

Metaphor: A Multi-stage, Object-Oriented Programming Language

Gregory Neverov and Paul Roe

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

Abstract. This paper presents a language (called Metaphor) for expressing staged programs in a strongly-typed, imperative, object-oriented environment. The language is based on a subset of C[#] or Java; it is multi-stage and provides static type checking of later stage code. Object-oriented frameworks usually offer a type introspection or *reflection* capability to discover information about types at run-time. Metaphor allows this reflection system to be incorporated into the language's staging constructs, thus allowing the generation of code based on the structure of types – a common application for code generation in these environments. The paper presents the language, gives a formal description of its type system and discusses a prototype implementation of the language as a compiler targeting the .NET Common Language Runtime.

1 Introduction

Multi-stage programming languages [1–4] take the programming task of code generation and support it as a first-class language feature. In a *typed* multi-stage language, type correctness guarantees that all code generated by a meta-program will be type safe. This typing simplifies programming and reduces errors.

Most research to date in the field of typed multi-stage languages has been done in a functional setting. In particular MetaML [1] has proved to be a successful language for expressing staged computation. In this paper we apply the theory and practice of languages like MetaML to mainstream object-oriented languages such as Java and C[#]. In doing so we can improve the level of abstraction and type-safety offered for run-time code generation in these object-oriented programming environments.

Modern object-oriented run-time environments such as Java and .NET offer run-time introspection of types (called *reflection*), and dynamic code generation and loading. Code generation libraries that provide a higher level of abstraction than raw bytes or strings are also quite popular on these platforms (e.g. the .NET framework's Reflection.Emit library and numerous third party libraries available for both Java and .NET.) These code generation libraries use reflection to refer to types in the code they are generating. To be able to generate the same sort

of code as one of these libraries, an object-oriented, multi-stage language needs to incorporate reflection into its staging constructs.

In this paper we present a small object-oriented language called Metaphor which supports multi-stage programming and reflection on types. Multi-stage programming allows the run-time generation of code. Reflection allows the run-time analysis of types. In combination they enable a programmer to write statically-typed programs that dynamically generate code based on the structure of types.

The reflection system in Java or C# is dynamically typed; indeed it is often used as a means of performing dynamically-typed operations in the midst of the statically-typed language. If such a reflection system were allowed to interact with staging constructs, then its dynamic typing would destroy the static typing of the multi-stage language. I.e. it would be possible to write a type correct meta-program that generates ill-typed code via reflection. We present a solution to this problem by introducing a type system that statically types reflection operations and therefore prevents them from generating ill-typed code.

Writing code induced by the structure of types is known as generic or polytypic programming [5–7]. Many programs object-oriented programmers write or generate use polytypic code, e.g. object serialisation/deserialisation, value equality of objects, object cloning (deep-copying) and enumerators or iterators for collections. Metaphor allows a programmer to define code-generating functions for these operations that are type-safe and produce code at run-time. Most existing approaches for generating code for these operations either only work at compile-time (e.g. templates, macros) or do not statically-type the generated code (e.g. library-based run-time code generation).

The contributions of this paper are:

1. the design and implementation of a typed, multi-stage programming language extension to a simple object-oriented language;
2. the integration of an object-oriented reflection system with the language's staging constructs such that a programmer can use the reflection system in the type-safe generation of code.

The paper is organised as follows. Section 2 discusses the basics of staging and reflection, and how they interact. Section 3 gives a detailed overview of the language's new constructs and how its type system is used to prevent staging errors. Section 4 gives an example of writing a serialiser generator in Metaphor. Section 5 gives a formal description of the language's type system, and Section 6 gives an account of the current implementation.

2 The Language

We introduce a small object-oriented language in the style of Java or C# that supports multi-stage programming and reflection on types. The base language includes common object-oriented constructs such as classes, methods, and fields, as well as general imperative programming constructs for assignment and control flow.

2.1 Staging

The base language is extended with the three staging constructs: brackets, escape and run – as used in MetaML [1]. These staging constructs enable the statically type-safe, run-time generation of code. The classic staged program – the power function – may be written thus:

```
class PowerExample {
  static <|int|> Power(<|int|> x, int n) {
    <|int|> p = <|1|>;
    while(n > 0) {
      p = <|~p * ~x|>;
      n = n - 1;
    }
    return p;
  }
}

delegate int Int2Int(int x);

static void Main() {
  <|Int2Int|> codePower3 =
    <|delegate Int2Int(int x) {
      return ~Power(<|x|>, 3);
    }|>;
  Int2Int power3 = codePower3.Run()
  Console.WriteLine(power3(2));
}
}
```

Instead of calculating the value of x^n , this power method generates code that will calculate x^n for a given value of n . The brackets `<|` and `|>` quote statements or expressions and create *code objects*. A code object of an expression has type `<|A|>`, where `A` is the type of the underlying expression; a code object of a statement has type `<|void|>`. Code objects are first-class values that can be manipulated and passed and returned from methods. The type system is designed to prevent ill-typed code from being produced. The escape construct (`~`) splices the value of a code object into a surrounding code object. The `Run` method compiles and executes a code object and returns the result of its execution.

The use of higher-order function values in staged programming is a common idiom. A staged program typically needs to produce code for a function rather than code that just computes a single value (like an integer). Therefore Metaphor supports C[#]-like *delegates* and *anonymous methods*. Delegates are essentially named function types. An anonymous method is defined by specifying a delegate type, a list of method parameters and a method body.

The `Main` method generates the code for a function that computes x^3 and invokes it. The call to `Power` returns the code object `<|1*x*x*x|>`. The `Main` method defines a anonymous method using the delegate type `Int2Int` (which

maps integers to integers) and splices this code object as the body of the anonymous method.

2.2 Reflection in Java and .NET

The Java and .NET frameworks include a *reflection* system that allows a program to discover information about its types at run-time. Types can be reified into objects, which a program can then manipulate like any other object. Such objects have type `Type`. Reflection has two typical uses: firstly, to dynamically perform actions on types or members when it is not possible to do so statically and secondly, for use with a code generation library (e.g. .NET's `Reflection.Emit`) to describe types and members in generated code.

The `Type` type has methods that provide information on the type it describes, e.g. name, base class, list of defined fields and methods, etc. Fields and methods are also reified into objects of the `Field` type and `Method` type respectively.

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

The above C# code shows how to use reflection to dynamically access the field `i`. `typeof(A)` reifies the type `A`. `GetField` gets a reified object for the field named `i`. `GetValue` gets the value of the field on a new instance of `A`. As can be seen reflection is dynamically typed.

2.3 Reflection and Staging

It is desirable to combine reflection with staging so that a staged program may use type analysis to control the generation of code. For example, this is needed in a serialiser generator that uses reflection to get a list of fields on an object and then generates code that accesses those fields. To support this we introduce a new construct called field splice which is written `.%`. This construct is similar to a field access except that its right operand is an expression that evaluates to a `Field` value, instead of an identifier. For example:

```
class A { int i; int j; }
Type t = typeof(A);
Field f = t.GetField("i");
<|new A() .%f|>
```

This program generates code that accesses a field on an instance of `A`, but the field to access is computed at run-time. There are two typing problems with this code expression.

1. The type system must statically prevent the generation of ill-typed code. In this code expression the type-checker does not know the run-time value of

`f` and so cannot check that it is a valid field on the type `A`. If `f` describes a field from some other type then this code expression will generate ill-typed code.

2. The code expression must be assigned a code type but the type of the field being accessed is not statically known.

These problems exist because the underlying reflection system is dynamically-typed and so does not maintain the type-safety of the language’s statically-typed staging constructs. To solve these problems in *Metaphor* we have enhanced the type system so that it can statically type-check the use of reflection in a program.

3 The Reflection Type System

We introduce a new type system for reflection that constrains the use of reflection in a staged program such that future stage code is type-safe. For simplicity we will only consider the reflection of types and fields. Methods can be handled as an extension to fields, i.e. fields with function types.

The reflection type system consists of –

1. general constructs that produce values of reflection types;
2. a set of reflection types;
3. staging constructs that consume values of reflection types in the generation of code.

An overview of how these parts fit together in the language is given in Figure 1.

3.1 Reflection Types

The new reflection types are parameterised over types to carry more information about the expressions they are assigned to.

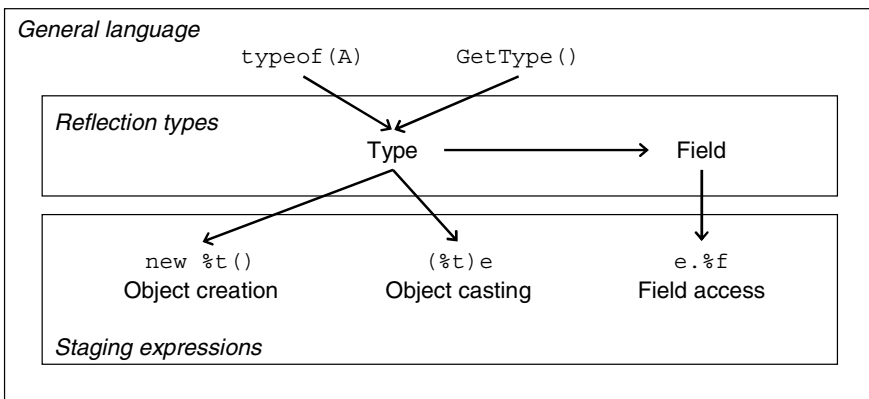


Fig. 1. Overview of reflection/staging language extension

The `Type` type takes one type parameter, i.e. `Type<A>`. There is only one value that has the type `Type<A>` and that is the value that reifies the type `A`. `A` could be a constant type (like `int`, `string`, etc.) or it could be a type variable. Metaphor uses existentially-quantified type variables to encapsulate the dynamic typing of the original reflection system. Hence the type `exists A.Type<A>` is the sum of all `Type` types and is equivalent to the unparameterised `Type` type in Java or .NET.

The `Field` type takes two type parameters, i.e. `Field<A,B>`. The first parameter, `A`, is the type the field is defined on. The second parameter, `B`, is the type of the field itself. Similarly, the type `exists A,B.Field<A,B>` is equivalent to the unparameterised `Field` type.

3.2 Reflection Expressions

A value involving the `Type` type can be introduced in three ways:

Context	Expression	Type
Static	<code>typeof(A)</code>	<code>Type<A></code>
Semi-static	<code>o.GetType()</code>	<code>exists B <: A.Type<A></code> <code>where o :: A</code>
Dynamic	<code>GetType(s)</code>	<code>exists A.Type<A></code> <code>where s :: String</code>

Static. The `typeof` operator produces the `Type` value for a given type. The value of this expression is known statically and these expressions are essentially the constants of the `Type` type.

Semi-static. The `GetType` method is available on every object and is used to find the object's dynamic (or run-time) type. The dynamic type must be a subtype of the static type of the expression. This information is conveyed in the type of `GetType` using bounded existential quantification. (Note: if the static type of `o` is a *sealed* type (i.e. cannot be inherited) then this case can become the static case, i.e. no existential quantification required.)

Dynamic. It is possible to produce a `Type` value through some arbitrary means, such as by parsing a string that contains a type name and returning the corresponding `Type` value. In this scenario no static information is available about the `Type` value returned from this expression. Hence unbounded existential quantification is used on the type variable `A`.

The type `Type<A>` contains a method named `GetFields` that returns an array of type `(exists B.Field<A,B>) []`. The elements of this array are `Field` values that describe the fields of `A`. The array is homogeneous in `A`, because the fields are all defined on the same type, but is heterogeneous in `B`, because the field type of each field is unknown and is in general different for each field. Existential quantification is used once again to express this information in a type.

Typically a programmer uses many ways to extract fields from types, e.g. getting a field by name or getting all fields of a particular type. These field operations can be given a type in the reflection type system and be implemented

using the single primitive `GetFields` method. For example, the type `Type<A>` might be declared like so:

```
class Type<A> {
    exists B.Field<A,B>[] GetFields();
    exists B.Field<A,B> GetFieldByName(string name);
    Field<A,B>[] GetFieldsWithType<B>(Type<B> type);
}
```

A Note on the Invariance of the Type Type. The use of bounded existential quantification instead of subtyping is needed because the `Type` type is invariant, i.e. `Type<A> ≤ Type` iff `A = B`. If the `Type` type was covariant it would be possible to access a field of a derived class on an object of a base class. E.g., assuming `B < A`, then value `typeof(B)` could be given the type `Type<A>`. Via this type, the fields returned by `typeof(B).GetFields()` would have type `exists C.Field<A,C>`, which indicates they are available on the supertype `A` when they may only be available on the subtype `B`.

3.3 Miscellaneous Expressions

Open. As the language uses existential types, there needs to be a way to *open* existentially-typed values, e.g.

```
open(t = GetType(s) as Type<A>) { ... }
```

Here the existentially-typed expression `GetType(s)` is bound to the variable `t` having the non-existential type `Type<A>`. The scope of `t` and the new type variable `A` is limited to the following statement block. There is no explicit *pack* operation because existential values can only be created by built-in reflection operations.

Typeif. Metaphor also provides a construct to scrutinise the value of a type variable, e.g.

```
typeid(A is int) { ... }
```

If the value of `A` matches the specified type (in this case `int`) then the `typeid` block is evaluated. Inside this block the type variable `A` is substituted with the type that was matched.

3.4 Staging Expressions

Types and fields are used in the syntax of code in three places: accessing a field on an object, creating a new object and casting an object. To make the link between the reflection and staging systems, three new staging constructs are added to consume reflection values and produce these three kinds of code. The syntax for

each construct is the syntax for the underlying unstaged construct embellished with a % symbol. Each construct is used to generate code for field access, object creation or object cast when the field or type to use in the code is not known until run-time. The constructs are illustrated in the following example.

```
class Foo { int i; }
class Bar {}
```

```
Type<A> t;
Field<A,B> f;
Type<C> u;
```

```
<|A|> x1 = <|new %t()|>; // staged object creation
<|B|> x2 = <|~x1.%f|>; // staged field access
<|C|> x3 = <|(%u)~x2|>; // staged object casting
```

In this example A, B and C are type variables. `x1` is assigned code that creates a new object, where the type of the object to create is defined by the value of `t`. `x2` is assigned code that accesses a field determined by `f`. This code expression is the example from the beginning of this section but now properly typed. Finally, `x3` is assigned code that makes a cast to a type determined by `u`.

Suppose that at run-time the type variables A, B and C are bound to the types `Foo`, `int` and `Bar`, respectively. Also suppose that `t` is bound to the `Type` value for `Foo`, `f` is bound to the `Field` value for `Foo.i` and `u` is bound to the `Type` value for `Bar`. Then when this code is executed the variables `x1`, `x2` and `x3` will have the following values:

```
x1 == <|new Foo()|>
x2 == <|new Foo().i|>
x3 == <|(Bar)new Foo().i|>
```

These code values statically (or literally) refer to the types and fields that were described by the run-time values of `t`, `f` and `u`.

4 Serialiser Generator Example

Serialisation is the process of taking a value and converting it into a stream. An object is typically serialised by recursively serialising each of the object's fields. Writing a serialiser is a fairly mechanical process and therefore serialisers are most often generated rather than hand-written.

Serialiser generators can be compile-time tools that typically produce textual source code, which is then difficult to compose with other generated code. In a multi-stage language a serialiser generator produces a code object that is modular and robust, and can be safely incorporated into a larger code generating application. The need to dynamically generate serialisers arises in distributed application frameworks which provide an infrastructure for applications to communicate data over a network. The types of data to communicate are not known

until deployment or run-time and so the framework must dynamically generate serialisers for them.

Below is the code for a serialiser generator in Metaphor – it is a staged program that takes a reified type value and produces code that will serialise that type. The staged serialiser was created by taking an unstaged serialiser and adding staging annotations in the manner of a *staged interpreter* [8].

```
static <|void|> Serialise<A>(Type<A> type, <|A|> obj) {
  typeif(A is int) { return <|WriteInt(~obj);|>; }
  else {
    <|void|> result = <|;|>;
    (exists B.Field<A,B> [] fields = type.GetFields();
    for(int i = 0; i < fields.Length; i++) {
      open(field = fields[i] as Field<A,B>) {
        Type<B> ft = typeof(B);
        <|B|> fv = <|~obj.%field|>;
        <|void|> code = Serialise<B>(ft, fv);
        result = <|~result; ~code;|>;
      }
    }
    return result;
  }
}
```

The `Serialise` method takes two arguments: the type to be serialised and the code for an object to be serialised. The universally-quantified type parameter `A` is used to form a relationship between the types of the two arguments, i.e. the type of the object being serialised must match the type the serialiser was generated for.

For simplicity we only consider a single primitive type `int`, which is serialised by the function `WriteInt`. For a non-primitive type, the program opens each field in the `fields` array using the type variable `B` to represent the abstract type of the current field. `B` also provides enough typing information to be able to make the recursive call to `Serialise`. The variable `result` accumulates the code for the serialiser. The generated serialiser does not use reflection and is known at compile-time to be type correct. For simplicity this example does not handle recursive types or the possibility of null field values.

To be able to use the generated serialiser we need to create an “invoking” function that we can pass the value we want serialised to. If the type we are going to generate a serialiser for is statically known (i.e. is a compile-time constant) then we can create an invoking function like so:

```
delegate void FooSerialiser(Foo obj);

// generate a serialiser for the type Foo
<|FooSerialiser|> codeFooSerialiser =
  <|delegate FooSerialiser(Foo obj) {
```

```

    ~Serialise<Foo>(typeof(Foo), <|obj|>);
    return;
}|>;
FooSerialiser fooSerialiser = codeFooSerialiser.Run();

//invoke the serialiser
fooSerialiser(new Foo());

```

Here we generate a serialiser for the statically-known type `Foo`, where `fooSerialiser` is the invoking function. If the type `Foo` is defined as below, then at run-time `codeFooSerialiser` will be assigned the following code object.

```

class Foo { int x; Bar y; }
class Bar { int z; }

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

```

If however the type to be serialised is not statically known then the serialiser can only be invoked through a dynamically-typed invoking function, like so:

```

delegate void AnySerialiser(object obj);

// get type to serialise
exists A.Type<A> type = GetType();

// generate a serialiser for an arbitrary type
<|AnySerialiser|> codeAnySerialiser;
open(type = type as Type<A>)
    codeAnySerialiser =
        <|delegate AnySerialiser(object obj) {
            A castObj = (%type) obj;
            ~Serialise<A>(type, <|castObj|>);
            return;
        }|>;
AnySerialiser anySerialiser = codeAnySerialiser.Run();

// invoke the serialiser
anySerialiser(obj);

```

Since we don't know the type of object being serialised, `anySerialiser` must take an `object` as its parameter. `GetType` is a function that computes the type to generate a serialiser for. In `codeAnySerialiser`, `obj` is cast to type `A` which is an abstract representation of the type the serialiser is generated for. `castObj` has the correct static type to pass in the call to `Serialise`. Although the invoking

function is dynamically-typed, the code inside it is not (except for the initial cast). The purpose of the cast is to prevent an object of the wrong type from being serialised. E.g. if the run-time type of `obj` is not compatible with the type described by the value of `type`, then the cast will fail since it would not be meaningful to serialise that object with this serialiser. The invoking function cannot take a parameter of type `A` because that type variable cannot escape the scope of the open block.

If `type` has the value `typeof(Foo)` (defined as before), then `codeAnySerialiser` will be assigned the following code object.

```
<|delegate AnySerialiser(object obj) {
  Foo castObj = (Foo) obj;
  WriteInt(castObj.x);
  WriteInt(castObj.y.z);
  return;
}|>
```

5 Formalisation

We present a formal language to express the core functionality of Metaphor. The constructs of the language can be divided into three categories:

General. General programming language constructs for an object-oriented language, similar to object-oriented calculi such as Featherweight Java [9] and C# minor [10].

Staging. Typical meta-programming language constructs for staged computation [1].

Reflection. Constructs that link the reflection system of the object-oriented runtime environment with the staging constructs to achieve typed object-oriented program generation – the chief contribution of this paper.

5.1 Syntax

Figure 2 shows the syntax of the language. The over-line notation, \overline{X} , is used to denote a sequence of X 's, e.g. a class declaration contains a sequence of field declarations $A_1 f_1, A_2 f_2, \dots, A_n f_n$. A program consists of an expression and a class table, CT , which maps class names to class declarations. A class declaration consists of a class name and a collection of named fields. Methods can be emulated by fields with function types. The domain of types contains class types, type variables, polymorphic function types, existential types, code types and the reflection types `Type` and `Field`. Existential quantification is restricted to reflection types as only these types can be existentially quantified in Metaphor. Type variables can only be bound to class types and other type variables.

The general expressions of the language comprise of basic object-oriented expressions as well as a `typeid` for testing the value of a type variable and an `open` for opening an existentially-typed value. Staging expressions are the usual brackets, `escape`, `run` and an explicit lift operator. The lift expression `%E`

Class names	c		
Field names	f		
Variables	x, y		
Type variables	α		
Class decl	$C ::= \text{Class } c \{ \overline{A} \overline{f} \}$		
Types	$A, B ::= S \mid R \mid \forall \overline{\alpha}. (\overline{A}) \rightarrow B \mid \exists \alpha. R \mid \langle A \rangle$	General types	
	$R ::= \text{Type } S \mid \text{Field } S_1 S_2$	Reflection types	
	$S ::= c \mid \alpha$	Simple types	
Expressions	$E ::=$		
General		Staging	
$x \mid$	Variable	$\langle E \rangle \mid$	Brackets
$\text{fun } B \ y \langle \overline{\alpha} \rangle (\overline{A} \ x) \{ E \} \mid$	Func. def.	$\sim E \mid$	Escape
$E \langle \overline{S} \rangle (\overline{E}) \mid$	Func. app.	$\text{run } E \mid$	Run
$E.f \mid$	Field access	$\%E \mid$	Lift
$\text{new } c \mid$	Obj. creation	Reflection	
$(c)E \mid$	Obj. casting	$E_1.\%E_2 \mid$	Staged field access
$\text{typeof}(\alpha \text{ is } S) E_1 \text{ else } E_2 \mid$	Type test	$\text{new } \%E \mid$	Staged obj. creation
$\text{open}(\alpha \ x = E_1) \text{ in } E_2 \mid$	Exist. open	$(\%E_1)E_2 \mid$	Staged obj. casting
		$\text{typeof } S$	Type of

Fig. 2. Syntax

evaluates the expression E in the preceding stage and embeds the resulting value as a literal in the current stage.

The reflection expressions incorporate the reflection system into the generation of code. They generalise the unstaged expressions for field access, object creation and casting. Instead of using a literal field or type as in the unstaged case they use a reflection expression that evaluates to a field or type. These reflection expressions must be evaluated at the stage before the code they are used in is executed. Consequently they must appear inside staging brackets.

The `typeof` expression produces a `Type` value from a literal type. Other operations on reflection values such as getting fields from types can be typed as a function in the language, e.g.:

$$\text{getFields} :: \forall \alpha. (\text{Type } \alpha, \text{String}) \rightarrow \exists \beta. \text{Field } \alpha \ \beta$$

5.2 Type Rules

The type system for this formal language is expressed as type rules in Figure 3. The natural numbers m and n define the staging level that a term is typed at. The level of a term is equal to the number of brackets minus the number of escapes and lifts that surround it. The type judgments use two environments:

- Δ is the environment of type variables. It contains the type variables annotated with the level they are declared at. It is not valid to use a type variable at a level less than what it is declared at.

Environments

$$\Delta ::= \emptyset \mid \Delta, \alpha^n \quad \Gamma ::= \emptyset \mid \Gamma, x: A^n$$

Types

$$\frac{CT(c) = \text{Class } c \{\bar{A} \bar{f}\}}{\Delta \vdash^n c} \quad \frac{\alpha^m \in \Delta \quad m \leq n}{\Delta \vdash^n \alpha} \quad \frac{\Delta \vdash^n S}{\Delta \vdash^n \text{Type } S}$$

$$\frac{\Delta \vdash^n S_1 \quad \Delta \vdash^n S_2}{\Delta \vdash^n \text{Field } S_1 S_2} \quad \frac{\Delta, \bar{\alpha}^n \vdash^n \bar{A} \quad \Delta, \bar{\alpha}^n \vdash^n B}{\Delta \vdash^n \forall \bar{\alpha}. (\bar{A}) \rightarrow B} \quad \frac{\Delta, \alpha^n \vdash^n R}{\Delta \vdash^n \exists \alpha. R} \quad \frac{\Delta \vdash^{n+1} A}{\Delta \vdash^n \langle A \rangle}$$

General Expressions

$$\frac{x: A^n \in \Gamma}{\Delta; \Gamma \vdash^n x: A} \quad \frac{\Delta, \bar{\alpha}^n \vdash^n \bar{A} \quad \Delta, \bar{\alpha}^n \vdash^n B \quad \Delta, \bar{\alpha}^n; \Gamma, \bar{x}: \bar{A}^n \vdash^n E: B}{\Delta; \Gamma \vdash^n \text{fun } B \ y \langle \bar{\alpha} \rangle \langle \bar{A} \ x \rangle \{E\}: \forall \bar{\alpha}. (\bar{A}) \rightarrow B}$$

$$\frac{\Delta \vdash^n \bar{S} \quad \Delta; \Gamma \vdash^n E: \forall \bar{\alpha}. (\bar{A}) \rightarrow B \quad \Delta; \Gamma \vdash^n \bar{E}: \bar{A}[\bar{\alpha}/\bar{S}]}{\Delta; \Gamma \vdash^n E \langle \bar{S} \rangle \langle \bar{E} \rangle: B[\bar{\alpha}/\bar{S}]} \quad \frac{\Delta \vdash^n c}{\Delta; \Gamma \vdash^n \text{new } c: c}$$

$$\frac{\Delta; \Gamma \vdash^n E: c \quad CT(c) = \text{Class } c \{\bar{A} \bar{f}\} \quad f = \bar{f}_i}{\Delta; \Gamma \vdash^n E.f: A_i} \quad \frac{\Delta \vdash^n c \quad \Delta; \Gamma \vdash^n E: A}{\Delta; \Gamma \vdash^n (c)E: c}$$

$$\frac{\Delta \vdash^n \alpha \quad \Delta \vdash^n S \quad \alpha \neq S \quad \Delta - \{\alpha\}; \Gamma[\alpha/S] \vdash^n E_1: A \quad \Delta; \Gamma \vdash^n E_2: A}{\Delta; \Gamma \vdash^n \text{typeif}(\alpha \text{ is } S) E_1 \text{ else } E_2: A}$$

$$\frac{\Delta; \Gamma \vdash^n E_1: \exists \beta. A \quad \Delta, \alpha^n; \Gamma, x: A[\beta/\alpha]^n \vdash^n E_2: B \quad \Delta \vdash^n B}{\Delta; \Gamma \vdash^n \text{open}(\alpha \ x = E_1) \text{ in } E_2: B}$$

Staging Expressions

$$\frac{\Delta; \Gamma \vdash^{n+1} E: A}{\Delta; \Gamma \vdash^n \langle E \rangle: \langle A \rangle} \quad \frac{\Delta; \Gamma \vdash^n E: \langle A \rangle}{\Delta; \Gamma \vdash^{n+1} \sim E: A} \quad \frac{\Delta; \Gamma \vdash^n E: \langle A \rangle}{\Delta; \Gamma \vdash^n \text{run } E: A} \quad \frac{\Delta; \Gamma \vdash^n E: A}{\Delta; \Gamma \vdash^{n+1} \% E: A}$$

Fig. 3. Type Rules

Reflection Expressions

$$\begin{array}{c}
 \frac{\Delta \vdash^n S}{\Delta; \Gamma \vdash^n \text{typeof } S : \text{Type } S} \\
 \\
 \frac{\Delta; \Gamma \vdash^{n+1} E_1 : A \quad \Delta; \Gamma \vdash^n E_2 : \text{Field } A B}{\Delta; \Gamma \vdash^{n+1} E_1.\%E_2 : B} \\
 \\
 \frac{\Delta; \Gamma \vdash^n E : \text{Type } A}{\Delta; \Gamma \vdash^{n+1} \text{new } \%E : A} \\
 \\
 \frac{\Delta; \Gamma \vdash^n E_1 : \text{Type } A \quad \Delta; \Gamma \vdash^{n+1} E_2 : B}{\Delta; \Gamma \vdash^{n+1} (\%E_1)E_2 : A}
 \end{array}$$

Fig. 3. Type Rules (continued)

- Γ is the environment of regular variables. It contains the names of variables, their types and the level they are declared at. A variable can only be used at the same level it is declared at, but variables from earlier stages can be lifted to the current stage using lift.

The judgment $\Delta \vdash^n A$ means that under the environment Δ the type A is valid at level n . The judgment $\Delta; \Gamma \vdash^n E : A$ means that under environments Δ and Γ the expression E has type A at level n .

The type rules for general expressions are standard. The type rules for staging expressions are typical for typed multi-stage languages in the style of MetaML. Of particular interest is how these type judgments control the staging level, e.g. brackets increase the level and escape and lift decrease the level and so cannot be used at level zero.

In the `typeid` expression, the type variable α is tested for equality against the type S . If they are equal then the expression E_1 is evaluated, otherwise E_2 is evaluated. When typing the E_1 branch, α is removed from Δ and any occurrence of α in Γ is substituted with S .

The reflection expressions are typed like specialised versions of lift. The reflection expressions type their `Type` or `Field` sub-expression at one level lower than where the reflection expression occurs. The `Type` or `Field` sub-expression is also evaluated at an earlier stage and its value is embedded as a literal type or field in the current stage code. Thus these constructs are equivalent to the standard lift construct except that they lift types or fields instead of values.

6 Implementation

We have implemented a prototype compiler for Metaphor that targets the .NET Common Language Runtime. The compiler has a traditional structure divided into three phases: parsing, type checking and code generation. During the first phase the whole source program is parsed including code that lies inside code brackets. The result of the parse phase is a parse tree.

In the next phase, the parse tree is type-checked and transformed into an abstract syntax tree (AST). The AST generated here is important because it also serves as the internal representation of code objects at run-time.

During the code generation phase, the AST is traversed and target language code is emitted for each node in the tree. The code generator emits stage zero code as it would in a unstaged language compiler. The code generator emits stage one and higher code as code that rebuilds the AST. When this compiled code is executed it reconstructs part of the compile-time AST that it was generated from. The run operator is implemented by invoking this code generation phase again at run-time. At run-time, code is compiled into an in-memory code buffer using the .NET framework's support for run-time code generation. Code can be generated and executed in the same process that invoked its generation.

Cross-stage persistence is handled using a global lookup table of objects. Cross-stage persisted objects are added into this table and their table indices are used by compiled code to refer back to the objects. The address of an object cannot be emitted in the generated code because the memory managed environment of the .NET CLR may change the object's location.

Variables in staged code must be renamed to prevent accidental name capture or collision. For example in the code below, the variable `x` is renamed in the value of `c3` to avoid a duplicate local variable declaration.

```
<|int|> c1 = <|int x = 1; x|>;
<|int|> c2 = <|int x = 1; x|>;
<|int|> c3 = <|~c1 + ~c2|>;
```

Therefore when this code is executed the value of `c3` would be something like:

```
<|int x1 = 1; int x2 = 1; x1 + x2|>
```

The parameterised reflection types in the source program are erased in the compiled program. They are replaced by the non-parameterised versions of reflection types used by the .NET CLR. In effect the reflection type system is just a compile-time check that has no residue in compiled code.

Metaphor suffers from some well-known problems with the typing of multi-stage languages, such as the unsafety of the run operator and scope extrusion in the presence of imperative side-effects [11]. These problems are areas of active research and current solutions [12, 13] to them could be applied to Metaphor.

7 Related Work

7.1 Generic Programming

Generic programming involves writing *polytypic functions* that implement algorithms that are induced by the structure of a type. Hinze and Peyton Jones [6] 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 [7] 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 [14] is a static 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.

7.2 Typed Multi-stage Languages

The multi-stage programming extensions used in Metaphor were inspired from the language MetaML [1, 2]. MetaML is an extension to ML that supports typed multi-stage programming with cross-stage persistence. MetaML is a functional language and is implemented as an interpreter written in SML.

MetaOCaml [15] is a newer language that has evolved from MetaML. It differs from MetaML in that it safely handles ML’s side-effecting constructs with respect to staging and is implemented as a modified OCaml byte-code compiler.

7.3 Higher-Level Imperative Code Generation

A number of imperative programming systems for run-time code generation exist but differ from Metaphor in their balance of type-safety versus expressiveness. ``C` [16] is fast run-time code generation extension to C. It inherits C’s weak type system but allows for more expressive code generation, e.g. functions with a dynamically determined number of parameters.

A run-time code generation extension [17] to Cyclone [18], and DynJava [19] a similar extension to Java offer stronger typing than ``C` but lose expressiveness. In these languages code values cannot be passed or returned from functions (i.e. lambda-abstracted), have no “run” operator and are not multi-stage. However these languages do cover the full set of their base languages features.

CodeBricks [20] is a software library designed to facilitate code composition in the .NET framework. It exposes a high-level abstraction to the programmer for composing code at the compiled (byte-code) level. Like all code generation/manipulation libraries it cannot statically type later stage code.

8 Conclusion and Future Work

We have described a typed, multi-stage, object-oriented language called Metaphor. The language can express staged programming in object-oriented languages like C[#] or Java. The language has a statically-typed reflection system for run-time type analysis that is used in conjunction with its staging constructs. This allows the programmer to implement type-safe polytypic functions by means of run-time code generation, or in other words to generate code based on the structure of types.

We plan to extend the language with dynamic type generation, i.e. not only can a type be analysed using reflection but a new type can also be created.

With this feature a meta-program could generate a new type and use it in an object-program. The multi-stage type system will prevent types from being used before they are created. The formal calculus presented in this paper will also be extended with this functionality and a soundness proof undertaken for this complete type system.

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

References

1. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997, New York: ACM (1997) 203–217
2. Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999)
3. Sheard, T.: Accomplishments and research challenges in meta-programming. In Taha, W., ed.: Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01). (2001) Invited talk.
4. Sheard, T., Benaissa, Z., Martel, M.: Introduction to multi-stage programming using MetaML. Technical report, Pacific Software Research Center, Oregon Graduate Institute (2000)
5. Jansson, P., Jeuring, J.: PolyP – A polytypic programming language extension. In: ACM Symposium on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997, New York, ACM Press (1997) 470–482
6. Hinze, R., Peyton Jones, S.: Derivable type classes. In Hutton, G., ed.: Haskell Workshop, Montreal, Canada (2000)
7. Cheney, J., Hinze, R.: A lightweight implementation of Generics and Dynamics. In: Haskell'02, Pittsburgh, Pennsylvania, USA (2002)
8. Sheard, T., Benaissa, Z., Pasalic, E.: DSL implementation using staging and monads. In: Domain-Specific Languages. (1999) 81–94
9. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In Meissner, L., ed.: Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99). Volume 34(10)., N. Y. (1999) 132–146
10. Kennedy, A., Syme, D.: Transposing F to C#. In: Proceedings of Workshop on Formal Techniques for Java-like Programs, Málaga, Spain. (2002)
11. Calcagno, C., Moggi, E., Taha, W.: Closed types as a simple approach to safe imperative multi-stage programming. In: Automata, Languages and Programming. (2000) 25–36
12. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL'03), New Orleans, Louisiana, ACM Press, New York (NY), USA (2003)
13. Calcagno, C., Moggi, E., Taha, W.: ML-like inference for classifiers. In: Proceedings of the European Symposium on Programming (ESOP 2004). LNCS, Springer-Verlag (2004)
14. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M., ed.: ACM SIGPLAN Haskell Workshop 02, ACM Press (2002) 1–16

15. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, Gensym, and Reflection. In: *Generative Programming and Component Engineering*. (2003)
16. Engler, D.R., Hsieh, W.C., Kaashoek, M.F.: `C: A language for high-level, efficient, and machine-independent dynamic code generation. In: *Symposium on Principles of Programming Languages*. (1996) 131–144
17. Hornof, L., Jim, T.: Certifying compilation and run-time code generation. In: *Partial Evaluation and Semantic-Based Program Manipulation*. (1999) 60–74
18. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: *USENIX Annual Technical Conference, Monterey, CA, June 2002*. (2002)
19. Oiwa, Y., Masuhara, H., Yonezawa, A.: DynJava: Type safe dynamic code generation in Java. In: *JSST Workshop on Programming and Programming Languages, Tokyo* (2001)
20. Attardi, G., Cisternino, A., Kennedy, A.: CodeBricks: Code fragments as building blocks. In: *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (2003)