IDE Support to Facilitate the Transition from Rapid Prototyping to Robust Software Production

Francisco Ortin
University of Oviedo
C/Calvo Sotelo s/n, 33007, Oviedo, Spain
ortin@lsi.uniovi.es

Anton Morant
Wolfson College, University of Oxford
Linton Road, Oxford, UK, OX26UD
anton.morant@comlab.ox.ac.uk

ABSTRACT

Dynamic languages are becoming increasingly popular for different software development scenarios such as rapid prototyping because of the flexibility and agile interactive development they offer. The benefits of dynamic languages are, however, counteracted by many limitations produced by the lack of static typing. In order to obtain the benefits of both approaches, some programming languages offer a hybrid dynamic and static type system. The existing IDEs for these hybrid typing languages do not provide any type-based feature when dynamic typing is used, lacking important IDE facilities offered for statically typed code. We have implemented a constraint-based type inference system that gathers type information of dynamic references at compile time. Using this type information, we have extended a professional IDE to offer those type-based features missed for dynamically typed code. Following the *Separation of Concerns* principle, the IDE has also been customized to facilitate the conversion of dynamically typed code into statically typed one, and vice versa.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Classifications – *C#, object-oriented languages*; D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments*

General Terms

Languages

Keywords

Hybrid dynamic and static typing, IDE support, type inference, autocomplete, separation of concerns.

1. INTRODUCTION

Dynamic languages have recently turned out to be really suitable for specific scenarios such as rapid prototyping, Web development, interactive programming, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. Common features of dynamic languages are metaprogramming, reflection, mobility, and dynamic reconfiguration and distribution. 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 C# 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 generate the structure of a high-level source code document. The Dynamic Language Runtime (DLR) adds to the .NET platform a set of services to facilitate the implementation of dynamic languages. A new `dynamic` type has been included in C# 4.0 to support dynamically typed 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 offers direct access to dynamically typed code in IronPython, IronRuby and the JavaScript code in Silverlight.

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 less opportunities for compiler optimizations. 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. 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 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 4). 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 [1]. There are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems, following the idea of static typing where possible, dynamic typing when needed [1].

We have developed an extension of C#, called *StaDyn*, which supports both static and dynamic typing. *StaDyn* 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. *StaDyn* allows the separation of the *dynamism* concern [2], facilitating the transition from rapidly developed prototypes to final robust and efficient applications.

In this paper, we present an extension of the Visual Studio (VS) IDE that facilitates this transition from rapid prototyping to robust software production. It also supports converting statically typed code into more flexible dynamically typed one, reducing the changes in the source code. Although the IDE extension is cur-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA.
Copyright 2011 ACM 978-1-4503-0599-0/11/05... $10.00.

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

rently based on *StaDyn*, the work presented can be also applied to the new `dynamic` type included in C# 4.0 –in fact, *StaDyn* is an extension of C# 3.0.

2. THE *STADYN* LANGUAGE

This section presents a summary of the distinguishing features of the *StaDyn* programming language. A more detailed description can be consulted in [3], and its formal specification in [4].

We have extended the use of the C# `var` implicitly typed local references. The type of references can still be explicitly declared, while it is also possible to use the `var` keyword to declare implicitly typed non-initialized local variables, parameters, return types and fields (see the example code in Figure 1). For this purpose, *StaDyn* implements a type inference (reconstruction) algorithm [5] and its type system has been extended to be constraint-based [6].

A `var` variable can have different types in the same scope. This is a common feature of dynamic languages, although *StaDyn* provides it with compile-time error detection. The example code in Figure 1[2] shows how the `alias` variable first holds an `int` value (line 34) and soon after an `Alias` object (line 38). The static type system of C# allows the compilation of the lines 35 and 39 in Figure 1.

Duck typing 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 they have a related inheritance hierarchy or not. The *StaDyn* programming language offers *static* duck typing. The benefit provided is not only that duck typing is supported, but also that it is statically typed. 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. In line 31 (Figure 1) the `x` field can be accessed because both `Circumferences` and `Rectangles` provide this field. In case the `figure` reference would also be pointing to a third `Triangle` object (as happens in Figure 2), an error message would be shown by the compiler because `Triangle` objects do not provide an `x` field.

Since this analysis is performed at compile time, the programmer benefits from both early type error detection and better runtime performance. We have defined a new interpretation of union and intersection types to implement this feature [4].

StaDyn permits the use of both static and dynamic `var` references –we do not include a new `dynamic` type as C# 4.0. The *dynamism* concern is not explicitly stated in the source code; it is specified in a separate XML file [3], transparently managed by the IDE. This makes it 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.

Depending on their *dynamism* concern, type checking and type inference is more restrictive (static) or lenient (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 [7]. As an example, the compiler would show an error if the `x` field of the `figure` reference in Figure 2 would be accessed, being `figure` declared as static: not all the possible types of `figure` provide an `x` field (i.e., `Triangle`). On the other hand, the compiler would accept the program if the `figure` reference was dynamic: at least one possible type of `figure` (both `Circumference` and `Rectangle`) provides an `x` field.

```
01: using System;
02: class Circumference {
03:   public var x, y, radius;
04: }
05: class Rectangle {
06:   public var x, y, width, height;
07: }
08: class Triangle {
09:   public var x1, y1, x2, y2, x3, y3;
10: }
11: class Alias {
12:   private var theObject;
13:   public Alias(var theObject) {
14:     this.theObject = theObject;
15:   }
16:   public void incX1(var inc) {
17:     theObject.x1 = theObject.x1+inc;
18:   }
19:   public var getTheObject() {
20:     return theObject;
21:   }
22: }
23: class Program {
24:   public static int f() {
25:     var figure;
26:     if (Random.Next() % 2 == 0)
27:       figure = new Circumference();
28:     else
29:       figure = new Rectangle();
30:     // static duck typing
31:     return figure.x; // int is inferred
32:   }
33:   public static void Main() {
34:     var alias = f();
35:     int twiceX = alias * 2;
36:     var triangle = new Triangle();
37:     // different types, same scope
38:     alias = new Alias(triangle);
39:     alias.incX1(0.5);
40:     int x = triangle.x1; // comp. error
41:     double d = triangle.x1;
42:   }
43: }
```

Figure 1. Example *StaDyn* core.

The problem of determining if a storage location may be accessed in more than one way is called *Alias Analysis* [8]. 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.

The `alias` reference in line 38 (Figure 1) holds an `Alias` object that points to a `Triangle`. The `incX1` message is passed to the `alias` object that indirectly changes the type of the `x1 triangle`'s field to `double`. Afterwards, when the `x1` field of the `triangle` object is accessed, the compiler shows an error in line 40 and accepts the assignment in line 41, because the type-based alias analysis detects the new `double` type of the `x1 triangle`'s field.

The alias analysis algorithm we have implemented is type-based [9] (uses type information to decide alias), inter-procedural [8] (makes use of inter-procedural flow information), context-sensitive [10] (differentiates between different calls to the same method), and may-alias [11] (detects all the objects a reference *may* point to; opposite to *must* point to).

The static type information gathered by the *StaDyn* compiler is used to optimize the generated .NET code. Runtime performance has been compared with C# 4.0 and VB 10, entailing significant performance improvements [3].

When using dynamic references, execution time shows a linear increase in the number of types inferred by the compiler. The maximum runtime performance benefit was obtained when the exact single type of a dynamic reference, being *StaDyn* is more than 2,322 and 3,195 times faster than VB and C#, respectively.

[2] The constructors of `Circumference`, `Rectangle` and `Triangle` assign random values to their fields. Their implementations are omitted for the sake of brevity.

This performance benefit drops when the number of possible types increases. The worse scenario is when the type system does not infer any type information of dynamic references. In this case, *StaDyn* is 2.95 and 4.42 times faster than VB and C#, respectively [3].

3. IDE SUPPORT

We have extended the functionalities of VS to make use of the features provided by *StaDyn*. We have employed the Managed Package Framework (MPF) to implement new VS extension packages (*VSPackages*). In particular, two new *StaDyn* project and *StaDyn* language extension packages have been developed.

The basic common features offered are the creation of *StaDyn* projects, the common VS editing services for *StaDyn* files, the same color syntax highlighting of C#, and the typical *build*, *rebuild*, *clean* and *start* commands. Figure 2 shows a snapshot of the VS extension where one of the four *StaDyn* files in a project is being edited. Notice that dynamic references (i.e., `figure`) are displayed in red, denoting a kind of caution because dynamic type errors might be produced at runtime.

Taking advantage of the *StaDyn* type inference (reconstruction) system, VS now displays all the possible messages that can be passed to any `var` reference. Unlike C#, IntelliSense works even with dynamic references. Figure 2 shows how the dynamic `figure` reference accepts the union of all the messages in `Circumference`, `Rectangle` and `Triangle`. In case it was static, the union type would only accept the intersection of their messages –see the type system formalization in [4].

As Figure 2 shows, the information of each message is displayed on the right of each member identifier as an IDE hint. Moreover, when the mouse moves over a `var` variable, VS shows a hint with the type information gathered. In Figure 2, a hint indicates that the inferred type of the `randomValue` statically typed variable is `int`[3].

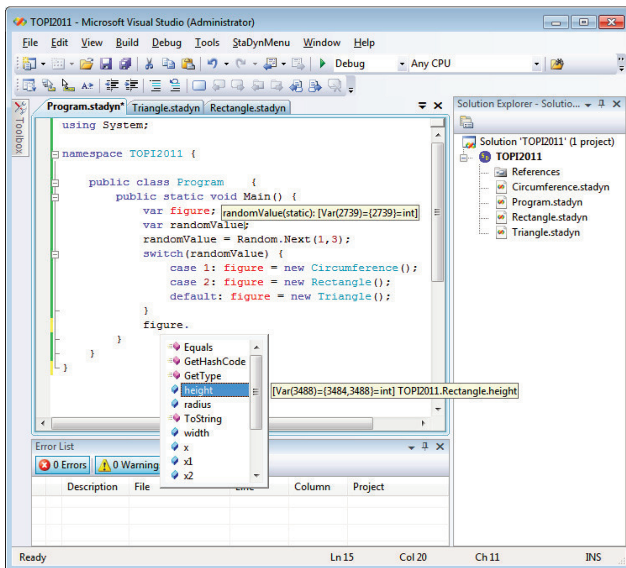

Figure 2. IntelliSense behavior with dynamic references.

The benefit of separating the *dynamism* concern (not explicitly stating it in the source code) is that it makes it easy to change dynamically typed code into statically typed one, and vice versa. In fact, *StaDyn* offers three different types of compilation: the default one that takes into consideration the specific dynamism of each single reference; the *everything dynamic* option, created for rapid prototyping, that considers every `var` reference as dynamic; and the *everything static* one that interprets all the `var` references as static. The two last options are displayed in the *StaDyn* menu shown in Figure 3, whereas the default option is included in the standard *Build* menu.

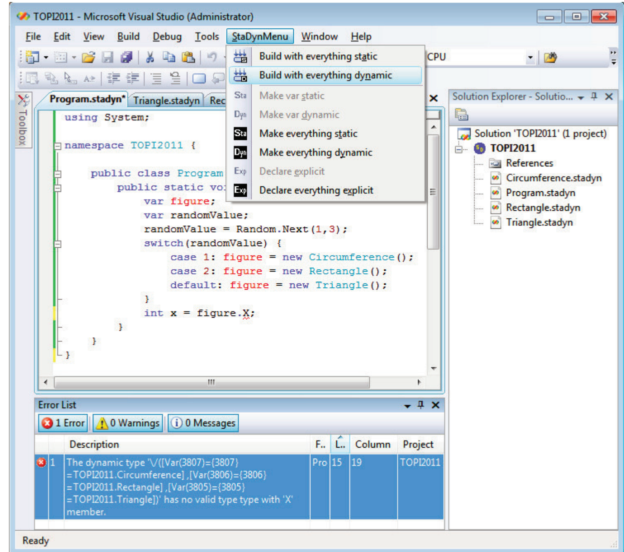

Figure 3. Compiler error using the *everything dynamic* option.

Figure 3 shows how the *StaDyn* compiler statically detects type errors, even when the *everything dynamic* compilation option is used. An error message saying *the dynamic type Circumference∨Rectangle∨Triangle has no valid X member* is shown, because none of these types offer any uppercase `X` public member. Therefore, setting a reference as dynamic does not imply that any message could be passed to it; static type-checking is still performed. This feature improves the robustness of dynamic (and hybrid) typing languages.

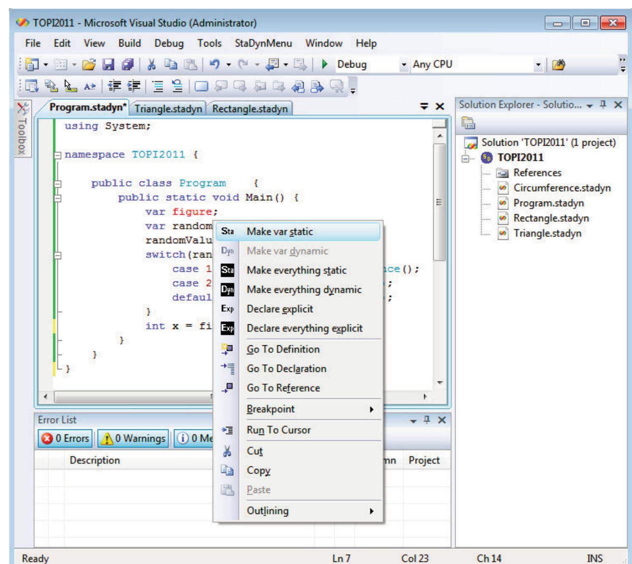

Figure 4. Changing the dynamism of a `var` reference.

---

[3] The numbers shown after *Var* are the unique identifiers set to each `var` reference [4].

Once the programmer finds out the error, he or she will modify the source code to correctly access the `x` (lowercase) field. If the program is compiled once again, the executable file will be generated. In this case, the compiler accepts passing the `x` message, because both `Circumference` and `Rectangle` (but not `Triangle`) types offer that field.

The generated 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, he or she may set the `figure` reference as static. As illustrated in Figure 4, it is only necessary to right click on the reference and select its dynamism. In this case, the compilation will produce an error message telling that `x` is not a valid member of `Triangle`. 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 the translation of the source code to a new programming language.

As Figure 4 shows, VS now allows making *every* `var` reference in a *StaDyn* file dynamic or static, without changing the source code (these options are also part of the new *StaDyn* menu). Using the static type inference system, it is also possible to explicitly declare `var` references. As an example, if we select this option with the `random` variable, its declaration will be changed from `var` to `int`.

4. RELATED WORK

There has been different hybrid dynamically and statically typed programming languages: from StrongTalk to C# 4.0, including Dylan, Cobra, Fantom, Boo, Thorn and VB .NET. Probably, the most similar to *StaDyn* is Boo. In Boo, a reference may be implicitly declared making the compiler infer its type (references could only have one unique type in the same scope), but 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. This approach follows the idea of separating the *dynamism* concern, but does not reduce the number of changes to be done in the source code. Boo also provides the BooLangStudio, a service language for VS 2008, currently released as an alpha version. Its features include syntax highlighting, building and debugging services and basic IntelliSense capabilities for statically typed references only.

Cobra is another hybrid typing programming language that provides an IDE support. The Cobra approach is similar to C# 4.0, offering a new `dynamic` for dynamic typing. The language is compiled to .NET assemblies. There are two IDEs, Visual Cobra and a plug-in for SharpDevelop, that offer editing, compiling and syntax highlighting. Neither of both facilitates the transition from dynamically typed code to statically typed one.

The Fantom programming language generates both JVM and .NET code, offering dynamic and static typing. Instead of adding a new type, dynamic typing is obtained with the `->` dynamic invoke operator. Unlike the dot operator, the dynamic invoke operator does not perform compile-time checking. Fantom does not follow the *Separation of Concerns* principle. The Fantom IDE can be installed either as a Netbeans plug-in or as a standalone IDE. Since no static type inference is performed, the Fantom IDE does not provide autocomplete with the dynamic invoke operator.

5. CONCLUSIONS

The benefits of both dynamic and static typing have made some programming languages include hybrid type systems to create both rapidly developed prototypes and robust and efficient software. However, the existing IDE for these languages do not facilitate the conversion of dynamically typed code into statically typed one and the other way around. For this purpose, we have extended the professional VS IDE, providing features such as autocomplete, type information and explicit type declaration for implicit dynamically typed references. Following the *Separation of Concerns* principle, we have implemented three different ways of compilation and different services to convert dynamically typed code into statically typed one (and vice versa), minimizing the changes in the application source code.

Although we have used the *StaDyn* programming language, our work could also be applied to other hybrid statically and dynamically typed languages such as C# 4.0. We are currently porting the code to VS 2010 –the current plug-in is only valid for VS 2008. Future work will be centered on making VS an interactive edit-debug-test environment similar to those provided for dynamic languages.

The current release of the *StaDyn* VS extension, its source code, and all the examples presented in this paper are freely available at http://www.reflection.uniovi.es/stadyn/download/2011/topi

6. ACKNOWLEDGMENTS

This work has been funded by Microsoft Research, under the project entitled *Extending dynamic features of the SSCLI*. 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*.

7. REFERENCES

[1] E. Meijer, and P. Drayton. Dynamic typing when needed: the end of the Cold War between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[2] W. Hürsch, and C. Lopes. *Separation of concerns*. Technical Report UN-CCS-95-03, Northeastern University, Boston, USA, 1995.

[3] F. Ortin, D. Zapico, J.B.G. Perez-Schofeld, and M. Garcia. Including both Static and Dynamic Typing in the same Programming Language. *IET Soft.*, 4(4): 268–282, August 2010.

[4] F. Ortin, and M. García. Union and Intersection Types to Support both Dynamic and Static Typing. *Inform. Process. Lett.*, 111(6): 278–286, February 2011.

[5] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3): 348–375, 1978.

[6] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Obj. Syst.*, 5(1): 35–55, January 1999.

[7] G. Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, October 2004.

[8] W. Landi, and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Programming Language Design and Implementation* (PLDI), June 1992.

[9] A. Diwan, K. McKinley, and J. Moss. Type-based alias analysis. In *Programming Language Design and Implementation* (PLDI), June 1991.

[10] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive inter-procedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation* (PLDI), June 1994.

[11] A. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.