The Journal of Systems and Software 113 (2016) 114–129

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Optimizing runtime performance of hybrid dynamically and statically typed languages for the .NET platform

Jose Quiroga [a], Francisco Ortin [a,*], David Llewellyn-Jones [b], Miguel Garcia [a]

[a] University of Oviedo, Computer Science Department, Calvo Sotelo s/n, Oviedo 33007, Spain
[b] Liverpool John Moores University, Department of Networked Systems and Security, James Parsons Building, Byrom Street, Liverpool L3 3AF, UK

ARTICLE INFO

Article history:
Received 16 June 2015
Revised 18 November 2015
Accepted 20 November 2015
Available online 30 November 2015

Keywords:
Hybrid static and dynamic typing
Dynamic language runtime
Runtime performance optimization

ABSTRACT

Dynamically typed languages have become popular in scenarios where high flexibility and adaptability are important issues. On the other hand, statically typed languages provide important benefits such as earlier type error detection and, usually, better runtime performance. The main objective of hybrid statically and dynamically typed languages is to provide the benefits of both approaches, combining the adaptability of dynamic typing and the robustness and performance of static typing. The dynamically typed code of hybrid languages for the .NET platform typically use the introspection services provided by the platform, incurring a significant performance penalty. We propose a set of transformation rules to replace the use of introspection with optimized code that uses the services of the Dynamic Language Runtime. These rules have been implemented as a binary optimization tool, and included as part of an existing open source compiler. Our system has been used to optimize 37 programs in 5 different languages, obtaining significant runtime performance improvements. The additional memory resources consumed by optimized programs have always been lower than the corresponding performance gains obtained.

1. Introduction

Dynamic languages have turned out to be suitable for specific scenarios such as rapid prototyping, Web development, interactive programming, dynamic aspect-oriented programming and runtime adaptive software (Ortin and Cueva, 2004). For example, in the Web development scenario, Ruby (Thomas et al., 2004) is used for the rapid development of database-backed Web applications with the Ruby on Rails framework (Thomas et al., 2005). This framework has confirmed the simplicity of implementing the DRY (Do not Repeat Yourself) (Hunt and Thomas, 1999) and the Convention over Configuration (Thomas et al., 2005) principles in a dynamic language. Nowadays, JavaScript (ECMA-357, 2005) is being widely employed to create interactive Web applications (Crane et al., 2005), while PHP is one of the most popular languages for developing Web-based views. Python (van Rossum et al., 2003) is used for many different purposes; two well-known examples are the Zope application server (Latteier et al., 2008) (a framework for building content management systems, intranets and custom applications) and the Django Web application framework (Django Software Foundation, 2015).

On the contrary, the type information gathered by statically typed languages is commonly used to provide two major benefits compared with the dynamic typing approach: early detection of type errors and, usually, significantly better runtime performance (Meijer and Drayton, 2004). Statically typed languages offer the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime – when the programmer efforts might be aimed at some other task, or even after the program has been deployed (Pierce, 2002). Moreover, avoiding the runtime type inspection and type checking performed by dynamically typed languages commonly involve a runtime performance improvement (Ortin et al., 2013, 2014a).

Since both approximations offer different benefits, some existing languages provide hybrid static and dynamic typing, such as Objective-C, Visual Basic, Boo, *StaDyn*, Fantom and Cobra. Additionally, the Groovy dynamically typed language has recently become hybrid, performing static type checking when the programmer writes explicit type annotations (Groovy 2.0) (Strachan, 2014). Likewise, the statically typed C# language has included the `dynamic` type in its version 4.0 (Bierman et al., 2010), indicating the compiler to postpone type checks until runtime.

The example hybrid statically and dynamically typed Visual Basic (VB) code in Fig. 1 shows the benefits and drawbacks of both typing

* Corresponding author. Tel.: +34 985 10 3172.
E-mail addresses: quirogajose@uniovi.es (J. Quiroga), ortin@uniovi.es, francisco.ortin@gmail.com (F. Ortin), D.Llewellyn-Jones@ljmu.ac.uk (D. Llewellyn-Jones), garciarmiguel@uniovi.es (M. Garcia).
URL: http://www.di.uniovi.es/~ortin (F. Ortin), http://www.flypig.co.uk/?style=3&page=research (D. Llewellyn-Jones), http://www.miguelgr.com (M. Garcia)

http://dx.doi.org/10.1016/j.jss.2015.11.041
0164-1212/

```
Module Figures
  Public Class Triangle
    Public edges(3) As Integer
    Public Sub New(edge1 As Integer,
             edge2 As Integer, edge3 As Integer)
      Me.edges = {edge1, edge2, edge3}
    End Sub
  End Class

  Public Class Square
    Public edges(4) As Integer
    Public Sub New(edge As Double)
      Me.edges = {edge, edge, edge, edge}
    End Sub
  End Class

  Public Class Circumference
    Public radius As Integer
    Public Sub New(rad As Integer)
      Me.radius = rad
    End Sub
  End Class

  Public Function TrianglePerimeter(
              poly As Triangle) As Double
    Dim result As Double = 0
    For Each edge In poly.edges
      result += edge
    Next
    Return result
  End Function

  Public Function PolygonPerimeter(poly) As Double
    Dim result As Double = 0
    For Each edge In poly.edges
      result += edge
    Next
    Return result
  End Function

  Sub Main()
    Dim perimeter As Double
    Dim triangle As Triangle = New Triangle(3,4,5)
    Dim square As Square = New Square(3)
    Dim circ As Circumference =
             New Circumference(4)

    perimeter = TrianglePerimeter(triangle)
    'compiler error
    perimeter = TrianglePerimeter(square)

    perimeter = PolygonPerimeter(triangle)
    perimeter = PolygonPerimeter(square)

    'runtime error
    perimeter = PolygonPerimeter(circ)
  End Sub
End Module
```

Fig. 1. Hybrid static and dynamic typing example in Visual Basic.

approaches. The statically typed `TrianglePerimeter` method computes the perimeter of a `Triangle` as the sum of the length of its `edges`. The first invocation in the `Main` function is accepted by the compiler; whereas the second one, which passes a `Square` object as argument, produces a compiler error. This error is produced even though the execution would produce no runtime error, because the perimeter of a `Square` can also be computed as the sum of its `edges`. In this case, the static type system is too restrictive, rejecting programs that would run without any error.

The `PolygonPerimeter` method implements the same algorithm but using dynamic typing. The `poly` parameter is declared as dynamically typed in VB by omitting its type. The flexibility of dynamic typing supports duck typing (Ortin et al., 2014b), meaning that any object that provides a collection of numeric `edges` can be passed as a parameter to `PolygonPerimeter`. Therefore, the first two invocations to `PolygonPerimeter` are executed without any error. However, the compiler does not type-check the `poly` parameter, and hence the third invocation produces a runtime error (the class `Circumference` does not provide an `edges` property).

As mentioned, the `poly.edges` expression in the `PolygonPerimeter` method is an example of duck typing, an important feature of dynamic languages. VB, and most hybrid languages for .Net and Java, implement this runtime type checking using introspection, causing a performance penalty (Ortin et al., 2014b). In general, the runtime type checks implemented by dynamic languages generally cause runtime performance costs (Redondo and Ortin, 2015). To minimize the use of these introspection services, a cache mechanism could be implemented to improve runtime performance of the dynamic inference of types.

The Dynamic Language Runtime (DLR) is a set of .Net libraries that provide, among other services, different cache levels for the typical operations of dynamically typed code. Although the DLR is exploited by the C# compiler (when the `dynamic` keyword is used) and some dynamic languages (e.g., IronPython 2+, IronRuby and PowerShell), it has not been used in any other hybrid typing language. Our research is based on the hypothesis that the DLR can be used to optimize different features of dynamic typing code in hybrid typing languages. The use of a runtime cache may incur a runtime performance penalty at start-up. It may also increase the memory resources used at runtime. Therefore, we must measure these values and evaluate when the use of the DLR may be appropriate.

The main contribution of this work is the optimization of the common dynamically typed operations of hybrid typing languages for the .Net platform using the DLR, evaluating the runtime performance gain obtained and the additional memory resources required. We have built a tool that processes binary .Net files compiled from the existing hybrid typing languages for that platform, and produces new binary files with the same behavior and better runtime performance. We have also included the proposed optimizations in the implementation of an existing compiler for .Net, obtaining similar results.

The rest of the paper is structured as follows. Section 2 describes the DLR architecture and its main components. The architecture of both the binary code optimizer and the optimizing compiler is presented in Section 3. That section also formalizes the transformation rules defined to optimize VB. Section 4 describes some implementation issues, and Section 5 presents the evaluation of runtime performance and memory consumption. Section 6 discusses related work, and Section 7 presents the conclusions and future work. Appendix A shows the dynamic typing operations supported by the DLR. Appendixes B, C, D, E and F present the optimization rules for VB, Boo, Cobra, Fantom and *StaDyn*, respectively.

2. The Dynamic Language Runtime

The Dynamic Language Runtime (DLR) is a set of libraries included in the .Net Framework 4 to support the implementation of dynamic languages (Chiles and Turner, 2015). The DLR is built on the top of the Common Language Runtime (CLR), the virtual machine of the .Net Framework. The DLR provides high-level services and optimizations common to most dynamic languages, such as a dynamic type checking, dynamic code generation and a runtime cache to optimize dynamic dispatch and method invocation (Chiles and Turner, 2015).