Supporting both Static and Dynamic Typing [1]

Francisco Ortin[2]

Computer Science Department
University of Oviedo
Oviedo, Spain

José Baltasar García Perez-Schofield[3]

Computer Science Department
University of Vigo
Ourense, Spain

Abstract

Dynamic languages are becoming increasingly popular for developing different kinds of applications such as adaptable and adaptive software, Web development, application frameworks, game engines, interactive programming, rapid prototyping, and dynamic aspect-oriented programming. These languages are built on the Smalltalk idea of supporting reasoning about (and customizing) program structure, behavior and environment at runtime. That is the reason why this trend is commonly referred to as *the revival of dynamic languages*.
Dynamism obtained by dynamic typing is, however, counteracted by two main limitations: early type error detection and runtime performance. To obtain the benefits of both dynamic and static typing, we have designed a programming language that provides both approaches. Following the *Separation of Concerns* principle, our programming language provides both dynamic and static typing. It is possible to customize the trade-off between runtime flexibility of dynamic typing and safety, performance and robustness of static typing. Moreover, the source code of the application stays unchanged. This paper presents an overview of the *StaDyn* programming language.

Keywords: Dynamic languages, separation of concerns, static and dynamic typing, type reconstruction, alias analysis.

1 Introduction

There is not a globally accepted definition of the term "dynamic language". Based on the features dynamic languages are held to have, it can be said that they are those languages that *support reasoning about and customizing their own structure, behavior and environment at runtime, supporting self-modifying features and dynamic code generation* [32]. Although these are also characteristics of scripting languages

[1] This work has been funded by *Microsoft Research*, under the project entitled *Extending dynamic features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution* Request for Proposals, 2006.

[2] Email: ortin@lsi.uniovi.es

[3] Email: jbgarcia@uvigo.es

[33] (both terms are commonly used indifferently), it is widely accepted that dynamic languages are an evolution of scripting languages that incorporate a number of technical advances such as support for modularization, object orientation, GUIs, or database access.

The main objective of dynamic languages is to model the dynamicity that is sometimes required in building high context-dependent software, due to the mobility of both the software itself and its users. Features such as meta-programming, reflection, mobility, dynamic reconfiguration and distribution are the domain of dynamic languages. This dynamism is, however, counteracted by the lack of static type checking, implying a considerable runtime performance penalty.

Dynamic languages have recently turned out to be really suitable for developing specific kinds of applications such as Web development, application frameworks, game engines, interactive programming, rapid prototyping, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. In the Web engineering area, Ruby [43] has been successfully used together with the Ruby on Rails framework for creating database-backed web applications [44]. This framework has confirmed the simplicity of implementing the DRY (*Don't Repeat Yourself*) [21] and the *Convention over Configuration* [44] principles with this kind of languages. Nowadays, JavaScript [15] is being widely employed to create interactive web applications with AJAX (*Asynchronous JavaScript And XML*) [10], and PHP (*PHP Hypertext Preprocessor*) is one of the most popular languages to develop Web-based views. Python [41] is used for many different purposes, being the Zope application server (a framework for building content management systems, intranets and custom applications) a good example [40]. Due to its small size, portability and ease of integration, Lua [23] has gained great popularity for extending games [24]. Finally, a wide range of dynamic aspect-oriented tools has been built over dynamic languages [38,4,43,31], being more flexible than the common static ones.

Due to the recent success of dynamic languages, usual static languages –such as Java or .Net– are gradually incorporating more dynamic features into its platforms. Taking Java as an example, the Reflection API became part of core Java with its release 1.1. This API offers introspection services to examine structures of object and classes at runtime, plus object creation and method invocation –having a substantial performance overhead. The Dynamic Proxy Class API was added to Java 1.3. It allows defining a class at runtime that implements any interface, funneling all its method calls to an `InvocationHandler`. In Java 1.6, the new Java Scripting API permitted dynamic scripting programs to be executed from, and have access to, the Java platform [25]. Finally the Java Specification Request 292 [26], expected to be included in Java 1.7, incorporates the new `invokedynamic` opcode to the Java Virtual Machine (JVM) in order to support the implementation of dynamically typed object oriented languages. Since the computational model of dynamic languages requires extending the JVM semantics, Sun Microsystems has launched the new Da Vinci project in January 2008 [13]. This project is aimed at prototyping a number of enhancements to the JVM, so that it can run non-Java languages, especially dynamic ones, with a performance level comparable to that of Java itself.

This trend has also been appreciated in the .Net platform. Although this platform was initially released with introspective and low-level dynamic code genera-

tion services, version 2.0 included *Dynamic Methods* and the `CodeDom` namespace to model and generate the structure of a high-level source code document. Finally, the *Dynamic Language Runtime* (DLR), announced by Microsoft in 2007, adds to the .Net platform a set of services to facilitate the implementation of dynamic languages [20].

The great flexibility of dynamic languages is, however, counteracted by limitations derived by the lack of static type checking. This deficiency implies two major drawbacks: no early detection of type errors, and a considerable runtime performance penalty. Static typing offers the programmer early detection of type errors, making possible to fix them immediately rather than discovering them at runtime; when the programmer's efforts might be aimed at some other task, or even after the program has been deployed [36]. Moreover, since runtime adaptability of dynamic languages is implemented with dynamic type systems, runtime type detection, inference and checking involves an important performance drawback.

This is the reason why the benefits of providing both typing approaches in the same language have been previously stated. Meijer and Drayton maintained that *instead of providing programmers with a black or white choice between static or dynamic typing, it could be useful to strive for softer type systems* [28]. Static typing allow earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance. Dynamic typing languages provide a solution to a kind of computational incompleteness inherent to statically-typed languages, offering, for example, storage of persistent data, inter-process communication, dynamic program behavior customization, or generative programming [1]. Hence, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [2]. That is, *static typing where possible, dynamic typing when needed* [28].

The main contribution of our work is to break the programmers' black or white choice between static or dynamic typing. Our programming language called *StaDyn* supports both static and dynamic typing separating the dynamism concern [22]. This programming language permits straightforward development of adaptable software and rapid prototyping, without sacrificing application robustness, performance and legibility of source code. The programmer may specify those parts of the code that need a higher adaptability (dynamic) and those that should guarantee a correct behavior at runtime (static). This separation permits turning rapidly developed prototypes into a final robust and efficient application. It is also possible to make parts of an application flexible, maintaining the robustness of the rest of the program.

In this paper, we describe an overview of our programming language that supports both dynamic and static typing. The rest of this paper is structured as follows. In the next section, we provide a motivation and background of dynamic and static languages. Section 3 describes the features of the *StaDyn* programming language and a brief identification of the techniques applied. Section 4 mentions the key implementation decisions and Section 5 discusses related work. Finally, Section 6 presents the ending conclusions and future work.

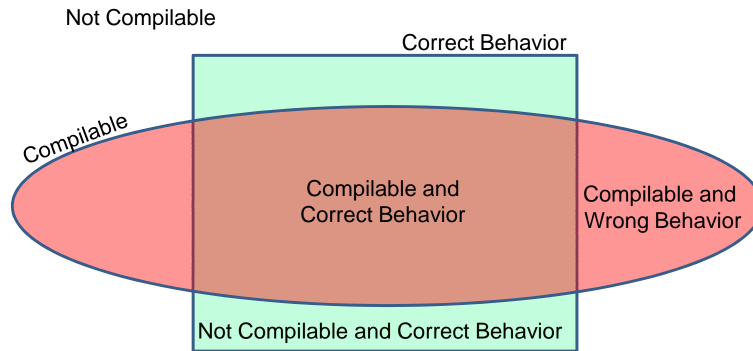

Fig. 1. Correct programs in a static language.

```
Object[] v=new Object[10];
for (int i = 0; i < 10; i++) {
    v[i] = "String " + i;
    int length = v[i].length();  // Compilation error
}
```

Fig. 2. Not compilable Java program that might behave correctly.

2 Motivation and Background

2.1 Static Typing Languages

A language is said to be *safe* if it produces no execution errors that go unnoticed and later cause arbitrary behavior [9]. Static languages ensure safety of programs by using type systems. However, these type systems do not compile some expressions that have a correct behavior at runtime (e.g. to pass a message called `m` to an object, the reference of that object must implement an interface that declares this public method). This happens because their static type systems require ensuring that only good expressions are typable. Figure 1 illustrates this situation.

Static typing is centered on making sure that programs behave correctly at runtime. This is the reason why languages with static typing employ a pessimistic policy regarding to program correctness. This pessimism causes compilation errors in programs that have a correct dynamic behavior. Java code shown in Figure 2 illustrates this limitation.

At the same time, static languages also permit the execution of programs that might cause the immediate stop of a running application, producing an execution error (e.g. array index out of bounds or null pointer exception).

2.2 Dynamic Typing Languages

The approach of dynamic languages is totally different. Instead of making sure that all correct expressions will be able to be executed, they make all syntactic correct programs compilable (Figure 3). This is a too optimistic approach that causes a lot of runtime errors that might have been detected at compile time. Dynamic languages do not apply any type checking at compile type, permitting too much wrong runtime behavior. They compile programs that might be identified as erroneous statically. The Visual Basic .Net source code in Figure 4 is an example of this too optimistic approach.

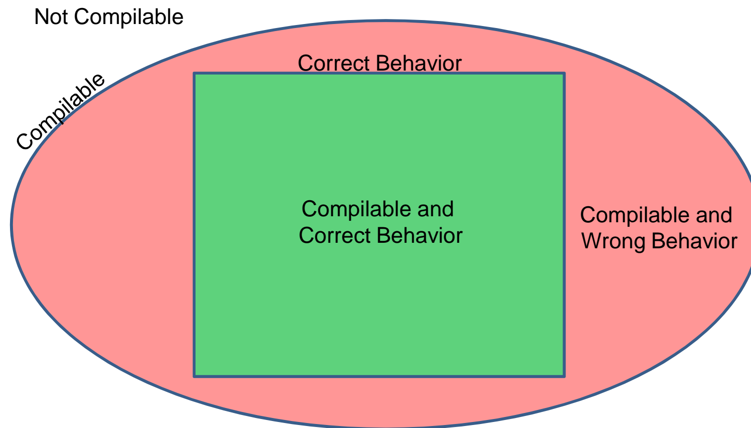

Fig. 3. Correct programs in a dynamic language.

```
Public Module MyModule
    Sub Main()
        Dim myObject
        Dim length As Integer
        myObject = New Object()
        length = myObject.length()  ' No compilation error
    End Sub
End Module
```

Fig. 4. Compilable Visual Basic program that behaves incorrectly.

2.3 Benefiting from both Approaches

The *StaDyn* programming language performs type inference at compile type, minimizing the *compilable and wrong behavior* region of dynamic languages (Figure 3) and the *not compilable and correct behavior* area of static languages (Figure 1). Consequently, *StaDyn* detects the compilation error of the dynamic program shown in Figure 4 (that Visual Basic does not detect) and compiles the correct static code in Figure 2 (that Java does not compile) –using the *StaDyn* syntax.

For both dynamic and static approaches, we use the same programming language, letting the programmer move form an optimistic, flexible and rapid development (dynamic) to a more robust and efficient (static) one. This can be done using the same source code, changing the compiler settings. We separate the concern of flexibility, robustness and performance, from functional requirements of the application.

3 The *StaDyn* Programming Language

In this section we describe the features of the *StaDyn* programming language, without detailing the techniques employed. Language features and a brief description of the techniques used are presented. Implementation issues are described in Section 4.

The *StaDyn* programming language is based on C# 3.0 [12]. Particularly, we extend the behavior of the new implicitly typed local references. In *StaDyn*, type of references can be explicitly declared, and it is also possible to use the `var` keyword to declare implicitly typed references. *StaDyn* permits the use of this kind of references

```
using System;
class Test {
    public static void Main() {
        Console.Write("Your age, please: ");
        var age = Console.In.ReadLine();
        Console.WriteLine("You are " + age + " years old.");
        age = Convert.ToInt32(age);
        Console.WriteLine(age.GetType());
        age++;
        Console.WriteLine("Happy birthday, you are " +
                              age + " years old now.");
        int length = age.Length; // * Compilation error
    }
}
```

Fig. 5. A reference with different types in the same scope.

wherever a type is used, whereas C# 3.0 only provides its use as local references. Therefore, `var` references in *StaDyn* are much more powerful that implicitly typed local variables in C# 3.0.

3.1 Multiple Types in the Same Scope

Static typing commonly forces a variable of type T to have the same type T within the scope in which it is bound to a value. Even languages with static type inference (type reconstruction) such as ML [30] or Haskell [19] do not permit the assignment of different types to the same polymorphic reference in the same scope.

This unique type assignment is not the common behavior in dynamic languages. These languages provide the use of one reference to hold different types in the same scope. This is easily implemented at runtime by a dynamic type system. However, our language should perform static type checking taking into account the concrete type of the reference. The *StaDyn* program shown in Figure 5 is an example of this capability.

The `age` reference in Figure 5 has different types in the same scope. Initially it is set to be a string, but later an integer is assigned to it. Correct static type inference is finally shown with the compilation error detected in the last line of code. Moreover, the best possible runtime performance is obtained because we do not need to use reflection to discover types at runtime.

To obtain this behavior we have developed a parametric polymorphism type system that provides type reconstruction when a `var` reference is used. We have implemented the Hindley-Milner type inference algorithm to infer types of local variables [29]. This algorithm has been modified to perform type reconstruction of `var` parameters and attributes (fields) –described in Section 3.4.

The unification algorithm provides parametric polymorphism, but it forces a reference to have the same static type in the scope it has been declared. To overcome this drawback we have developed a version of the SSA (*Single Static Assignment*) algorithm [11]. Since type inference Since type inference must be performed after the SSA algorithm, we have implemented this algorithm as an AST (*Abstract Syntax Tree*) transformation. The implementation of this algorithm follows the Visitor design pattern [17].

```
using System;
class Test {
    public static void Main() {
        Console.Write("Your age, please: ");
        var age0 = Console.In.ReadLine();
        Console.WriteLine("You are " + age0 + " years old.");
        age1 = Convert.ToInt32(age0);
        Console.WriteLine(age1.GetType());
        age2 = age1 + 1;
        Console.WriteLine("Happy birthday, you are " +
                              age2 + " years old now.");
        int length = age2.Length; // * Compilation error
    }
}
```

Fig. 6. Corresponding program before the SSA transformation.

```
var exception;
if (new Random().NextDouble()<0.5)
    exception = new ApplicationException("An application exception.");
else
    exception = new SystemException("A system exception");
Console.WriteLine(exception.Message);
```

Fig. 7. Static *duck typing*.

Figure 6 shows the corresponding program of applying the AST transformation to the source code in Figure 5. The AST represented by the source code in Figure 6 is the actual input to the unification algorithm.

3.2 Duck Typing

The static type system of *StaDyn* is *flow-sensitive*. This means that it takes into account the flow context of each `var` reference. It gathers *concrete* type information (opposite to classic *abstract* type systems) [37] knowing all the possible types a `var` reference may have. Instead of declaring a reference with an abstract type that embraces all the possible concrete values, a `var` reference infers the union of all its possible concrete type. Notice that different types depending on flow context could be inferred for the same reference by means of the type inference mechanism described in previous point.

Following this scheme, the *StaDyn* programming language offers *duck typing* at compile type. Duck typing (*if it walks like a duck and quacks like a duck, it must be a duck*) [43] means that an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether the objects have a related inheritance hierarchy or not. This is a powerful feature offered by dynamic languages.

The benefit provided by *StaDyn* is not only that it supports *duck typing*, but that this feature is provided statically. Whenever a `var` reference may point to a set of objects that implement a public `m` method, it can be safely invoked. These objects need not to implement a common interface or class with the `m` method. Since this analysis is performed at compile time, the programmer benefits from both early type error detection and runtime performance.

```
var exception;
Random random = new Random();
switch (random.Next(1,4)) {
case 1:
     exception = new ApplicationException("An application exception.");
     break;
case 2:
     exception = new SystemException("A system exception");
     break;
case 3:
     exception = "This is not an exception";
     break;
 }
 Console.WriteLine(exception.ToString());
 Console.WriteLine(exception.Message); // * Compilation error?
```

Fig. 8. Pessimistic behavior of static references.

Source code in Figure 7 shows this feature. The reference `exception` may point to an `ApplicationException` or to a `SystemException` object. Both objects have the `Message` property and, therefore, it is safe to use this property. It is not necessary to define a common interface or class to pass this message. Since type inference system is *flow-sensitive* and uses concrete types, the programmer obtains a safe duck-typing system.

Union types are the key technique we have used to obtain this concrete-type flow-sensitiveness [35]. Concrete types are obtained by the abovementioned unification algorithm. Whenever a branch is detected, a union type is created with all the possible concrete types inferred. Type checking of union types depends on the dynamism concern, which is explained in the following section.

3.3 Separation of the Dynamism Concern

StaDyn permits the use of both static and dynamic `var` references. Depending of this concern, type checking and type inference is more pessimistic (static) or optimistic (dynamic). Since the dynamism concern is not explicitly stated in the source code, *StaDyn* promotes the conversion of dynamic references into static ones, and vice versa.

Source code in Figure 8 adds another alternative in the assignment of the `exception` reference. The `ToString` message is correct because it is offered by the three possible objects created. However, the `Message` property depends on the level of dynamism the programmer requires. By default, the compiler uses the *everythingStatic* option, and the following error message is shown:

> *Error ErrorManagement.UnknownMemberError (Semantic error). 'Message': no suitable member found.*

However, we can be very optimistic and set all the `var` references in the program as dynamic. In this case, the compiler accepts a message if there is one possibility in which that program behaves correctly. The executable file is generated if we compile the program in Figure 8 with the *everythingDynamic* option.

Actually, we do not need to set all the `var` references in the program as dynamic. It is possible to specify the dynamism of each single reference by means of a XML

```
<?xml version="1.0" encoding="utf-8"?>
<application name="sample3">
  <namespace name="GettingStarted">
    <class name="Test">
          <method name="Main">
            <dynvar name="exception" />
          </method>
        </class>
    </namespace>
</application>
```

Fig. 9. Setting dynamic the `exception` reference.

```
var reference;
if (new Random().NextDouble() < 0.5)
    reference = "String";
else
    reference = 3;
Console.WriteLine(reference.Message);
```

Fig. 10. Optimistic behavior of dynamic references.

file. The XML document shown in Figure 9 is the `sample3.dyn` file that sets the exception reference as dynamic. Each source code file could have a corresponding XML document specifying its dynamism concern.

It is worth noting that setting a reference as dynamic does not imply that every message could be send. Static typing is still performed. The major change is that type checking is more optimistic. This shows how the dynamism concern implies a modification of the type checking behavior performed over union types. If the implicitly typed `var` reference inferred with a union type were static, type checking is performed over all its possible concrete types. However, if the reference were dynamic, type checking is performed over those concrete types that do not produce a type error; if none exists, a type error is shown.

Figure 10 shows this behavior. Even though this code is compiled with the *everythingDynamic* option, the compiler shows the following a static error:

> *Error ErrorManagement.NoTypeHasMember (Semantic error). The dynamic type '([Var(6)=6=string] ,[Var(5)=5=int])' has no valid type type with 'Message' member.*

This behavior shows how static typing is performed even in dynamic scenarios, providing better early type error detection. This lack in dynamic languages is shown in the Visual Basic code in Figure 4. *StaDyn* is capable of detecting the compilation error that Visual Basic does not find.

3.4 Implicitly-Typed Parameter and Attribute References

Concrete type reconstruction is not limited to local variables. *StaDyn* performs a global *flow-sensitive* analysis of implicit `var` references. The result is a powerful parametric polymorphism (generics) much more straightforward than the one offered by Java, C# (bounded) and C++ (unbounded) [8].

Implicitly-typed parameter references cannot be unified to a single concrete type. Since they represent any actual type, we cannot perform type inference the same

```
public static var upper(var parameter) {
    return parameter.ToUpper();
}
public static var getString(var parameter) {
    return parameter.ToString();
}
```

Fig. 11. Implicitly typed parameter references.

way as we did with local references. This necessity is shown in Figure 11. Both methods require the parameter to implement a specific method, returning its value. In the `getString` method, any object could be passed as a parameter because every object accepts the `ToString` message. In the `upper` method, the parameter should be any object able to respond to the `ToUpper` message. Depending on the type of the actual parameter, the compiler should generate the corresponding compilation error.

For this purpose we enhanced our type system to be constraint-based. Types of methods in our object-oriented language may have an ordered list of constraints specifying the set of restrictions that must be satisfied by the parameters. In our example, the type of the `upper` method is

$$\forall\alpha\beta.\alpha \rightarrow \beta|\alpha : Class\,(ToUpper : void \rightarrow \beta)$$

Using implicitly-typed attribute references, it is possible to create a generic `Wrapper` class as shown in Figure 12. The `Wrapper` class is capable of wrapping any type. Each time we call the `set` method, the new concrete type of the parameter is saved as the object type. By using this mechanism the two lines with comments report compilation errors. This coding style is type safe and it is easier that the parametric polymorphism used in C++ and much more straightforward than the bounded polymorphism offered by Java and C#. At the same time, runtime performance is as good as if we were specified types explicitly. Although the source code makes use of `var` references, concrete types are known at compile time. The generated code uses static type information and, therefore, runtime performance is better that the one obtained in dynamic languages.

Implicitly-typed attributes extend the constraint-based behavior of parameter references in the sense that concrete type of the implicit parameter (the object used in every non-static method invocation) could be modified at message passing. In our example, the type of the `wrapper` attribute is modified each time the `set` method (and the constructor) is invocated. This does not implies a modification of the whole `Wrapper` type, but only the type of the single `wrapper` object –due to the concrete type system employed.

For this purpose we have added a new kind of constraint to the type system: the *assignment* constraint. Each time a value is assigned to an attribute type with a fresh type variable, an assignment constraint is added to the method being analyzed. This constraint postpones the unification of the concrete type of `attribute`, to be performed later with the actual type used in the invocation. Therefore, the unification algorithm mentioned in Section 3.1 is executed when the method is called, using the concrete type of the actual object.

```
class Wrapper {
    private var attribute;
    public Wrapper(var attribute) {
        this.attribute = attribute;
    }
    public var get() {
        return attribute;
    }
    public void set(var attribute) {
        this.attribute = attribute;
    }
}
class Test {
    public static void Main() {
        string aString;
        int aInt;
        Wrapper wrapper = new Wrapper("Hello");
        aString = wrapper.get();
        aInt = wrapper.get(); // * Compilation error
        wrapper.set(3);
        aString = wrapper.get(); // * Compilation error
        aInt = wrapper.get();
    }
}
```

Fig. 12. Implicitly typed attribute references.

```
aString = getString(reference); // * Correct!
aString = upper(reference); // * Compilation error
              // * (correct if we set parameter to dynamic)
```

Fig. 13. Dynamic and static code interoperation.

3.5 Interaction between Static and Dynamic Typing

StaDyn performs static type checking of both dynamic and static `var` references. This makes possible the combination of static and dynamic code in the same application, because in both scenarios the compiler has type information.

Code in Figure 13 uses the `getString` and `upper` methods of Figure 11. `reference` may point to a string or to an exception. Therefore, it is type safe to invoke the `getString` method, but an erroneous dynamic behavior might be obtained calling to the `upper` method. This is the reason why, if the reference is set as static a compilation error is shown. Otherwise, if `reference` is set as dynamic, the same source code will be compiled without errors –although this optimism might involve errors at runtime.

Since dynamic and static code behaves differently, it is necessary to describe interoperation between both types of references. In case we set `reference` as a dynamic reference, could it be passed as an argument to the `upper` or `getString` methods? That is, how could optimistic (dynamic) code interoperate with pessimistic (static) one? Figure 13 shows an example.

The first invocation is correct regardless of the dynamism of `parameter`. Being optimistic or pessimistic, the argument responds to the `ToString` method correctly. However, it is not the same in the second scenario. By default, a compilation error

```
class Test {
    private var testField;
    public void setField(var param) {
        this.testField = param;
    }
    public var getField() {
        return this.testField;
    }
    public static void Main() {
        var wrapper = new Wrapper("hi");
        var test = new Test();
        test.setField(wrapper);
        string s = test.getField().get(); // * Correct!
        wrapper.set(true);
        bool b = test.getField().get();   // * Correct!
        string s = test.getField().get(); // * Compilation Error
    }
}
```

Fig. 14. Alias analysis.

is obtained, because the parameter `reference` is static and `reference` may point to an exception, causing an erroneous behavior. However, if we set the parameter of the `upper` method as dynamic, the compilation is correct.

This way, dynamic code could easily interoperate with static one: dynamic references should satisfy the constraints inferred in every possible concrete type. Promotion of static references to dynamic ones is more flexible: static references should satisfy at least one constraint from the set of alternatives.

3.6 Alias Analysis for Concrete Type Evolution

The problem of determining if a storage location may be accessed in more than one way is called *Alias Analysis* [27]. Two references are aliased if they point to the same object. Although alias analysis is mainly used for optimizations, we have used it to know the concrete types a reference may point to.

Code in Figure 14 uses the `Wrapper` class previously shown. Initially the `wrapper` reference points to a `string` object. Then a `Test` object that references to the original `Wrapper` object is created. If we get the object inside the `wrapper` object inside the `test` object, we get a `string` object. Then a `bool` attribute is set to the `wrapper` object. Repeating the previous access to the object inside the `wrapper` object inside the `test` object, a `bool` object is obtained.

The alias analysis algorithm implemented is type-based (uses type information to decide alias) [14], inter-procedural (makes use of inter-procedural flow information) [27], context-sensitive (differentiates between different calls to the same method) [16], and may-alias (detects all the objects a reference may point to; opposite to *must* point to) [3].

Alias analysis is an important tool for our type-reconstructive concrete type system, and it is the key technique we are presently using to implement our current stage: structural reflective type evolution –see Section 6.

4 Implementation

All the programming language features described in this paper have been implemented over the CLR 2.0 platform, using the C# 2.0 programming language. Current release of the *StaDyn* programming language, its source code, and all the examples presented in this paper are freely available at http://www.reflection.uniovi.es/stadyn

Our compiler is a multiple-pass language processor that follows the *Pipes and Filters* architecture pattern [7]. We have used the AntLR language processor tool to implement lexical and syntactic analysis [34]. *Abstract Syntax Trees* (ASTs) have been implemented following the *Composite* design pattern [17] and each pass over the AST implements the *Visitor* design pattern [17].

Currently we have developed the following AST visits: two visitors for the SSA algorithm; two visitors to load types into the types table; one visitor for symbol identification [46] and another one for type inference; and two visitors to generate code. Once the final compiler is finished, the number of AST visits will be reduced. The complexity of our implementation is O(n), being n the number of abstract nodes in the AST. At present, we use the CLR 2.0 as the sole compiler's back-end. However, we have designed the code generator module following the *Bridge* design pattern to add both the DLR (*Dynamic Language Runtime*) [20] and the ЯRotor [39] back-ends in the future .

5 Related Work

There have been few approaches implemented to include static and dynamic typing in the same programming language. Although several theoretical works exist, there is a lack on real implementations of these proposals.

5.1 Visual Basic .Net

The Visual Basic .Net programming language incorporates both dynamic and static typing [45]. Compiled applications run over the .Net platform using the same virtual machine.

The main benefit of its dynamic type system is that it supports *duck typing*. However, since no static type inference is performed with dynamic references, there are interoperation lacks between dynamic and static code. If it is necessary to use a dynamic object in static code, a cast must be performed. Therefore, all the type checking is performed at runtime. At the same time, dynamic references do not produce any type error at compile time.

Another limitation of Visual Basic .Net is that dynamism is included in the source code. This forces the programmer to explicitly state which references are static and which are dynamic, making difficult the transition between both scenarios.

5.2 *Boo*

Boo is a recent object-oriented programming language that is both statically and dynamically typed with a python inspired syntax [5]. In Boo references may be declared without specifying its type and the compiler performs type inference. However, references could only have a unique type in the same scope. Moreover, fields and parameters could not be declared without specifying its type.

Boo offers dynamic type inference with a special type called `duck`. If a reference has this type, any operation could be performed over that reference –no static typing is performed. Converting a dynamic type reference to a static one implies a type cast.

Although this behavior is similar to the one offered by Visual Basic .Net, the Boo compiler provides the *ducky* option that interprets the `Object` type as if it were `duck`. Turning on the *ducky* option allows the programmer to test out the code more quickly, and makes coding in Boo feel much more like coding in a dynamic language. So, when the programmer has tested the application, she may wish to turn the *ducky* option back off and add various type declarations and casts. However, this scheme does not follow the *Separation of Concerns* principle [22]; it sets all the object references as dynamic.

5.3 *Dylan*

Dylan is a high-level programming language, designed to allow efficient, static compilation of features normally associated with dynamic languages [42]. Dylan permits both explicit and implicit variables declaration. It also supports two compilation scenarios: production and interactive.

In the interactive mode, all the types are ignored and no static type checking is performed. This behavior is similar to the one offered by dynamic languages. When the production configuration is selected, explicitly typed variables are checked following a common static type system. However, types of generic references (references without type declaration) are not inferred at compile type –they are checked at runtime.

5.4 *Strongtalk*

Strongtalk is a major re-thinking of the Smalltalk-80 programming language [6]. It retains the basic Smalltalk syntax and semantics [18], but a type system is added to provide more reliability and much more runtime performance. Types added by Strongtalk made the best performance optimization ever achieved over Smalltalk.

One key issue in Strongtalk is that its type system is completely optional. This assumes that is the programmer's responsibility to ensure that types are sound in regard to dynamic behavior. Type errors are checked at compile type, but they do not guarantee a correct dynamic behavior. This type system is not completely safe, but implies a significant performance improvement.

6 Conclusions and Future Work

The main contribution of our work is that supporting both dynamic and static typing in the same programming language, following the *Separation of Concerns* principle, provides runtime flexibility, type safety, runtime performance, direct dynamic and static code interoperation, language simplicity, and movement from rapid prototyping to robust software development.

If runtime flexibility is needed to make parts of an application (or the whole program) more flexible, it is only necessary to specify it in separate files, without changing the source code. These parts of the application will use a more *optimistic* type checking scheme. Otherwise, if parts of an application are identified as critical in performance and safety, static references will be used and the same code will be compiled in a *pessimistic* way. Source code can be moved from *optimistic* to *pessimistic* and the other way round. This feature is achieved by means of modularizing the dynamism concern into separate XML files.

StaDyn performs type inference in dynamic and static scenarios. When types could be inferred at compile time (not always possible), runtime performance and safety will be better than the one offered by dynamic languages. Moreover, JIT compilation and adaptive hotspot optimization of the CLR are used in both scenarios at runtime.

Another benefit of applying the same type system for both dynamic and static typing is language interoperation. Since we do type inference in dynamic and static compilation modes, it is possible to make dynamic and static code interoperate. Currently, IronPython and the Java Scripting API are two common examples that show how dynamic languages can interoperate with static ones, but not the other way round. If the programmer creates a class instance in Python, the static code cannot directly retrieve its type. This lack is due to the fact that Python does no static type checking. Therefore, types created by the programmer in Python are not included in the type system of the static program.

Future work will be centered in adding two important features available in most dynamic languages: structural reflection and dynamic code generation. Structural reflection permits the dynamic addition, deletion and modification of classes and objects. *StaDyn* will analyze concrete type evolution by means of its alias analysis mechanism. Only dynamic references will permit type evolution. When the name of the member to be added, removed or modified is known statically, it will be used to modify the concrete type. Otherwise, a dynamic type inference system will be used with dynamic references. Dynamic code generation implies producing (part of) programs at runtime. The compiler API will be added to the runtime, inferring types at execution. If some compile error exists, the execution of this code will throw an exception.

References

[1] Abadi, M., L. Cardelli, B.C. Pierce, and G. Plotkin, *Dynamic typing in a statically typed language*, ACM Transactions on Programming Languages and Systems 13, 2 (1991).

[2] Abadi, M., L. Cardelli, B.C. Pierce, and G. Plotkin, “Dynamic typing in polymorphic languages”, SRC Research Report 120, Digital, 1994.

[3] Appel, A.W., "Modern Compiler Implementation in ML", Cambridge University Press, 1998.

[4] Böllert, K., *On weaving aspects*, in *European Conference on Object-Oriented Programming (ECOOP), Workshop on Aspect Oriented Programming* (1999).

[5] *Boo home page*, http://boo.codehaus.org

[6] Bracha, G., and D. Griswold, *Strongtalk: Typechecking Smalltalk in a Production Environment*, in *Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications* (1993).

[7] Buschmann, F., "Pattern-Oriented Software Architecture, a System of Patterns", John Wiley & Sons, 1996.

[8] Canning, P., W. Cook, W. Hill, W. Olthoff, and J.C. Mitchell, *F-bounded polymorphism for object-oriented programming*, in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ACM Press (1989).

[9] Cardelli, L., "Type Systems", The Handbook of Computer Science and Engineering, 1997.

[10] Crane, D., E. Pascarello, and D. James, "Ajax in Action", Manning Publications, 2005.

[11] Cytron, R., J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems 13 , 4 (1991).

[12] *The C# 3.0 Language Specification*, Microsoft Developer Network, http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc

[13] *The Da Vinci Machine, a multi-language renaissance for the Java Virtual Machine architecture*, Sun Micryosystems OpenJDK, http://openjdk.java.net/projects/mlvm

[14] Diwan, A., K.S. McKinley, and J.E.B. Moss, *Type-Based Alias Analysis*, in *SIGPLAN Conference on Programming Language Design and Implementation* (1998)

[15] Standard ECMA-357, "ECMAScript for XML (E4X) Specification, 2nd edition", European Computer Manufacturers Association, 2005.

[16] Emami, M., R. Ghiya, and L. Hendren, *Context-sensitive inter-procedural points-to analysis in the presence of function pointers*, in *Proceedings of ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (1994).

[17] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[18] Goldberg, A., and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.

[19] Hudak, P., S. P. Jones, and P. Wadler, "Report on the programming language Haskell version 1.1", Technical report, Departments of Computer Science, University of Glasgow and Yale Universtiy, 1991.

[20] Hugunin, J., *Just Glue It! Ruby and the DLR in Silverlight*, in *MIX Conference*, 2007.

[21] Hunt, A., and D. Thomas, "The Pragmatic Programmer", Addison-Wesley, 2000.

[22] Hürsch, W.L., and C.V.Lopes, "Separation of Concerns", Technical Report UN-CCS-95-03, Northeastern University, Boston, USA, 1995.

[23] Ierusalimschy, R., L.H. de Figueiredo, and W.C. Filho, *Lua –an extensible extension language*, Software Practice & Experience 26, 6 (1996).

[24] Ierusalimschy, R., L.H. de Figueiredo, W. Celes, *The evolution of an extension language: A history of Lua*, in *V Brazilian Symposium on Programming Languages* (2001).

[25] JSR 223, *Scripting for the Java Platform*, http://www.jcp.org/en/jsr/detail?id=223

[26] JSR 292, *Supporting Dynamically Typed Languages on the Java Platform*, http://www.jcp.org/en/jsr/detail?id=292

[27] Landi, W., and B.G. Ryder, *A Safe Approximation Algorithm for Interprocedural Pointer Aliasing*, in *Conference on Programming Language Design and Implementation*, 1992.

[28] Meijer, E., and P. Drayton, P., *Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, in *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages* (2004)

[29] Milner, R., *A theory of type polymorphism in programming*, Journal of Computer and System Sciences (1978).

[30] Milner, R., M. Tofte, and R. Harper, "The Definition of Standard ML", MIT Press, 1990.

[31] Ortin, F., and J.M Cueva, *Dynamic Adaptation of Application Aspects*, Journal of Systems and Software 71, 3 (2004).

[32] Ortin, F., *Extending Rotor with Structural Reflection to support Reflective Languages*, in *Microsoft Research SSCLI RFP II Capstone Workshop*, Redmond, Washington (2005).

[33] Ousterhout, J.K., *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer 31, 3 (1998).

[34] Parr, T., "The Definitive ANTLR Reference: Building Domain-Specific Languages", Pragmatic Bookshelf, 2007.

[35] Pierce, B.C., "Programming with intersection types, union types, and polymorphism", Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[36] Pierce, B.P., "Types and Programming Languages", The MIT Press. February, 2002.

[37] Plevyak, J., and A.A. Chien, *Precise concrete type inference for object-oriented languages*, SIGPLAN Notices 29, 10, Proceeding of the OOPSLA Conference,(1994).

[38] *The Pythius website*, http://pythius.sourceforge.net

[39] Redondo, J.M., F. Ortin, and J.M. Cueva, *Optimizing Reflective Primitives of Dynamic Languages*, International Journal of Software Engineering and Knowledge Engineering 18, 6 (2008).

[40] Richter, S., "Zope 3 Developer's Handbook", Sams, 2005.

[41] van Rossum, G., L. Fred, and J.R. Drake, "The Python Language Reference Manual", Network Theory, 2003.

[42] Shalit, A., "The Dylan reference manual: the definitive guide to the new object-oriented dynamic language", Addison Wesley Longman Publishing Co. (1996).

[43] Thomas, D., C. Fowler, and A. Hunt, "Programming Ruby", 2nd Edition, Addison-Wesley Professional, 2004.

[44] Thomas, D., D.H. Hansson, A. Schwarz, T. Fuchs, L. Breed, and M. Clark, "Agile Web Development with Rails. A Pragmatic Guide", Pragmatic Bookshelf, 2005.

[45] *Visual Basic .NET Language Specification*, Microsoft Development Network, http://msdn2.microsoft.com/en-us/library/aa712050(VS.71).aspx

[46] Watt, D., and D. Brown, "Programming Language Processors in Java: Compilers and Interpreters", Prentice Hall, 2000.