A PROGRAMMING LANGUAGE TO FACILITATE THE TRANSITION FROM RAPID PROTOTYPING TO EFFICIENT SOFTWARE PRODUCTION

Francisco Ortin, Daniel Zapico and Miguel Garcia
Computer Science Department, University of Oviedo, Calvo Sotelo s/n, Oviedo, Spain

Keywords: Dynamic languages, Rapid prototyping, Separation of concerts, Union types, Alias analysis, Type inference.

Abstract: Dynamic languages are becoming increasingly popular for developing different kinds of applications, being rapid prototyping one of the scenarios where they are widely used. 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 to provide both approaches. Our language implementation keeps gathering type information at compile time, even when dynamic references are used. This type information is used to offer compile-time type error detection, direct interoperation between static and dynamic code, and better runtime performance. Following the Separation of Concerns principle, dynamically typed references can be easily turned into statically typed ones without changing the application source code, facilitating the transition from rapid prototyping to efficient software production. This paper describes the key techniques used in the implementation of *StaDyn* to obtain these benefits.

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, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. Common features of dynamic languages are meta-programming, reflection, mobility, and dynamic reconfiguration and distribution. One of the scenarios where they are widely used is the rapid development of software prototypes. Their ability to address quickly changing software requirements and their fast interactive edit-debug-test development method make dynamic languages ideal for the rapid creation of prototypes.

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 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 (Hugunin, 2007). Finally, Microsoft has just included a dynamic typing feature in C# 4.0, as part of the Visual Studio 2010. This new feature of C# 4.0 is the Microsoft response to the emerging use of dynamic languages such as Python or Ruby. 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 (Hugunin, 2007).

The great suitability of dynamic languages for rapid prototyping 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

Ortin F., Zapico D. and Garcia M. (2010).
A PROGRAMMING LANGUAGE TO FACILITATE THE TRANSITION FROM RAPID PROTOTYPING TO EFFICIENT SOFTWARE PRODUCTION.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 40-50
DOI: 10.5220/0002978500400050

some other task, or even after the program has been deployed (Pierce, 2002). 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.

Twitter can be seen as a practical example of how dynamic languages are suitable for rapid prototyping but they are not as appropriate as static ones for scalable, robust and efficient software production. Twitter began its life as a Ruby on Rails application but, in 2008 the Twitter developers started replacing the back-end Ruby services with applications running on the JVM written in Scala (Venners, 2009). They changed to Scala because, in their opinion, *the Ruby language lacks some things that contribute to reliable, high performance code*, and they want their code *to be correct and maintainable* (Venners, 2009).

Since translating an implementation from one programming language to another is not a straightforward task, 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 (Meijer and Drayton, 2004). Static typing allows 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 (Abadi et al., 1991). Hence, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems (Abadi et al., 1994). That is, *static typing where possible, dynamic typing when needed* (Meijer and Drayton, 2004).

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 the rapid development of dynamically typed prototypes, and the later conversion to the final application with a high level of robustness and runtime performance. The programmer indicates whether high flexibility is required (dynamic typing) or *correct*[1] execution (static) is preferred. 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. The result is that *StaDyn* allows the separation of the dynamism concern (Hürsch and Lopes, 1995). This feature facilitates turning rapidly developed prototypes into a final robust and efficient application.

In this paper, we present an overview of the techniques we have used to design and implement our programming language in order to facilitate the transition from rapid prototyping to efficient software production. 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 presents the key implementation decisions, and the results of a runtime performance assessment is presented in Section 5, and Section 6 discusses related work. Finally, Section 7 presents the conclusions and future work.

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

2 STATIC AND DYNAMIC TYPING

2.1 Static Typing

A language is said to be *safe* if it produces no execution errors that go unnoticed and later cause arbitrary behavior (Tucker, 1997), following the notion that well-typed programs should not go *wrong* (i.e., reach a *stuck* state on its execution) (Pierce, 2002). 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.

In order to ensure that no type error is produced at runtime, statically typed languages employ a pessimistic policy regarding to program compilation. This pessimism causes compilation errors in programs that do not produce any error at runtime. C# code shown in Figure 1 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.

This intend of validating the correctness of a pro-

```
public class Test {
  public static void Main() {
    object[] v = new object[10];
    int summation = 0;
    for (int i = 0; i < 10; i++) {
      v[i] = i+1;
      summation += v[i]; // Compilation error
    }
  }
}
```

Figure 1: Not compilable C# program that would not produce any runtime error.

gram before its execution makes statically typed languages to be less appropriate for rapid prototyping. Almost every programming language that supports interactive development environments (those where code can be evaluated, tested, modified and reevaluated while the application is running) implements a dynamic type system.

2.2 Dynamic Typing

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. This is a too optimistic approach that causes a high number of runtime type errors (commonly throwing runtime exceptions) that might have been detected at compile time. This approach compiles programs that might have been identified as erroneous statically. The C# 4.0 source code in Figure 2 is an example of this too optimistic approach. C# 4.0 has just included the `dynamic` type to permit the use of dynamically typed references. This erroneous program is compilable, although a static type system might have detected the error before its execution.

```
public class Test {
  public static void Main() {
    dynamic myObject = "StaDyn";
    // No compilation error
    System.Console.Write(myObject*2);
  }
}
```

Figure 2: Compilable C# 4.0 program that generates runtime type errors.

2.3 The *StaDyn* Hybrid Approach

The *StaDyn* programming language performs type inference at compile time, even over dynamic references. The type information gathered statically is used to both increase the robustness of the programming language (notifying type error at compile time) and improve its runtime performance. Type-checking over dynamic references is more lenient in order to facilitate the rapid development of prototypes. However, if the type-checker detects that a type error is undoubtedly going to be produced at runtime, an error message is shown and compilation is stopped. Type-checking over static references is performed the same as in C#.

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 the techniques employed. A formal description of its type system is depicted in (Ortin and Perez-Schofield, 2008) and (Ortin, 2009). Implementation issues are presented in Section 4.

The *StaDyn* programming language is an extension of C# 3.0. Although our work could be applied to any object-oriented statically-typed programming language, we have used C# 3.0 to extend the behavior 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 dynamic property of `var` references is specified 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 changes the dynamism of a `var` reference, 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 (this is the reason why we did not include a new `dynamic` reserved word, like C# 4.0). It is also possible to make parts of an application more adaptable, maintaining the robustness and runtime performance of the rest of the program.

3.1 Single Static Assignment

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 (Milner et al., 1990) or Haskell (Hudak et al., 1991) 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 3 is an example of this capability. The `number` reference has different types in the same scope. It is initially set to a string, and a double 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.

```
using System;
class Test {
  public static void Main() {
    Console.Write("Enter a number, please: ");
    var number = Console.In.ReadLine();
    Console.WriteLine("Number of digits: {0}.",
                             number.Length);
    number = Math.Pow(Convert.ToInt32(number), 2);
    Console.WriteLine("The square is {0}.", number);
    int digits = number.Length; // * Compilation error
  }
}
```

Figure 3: A reference with different types in the same scope.

In order to obtain this behavior, we have developed an implicit parametric polymorphic type system (Cardelli, 1988) 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 (Milner, 1978). 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 (Cytron et al., 1991). 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.

Figure 4 shows the corresponding program after applying the AST transformation to the source code in Figure 3. The AST represented by the source code in Figure 4 is the actual input to the type inference system. Each `number` reference will be inferred to a single static type –in our example, string and double, respectively.

```
using System;
class Test {
  public static void Main() {
    Console.Write("Enter a number, please: ");
    var number0 = Console.In.ReadLine();
    Console.WriteLine("Number of digits: {0}.",
                             number0.Length);
    var number1 = Math.Pow(Convert.ToInt32(number0), 2);
    Console.WriteLine("The square is {0}.", number1);
    int digits = number1.Length; // * Compilation error
  }
}
```

Figure 4: Corresponding program after the SSA transformation.

3.2 Static Duck Typing

Duck typing[2] is a property offered by most 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. One of the outcomes of duck typing is that it supports polymorphism without using inheritance. Therefore, the role of the abstract methods and interface as a mechanism to specify a contract is made redundant. Since it is not necessary to define polymorphic inheritance hierarchies, software can be developed more quickly.

There exist statically typed programming languages such as Scala (Odersky et al., 2002) or OCaml (Rémy and Vouillon, 1998) 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

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

provides implicit structural typing, variables should only have one type in the same scope, and this type is the most general possible (principal) type (Freeman and Pfenning, 1991). 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 may point 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 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) (Plevyak and Chien, 1994) 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 5 shows this feature. The reference `reference` may point to either an `StringBuilder` or a `String` object. Both objects have the `Length` 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 –in fact, their only common superclass is `Object`. Since type inference system is *flow-sensitive* and uses *concrete* types, the programmer obtains a safe static duck-typing system.

```
public static int f(bool condition) {
  var reference;
  if (condition)
    reference = new StringBuilder("ICSOFT");
  else
    reference = "ICSOFT'2010";
  return reference.Length;
}
```

Figure 5: Static *duck* typing.

The key technique we have used to obtain this concrete-type flow-sensitiveness is *union types* (Pierce, 1992). Concrete types are first obtained by the abovementioned 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).

3.3 From Dynamic to Static Typing

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 (Bracha, 2004) and (Haldiman et al., 2009). 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.

```
using System;
using System.Text;
public class Test {
  public static int g(string str) {
    var reference;
    switch(Random.Next(1,3)) {
    case 1: reference=new StringBuilder(str);
            break;
    case 2: reference = str;
            break;
    default: reference=new Exception(str);
    }
    return reference.Lenght;
  }
}
```

Figure 6: Static `var` reference.

The source code in Figure 6 defines a `g` method, where `reference` may point to a `StringBuilder`, `String` or `Exception` object. If we want to compile this code to rapidly develop a prototype, we can pass the compiler the *everythingDynamic* option. However, although we are compiling the code in the optimistic configuration, the compiler shows the following error message:

> *Error No Type Has Member (Semantic error). The dynamic type '$\bigvee$([Var(8)=StringBuilder] ,[Var(7)=String] ,[Var(6)=Exception])' has no valid type type with 'Lenght' member.*

The error is produced because no public `Lenght` property (it has been misspelled) is implemented in the `String`, `StringBuffer` or `Exception` classes.

This message shows how type-checking is performed at compile time even in dynamic scenarios, providing early type error detection. This feature improves the way most dynamic languages work. As an example, in the erroneous program in Figure 2 that C# compiles without any error, *StaDyn* detects the error at compile time.

It is worth noting that setting a reference as dynamic does not imply that every message could be passed to that reference; static type-checking is still performed. The major change is that the type system is more 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, then a type error is shown.

Once the programmer has found out the misspelling error, she will modify the source code to correctly access the `Length` property. If the program is once again compiled with the *everythingDynamic* option, the executable file is generated. In this case, the compiler accepts passing the `Length` message, because both `String` and `StringBubuilder` (but not `Exception`) types offer that property. With dynamic references, type checking succeeds if at least one of the types that compose the union type is valid. The actual type will be discovered at runtime, checking that the `Length` property can be actually accessed, or throwing `MissingMethodException` otherwise.

Actually, the programmer does not need to set all the `var` references in a compilation unit 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. Each *StaDyn* source code file may have a corresponding XML document specifying its dynamism concern.

The generated `g` function program will not produce any runtime type error because the random number that is generated will always be 1 or 2. However, if the programmer, once the prototype has been tested, wants to generate the application using the static type system, she may use the *everythingStatic* option. When this option is used, no XML dynamism file is analyzed and static typing is performed over every `var` reference in that compilation unit. In this case, the compilation of the `g` method will produce an error message telling that `Length` is not a property of `Exception`. The programmer should then modify the source code to compile this program with the robustness and efficiency of a static type system, but without requiring to translate the source code to a new programming language.

3.4 Constraint-based Type System

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 (Cardelli, 1988) more straightforward for the programmer than the one offered by Java, C# (F-bounded) and C++ (unbounded) (Canning et al., 1989).

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 7. 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.

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

Figure 7: Implicitly typed parameters.

For this purpose we have enhanced the *StaDyn* type system to be constraint-based (Odersky et al., 1999). Types of methods in our object-oriented language have an ordered set of constraints specifying the set of restrictions that must be fulfillled by the parameters. In our example, the type of the `upper` method is:

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

This means that the type of the parameter (α) should implement a public `ToUpper` method with no parameters (*void*), 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 no error, but it also infers the resulting type as a string. Type constraint fulfillment is, thus, part of the type inference mechanism (the concrete algorithm could be consulted in (Ortin, 2009)).

3.5 Assignment Constraints

Using implicitly typed attribute references, it is possible to create the generic `Node` class shown in Figure 8. The `Node` class can hold any data of any type. Each time the `setData` method is called, the new concrete type of the parameter is saved as the `data` field 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.

```
public class Node {
  private var data;
  private var next;
  public Node(var data, var next) {
    this.data = data;
    this.next = next;
  }
  public var getData() { return data; }
  public void setData(var data) {
    this.data=data;
  }
}
public class Test {
  public static void Main() {
    var node = new Node(1, 0);
    int n = node.getData();
    bool b = node.getData(); // * Error
    node.setData(true);
    int n = node.getData();  // * Error
    bool b = node.getData();
  }
}
```

Figure 8: Implicitly typed attributes.

Implicitly typed attributes extend the constraint-based behavior 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 `node` attribute is modified each time the `setData` method (and the constructor) is invoked. This does not imply a modification of the whole `Node` type, only the type of the single `node` object –thanks to the *concrete* type system employed.

For this purpose we have added a new kind of *assignment* constraint to the type system (Ortin, 2009). 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 (Ortin, 2009)).

3.6 Using both Static and Dynamic References

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 9 uses the `getString` and `upper` methods of Figure 7. `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 7) arises. That is, how optimistic (dynamic) code could interoperate with pessimistic (static) one. An example is shown in Figure 9.

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

Figure 9: Dynamic and static code interoperation.

The first invocation is correct regardless of the dynamism of `parameter`. 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 (Ortin, 2009). Promotion of static references to dynamic ones is more flexible: static references should fulfill at least one constraint from the set of alternatives.

3.7 Type-based Alias Analysis

The problem of determining if a storage location may be accessed in more than one way is called *Alias Analysis* (Landi and Ryder, 1992). 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 10 uses the `Node` class previously shown. Initially, the `list` reference points to a node whose data is an integer. If we get the `data` inside the `Node` object inside the `List` object, we get an integer. Then a new `Node` that holds a `bool` value is inserted at the beginning of the `list`. Repeating the previous access to the `data` inside the `Node` object inside the `List` object, a `bool` object is then obtained.

```
public class List {
  private var list;
  public List(var data) {
    this.list = new Node(data, 0);
  }
  public void insert(var data) {
    this.list = new Node(data, this.list);
  }
  public static void Main() {
    var aList = new List(1);
    int n1 = aList.list.getData();
    aList.insert(true);
    int n2 = aList.list.getData(); // * Error
    bool b = aList.list.getData();
  }
}
```

Figure 10: Alias analysis.

The alias analysis algorithm implemented is type-based (uses type information to decide alias) (Diwan et al., 1991), inter-procedural (makes use of inter-procedural flow information) (Landi and Ryder, 1992), context-sensitive (differentiates between different calls to the same method) (Emami et al., 1994), and may-alias (detects all the objects a reference *may* point to; opposite to *must* point to) (Appel, 1998).

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.

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. We have used the AntLR language processor tool to implement lexical and syntactic analysis (Parr, 2007). Abstract Syntax Trees (ASTs) have been implemented following the *Composite* design pattern and each pass over the AST implements the *Visitor* design pattern.

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 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 (Ortin et al., 2007).

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 back-end. However, we have designed the code generator module following the *Bridge* design pattern to add both the DLR (Dynamic Language Runtime) (Hugunin, 2007) and the ЯROTOR (Redondo and Ortin, 2008) back-ends in the future.

5 RUNTIME PERFORMANCE

We have evaluated the performance benefits obtained with the inclusion of dynamic and static typing in the same programming language. In this paper we only summarize the results –detailed information can be consulted in (Ortin et al., 2010). We have compared C# 4.0 Beta 2, Visual Basic 10 (VB) and current version of *StaDyn*. These have been the results:

- Tests with explicit type declaration revealed that the three implementations offer quite similar runtime performance. C# offers the best runtime performance, being VB almost as fast as C# (C# is only 0.64% better than VB). Finally, runtime performance of *StaDyn*, when variables are explicitly typed, is 9.22% lower than VB, and 9.92% in comparison with C#. This difference may be 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 (in fact, the code generated is exactly the same). In this special scenario, *StaDyn* shows a huge performance improvement: *StaDyn* is more than 2,322 and 3,195 times faster than VB and C# respectively. This vast difference is caused by the lack of static type inference of both VB and C#. 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 (Ortin et al., 2009), the execution time is significantly incremented.

- While the number of possible types inferred by our compiler increases, execution time shows a linear raising regarding to the number of types inferred by the compiler. However, C# and VB maintain their runtime performance almost constant. Therefore, the performance benefit drops while the number of possible types increases. As an example, the runtime performance benefit of *StaDyn* drops to 40 and 56 times better than VB and C# respectively, when the compiler infers 100 possible types for a `var` reference.
- 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#, respectively, employ to run the same program.

Differences between our approach and both C# and VB are justified by the amount of type information gathered by the compiler. *StaDyn* continues collecting type information even when references are set as dynamic. Nevertheless, both C# 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

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 (Bracha and Griswold, 1993). It retains the basic Smalltalk syntax and semantics, but a type system is added to provide more reliability and a better runtime performance. The Strongtalk type system is completely optional, following the idea of *pluggable* type system (Bracha, 2004). This feature facilitates the transition from rapid prototyping to efficient software production.

Dylan is a high-level programming language, designed to allow efficient compilation of features commonly associated with dynamic languages (Shalit, 1996). 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. When the production configuration is selected, explicitly typed variables are checked using a static type system.

Boo is a recent object-oriented programming language that is both statically and dynamically typed (Codehaus Foundation, 2006). 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. The Boo compiler provides the *ducky* option that interprets the `Object` type as if it was `duck`, i.e. dynamically typed.

The *Visual Basic .Net* programming language incorporates both dynamic and static typing. 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.

As mentioned, C# 4.0 includes the support of dynamically typed objects. 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.

There are some theoretical research works on hybrid dynamic and static typing as well. *Soft Typing* (Cartwright and Fagan, 1991) was one of the first theoretical works that applied static typing to a dynamically typed language such as Scheme. 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 (Abadi et al., 1991) 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* (Thatte, 1990), *Hybrid Typing* (Flanagan et al., 2006) and *Gradual Typing* (Siek and Taha, 2007) 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. None of them separates the dynamism concern. Gradual typing already identified unification-based constraint resolution as a suitable approach to integrate both dynamic and static typing (Siek and Vachharajani, 2008). 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.

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, facilitating the transition from rapid prototyping to efficient software production. Each compilation unit can be built in a dynamic or static configuration, and the dynamism of each single reference can be also specified without changing the semantics of the whole program.

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.

StaDyn performs type inference over dynamic and static references, improving runtime performance and robustness. A 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 centered on integrating *StaDyn* in a visual IDE suitable for both rapid prototyping and final efficient application development. Our idea is to extend Visual Studio 2010 to offer services such as interactive code evaluation, substitution of implicitly typed dynamic references with explicit static ones, refactoring facilities to make dynamic code become statically typed, and showing type errors and warnings even over dynamic code.

Current release of the *StaDyn* programming language and its source code are freely available at http://www.reflection.uniovi.es/stadyn. A formal description of the *StaDyn* type system is detailed in (Ortin, 2009).

ACKNOWLEDGEMENTS

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. It has been also funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation; project TIN2008-00276 entitled *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software*.

REFERENCES

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

Abadi, M., Cardelli, L., Pierce, B., Rémy, D., and Taylor, R. W. (1994). Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5:92–103.

Appel, A. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

Bracha, G. (2004). Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*.

Bracha, G. and Griswold, D. (1993). Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 215–230, New York, NY, USA. ACM.

Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. (1989). F-bounded polymorphism for object-oriented programming. In Press, A., editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280.

Cardelli, L. (1988). Basic polymorphic typechecking. *Science of Computer Programming*, (8):147–172.

Cartwright, R. and Fagan, M. (1991). Soft Typing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, pages 278–292.

Codehaus Foundation (2006). Boo, a wrist friendly language for the CLI. http://boo.codehaus.org.

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

Diwan, A., McKinley, K., and Moss, J. (1991). Type-based alias analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, pages 106–117.

Emami, M., Ghiya, R., and Hendren, L. (1994). 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*, pages 224–256.

Flanagan, C., Freund, S., and Tomb, A. (2006). Hybrid types, invariants, and refinements for imperative objects. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)*.

Freeman, T. and Pfenning, F. (1991). Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277.

Haldiman, N., Denker, M., and Nierstrasz, O. (2009). Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures*, 35:48–62.

Hudak, P., Jones, S., , and Wadler, P. (1991). Report on the programming language haskell version 1.1. Technical report, Departments of Computer Science, University of Glasgow and Yale University.

Hugunin, J. (2007). Just glue it! Ruby and the DLR in Silverlight. In *MIX Conference*.

Hürsch, W. and Lopes, C. (1995). Separation of concerns. Technical Report UN-CCS-95-03, Northeastern University, Boston, USA.

Landi, W. and Ryder, B. (1992). A safe approximate algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 473–489.

Meijer, E. and Drayton, P. (2004). Dynamic typing when needed: The end of the cold war between programming languages. In *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages*.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML*. The MIT Press.

Odersky, M., anb C. Röckl, V. C., and Zenger, M. (2002). A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 201–224.

Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55.

Ortin, F. (2009). The StaDyn core type system. Technical report, Computer Science Department, University of Oviedo, Spain.

Ortin, F. and Perez-Schofield, J. B. G. (2008). Supporting both static and dynamic typing. In *Programming and Languages Conference (PROLE)*, pages 215–232.

Ortin, F., Redondo, J. M., and Perez-Schofield, J. B. G. (2009). Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming*, 74.

Ortin, F., Zapico, D., and Cueva, J. M. (2007). Design patterns for teaching type checking in a compiler construction course. *IEEE Transactions on Education*, 50(3):273–283.

Ortin, F., Zapico, D., Perez-Schofield, J. B. G., and Garcia, M. (2010). Including both static and dynamic typing in the same programming language. *IET Software*, (to be published).

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf.

Pierce, B. C. (1992). Programming with intersection types and bounded polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA.

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

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

Redondo, J. M. and Ortin, F. (2008). Optimizing reflective primitives of dynamic languages. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):759–783.

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

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

Siek, J. G. and Taha, W. (2007). Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*.

Siek, J. G. and Vachharajani, M. (2008). Gradual typing with unification-based inference. In *Dynamic Languages Symposium*.

Thatte, S. (1990). Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 367–381, New York, NY, USA. ACM.

Tucker, A. B. (1997). *Type Systems. The Computer Science and Engineering Handbook*. CRC Press.

Venners, B. (2009). Twitter on Scala. a conversation with Steve Jenson, Alex Payne, and Robey Pointer. *Artima Developer*.