Including both Static and Dynamic Typing in the same Programming Language

Francisco Ortin[*]   Daniel Zapico[*]

J. Baltasar G. Perez-Schofield[†]   Miguel Garcia[*]

December 1, 2009

Abstract

Dynamic languages are becoming increasingly popular for different software development scenarios such as Web engineering, rapid prototyping, or the construction of applications that require runtime adaptiveness. These languages are built on the idea of supporting reasoning about (and customizing) program structure, behaviour and environment at runtime. The dynamism offered by dynamic languages is, however, counteracted by two main limitations: no early type error detection and fewer opportunities for compiler optimizations. To obtain the benefits of both dynamically and statically typed languages, we have designed the *StaDyn* programming language that provides both approaches. *StaDyn* keeps gathering type information at compile time, even when dynamic variables are used. This type information is used to offer early type error detection, direct interoperation between static and dynamic code, and better runtime performance. Following the *Separation of Concerns* principle, it is possible to customize the trade-off between runtime flexibility of dynamic typing, and safety, performance and robustness of static typing. A runtime performance

---

[*]Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007, Oviedo, Spain, `ortin@lsi.uniovi.es`

[†]Computer Science Department, University of Vigo, As Lagoas s/n, 32004, Orense, Spain, `jbgarcia@uvigo.es`

assessment is presented to show an estimate of the benefits of combining dynamic and static typing in the same programming language.

1 Introduction

Dynamic languages have recently turned out to be really suitable for specific scenarios such as Web development, application frameworks, game scripting, interactive programming, rapid prototyping, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software. Common features of dynamic languages are meta-programming, reflection, mobility, and dynamic reconfiguration and distribution.

In the Web engineering area, Ruby [1] has been successfully used together with the *Ruby on Rails* framework for creating database-backed web applications [2]. This framework has confirmed the simplicity of implementing the DRY (*Don't Repeat Yourself*) [3] and the *Convention over Configuration* [2] principles with this kind of languages. Nowadays, JavaScript [4] is being widely employed to create interactive Web applications with AJAX (*Asynchronous JavaScript And XML*) [5], while PHP (*PHP Hypertext Preprocessor*) is one of the most popular languages to develop Web-based views. Python [6] is used for many different purposes, being the *Zope* application server [7] (a framework for building content management systems, intranets and custom applications) and the *Django* Web application framework [8] two well-known examples. Due to its small size, portability and ease of integration, *Lua* [9] has gained great popularity for extending games [10]. Finally, a wide range of dynamic aspect-oriented tools has been built over dynamic languages [11, 12, 1, 13], offering a higher runtime adaptiveness

than the common static ones.

Due to the recent success of dynamic languages, other statically typed ones –such as Java or C#– are gradually incorporating more dynamic features into their platforms. Taking Java as an example, the *Reflection API* became part of core Java platform 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 –involving 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, funnelling all its method calls to an `InvocationHandler`. In Java 1.6, the new *Java Scripting* API permits dynamic scripting programs to be executed from, and have access to, the Java platform [14]. Finally the *Java Specification Request* 292 [15], 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 launched the *Da Vinci Machine* project in January 2008 [16]. 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. This platform was initially released with introspective and low-level dynamic code generation services. Version 2.0 included dynamic methods and the `CodeDom` namespace to model and generate the structure of a high-level source code document. The *Dynamic Language Runtime* (DLR), first announced by Microsoft in 2007, adds to the .NET platform a set of services to facilitate the implementation of dynamic languages [17]. Finally, Microsoft has just

3

included a dynamic typing feature in C# 4.0, as part of the Visual Studio 2010 [18]. This new feature of C# 4.0 is the Microsoft response to the emerging use of dynamic languages such as Python [6] or Ruby [1]. C# 4.0 offers a new `dynamic` keyword to support dynamically typed C# code. When a reference is declared as `dynamic`, the compiler performs no static type checking, making all the type verifications at runtime. With this new characteristic, C# 4.0 will offer direct access to dynamically typed code in IronPython, IronRuby and the JavaScript code in Silverlight. Dynamic code in C# 4.0 makes use of the DLR services [17].

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 commonly a considerable runtime performance penalty. Static typing offers the programmer the detection of type errors at compile time, 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 [19]. Moreover, since runtime adaptability of dynamic languages is mostly implemented with dynamic type systems, runtime type inspection and checking commonly involves a significant performance penalty.

Since both approximations offer different benefits, there have been former works on providing both typing approaches in the same language (see Section 6). 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 [20]. Static typing allows earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance.

4

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 behaviour customization, or generative programming [21]. Hence, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [22]. That is, *static typing where possible, dynamic typing when needed* [20].

Our work breaks the programmers' black or white choice between static or dynamic typing. The programming language presented in this paper, called *StaDyn*, supports both static and dynamic typing. 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 where high adaptability is required (dynamic) and those where *correct*[1] execution (static) should be guaranteed –i.e. *StaDyn* separates the dynamism concern [23]. This separation facilitates turning rapidly developed prototypes into a final robust and efficient application. It is also possible to combine both approaches, making parts of an application more flexible, whereas the rest of the program maintains its robustness and runtime performance.

In this paper, we present an overview of the techniques we used to design and implement our programming language in order to support both dynamic and static typing. The rest of this paper is structured as follows. In the next section, we provide the 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 employed. Section 4

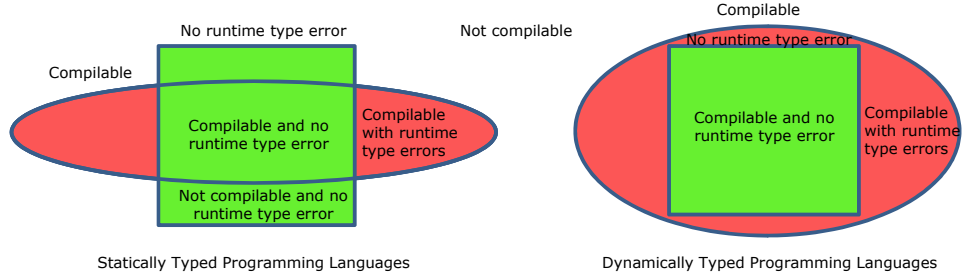[1]We use *correct* to indicate programs without runtime type errors.

Figure 1: Program execution in statically and dynamically typed languages.

presents the key implementation decisions, an initial runtime performance assessment is presented in Section 5, and Section 6 discusses related work. Finally, Section 7 presents the ending conclusions and future work.

2 Static Typing vs. Dynamic Typing

2.1 Statically Typed Languages

A language is said to be *safe* if it produces no execution errors that go unnoticed and later cause arbitrary behaviour [24], following the notion that well-typed programs should not go *wrong* (i.e., reach a *stuck* state on its execution) [19]. Statically typed languages ensure type safety of programs by means of static type systems. However, these type systems do not compile some expressions that do not produce any type error at runtime (e.g., in .NET and Java it is not possible to pass the m message to an Object reference, although the object actually implements a public m method). This happens because their static type systems require ensuring that compiled expressions do not generate any type error at runtime. Left part of Figure 1 illustrates this situation (the *not compilable and no runtime type error* region).

Static typing is centred on making sure that no type error is produced at runtime. This is the reason why languages with static typing employ a pessimistic policy regarding to program compilation. This pessimism causes

6

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

Figure 2: Not compilable C# program that would not produce any runtime error.

compilation errors in programs that do not produce any runtime error. C# code shown in Figure 2 is an example program of this scenario. Although the program does not produce any error at runtime, the C# type system does not recognize it as a valid compilable program.

At the same time, static languages also permit the execution of programs that might cause an erroneous execution (e.g. array index out of bounds or null pointer access) –the *compilable with runtime type errors* region in Figure 1.

2.2   Dynamically Typed Languages

The approach of dynamic languages is the opposite one. Instead of making sure that all valid expressions will be executed without any error, they make all the syntactic valid programs compilable (right part of Figure 1). This is a too optimistic approach that causes a high number of runtime type errors that might have been detected at compile time. This situation, where dynamic languages commonly throw runtime exceptions, is what is represented in Figure 1 as *compilable with runtime type errors*. This approach permits too many runtime type errors, compiling programs that might have been identified as erroneous statically. The Visual Basic .NET source code in Figure 3 is an example of this too optimistic approach. This erroneous program is compilable, although a static type system might have detected the error before its execution.

7

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

Figure 3: Compilable Visual Basic program that generates runtime type errors.

2.3   Support of both Approaches

The *StaDyn* programming language performs type inference at compile type, minimizing the *compilable with runtime type errors* region of dynamic languages (right part of Figure 1) and the *not compilable and no runtime type error* area of static languages (left part of Figure 1). Consequently, *StaDyn* detects the compilation error of the dynamic program shown in Figure 3 (that Visual Basic does not detect) and compiles the valid static code in Figure 2 (that C# does not compile) –using its own syntax.

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

3   The *StaDyn* Programming Language

This section presents the features of the *StaDyn* programming language, identifying –but not detailing– the techniques employed. A formal description of its type system is depicted in [25]. Implementation issues are pre-

8

sented in Section 4.

The *StaDyn* programming language is an extension of C# 3.0 [26]. Although the work presented in this paper could be applied to any object-oriented statically-typed programming language, we have used C# 3.0 to extend the behaviour of its implicitly typed local references. In *StaDyn*, the type of references can still be explicitly declared, while it is also possible to use the `var` keyword to declare implicitly typed references. *StaDyn* includes this keyword as a new type (it can be used to declare local variables, fields, method parameters and return types), whereas C# 3.0 only provides its use in the declaration of initialized local references. Therefore, `var` references in *StaDyn* are much more powerful than implicitly typed local variables in C# 3.0.

The dynamism of `var` references is placed in a separate file (an XML document). The programmer does not need to manipulate these XML documents directly, leaving this task to the IDE. When the programmer (un)sets a reference as dynamic, the IDE transparently modifies the corresponding XML file. Depending on the dynamism of a `var` reference, type checking and type inference is performed pessimistically (for static references) or optimistically (for dynamic ones). Since the dynamism concern is not explicitly stated in the source code, *StaDyn* facilitates the conversion of dynamic references into static ones, and vice versa. This separation facilitates the process of turning rapidly developed prototypes into final robust and efficient applications. It is also possible to make parts of an application more adaptable, maintaining the robustness and runtime performance of the rest of the program.

```csharp
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
    }
}
```

Figure 4: A reference with different types in the same scope.

3.1 Multiple Types in the Same Scope

Existing statically typed languages force 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 [27] or Haskell [28] do not permit the assignment of different types to the same polymorphic reference in the same scope.

However, dynamic languages provide the use of one reference to hold different types in the same scope. This is easily implemented at runtime with a dynamic type system. However, *StaDyn* offers this feature statically, taking into account the concrete type of each reference. The *StaDyn* program shown in Figure 4 is an example of this capability. The `age` reference has different types in the same scope. It is initially set to a string, and an integer is later assigned to it. The static type inference mechanism implemented in *StaDyn* detects the error in the last line of code. Moreover, a better runtime performance is obtained because it is not necessary to use reflection to discover types at runtime.

In order to obtain this behaviour, we have developed an implicit para-

```csharp
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
    }
}
```

Figure 5: Corresponding program after the SSA transformation.

metric polymorphic type system [29] 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 [30]. This algorithm has been modified to perform type reconstruction of `var` parameters and attributes (fields) –described in sections 3.4 and 3.5.

The unification algorithm used in the Hindley-Milner type system 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 [31]. This algorithm guarantees that every reference is assigned exactly once by means of creating new temporary references. Since type inference is performed after the SSA algorithm, we have implemented it as a previous AST (*Abstract Syntax Tree*) transformation. The implementation of this algorithm follows the *Visitor* design pattern [32].

Figure 5 shows the corresponding program after applying the AST transformation to the source code in Figure 4. The AST represented by the source code in Figure 5 is the actual input to the type inference system. Each `age` reference will be inferred to a single static type.

3.2 Duck Typing

Duck typing[2] [1] is a property of dynamic languages that means that an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether those objects have a related inheritance hierarchy or not. This is a powerful feature offered by most dynamic languages.

There exist statically typed programming languages such as Scala [33] or OCaml [34] that offer structural typing, providing part of the benefits of *duck typing*. However, the structural typing implementation of Scala is not implicit, forcing the programmer to explicitly declare part of the structure of types. In addition, intersection types should be used when more than one operation is applied to a variable, making programming more complicated. Although OCaml provides implicit structural typing, variables should only have one type in the same scope, and this type is the most general possible (principal) type [35]. Principal types are more restrictive than *duck typing*, because they do not consider all the possible (concrete) values a variable may hold.

The *StaDyn* programming language offers static duck typing. The benefit provided by *StaDyn* is not only that it supports (implicit) *duck typing*, but also that it is provided statically. Whenever a `var` reference points to a set of objects that implement a public `m` method, the `m` message could be safely passed. These objects do not need to implement a common interface or an (abstract) 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.

We have implemented static *duck typing* making the static type system

---
[2] *If it walks like a duck and quacks like a duck, it must be a duck.*

```
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);
```

Figure 6: Static *duck* typing.

of *StaDyn 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) [36] knowing all the possible types a `var` reference may hold. Instead of declaring a reference with an abstract type that embraces all the possible concrete values, the compiler infers the union of all possible concrete types a `var` reference may point to. Notice that different types depending on flow context could be inferred for the same reference, using the type inference mechanism mentioned above.

Code in Figure 6 shows this feature. The reference `exception` may point to either an `ApplicationException` or a `SystemException` object. Both objects have the `Message` property and, therefore, it is statically safe to access to 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 static duck-typing system.

The key technique we have used to obtain this concrete-type flow-sensitiveness is *union types* [37]. Concrete types are first obtained by the above-mentioned unification algorithm (applied in assignments and method calls). 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 (next section).

13

3.3 Separation of the Dynamism Concern

StaDyn permits the use of both static and dynamic `var` references. Depending on their dynamism concern, type checking and type inference would be more pessimistic (static) or optimistic (dynamic), but the semantics of the programming language is not changed (i.e., program execution does not depend on its dynamism). This idea follows the *pluggable* type system approach described in [38] and [39]. Since the dynamism concern is not explicitly stated in the source code, it is possible to customize the trade-off between runtime flexibility of dynamic typing, and runtime performance and robustness of static typing. It is not necessary to modify the application source code to change its dynamism. Therefore, dynamic references could be converted into static ones, and vice versa, without changing the application source code.

Source code in Figure 7 adds another alternative in the assignment of the `exception` reference. The `ToString` message is valid because it is offered by all the three possible objects. 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 UnknownMemberError (Semantic error). 'Message': no suitable member found.

However, if the programmer prefers to be more optimistic, she or he could set all the `var` references in the module as dynamic. If the program in Figure 7 is compiled with the *everythingDynamic* option, the executable file is generated. In this case, the compiler accepts passing `Message`, because there is at least one possibility that the program executes without any error (i.e., type checking succeeds if at least one of the types that compose

14

```
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?
```

Figure 7: Static `var` reference.

```
<?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>
```

Figure 8: Sample XML document specifying the dynamism of the `exception` reference.

the union type is valid). The actual type will be discovered at runtime, checking that the `Message` property can be actually accessed, or throwing `MissingMethodException` otherwise.

Actually, the programmer does not need to set all the `var` references in a program (or assembly) as dynamic. It is possible to specify the dynamism of each single reference by means of a XML file. As discussed above, the programmer does not manipulate these XML documents directly, leaving this task to the IDE. The XML document shown in Figure 8 only sets as dynamic the `exception` reference in Figure 7. Each *StaDyn* source code file may have a corresponding XML document specifying its dynamism concern.

It is worth noting that setting a reference as dynamic does not imply that

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

Figure 9: Dynamic `var` reference.

every message could be passed to that reference; static type-checking is still performed. The major change is that the type system is more –but not too– optimistic when dynamic `var` references are used. The dynamism concern implies a modification of type checking over union types. If the implicitly typed `var` reference inferred with a union type is static, type checking is performed over all its possible concrete types. However, if the reference is 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 9 shows how dynamic references may produce static errors as well. Even though its code is compiled with the *everythingDynamic* option, the compiler shows the following static error:

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

This example shows how static typing is performed even in dynamic scenarios, providing early type error detection (runtime performance improvement is discussed in Section 5). This limitation of dynamic languages is shown in the Visual Basic code in Figure 3, which *StaDyn* detects as erroneous, whereas Visual Basic does not.

16

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

Figure 10: Implicitly typed parameters.

3.4 Implicitly Typed Parameters

Concrete type reconstruction is not limited to local variables. *StaDyn* performs a global *flow-sensitive* analysis of implicit `var` references. The result is an implicit parametric polymorphism [29] more straightforward for the programmer than the one offered by Java, C# (F-bounded) and C++ (unbounded) [40].

Implicitly typed parameter references cannot be unified to a single concrete type. Since they represent any actual type of an argument, they cannot be inferred the same way as local references. This necessity is shown in the source code of Figure 10. 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 capable of responding to the `ToUpper` message. Depending on the type of the actual parameter, the *StaDyn* compiler generates the corresponding compilation error.

For this purpose we have enhanced the *StaDyn* type system to be constraint-based [41]. Types of methods in our object-oriented language have an ordered set of constraints specifying the set of restrictions that must be fulfiled by the parameters. In our example, the type of the `upper` method is:

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

17

This means that the type of the parameter (α) should implement a public `ToUpper` method with no parameters, and the type returned by `ToUpper` (β) will be also returned by `upper`. Therefore, if an integer is passed to the `upper` method, a compiler error is shown. However, if a string is passed instead, the compiler reports not only any error, but it also infers the resulting type as a string. Type constraint fulfilment is, thus, part of the type inference mechanism (the concrete algorithm could be consulted in [25]).

3.5   Implicitly Typed Attributes

Using implicitly typed attribute references, it is possible to create the generic `Wrapper` class shown in Figure 11. The `Wrapper` class can wrap any object of any type. Each time the `set` method is called, the new concrete type of the parameter is saved as the `attribute` type. By using this mechanism, the two lines with comments report compilation errors. This coding style is polymorphic and it is more legible that the parametric polymorphism used in C++ and much more straightforward than the F-bounded polymorphism offered by Java and C#. At the same time, runtime performance is equivalent to explicit type declaration (see Section 5). Since possible concrete types of `var` references are known at compile time, the compiler has more opportunities to optimize the generated code, improving runtime performance.

Implicitly typed attributes extend the constraint-based behaviour of parameter references in the sense that the concrete type of the implicit object parameter (the object used in every non-static method invocation) could be modified on a method invocation expression. In our example, the type of the `wrapper` attribute is modified each time the `set` method (and the constructor) is invoked. This does not imply a modification of the whole

```
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();
    }
}
```

Figure 11: Implicitly typed attributes.

`Wrapper` type, only the type of the single `wrapper` object –thanks to the *concrete* type system employed.

For this purpose we have added a new kind of *assignment* constraint to the type system [25]. Each time a value is assigned to a `var` attribute, an assignment constraint is added to the method being analyzed. This constraint postpones the unification of the concrete type of the attribute to be performed later, when an actual object is used in the invocation. Therefore, the unification algorithm is used to type-check method invocation expressions, using the concrete type of the actual object (a detailed description of the unification algorithm can be consulted in [25]).

3.6 Interaction between Static and Dynamic Types

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 the compiler gathers type information in both scenarios.

Code in Figure 12 uses the `getString` and `upper` methods of Figure 10. `reference` may point to a string or integer. Therefore, it is safe to invoke the `getString` method, but a dynamic type error might be obtained when the `upper` method is called.

Since type-checking of dynamic and static code is different, it is necessary to describe interoperation between both types of references. In case `reference` had been set as a dynamic, the question of whether or not it could have been passed as an argument to the `upper` or `getString` methods (Figure 10) arises. That is, how optimistic (dynamic) code could interoperate with pessimistic (static) one. An example is shown in Figure 12.

The first invocation is correct regardless of the dynamism of `parameter`.

```
var reference;
string aString;
if (new Random().NextDouble() < 0.5)
    reference = "String";
else
    reference = 3;
aString = getString(reference); // * Correct!
aString = upper(reference);     // * Compilation error
                  // * (correct if we set parameter to dynamic)
```

Figure 12: Dynamic and static code interoperation.

Being either 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 is obtained, because the parameter `reference` is static and it may point to an integer, which does not implement a public `ToUpper` method. However, if we set the parameter of the `upper` method as dynamic, the compilation will succeed.

This type-checking is obtained taking into consideration the dynamism of references in the subtyping relation of the language. A dynamic reference is a subtype of a static one when all the concrete types of the dynamic reference promote to the static one [25]. Promotion of static references to dynamic ones is more flexible: static references should fulfil at least one constraint from the set of alternatives.

3.7   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* [42]. 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 of the objects a reference may point to.

Code in Figure 13 uses the `Wrapper` class previously shown. Initially, the `wrapper` reference points to a string object. Then a `Test` object that

21

```
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
    }
}
```

Figure 13: Alias analysis.

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 then obtained.

The alias analysis algorithm implemented is type-based (uses type information to decide alias) [43], inter-procedural (makes use of inter-procedural flow information) [42], context-sensitive (differentiates between different calls to the same method) [44], and may-alias (detects all the objects a reference *may* point to; opposite to *must* point to) [45].

Alias analysis is an important tool for our type-reconstructive concrete type system, and it is the key technique to implement the next (future) stage: structural reflective type evolution –see Section 7.

22

4 Implementation

All the programming language features described in this paper have been implemented over the .NET Framework 3.5 platform, using the C# 3.0 programming language. Our compiler is a multiple-pass language processor that follows the *Pipes and Filters* architectural pattern [46]. We have used the AntLR language processor tool to implement lexical and syntactic analysis [47]. Abstract Syntax Trees (ASTs) have been implemented following the *Composite* design pattern [32] and each pass over the AST implements the *Visitor* design pattern [32].

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 [48] 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 to optimize the implementation. The type system has been implemented following the guidelines described in [49].

We generate .NET intermediate language and then assemble it to produce the binaries. At present, we use the CLR 2.0 as the unique compiler's backend. However, we have designed the code generator module following the *Bridge* design pattern to add both the DLR (Dynamic Language Runtime) [17] and the ЯRotor [50] back-ends in the future.

5 Runtime Performance Assessment

We have done an initial assessment of the runtime performance benefits obtained with the inclusion of dynamic and static typing in the same programming language. The aim of the evaluation is to obtain an initial estimate of what is the performance improvement obtained with the combination

of both typing approaches. For that purpose we have developed a simple micro-benchmark that takes the following scenarios into account:

1. Explicit static type declaration. No `var` references are used at all, explicitly stating the type of every variable.

2. Implicit dynamic type reference declaration, when the compiler manages to infer types. Although dynamic `var` references are used, the compiler could infer possible types statically. Figure 4 is a basic example of this kind of type inference, when the exact concrete type can be inferred. In the case of the code in Figure 7, three possible types could be inferred by the compiler. In this micro-benchmark we have developed this scenario when 1, 5, 10, 50 or 100 possible types could be inferred statically.

3. Implicit dynamic type reference declaration, when the compiler cannot infer any type at all. In this scenario, dynamic references are used as parameters. The argument reference randomly holds an object from 100 different types.

We have selected the most commonly used operation in object oriented programs: method invocation. A polymorphic method that performs a basic arithmetical operation is called in a loop of 100,000 iterations. Its implementation depends on the type of its parameters, local variables, object fields and the object itself (it is polymorphic). Since the difference of our approach is the type information gathered by the compiler, the primitive to measure is not really significant because, excluding reflection services, most low-level operations in .NET are statically typed.

The same programs (previous enumeration) have been compiled with the following compilers:

- C# 3.0. We have run our benchmark in the production C# version 3.0 that is shipped with Visual Studio 2008. Since C# 3.0 does not support dynamic typing, we only run the explicit type declaration test.

- The C# 4.0 implementation of Visual Studio 2010 Beta 1. The final product is expected to be released in March 2010. It combines static and dynamic typing (see Section 6). Its back-end is the final version of the DLR that is also released together with SilverLight and IronPython 2.0.

- Visual Basic 10. The Visual Basic (VB) programming language also supports dynamic typing [20]. A dynamic reference is declared with the `Dim` reserved word, without setting a type. With this syntax, the compiler does not gather any type information statically, and type checking is performed at runtime. The main difference between VB 10 and C# 4.0 is that the former uses the CLR, whereas the latter employs the DLR.

- *StaDyn*. The same programs coded in C# 4.0 are simply translated into *StaDyn* by replacing the `dynamic` reserved work with `var`. We compile the four programs with the *everythingDynamic* option.

We have not included other dynamic programming languages such as Python or Ruby to avoid the introduction of a bias in the translation of source code (translation of C# to VB is direct). At the same time, all the languages we have used generate code for the .NET framework, so it facilitates the comparison of performance results. This way, the measurements obtained show the performance improvement of gathering type information of dynamic code at compile time.
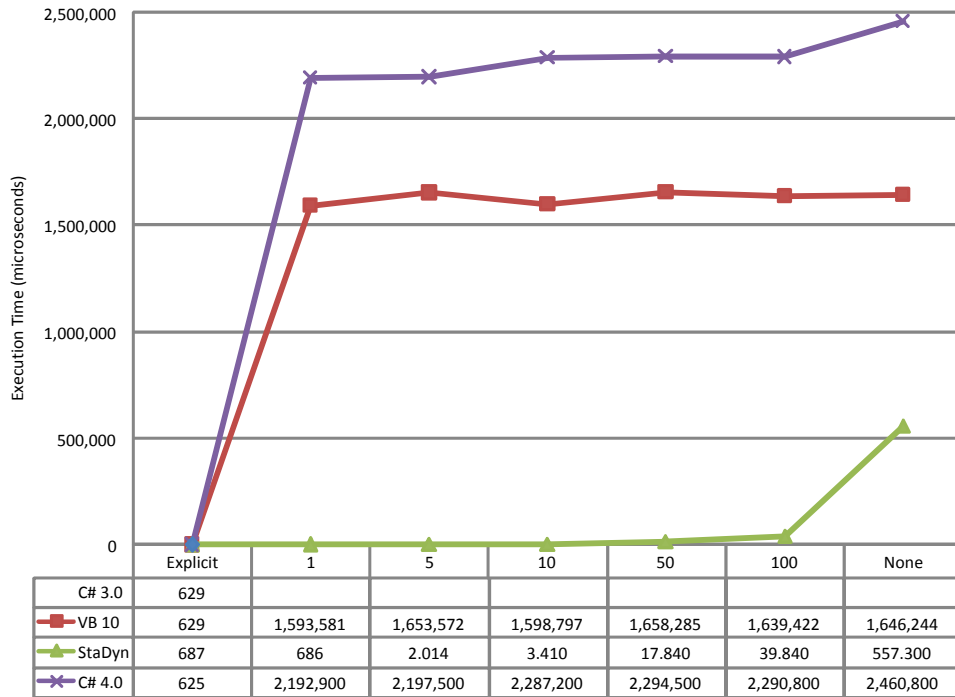
	Explicit	1	5	10	50	100	None
C# 3.0	629						
VB 10	629	1,593,581	1,653,572	1,598,797	1,658,285	1,639,422	1,646,244
StaDyn	687	686	2.014	3.410	17.840	39.840	557.300
C# 4.0	625	2,192,900	2,197,500	2,287,200	2,294,500	2,290,800	2,460,800

Figure 14: Execution time in C# 3.0, Visual Basic 10, *StaDyn* and C# 4.0.

All the programs have been executed over the .NET framework 4.0 Beta 1 on a lightly loaded 2.67 GHz Core 2 Duo system with 2 GB of RAM running Windows 2008 Server Standard, build 6001. Every test has been compiled without debugging information and full optimized.

Figure 14 shows the results expressed in microseconds. Tests with explicit type declaration reveal that the four implementations offer quite similar runtime performance. C# 4.0 offers the best runtime performance, being C# 3.0 and VB 10 almost as fast as C# 4.0 (C# 4.0 is only 0.64% better than VB and C# 3.0). Finally, runtime performance of *StaDyn*, when variables are explicitly typed, is 9.22% lower than C# 3.0 and VB, and 9.92% in comparison with C# 4.0. This difference is caused by the greater number of optimizations that these production compilers perform in relation to our implementation.

The performance assessment of *StaDyn* when the exact single type of a `var` reference is inferred shows the repercussion of our approach. Runtime performance is the same as when using explicitly typed references[3] (in fact, the code generated is exactly the same). In this special scenario, *StaDyn* shows a huge performance improvement. If the compiler infers the exact type of `var` references, *StaDyn* is more than 2,322 and 3,195 times faster than VB and C# 4.0 respectively (notice that C# 3.0 does not support this feature). This vast difference is caused by the lack of static type inference of both VB and C# 4.0. When a reference is declared as dynamic, *every* operation over that reference is performed at runtime using reflection. Since the usage of reflective operations in the .NET platform has an important performance cost [51], the execution time is significantly incremented.

Figure 14 shows the progression of runtime performance when the compiler infers 1, 5, 10, 50 or 100 possible types. In the case of VB, runtime performance is almost constant: the mean value is 1,628,731 microseconds with a standard derivation of 1.876%. In the case of C# 4.0, runtime performance slightly raises from 2.193 seconds (1 possible type) to 2.290 (100 possible types). When there is no type information at all, it reaches the highest execution time: 2.46 seconds.

The runtime performance of *StaDyn* programs evolve in a different way. Execution time shows a linear raising regarding to the number of types inferred by the compiler. Therefore, the performance benefit drops while the number of possible types increases. As an example, *StaDyn* is more than 40 and 56 times faster than VB and C# respectively, when the compiler infers 100 possible types for a `var` reference.

---

[3]Actually, there is one microsecond improvement, but it may be due to any assessment bias such as the load of the operating system.

27

The final comparison to be established is when the compiler gathers
no static type information at all. In this case, runtime performance is the
worst in the three programming languages, because method invocation is
performed using reflection. However, *StaDyn* requires 33.85% and 22.65%
the time that VB and C# employ to run the same program.

Differences between our approach and both C# 4.0 and VB are justi-
fied by the amount of type information gathered by the compiler. *StaDyn*
continues collecting type information even when references are set as dy-
namic. Nevertheless, both C# 4.0 and VB perform no static type inference
once a reference has been declared as dynamic. This is the reason why
StaDyn offers the same runtime performance with explicit type declaration
and inference of the exact single type, involving a remarkable performance
improvement.

6 Related Work

Although several theoretical works exist, there have been few implementa-
tion approaches to include static and dynamic typing in the same program-
ming language.

6.1 Programming Languages Implementation

Strongtalk was one of the first programming language implementation that
included both dynamic and static typing in the same programming language.
Strongtalk is a major re-thinking of the Smalltalk-80 programming language
[52]. It retains the basic Smalltalk syntax and semantics [53], but a type sys-
tem is added to provide more reliability and a better runtime performance.
The Strongtalk type system is completely optional. This assumes that it is
the programmer's responsibility to ensure that types are sound in regard to

dynamic behaviour. Type checking is performed at compile-time, but it does not guarantee an execution without type errors. Although its type system is not completely safe, it implies a significant performance improvement.

Dylan is a high-level programming language, designed to allow efficient compilation of features commonly associated with dynamic languages [54]. Dylan permits both explicit and implicit variable 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 behaviour is similar to the one offered by dynamic languages. When the production configuration is selected, explicitly typed variables are checked using a static type system. However, types of generic references (references without type declaration) are not inferred at compile type –they are always checked at runtime.

The *Visual Basic .Net* programming language incorporates both dynamic and static typing [55]. 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, there are interoperation lacks between dynamic and static code because no static type inference is performed over dynamic references. Every type can be converted to a dynamic one, and vice versa. Therefore, all the type checking over dynamic references 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 it does not separate the dynamism concern: it forces the programmer to explicitly state in the source code which references are static and which ones are dynamic.

Boo is a recent object-oriented programming language that is both statically and dynamically typed with a Python inspired syntax [56]. In Boo,

references may be declared without specifying its type and the compiler performs type inference. However, references could only have one 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`. Any operation could be performed over a `duck` reference –no static typing is performed. It is allowed to convert any dynamic reference to a static one without any cast. Although this behaviour 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 was `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.

As mentioned, C# 4.0 includes the support of dynamically typed objects [18]. A new `dynamic` keyword has been added as a new type. The compiler performs no static type checking over any `dynamic` reference, making all the type verifications at runtime. The main objective of this enhancement is to offer direct access to dynamically typed code in IronPython, IronRuby and the JavaScript code in Silverlight. Dynamic code in C# 4.0 makes use of the DLR services [17].

6.2 Theoretical Research

Soft Typing [57] was one of the first theoretical works that applied static typing to a dynamically typed language such as Scheme [58]. However, soft typing does not control which parts in a program are statically checked, and static type information is not used to optimize the generated code either. The approach in [21] adds a `Dynamic` type to lambda calculus, including two

conversion operations (`dynamic` and `type-case`), producing a verbose code deeply dependent on its dynamism. The works of *Quasi-Static Typing* [59], *Hybrid Typing* [60] and *Gradual Typing* [61] perform implicit conversion between dynamic and static code, employing subtyping relations in the case of quasi-static and hybrid typing, and a consistency relation in gradual typing. Gradual typing already identified unification-based constraint resolution as a suitable approach to integrate both dynamic and static typing [62]. However, with gradual typing a dynamic type is implicitly converted into static without any static type-checking, because type inference is not performed over dynamic references. The main difference between these approaches and the work presented in this paper is that we perform type-checking even when dynamic types are used, detecting some type errors in dynamic code, improving its robustness and performance.

7   Conclusions

The *StaDyn* programming language includes both dynamic and static typing in the same programming language, improving the runtime flexibility and simplicity of the statically typed languages, and robustness and performance of the static ones. *StaDyn* allows both dynamic and static references in the same program, and it has been designed to make it easier to convert dynamically typed code into statically typed one, and vice versa. This paper describes the key techniques we have used to achieve these objectives.

Dynamic and static code can be seamlessly integrated because they share the same type system. Type inference is performed over dynamic and static references, facilitating the interoperation between dynamic and static code. Currently, IronPython and the Java Scripting API are two examples of the existing limitations of dynamic and static code interoperation (with C#

and Java respectively). Dynamic languages directly access to static ones, but not the other way round. For instance, if the programmer creates a class instance in Python, the statically typed code cannot directly retrieve its type. This lack is due to the fact that Python does not perform any static type checking at all. Therefore, types created by the programmer in Python are not included in the type system of the statically typed program.

StaDyn performs type inference over dynamic and static references, improving runtime performance and robustness. An initial runtime performance assessment has confirmed how performing type inference over dynamic references involves an important performance benefit. Although this benefit decreases as the number of possible inferred types increases, runtime performance of *StaDyn* is still significantly better than C# and VB when no type information of `var` references is inferred at all.

Future work will be centred on adding structural reflection to *StaDyn*. Structural reflection permits the dynamic addition, deletion and modification of members to classes and objects. *StaDyn* will perform concrete type evolution by means of its alias analysis mechanism. We are also working on including the DLR and ЯRotor back-ends to our current implementation.

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`. A formal description of the *StaDyn* type system is detailed in [25].

8   Acknowledgments

References

[1] Thomas, D., Fowler, C., and Hunt, A.: 'Programming Ruby, 2nd Edition'. Pragmatic Bookshelf, 2004.

[2] Thomas, D., Hansson, D.H., Schwarz, A., Fuchs, T., Breed, L., and Clark, M.: 'Agile Web Development with Rails. A Pragmatic Guide'. Pragmatic Bookshelf, 2005.

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

[4] ECMA-357: 'ECMAScript for XML (E4X) Specification, 2nd edition'. European Computer Manufacturers Association, 2005.

[5] Crane, D., Pascarello, E., and James, D.: 'Ajax in Action'. Manning Publications, 2005.

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

[7] Latteier, A., Pelletier, M., McDonough, C., and Sabaini, P.: 'The Zope Book', 2008. `http://www.zope.org/Documentation/Books/ZopeBook/`

[8] Django, the Web framework for perfectionists with deadlines: `http://www.djangoproject.com`, accessed September 2009.

[9] Ierusalimschy, R., de Figueiredo, L.H., and Filho, W.C.: 'Lua –an extensible extension language'. Software Practice & Experience, 26, (6), 1996, pp. 635–652.

[10] Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: 'The evolution Lua'. Proceedings of the third ACM SIGPLAN conference on History of programming languages, 2007, pp. 1–26.

[11] The Pythius website: `http://pythius.sourceforge.net`, accessed September 2009.

[12] Böllert, K.: 'On weaving aspects'. European Conference on Object-Oriented Programming (ECOOP), Workshop on Aspect Oriented Programming, 1999, pp. pp. 301–302.

[13] Ortin, F., and Cueva, J.M.: 'Dynamic Adaptation of Application Aspects'. Journal of Systems and Software 71, (3), 2004, pp. 229–243.

[14] JSR 223: 'Scripting for the Java Platform'. `http://www.jcp.org/en/jsr/detail?id=223`

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

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

[17] Hugunin, J.: 'Just Glue It! Ruby and the DLR in Silverlight'. MIX Conference, 2007.

[18] Torgersen, M.: 'New features in C# 4.0'. Microsoft Corporation, 2009.

[19] Pierce, B.C.: 'Types and Programming Languages'. The MIT Press, 2002.

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

[21] Abadi, M., Cardelli, L., Pierce, B.C., and Plotkin, G.: 'Dynamic typing in a statically typed language'. ACM Transactions on Programming Languages and Systems, 13, (2), (1991), pp. 237–268.

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

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

[24] Cardelli, L.: 'Type Systems'. The Handbook of Computer Science and Engineering, 1997.

[25] Ortin, F:. 'The StaDyn core Type System'. Technical Report. Computer Science Department, University of Oviedo. August 2009. `http://www.reflection.uniovi.es/stadyn/publications/stadyn.core.type.system.pdf`

[26] '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`

[27] Milner, R., Tofte, M., and Harper, R.: 'The Definition of Standard ML'. MIT Press, 1990.

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

[29] Cardelli, L.: 'Basic Polymorphic Typechecking'. Science of Computer Programming 8, 1998, pp. 147–172.

[30] Milner, R.: 'A theory of type polymorphism in programming'. Journal of Computer and System Sciences, 1978, pp. 348–375.

[31] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, F.K.: 'Efficiently computing static single assignment form and the control dependence graph'. ACM Transactions on Programming Languages and Systems 13 , (4), 1991, pp. 451–490.

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

[33] Odersky, M., Cremet, V., Röckl, C., and Zenger, M.: 'A Nominal Theory of Objects with Dependent Types'. European Conference on Object-Oriented Programming, 2002, pp. 201–224.

[34] Rémy, D. and Vouillon, J.: Objective ML: 'An effective object-oriented extension to ML'. Theory And Practice of Object Systems, 4, (1), 1998, pp. 27–50.

[35] Freeman, T., Pfenning, F.: 'Refinement types for ML'. Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, Toronto, Ontario, Canada, 1991, pp. 268–277.

[36] Plevyak, J., and Chien, A.A.: 'Precise concrete type inference for object-oriented languages'. SIGPLAN Notices 29, 10, Proceeding of the OOPSLA Conference (1994), pp. 324–340.

[37] 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.

[38] Bracha, G. 'Pluggable type systems'. OOPSLA workshop on revival of dynamic languages, October 2004.

[39] Haldiman, N., Denker, M., and Nierstrasz, O.: 'Practical, pluggable types for a dynamic language'. Computer Languages, Systems & Structures, 35, (1), April 2009, pp. 48–62.

[40] Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J.C.: 'F-bounded polymorphism for object-oriented programming'. Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM Press (1989), pp. 273–280.

[41] Odersky, M., Sulzmann, M., and Wehr, M.: 'Type inference with constrained types'. Theory and Practice of Object Systems 5, 1 (1999).

[42] Landi, W., and Ryder, B.G.: 'A Safe Approximate Algorithm for Interprocedural Pointer Aliasing'. Conference on Programming Language Design and Implementation, 1992, pp. 473–489.

[43] Diwan, A., McKinley, K.S., and Moss, J.E.B.: 'Type-Based Alias Analysis'. SIGPLAN Conference on Programming Language Design and Implementation, 1998, pp. 106–117.

[44] Emami, M., Ghiya, R., and Hendren, L.: 'Context-sensitive interprocedural points-to analysis in the presence of function pointers'. Proceedings of ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, 1994, pp. 242–256.

[45] Appel, A.W.: 'Modern Compiler Implementation in ML', Cambridge University Press, 1998.

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

[47] Parr, T.: 'The Definitive ANTLR Reference: Building Domain-Specific Languages'. Pragmatic Bookshelf, 2007.

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

[49] Ortin, F., Zapico, D., and Cueva, J.M.: 'Design Patterns for Teaching Type Checking in a Compiler Construction Course'. IEEE Transactions on Education, 50, (3), 2007, pp. 273–283.

[50] Redondo, J.M., Ortin, F. and Cueva, J.M.: 'Optimizing Reflective Primitives of Dynamic Languages'. International Journal of Software Engineering and Knowledge Engineering, 18, (6), 2008, pp. 759–783.

[51] Ortin, F., Redondo, J.M., Perez-Schofield, J.B.G.: 'Efficient Virtual Machine Support of Runtime Structural Reflection'. Science of Computer Programming, 74, (10), 2009, pp. 836–860.

[52] Bracha, G., and Griswold D.: 'Strongtalk: Typechecking Smalltalk in a Production Environment'. Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications, 1993, pp. 215–230.

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

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

[55] Vick, P.: 'The Microsoft Visual Basic Language Specification'. Microsoft Corporation, 2007.

[56] Boo home page: `http://boo.codehaus.org`, accessed September 2009.

[57] Cartwright, R., and Fagan, M.: 'Soft typing'. Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation, 1991.

[58] Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., and Felleisen, M.: 'Catching bugs in the web of program invariants'. Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation, 1996.

[59] Thatte, S.: 'Quasi-static typing'. Proceedings of the ACM Symposium on Principles of programming languages, 1990.

[60] Flanagan, C., Freund, S.N., and Tomb, A.: 'Hybrid types, invariants, and refinements for imperative objects'. International Workshop on Foundations and Developments of Object-Oriented Languages, 2006.

[61] Siek, J., and Taha, W.: 'Gradual Typing for Objects'. Proceedings of the 21st European Conference on Object-Oriented Programing, Lecture Notes In Computer Science 4609, 2007.

[62] Siek, J., and Vachharajani, M.: 'Gradual typing with unification-based inference'. Proceedings of the Symposium on Dynamic Languages, 2008.