then into the type variables \( T \) and \( U \). Once
the generation of the function has started the first thing to be done is add the function’s parameter and the function itself to the environment, after which the return expression can be compiled. The variables are removed from the environment when generation is complete.

delegate U Func<T>(T x);

class Function : Expr
{
    string funcName;
    string argName;
    string argType;
    Expr retExpr;
    string retType;

    override Code Compile(Dictionary<string,Code> env)
    {
        typecast<T>(t = Type.Parse(argType) as Type<T>)
        typecast<U>(u = Type.Parse(retType) as Type<U>)
        {
            Code<Func<T,U>> func = delegate f(T x) {
                env.Add(argName, x);
                env.Add(funcName, f);
                return Code<U> retExpr.Compile(env);}
            env.Remove(argName);
            env.Remove(funcName);
            return func;
        }
    }
}

The Apply class compiles a function application.

class Apply : Expr
{
    Expr func:
    Expr arg:

    override Code Compile(Dictionary<string,Code> env)
    {
        typecast<T,U>(tFunc = func.Compile(env) as Code<Func<T,U>>)
        typecast<T>(tArg = arg.Compile(env) as Code<T>)
        return Code<Code<T,U>>(-tFunc, -tArg);
    }
}
The result of compilation is a code object that can then be run or saved using the standard methods on code objects. The three phases of program processing: parsing, compiling and running can be combined into a single function. Below is a top-level 'eval' function that parses, compiles and runs a program in this example DSEL.

```csharp
object Eval(string source)
{
    Expr ast = Parse(source);
    Code code = ast.Compile(new Dictionary<string,Code>());
    return code.Run();
}
```

### 4.7.5 Performance data

The table below lists staged performance statistics for the serializer programs from this section. The table lists the un-staged execution time ($U$), the staged execution time ($S$), the time to generate, compile and JIT-compile the staged program ($G$), the speedup ($s$) and the break-even point ($n$). The tests were performed on an Intel Pentium 4 processor at 3 GHz clockspeed running Microsoft Windows XP and CLR 2.0.

<table>
<thead>
<tr>
<th>Program</th>
<th>$U$ (µs)</th>
<th>$S$ (µs)</th>
<th>$G$ (µs)</th>
<th>$s$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic serializer</td>
<td>3.0</td>
<td>0.11</td>
<td>940</td>
<td>25</td>
<td>330</td>
</tr>
<tr>
<td>Recursive serializer</td>
<td>3.9</td>
<td>0.60</td>
<td>2800</td>
<td>6.4</td>
<td>850</td>
</tr>
<tr>
<td>Sub-typing serializer</td>
<td>4.1</td>
<td>1.3</td>
<td>3800</td>
<td>3.2</td>
<td>1400</td>
</tr>
</tbody>
</table>

The basic serializer serializes values of the value type `Foo` below.

```csharp
struct Foo { Bar x: int y; }

struct Bar { int z; }
```

The remaining serializers (for reference types) serialize objects of type `Foo` below.

```csharp
class Foo { int x: Bar y; }

class Bar { Foo z: int w; }
```

The instances of `Foo` and `Bar` actually serialized are mutually recursive.
All examples underwent a reasonable speedup as a result of staging. This indicates the use of reflection API is a considerable portion of an un-staged polytypic program’s run-time. Staging the program eliminates this performance overhead and validates the idea of polytypic staged programming.

The speedup is less for the reference-type serializers because both the un-staged and staged versions need to manage the ‘visited’ objects table (to preserve object identity and guard against cyclic object graphs) which starts to dominate the time taken to traverse the object graph (which is what is actually being staged). The sub-typing serializer is slower again because of its run-time function lookup which further dwarfs the time spent in specialized code. The generation time for reference-type serializers is greater due to slow access to the hash-table used to memoize generated code for previously seen types.

The speedup results for staged serialization are typically better than the speedup results for non-reflective programs reported in Section 3.4.2. In this earlier section, the major code optimizations performed as a result of staging were unrolling loops and in-lining recursive functional calls: in other words, simplifying the branching structure of the code. These are fairly low-level optimizations and as such the CLR’s JIT compiler probably already does a fairly good job of compiling byte-code into optimized native code (e.g., by performing a limited form of unrolling and in-lining itself), hence undermining the optimizations made in the staged source program. The staged programs in this section optimize code branch structure too but they chiefly optimize reflection API method calls. Reflection API methods are not optimized by the JIT compiler and are inherently slow operations. Therefore polytypic or reflective programs benefit more than regular programs from staging on the CLR.

4.8 Conclusion

This section has investigated three approaches for integrating a C#/Java-style reflection API with multi-stage programming. This combination is important because the reflection API dynamically performs primitive language operations such as field ac-
cess and object creation. If these reflection actions are staged then it means a program can divide its use of the reflection API into two stages. The first stage performs expensive type analysis operations. Code generated for the second stage has direct access to objects without going through the reflection API, hence giving it a performance improvement. This means the use of reflection in a program can be partially evaluated away. A programmer can write code that uses reflection but at run-time specialized code is generated to perform the program’s functionality without the use of reflection.

Each approach in this section provided a varying degree of type safety. The dynamically-typed approach exposed the dynamic typing of the standard reflection API. This means that errors may occur at generation-time when reflection is used with staged code. This breaks the multi-stage programming ideal that all code generation errors should be detected at compile-time by a static type system. On the positive side the dynamically-typed approach is easy to design and implement in a language. It requires minimal extension to the language—only one new construct—and no extension to the type system.

The results of the static approach are the exact opposite. The statically-typed approach can prevent errors in staged reflective code at compile-time but is a considerable burden on the language design and implementation. The approach introduces two new language constructs (one to open existential types and another to compare type variables) but because the usage of these constructs is frequent and cumbersome there is a potential host of syntactic sugaring constructs possible to simplify the syntax (e.g., for iteration over collections of existentially-typed values). The most heavy-weight feature of the statically-typed approach is its existential type system. Although crucial to bringing about the static type safety of the reflection API, and hence staged reflection, this type system extension has a far-reaching influence on the language as a whole. At least without support from the CLR it was decided that implementing the existential type system merely to support staged reflection is infeasible.

The hybrid approach provides a balance between the simplicity of the dynamic approach and complexity of the static approach. The approach exploits a feature of parametric polymorphism in object-oriented programming to partially emulate existen-
tial types for free. This allows keeping the language design and implementation simple and enabling compile-time type checking of generated code in most cases. The approach added one new language construct, but this is more of an extension to C#'s existing type cast construct than an entirely new one. The hybrid approach is used in the actual implementation of Metaphor.

A common trait of all three approaches is the need for a construct that converts a reflection type object into an actual type: essentially an inverse of the `typeof` operator. The need for such an operation is obvious because a polytypic program must analyse a type using the reflection API but then use an actual type inside quotation brackets to generate the specialized code. This operation was a sticking point in the implementation of all approaches because there is no corresponding functionality in the CLR. The problem was successfully avoided in the hybrid approach by placing restrictions on how a type variable derived from a reflection type object can be used.
5 Conclusion

This thesis has investigated the design and implementation of a multi-stage programming language based on C#. The language is called Metaphor and is implemented as a compiler targeting the CLR. Multi-stage programming offers a way of reducing the performance penalty of excess abstraction through run-time code generation. Bringing multi-stage programming to a mainstream programming environment such as the CLR means that more programmers can exploit staged computation in their code, thus enabling the creation of high-performance, self-optimizing libraries, such as for scientific computing.

Section 3 saw the application of multi-stage programming theory to a C#-like language, resulting in the creation of the multi-stage language Metaphor. Previous work on multi-stage languages had been limited to the domain of functional languages; hence Metaphor is a new development in this regard. The object-oriented and imperative environment of C# presented some new challenges for multi-stage language design and these issues were discussed in detail in Section 3.3. From another angle, Metaphor also introduces a formally-sound, type-safe, semantics-based approach to code generation on modern object-oriented platforms: an improvement on existing approaches of meta programming and run-time code generation for this environment.

One way that the CLR and JVM run-time environments differ from those of functional languages is their support for type introspection: the ability to discover type information at run-time and dynamically manipulate objects. Since type introspection is an integral part of programming in modern object-oriented languages, its interaction with multi-stage programming is necessary to eliminate all forms of excess abstraction that a program may be using. A program that uses type introspection can improve its performance by run-time code generation through multi-stage programming: expensive type analysis is performed in the first stage to yield efficient specialized code in the second. This ability enables the staging (and hence run-time optimization) of polytypic algorithms such as serialization.
Section 4 examines how the CLR’s reflection API can be integrated with the multi-stage programming constructs of Metaphor. In doing so it is considered important to preserve, as much as possible, the static typing property of multi-stage languages which prevents code generation errors at compile-time. Three different approaches were considered that had varying degrees of static type safety. A fully-static type system was developed but considered too heavyweight to be practical for both the language user and language implementer. Ultimately a lightweight system that provided compile-time checking of code generation in most useful example applications was considered, on balance, to be superior.

The aim of this research is to investigate the interaction between multi-stage programming and object-oriented type introspection to enable the creation of staged, polytypic programs. This aim is successfully achieved through

1. the invention of the programming language Metaphor,

2. its demonstrated ability to express staged polytypic programs, and

3. the speedup in execution time gained by staging a polytypic program.

In the end Metaphor is a successful prototype language for multi-stage programming in an object-oriented environment. It is capable of staging regular algorithms and, more interestingly, polytypic algorithms that employ type analysis as part of their operation.

The Metaphor compiler and sample programs are available for download at http://www.fit.qut.edu.au/~neverov/.

5.1 Future work

The most significant open research problem currently facing multi-stage programming in imperative environments (including impure functional environments like OCaml) is that of scope extrusion (see Section 3.3.1). Finding a simple type system to
prevent scope extrusion that is capable of type inference and straightforward to implement would be a great gain for imperative multi-staged programming.

The crucial factor in the practical usefulness of any run-time code generation system is the speed at which it can generate code. Metaphor uses the CLR’s built-in run-time code generation (RTCG) API, Reflection.Emit, for its code generation needs. This API provides a high-level approach for generating CIL (the byte-code of the CLR). Using a more direct method of emitting CIL would improve the efficiency of the code generator and the practical usefulness of Metaphor. Interestingly, the RTCG API being too abstract for optimal performance is exactly the problem that multi-stage programming solves. An ideal solution would be a staged RTCG library that provides a similarly high-level interface as Reflection.Emit but generates code for the fast emitting of a CIL instruction stream. The Metaphor compiler would need to be bootstrapped to take advantage of this library however.
6 Bibliography


