Towards a Fully-reflective Meta-programming Language

Gregory Neverov **Paul Roe**

Centre for Information Technology Innovation,
Queensland University of Technology,
Brisbane, Australia
Email: {g.neverov,p.roe}@qut.edu.au

Abstract

The term *meta-programming language* is used to describe languages that have some capability for manipulating code. A *multi-stage language* is a kind of meta-programming language that allows static type-checking of dynamically generated code. The expressiveness and type-safety of multi-stage languages have led to their success in many applications that require code generation. This paper presents the design of a multi-stage language that is an extension to a traditional object-oriented language (e.g. C# or Java). The language has a static type system and allows types and code to be manipulated dynamically, hence giving it full reflection over the structure of a program. The language is discussed through a series of examples on run-time optimisation, serialiser generation and compiler construction. A prototype compiler for the language has been implemented which targets Microsoft's Common Language Runtime (.NET).

Keywords: Compilers, object-oriented systems, programming languages.

1 Introduction

Many software applications need to operate on programs as their input or output, e.g. compilers, interpreters, partial evaluators, instrumenters, optimisers, aspect-oriented systems, static and dynamic code generation, staged computation, domain-specific languages. For example, a traditional compiler reads a program as textual source code, converts the program into an intermediate representation (e.g. an abstract syntax tree) for type-checking and optimisation, and then generates code for the program in some target language. An instrumenter in an aspect-oriented system loads a program, analyses and injects code at certain points and returns an instrumented program. An optimisation in a high performance application needs to generate and compile specialised code at run-time. All these applications require writing programs that manipulate a representation of another program at run-time – a concept we term as *program reflection*.

Program reflection is a difficult and error-prone task, so programmers build abstractions to make the task simpler and easier. These abstractions vary in sophistication from programming languages (often called meta-programming languages) to software libraries to ad hoc approaches. Program reflection can be divided into four kinds determined by the operation being performed, either generation or analysis, and the part of the program being operated on, either types or code. For example a compiler for an object-oriented language needs to generate types and code for the source program being compiled and also analyse types in referenced modules to type check their use in the current program. A wide range of approaches exist for making program reflection tractable but each approach has limitations on the kinds of program reflection it can express and on the degree in which it aids software development.

Underpinning the process of program reflection is the recognition that the programs being manipulated are most often *typed*. The advent of virtual machine platforms has made typed languages the prevalent way of representing programs. So much so that Java and C# are strongly-typed languages in both their source and compiled (i.e. byte-code) forms. Given this ubiquity of typed languages we would like to be able to preserve the type correctness of a program throughout program reflection.

1.1 Contribution

In this paper we discuss the language Metaphor (Neverov & Roe 2004) – a novel meta-programming language for modern object-oriented platforms that provides a strongly-typed system for program reflection. The language is an application of research on multi-stage languages (Taha 1999) to an imperative, object-oriented environment. It is an extension of traditional multi-stage languages in that it treats types as well as code as first-class values. The language covers all four kinds of program reflection: code and type generation and analysis. The fully reflective nature of Metaphor allows any Metaphor program to be generated as the output of another Metaphor program.

To be a powerful system for program reflection the language satisfies these criteria:

Uniform The same environment is used for all kinds of program reflection. This simplifies applications that require more than one kind of program reflection.

High-level Operates on the level of a source language based on C# or Java.

Expressive Program reflection operations can be expressed clearly and concisely using the lightweight syntax of quasi-quotation and pattern matching.

Reduces errors The multi-stage type system prevents many errors in a code generator at compile time.

The ultimate goal of Metaphor is to improve the ease at which program reflection can be done in an object-oriented, virtual machine environment. Making program reflection a less prohibitive technique will allow it to be more frequently used by applications which can benefit from it. A compiler for the language is under ongoing development.

1.2 Organisation

Section 2 gives a review of various methods for code generation that are related to the idea of program reflection and the language Metaphor as presented in this paper. Section 3 gives a description of the Metaphor programming language. The language is explained through two examples, one for run-time optimisation of a function and another for dynamically generating code for a serialiser. Section 4 discusses how the language can be used as an approach for writing compilers and Section 5 gives another example of a toy object-oriented compiler written in Metaphor. Finally Section 6 gives a brief overview of the current state of development of the Metaphor compiler.

2 Related work

2.1 Partial evaluation

Partial evaluation (Jones, Gomard & Sestoft 1993) is a powerful program optimisation technique. Given a program some of its inputs are marked as being static (and remaining inputs are consequently dynamic). The first phase of partial evaluation is to analyse the program with its marked static inputs and determine what parts of the program can be completely executed if only that static input is known. The output of this phase is an annotated program that separates the original program into two stages. The first stage can be executed with only the static input, and the second stage can be executed with the remaining dynamic input. The second phase of partial evaluation uses the annotated program to specialise the original program for actual values of the static input. The output of this phase is a residual program which accepts the remaining dynamic input of the original program. The performance of the residual program is greater than that of the original program because some computation has been performed statically outside of the program. Typically, the residual program will be executed a number of times to amortise the cost of its generation.

Tempo (Consel, Hornof, Marlet, Muller, Thibault & Volanschi 1998) and JSpec (Schultz & Consel n.d.) are partial evaluators for C and Java respectively. Fabius (Leone & Lee 1998) and DyC (Grant, Mock, Philipose, Chambers & Eggers 2000) are dynamic compilation frameworks for ML and C. These systems allow programs to use partial evaluation at run-time as a run-time optimisation technique.

2.2 Staged languages

Staged languages have grown out of the languages used to express annotated programs in a partial evaluation system. Staged languages are programming languages that allow the programmer to explicitly specify the separation of stages in a program. Staged languages are either two-stage, meaning a program can be separated into a static and dynamic stage, or multi-stage, meaning a program can be separated into any number of stages of execution. Staged languages are advantageous to partial evaluation because they allow the programmer finer and direct control over the staging of a program. It is difficult for a partial evaluator to automatically determine the best staging of a program because of the complexity of program analysis required.

Staged languages are typically created by extending an existing general-purpose language. Staged languages provide a quasi-quotation syntax for constructing code values. Quasi-quotation allows the programmer to specify staged code in the language's concrete syntax instead of writing verbose code that constructs AST nodes. In this way writing code to generate an expression is just as easy as writing the expression itself. Staged languages typically only handle code generation.

MetaML (Taha & Sheard 1997) and MetaOCaml (Calcagno, Taha, Huang & Leroy 2003) are multi-stage extensions to the functional languages ML and OCaml. These languages are statically typed and use their type systems to encode semantic information about the code they manipulate. The multi-stage type system prevents the programmer from producing code that is ill-typed. The basic goal of the type system is to ensure that every type-correct program can at run-time only generate type-correct code. Many errors in code generating applications are caused by obvious mistakes in the code generator that can be detected statically. The type system's static guarantee about the correctness of a code generator significantly reduces these errors.

`C (Engler, Hsieh & Kaashoek 1996) is a two-stage extension to C. It inherits C's weak type system but allows for more expressive code generation than can be achieved with a statically typed language. DynJava (Oiwa, Masuhara & Yonezawa 2001) and a code generating dialect of Cyclone (Hornof & Jim 1999) are two-stage extensions to Java and Cyclone. These languages have a static type system that prevents errors in code generation but are less expressive than MetaML and MetaOCaml (even within two stages).

2.3 Template meta-programming

Meta-programming languages also exist that generate code at compile-time instead of run-time. They are often called *templates* or *macros*.C++ uses a template language to generate class declarations at compile-time. Template Meta Haskell (Sheard & Peyton Jones 2002) uses Haskell as a static meta-language for generating and analysing Haskell code. Template Meta Haskell exposes code through quasi-quotation and an AST data model, and allows more expressive code generation than C++ templates.

2.4 Code generation libraries

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 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). Type analysis is provided by both these platforms as part of the virtual machine. Commonly called *reflection* it allows a program to discover information about types at run-time. The reflection system in the CLR also extends to type and code generation. A number of libraries (BCEL (Dahm 1999), AbsIL (Syme 2002), PERWAPI (Corney 2003)) exist on both platforms that provide full program reflection but these libraries work on the

	Generation	Analysis
Code	partial evaluation, staged languages, RTCG libraries	partial evaluation, RTCG libraries
Types	RTCG libraries	run-time platform

Table 1: Summary of program reflection

low level of byte-code and 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.

2.5 Comparison

Table 1 summarises the current approaches to program reflection.

The design of Metaphor was heavily influenced by the multi-stage languages MetaML and MetaOCaml. Metaphor differs from other multi-stage languages because it –

- includes generation of types. Current multi-stage languages only handle the generation of code, not types.
- is an extension of an imperative, object-oriented language. Current multi-stage languages are extensions of functional languages. Aspects of multi-stage programming need to be recast in a different language setting.
- is implemented on an industrial virtual machine platform. The implementation can take advantage of, or needs to interact with features of this environment. E.g. memory management, serialisation, support for run-time type analysis and code generation.

Metaphor also raises the level of abstraction of current practices of program reflection in object-oriented virtual machines. In maintains a familiar type reflection system but enhances it with static typing to prevent errors. It also operates on the level of a source language as opposed to byte-code.

3 Programming in Metaphor

Metaphor is a multi-stage programming extension to a subset of C#. The language includes common object-oriented constructs such as classes, methods, and fields, as well as general imperative programming constructs for assignment and control flow. Metaphor extends the base language with types that correspond to different parts of a program (types, fields, code) and constructs for manipulating values of these types for the generation or analysis of programs. Types are manipulated at run-time by an approach similar to the type reflection system in Java and the CLR but with more static type information. The run-time representation of code is known as a *code object*. Code objects have a corresponding *code type* and are created by enclosing the syntax for a piece of code inside code quoting brackets. Metaphor uses the three classic staging constructs from MetaML: brackets `<|x|>`, escape `~x` and run `x.Run()` for manipulating code objects.

The class of programs that can be represented as values in Metaphor is equal to the class of programs that can be written in Metaphor – i.e. any program that a programmer could write directly, can also be created as a run-time code value. The property that any program can be generated by another program demonstrates the *multi-stage* nature of Metaphor.

The usage of Metaphor's staging features is best explained through examples.

3.1 Run-time optimisation

The classic example from staged computation and partial evaluation is the power function – i.e. calculating x^n for a positive integer n. The result is calculated by iterating over n and multiplying by x. This function is ideal for run-time optimisation because if the value of n is known then the loop it is used in can be unrolled.

Figure 1 shows the code for a *staged* power function – i.e. it produces specialised code for calculating x^n for a given value of n. The type `<|int|>` is a *code type* which is used to type code objects. The type inside the brackets `<|...|>` (in this case `int`) is the type of the code contained in the code object. This type can be any type in the language including another code type. If a code object has the code type `<|T|>`, it means that if the code contained in the code object were run it would produce a value of type `T`.

Line 1 defines the signature of the power function. The first parameter `x` is an `int` code object and the dynamic input to the function. The second parameter `n` is the static input that will be used to specialise the function. The function returns another `int` code object that will contain the code `x` repeated `n` times separated by multiplications. Line 2 declares a local variable `a` of `int` code type and initialises it to a code object containing the integer literal 1. Code objects are constructed by quoting a piece of code with brackets `<|...|>`. Whatever code is between the brackets will be contained by the resultant code object. The variable `a` is used as an accumulator to build up a code object inside the while loop. Line 4 assigns to `a` a new code object that contains the code contained in the old value of `a` multiplied by the code contained in `x`. The tildes are the escape staging operator and are used to splice the contents of a code object into the contents of a surrounding code object. Consequently the escape operator can only appear inside brackets.

Line 9 declares a new delegate[1] type `Int2Int` that maps `int`s to `int`s. The `Main` method will at run-time generate code for a function that calculates x^3 and invoke this function. It does this by first building a code object for the function in lines 11-14. The delegate type `Int2Int` is used to define an anonymous method, similar to the anonymous methods in C#. Inside the escape operator on line 13, the `Power` method is called passing the formal parameter of the anonymous method as the code to use to build the multiplication chain. The `int` code object returned from the call to `Power` is spliced as the return expression for the anonymous method. When execution passes line 14 the local variable `codeCube` will have the following value.

```
<|delegate Int2Int(int x) {
  return 1*x*x*x;
}|>
```

A code object of type `<|T|>` has two methods available on it.

```
T Run();
void Save(string name);
```

The `Run` method compiles the code object into CIL[2], executes that code and returns the result of execution. The `Save` method compiles the code into CIL and saves the compiled code to disk with the specified file `name`. The saved code is in a CLR module and can be used from any CLR application (e.g. ones written in C#, VB.NET., etc.). Line 15 calls the `Run` method on `codeCube` to produce a delegate that refers to the compiled anonymous method. This delegate is then invoked on line 16.

[1]Delegates are a common feature of the CLR and are essentially named function types.

[2]CIL is the intermediate byte-code language used by the CLR.

```
static <|int|> Power(<|int|> x, int n) {
  <|int|> a = <|1|>;
  while(n > 0) {
    a = <|~a * ~x|>;
    n = n - 1;
  }
  return a;
}
delegate int Int2Int(int x);
static void Main() {
  <|Int2Int|> codeCube =
    <|delegate Int2Int(int x) {
      return ~Power(<|x|>, 3);
    }|>;
  Int2Int cube = codeCube.Run()
  Console.WriteLine(cube(2));
}
```

Figure 1: Staged power function

3.2 Type system

The multi-stage type system is designed to prevent errors in code generation by tracking the use of staging constructs. The type system used in Metaphor is based on the type system used in MetaML.

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.

```
<|string|> x = ...;
<|Math.Abs(~x)|> // 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 has an associated *stage*. The stage 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 in stage 0 is executed and code in stage 1 and higher is generated. The `Run` method makes a transition between stages by starting execution of a code object at stage 0.

Every variable has an associated declaration stage and use stage. If the use stage is greater than the declaration stage then the variable will have a run-time value when it is used in generated code and therefore the value of the variable is emitted in the generated code rather than a reference to the variable. For example,

```
int x = 2; // x decl at stage 0
<|int|> y = <|x + 1|>; // x use at stage 1
```

When this code executes the code 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*.

The use stage being equal to the declaration stage is the standard case and nothing special happens here. If the use stage is less than the declaration stage, 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 in stage 1 and used in stage 0 but the variable doesn't exist in stage 0 and so is effectively out of scope.

```
<|<|int|> x = ...; // x decl at stage 1
  int y = ~x;|> // error: x use at stage 0
```

3.3 Serialiser generator

Metaphor provides a type reflection system similar to the reflection system used in .NET and Java. The type reflection system can be used with the staging constructs to generate code for object creation, member access and type casting expressions where the types and members involved in those expressions are not known statically.

Similar to .NET and Java there are types that represents types (or classes), methods and fields. However these types are parameterised to contain more static information about the type, method or field they refer to. In the below code, `t` is a type object representing the type `Foo`. The variable `f` could be used to store a field object for the field `x` on `Foo`. The type `Field<Foo, int>` means that the field exists on the type `Foo` and the type of the field is `int`.

```
class Foo { int x; }
Type<Foo> t = typeof(Foo);
Field<Foo, int> f;
```

Figure 2 shows the code for a serialiser generator. The code uses type analysis to analyse the fields of a given type and generates code that will serialise objects of that type. The `Serialise` method is generic method with a type parameter `A` that represents the type to generate a serialiser for. The method's value parameter `<|int|>` holds the code containing the object to be serialised. On line 2 the `typeif` construct is used to test the value of a type variable. The following block is executed if the variable `A` is the type `int`; otherwise the else block is executed. In the true branch of the `typeif`, the variable `obj` has type `<|int|>` as all occurrences of `A` have been replaced by `int`. Objects of type `int` are serialised by calling a primitive serialisation function `WriteInt`.

If `A` is not an integer, then it will be serialised by recursively serialising each field belonging to it. Line 5 declares a local variable `result` which will be used as an accumulator. The value of `result` is initialised to the empty statement. Lines 6-7 retrieve an array of reflection field objects representing the fields on the type `A`. All the field objects in this array are valid fields on the type `A`, but each field has a different field type. This information is expressed in the rank-2 existential type (Pierce 2002) of the local variable `fields`, which uses the type variable `B` to abstract the varying field types in the array.

On line 9, `fields[i]` has type `exists B.Field<A,B>`, i.e. a field object from the class `A` where the type of the field is not known. Informally this type could be thought of as `Field<A,?>`, where the `?` indicates missing type information. The `open` statement takes an existentially typed expression (such as `fields[i]`) and binds a new type variable to represent the missing type information. The `open` statement on line 9, assigns the value of `fields[i]` to `field` and binds a new type variable `B` that represents the field type of the field object under consideration. The variable `field` does not have an existential type but instead has a regular type `Field<A,B>`, where `B` is a valid type variable that can be used in the scope of the `open` statement.

In line 10, `~obj` splices in the code contained in `obj` (this code has type `A`) and `.~field` accesses the field referred to by `field`. This field is accessible on the type `A` and returns the type `B`, so this code expression is type correct and produces a code object of type `<|B|>`. The tilde in `.~` is used to signify that the expression to the right of the `.` is an expression that evaluates to a reflection field object rather than an identifier, which would be expected for an ordinary field access. Line 11 recursively calls `Serialise` passing the type of the field, `B`, and the code to access

```
 1: static <|void|> Serialise<A>(<|A|> obj) {
 2:   typeif(A is int) {
 3:     return <|WriteInt(~obj);|>;
 4:   } else {
 5:     <|void|> result = <|;|>;
 6:     (exists B.Field<A,B>)[] fields
 7:       = typeof(A).GetFields();
 8:     for(int i = 0; i < fields.Length; i++) {
 9:       open<B>(Field<A,B> field = fields[i]) {
10:         <|B|> fv = <|~obj.~field|>;
11:         <|void|> code = Serialise<B>(fv);
12:         result = <|~result; ~code;|>;
13:       }
14:     }
15:     return result;
16:   }
17: }
```

Figure 2: Serialiser generator

```
exists A.Type<A> MakeVector<T>(int n) {
  return <|class Vector {
    T[] array;
    Vector() { array = new T[n]; }
    T GetItem(int i) { return array[i % n]; }
  }|>
}
```

Figure 3: Type generation

this field, `fv`. The produced code is combined with `result` on line 12.

For the types

```
class Foo { int x; Bar y; }
class Bar { int z; }
delegate void Serialiser(Foo obj);
```

the code

```
<|delegate Serialiser(Foo obj) {
  ~Serialise<Foo>(<|obj|>);
}|>;
```

will evaluate to

```
<|delegate Serialiser(Foo obj) {
  WriteInt(obj.x);
  WriteInt(obj.y.z);
}|>
```

3.4 Type generation

To generate types, class declarations can appear inside brackets. The level of expressivity in type generation that can be currently achieved in Metaphor is similar to what can be done with C++ templates. (N.B. Metaphor is used to generate types at run-time, as opposed to C++ which generates types at compile-time.) Figure 3 shows a function that produces a type for a vector specialised for a particular element type `T` and size `n`.

4 Writing a compiler

Code objects in Metaphor are useful as an intermediate code representation for compilers. A compiler front-end can be written to generate a code object for the program being compiled. The compiler can take advantage of existing source-to-source transformation libraries to optimise or instrument the code. Finally a compiler back-end could be written that transforms the code value into a specific target language or the default Metaphor back-end can be used to generate code on the platform Metaphor is implemented in (in this case the CLR).

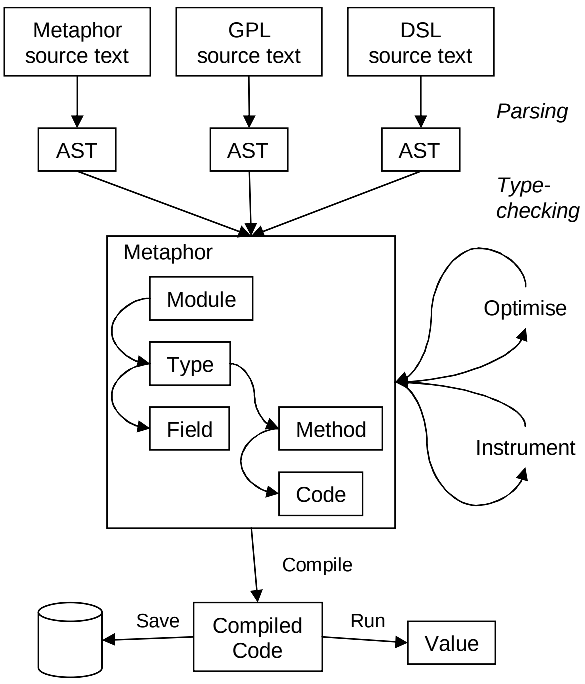

Figure 4: Overview of compiler structure

The conventional approach to writing a compiler in a multi-stage language is by staging an interpeter for the language being compiled (Sheard, Benaissa & Pasalic 1999). This approach constructs a compiler based on the semantics of the language which have been encoded as an interpreter. Compilers constructed in this manner can be easier to produce and are less prone to errors. Compilers written in Metaphor can also make use of its type generation capabilities. This will allow staging techniques to be used to produce compilers for a larger class of languages, namely languages with programmer-definable types.

A typical structure of a compiler is shown in Figure 4. The language to compile could be Metaphor itself, a general-purpose language (e.g. C#, Java) or a domain-specific language. The first phase is to write a parser for each language that transforms textual source code into a language-specific abstract syntax tree (AST). The next phase is type checking which analyses the AST for semantic errors and possibly annotates it with additional information. These two phases do not use any of the extended features of Metaphor and thus could also be written in C#. The final phase translates the AST into a code object which is the common intermediate format that all the languages compile to. This translation introduces more type information into the intensional representation of the program. The AST is an untyped representation in the sense that the type of an AST node does not convey the type of the code that it holds. The code object (which is essentially another AST) however does contain this extra type information. Hence this phase is responsible for converting an untyped program tree into a typed one. A lot of type checking on the source language is performed as a side-effect of the AST's translation in to a code object. Depending on the semantic difference between the source language and Metaphor, the initial type checking phase on the AST may be redundant.

Representing a program as a code object is better than using code generation libraries because of the stronger static typing Metaphor provides. When the compiler itself is type checked it guarantees that the compiler will not generate invalid code. Type checking errors when the compiler is compiled will most

$$\begin{array}{lll}
\text{Types} & T & \\
\text{Stmt} & S ::= & T\ x;\ S \mid E_1 := E_2; \mid \mathsf{while}\ E\ S \\
\text{Expr} & E ::= & x \mid E.f
\end{array}$$

Figure 5: Syntax

$$\begin{array}{lll}
\llbracket T\ x;\ S \rrbracket & \to & T\ x\ \texttt{= new}\ T\texttt{();}\ \ \llbracket S \rrbracket \\
\llbracket E_1 := E_2; \rrbracket & \to & \llbracket E_1 \rrbracket\ \texttt{=}\ \llbracket E_2 \rrbracket\texttt{;} \\
\llbracket \mathsf{while}\ E\ S \rrbracket & \to & \texttt{while(}\llbracket E \rrbracket\ \texttt{!= null)}\ \llbracket S \rrbracket \\
\llbracket x \rrbracket & \to & x \\
\llbracket E.f \rrbracket & \to & \llbracket E \rrbracket\texttt{.}f
\end{array}$$

Figure 6: Semantics

likely reveal errors in the compiler's type-checker's implementation. Of course it is still possible for the compiler to generate code that is valid but does not implement the semantics of the source language.

5 Example compiler

In this section we give an example of writing a compiler in Metaphor for a toy object-oriented language.

5.1 Specification

The abstract syntax for the toy language is shown in Figure 5. The language is stratified into types, statements and expressions. Types are simply class names that refer to types located elsewhere (e.g. in an external module). There are three statements: local variable declaration, assignment and while loop; and two expressions: variable and field access, where f is the name of the field.

The semantics of the language are shown in Figure 6 as a translation to C# or Metaphor code. The translation $\llbracket A \rrbracket \to B$ translates a toy language term A into the C# term B. A local variable declaration declares a new local variable and initialises it using its type's default constructor (assuming it exists). A while loop loops over its body until its condition expression evaluates to `null`. The translation of the remaining constructs is standard.

5.2 Implementation

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

The AST node `MyType`[3] is used to represent a type. This node contains a `Compile` method that produces a reflection type object for the type referred to by `name`. The implementation of this method would typically search one of more referenced assemblies for a type with a matching name and raise an error if no match is found.

```
class MyType {
  string name;
  exists A.Type<A> Compile();
}
```

The helper class `Env` is used as an environment that maps variable names to code objects. The abstract base classes `Stmt` and `Expr` are AST nodes for statements and expressions respectively. They each have a virtual `Compile` method that takes an environment and returns a code object containing the translation of the statement or expression. Statements always compile to code with no return value and so this method returns `<|void|>`. Expression however compile to code that does return a value but the type of this value is not known statically, hence the use of an existential code type. The existential return type can be viewed as a way of passing the inferred type of the expression back to the caller of `Compile`. Expressions potentially can be used on the left-hand side of an assignment, so `Compile` must also produce a code object with a `ref` code type. A `ref` code type is the same as a regular code type except that it can be spliced into the left-hand side of an assignment. In an AST for a larger language, expressions would be separated into assignable (left-hand) and regular (right-hand) expressions so that all expressions do not have to compile to a `ref` code type.

```
class Env {
  void Add(string name, exists A.<|A|> var);
  exists A.<|A|> Lookup(string name);
}
class Stmt {
 <|void|> Compile(Env env);
}
class Expr {
  exists A.<|ref A|> Compile(Env env);
}
```

The `Compile` method for each kind of AST node can be implemented by almost directly following the translation in Figure 6. The `Declaration` class compiles a local variable declaration by compiling the variable's type, constructing code for an initialised local variable, adding the variable to the environment and splicing the code from a recursive call to `Compile`.

```
class Declaration : Stmt {
  MyType type;
  string name;
  Stmt stmt;
  <|void|> Compile(Env env) {
     open<A>(Type<A> code = type.Compile())
     return <|A x = new ~code();
       ~{env.Add(name, <|x|>); stmt.Compile(env)}|>;
   }
}
```

Type-checking of the toy language is performed as a corollary of its compilation into a code object. The only construct of the toy language that needs type-checking is assignment – i.e. the left-hand and right-hand sides of an assignment must be the same type. In the `Assignment` class both sides of an assignment are compiled and the type variables `A` and `B` are used to refer to the inferred types of the assignment's sides. The `typeif` construct tests equality of these two types and returns a code object for the assignment if they are equal. If the equality test fails then there is a type error in the source toy program which should be handled in some way.

```
class Assignment : Stmt {
  Expr lhs;
  Expr rhs;
  <|void|> Compile(Env env) {
    open<A>(<|ref A|> rhsCode = rhs.Compile(env))
    open<B>(<|ref B|> lhsCode = lhs.Compile(env))
    typeif(A is B) return <|~lhsCode = ~rhsCode;|>;
    else ... //handle type error
   }
}
```

The `While` class compiles a while loop.

```
class While : Stmt {
  Expr cond;
  Stmt stmt;
  <|void|> Compile(Env env) {
    open<A>(<|ref A|> condCode = cond.Compile(env))
    return <|while(~condCode != null)
```

[3] The class is called `MyType` to disambiguate it from the Metaphor type `Type`.

```
      ~stmt.Compile(env);|>;
   }
}
```

The `Variable` class compiles a variable by looking-up its name in the environment. The `Lookup` method would raise an error if the variable's name is not found.

```
class Variable : Expr {
  string name;
  exists A.<|ref A|> Compile(Env env) {
    return env.Lookup(name);
  }
}
```

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 field by name on the inferred expression type (represented by the type variable `A`). If the field is found then the reflection field object is used to build code otherwise an error is raised.

```
class FieldAccess : Expr {
  Expr expr;
  string name;
  exists A.<|ref A|> Compile(Env env) {
    open<A>(<|ref A|> code = expr.Compile(env))
    open<B>(Field<A,B> field =
      typeof(A).GetField(name))
    if(field != null) return <|~code.~field|>;
    else ... //error field not found
  }
}
```

The result of compilation is a code object which can then be run or saved using the standard methods on code objects. The three phases of program processing: parsing, compiling and running (or saving) can be combined into a single function. The below function parses, compiles and runs a toy language program.

```
void Execute(string source) {
  Stmt ast = Parse(source);
  <|void|> code = ast.Compile(new Env());
  code.Run();
}
```

6 Implementation

The Metaphor language is in the process of being implemented as a compiler. The code generation and type analysis aspects of the language have been implemented. A basic template approach to type generation as described in this paper has also been implemented. Code analysis is currently being designed and implemented.

For future work we plan develop are more expressive form of type generation where a type can be built compositionally out of its components instead of as a single template. With the completion of work on code analysis, all four kinds of program reflection will be able to be expressed thus achieving a fully-reflective meta-programming language. Currently the Metaphor compiler is written in C#. We would like to be able to produce a bootstrapping compiler in Metaphor using the compiler construction method discussed in this paper.

The current Metaphor compiler is available for download at
http://sky.fit.qut.edu.au/~neverov/metaphor.

7 Conclusion

We have presented the idea of program reflection – the ability for a program to operate on other programs at run-time as first-class values. In an object-oriented virtual machine, program reflection amounts to being able to dynamically generate and analyse types and code. Program reflection is useful for many applications that deal with code in some way, e.g. compilers, run-time optimisation, etc. We have presented the programming language Metaphor that performs program reflection in an expressive, high-level and statically type-safe manner. The language improves the level of abstraction of current practices for code generation in Java and .NET, which are based on software libraries. The language is an expansion of multi-stage languages in that treats types as well as code as values that can be manipulated at run-time.

References

Calcagno, C., Taha, W., Huang, L. & Leroy, X. (2003), Implementing multi-stage languages using ASTs, Gensym, and Reflection, *in* 'Generative Programming and Component Engineering'.

Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S. & Volanschi, E.-N. (1998), 'Tempo: specializing systems applications and beyond', *ACM Computing Surveys* **30**(3es).

Corney, D. (2003), 'PE Reader/Writer API'. **URL:** *http://www.citi.qut.edu.au/research/-plas/projects/cp_files/pefile.jsp*

Dahm, M. (1999), Byte code engineering, *in* 'Java-Informations-Tage', pp. 267–277.

Engler, D. R., Hsieh, W. C. & Kaashoek, M. F. (1996), `C: A language for high-level, efficient, and machine-independent dynamic code generation, *in* 'Symposium on Principles of Programming Languages', pp. 131–144.

Grant, B., Mock, M., Philipose, M., Chambers, C. & Eggers, S. J. (2000), 'DyC: an expressive annotation-directed dynamic compiler for C', *Theoretical Computer Science* **248**(1–2), 147–199.

Hornof, L. & Jim, T. (1999), Certifying compilation and run-time code generation, *in* 'Partial Evaluation and Semantic-Based Program Manipulation', pp. 60–74.

Jones, N., Gomard, C. & Sestoft, P. (1993), *Partial Evaluation and Automatic Program Generation*, Prentice Hall International.

Leone, M. & Lee, P. (1998), 'Dynamic specialization in the Fabius system', *ACM Computing Surveys* **30**(3es).

Neverov, G. & Roe, P. (2004), Metaphor: A multi-stage, object-oriented programming language, *in* 'Generative Programming and Component Engineering'.

Oiwa, Y., Masuhara, H. & Yonezawa, A. (2001), DynJava: Type safe dynamic code generation in Java, *in* 'JSST Workshop on Programming and Programming Languages', Tokyo.

Pierce, B. (2002), *Types and Programming Languages*, MIT Press.

Schultz, U. & Consel, C. (n.d.), 'Automatic program specialization for Java'.

Sheard, T., Benaissa, Z.-E.-A. & Pasalic, E. (1999), DSL implementation using staging and monads, *in* 'Domain-Specific Languages', pp. 81–94.

Sheard, T. & Peyton Jones, S. (2002), Template metaprogramming for Haskell, *in* M. Chakravarty, ed., 'ACM SIGPLAN Haskell Workshop 02', ACM Press, p. 116.

Syme, D. (2002), 'Abstract IL'.
URL: *http://research.microsoft.com/projects/-ilx/absil.aspx*

Taha, W. (1999), Multi-Stage Programming: Its Theory and Applications, PhD thesis, Oregon Graduate Institute of Science and Technology.

Taha, W. & Sheard, T. (1997), Multi-stage programming with explicit annotations, *in* 'Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997', New York: ACM, pp. 203–217.