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, 33007, Oviedo, Spain*
[b] *Liverpool John Moores University, Department of Networked Systems and Security, James Parsons Building, Byrom Street, Liverpool, L3 3AF, UK*

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, 33007, Oviedo, Spain*
[b] *Liverpool John Moores University, Department of Networked Systems and Security, James Parsons Building, Byrom Street, Liverpool, L3 3AF, UK*

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 con-

*Corresponding author

Email addresses: `quirogajose@uniovi.es` (Jose Quiroga), `ortin@uniovi.es` (Francisco Ortin), `D.Llewellyn-Jones@ljmu.ac.uk` (David Llewellyn-Jones), `garciarmiguel@uniovi.es` (Miguel Garcia)

URL: `http://www.di.uniovi.es/~ortin` (Francisco Ortin), `http://www.flypig.co.uk/?style=3&page=research` (David Llewellyn-Jones), `http://www.miguelgr.com` (Miguel Garcia)

sumed by optimized programs have always been lower than the corresponding performance gains obtained.

Keywords: hybrid static and dynamic typing, dynamic language runtime, runtime performance optimization, .NET

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 [1]. For example, in the Web development scenario, Ruby [2] is used for the rapid development of database-backed Web applications with the Ruby on Rails framework [3]. This framework has confirmed the simplicity of implementing the DRY (Do not Repeat Yourself) [4] and the Convention over Configuration [3] principles in a dynamic language. Nowadays, JavaScript [5] is being widely employed to create interactive Web applications [6], while PHP is one of the most popular languages for developing Web-based views. Python [7] is used for many different purposes; two well-known examples are the Zope application server [8] (a framework for building content management systems, intranets and custom applications) and the Django Web application framework [9].

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 [10]. 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 [11]. Moreover, avoiding the runtime type inspection and type checking performed by dynamically typed languages commonly involve a runtime performance improvement [12, 13].

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) [14]. Likewise, the statically typed C# language has included the dynamic type in its version 4.0 [15], indicating the compiler to postpone type checks until runtime.

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

```
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
```

Figure 1: Hybrid static and dynamic typing example in Visual Basic.

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 [16], 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 [16]. In general, the runtime type checks implemented by dynamic languages generally cause runtime performance costs [17]. 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.

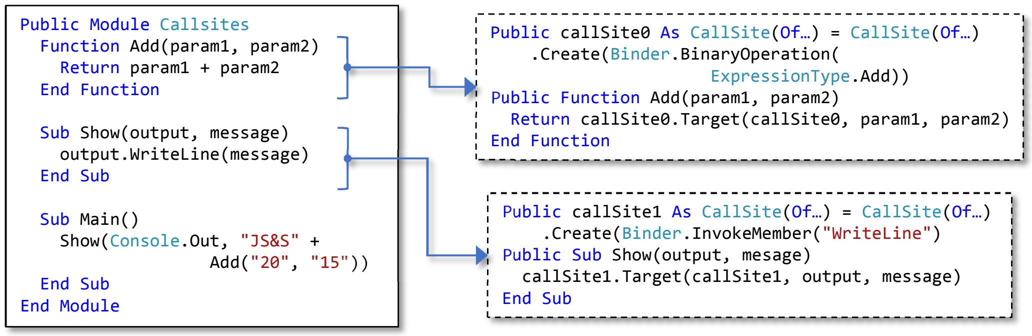
2. The Dynamic Language Runtime

The Dynamic Language Runtime (DLR) is a set of libraries included in the .NET Framework 4 to support the implementation of dynamic languages [18]. 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 [18]. Therefore, it facilitates the development of dynamic languages for the .NET platform, and provides interoperability among them. The DLR services are currently used in the implementation of the IronPython 2+, IronRuby and PowerShell dynamic languages. It is also used in C# 4+ to support the new `dynamic` type. This section briefly describes the components of the DLR used in our work; more detailed information can be consulted in [18].

The key elements of the DLR are call-sites, binders and its runtime cache. A call-site is any expression with (at least) one dynamically typed operand. The DLR adds the `CallSite` class to the .NET Framework to provide the dynamic typing services and optimizations for dynamically typed expression. Figure 2[1] shows two examples of dynamically typed operations executed with (right-hand side) and without (left-hand side) DLR `CallSites`. The complete list of the dynamically typed operations supported by the DLR is shown in Appendix A [18].

Figure 2 shows how a new `CallSite` instance is created for each single dynamically typed expression (the addition in `Add` and the method invocation in `Show`). Every `CallSite` receives a `CallSiteBinder` as an argument upon construction. A `CallSiteBinder` encapsulates the specific kind of expression represented by a `CallSite` (e.g., binary addition and method invo-

[1]The VB code has been simplified the following way: 1) `CallSite` type definitions are shortened, 2) lazy initializations of `CallSites` have been replaced by initializations in the declaration; and 3) arguments of `CallSiteBinders` have been omitted.



```
Public Module Callsites
  Function Add(param1, param2)
    Return param1 + param2
  End Function

  Sub Show(output, message)
    output.WriteLine(message)
  End Sub

  Sub Main()
    Show(Console.Out, "JS&S" +
                       Add("20", "15"))
  End Sub
End Module
```

```
Public callSite0 As CallSite(Of…) = CallSite(Of…)
    .Create(Binder.BinaryOperation(
                      ExpressionType.Add))
Public Function Add(param1, param2)
  Return callSite0.Target(callSite0, param1, param2)
End Function
```

```
Public callSite1 As CallSite(Of…) = CallSite(Of…)
    .Create(Binder.InvokeMember("WriteLine")
Public Sub Show(output, mesage)
  callSite1.Target(callSite1, output, message)
End Sub
```

Figure 2: Example VB program with (right-hand side) and without (left-hand side) DLR optimizations.

cation). With this information, the `CallSiteBinder` dynamically generates a method that computes that expression. Since the method is generated at runtime, the particular dynamic types of the operands are known. Therefore, the generated code does not need to consult the operand types, implying a runtime performance benefit [18, 13]. The types of the operands are stored in a cache implemented by the `CallSite`. Later invocations to the `CallSite` may produce a cache hit, if the operand types remain unchanged. Otherwise, a cache miss is produced; and another method is generated by the `CallSiteBinder`. `CallSites` implement three distinct cache levels, using introspection upon the third cache miss [18].

We previously measured that the runtime cache provided by the DLR provides a significant performance improvement compared to the use of introspection [16]. The key insight behind our work is to replace the dynamically typed operations (including the introspective ones) used by .NET languages with DLR `CallSites`, and evaluate if the new code provides significant performance improvements. Besides, we should measure the cost of the dynamic code generation method implemented by the DLR, because it may incur a performance penalty at start-up. The additional memory resources consumed by the DLR must also be evaluated.

3. Optimization of .Net Hybrid Typing Languages

As mentioned, we optimize the existing hybrid typing languages for the .NET platform, using the services provided by the DLR. These optimizations have been applied to the language implementations following the two differ-

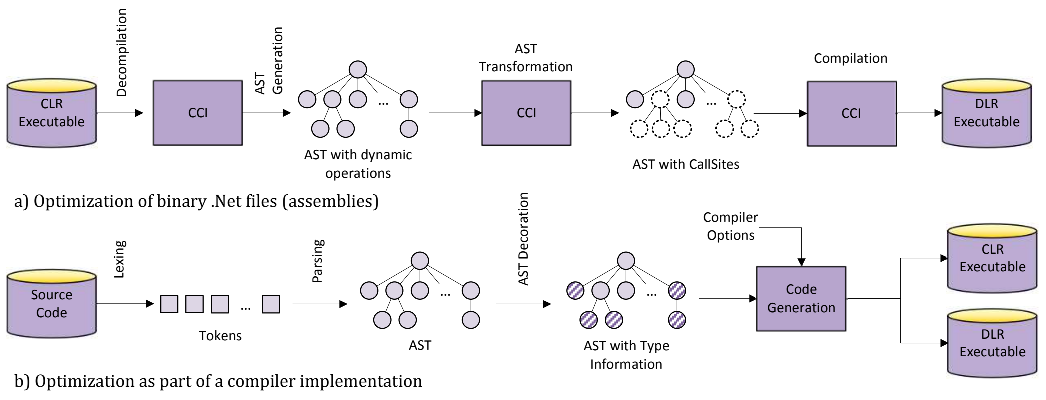


Figure 3: Architecture of the two optimization approaches.

ent approaches shown in Figure 3: as an optimizer of .NET executable files (Figure 3.a), and as part of an open source compiler (Figure 3.b).

Figure 3.a shows the binary optimization approach implemented for programs coded in VB, Boo, Cobra and Fantom. Using the Microsoft Research Common Compiler Infrastructure (CCI) [19], the Abstract Syntax Trees (ASTs) of binary files (i.e., assemblies) are obtained. Our optimizer traverses each AST, searching for dynamically typed expressions. Those expressions are replaced by semantically equivalent expressions that use DLR `CallSite`s (Section 3.1). Finally, the ASTs are saved as new optimized binary files that use the DLR.

The proposed optimizations have also been included in a compiler (Figure 3.b). We have modified the existing implementation of the *StaDyn* hybrid typing language [20]. *StaDyn* is an extension of C# that provides type inference of `dynamic` references. The *StaDyn* compiler performs type inference with 5 traversals of the AST [21]. Afterwards, the code generation phase generates binary files for the CLR. We have added a new `server` command-line option to the compiler. When this option is passed, we optimize the only dynamically typed references that the *StaDyn* compiler does not manage to infer: `dynamic` method arguments (Appendix F). Otherwise, the types of the `dynamic` parameters are inspected using introspection –the types of local variables and fields are inferred by the compiler using union and intersection types [22].

3.1. Runtime Performance Optimizations

In this section, we formalize the performance optimizations implemented for VB, which follow the .NET binary optimization approach presented in Figure 3.a. Appendixes C, D and E detail the binary optimizations for Boo, Cobra and Fantom, respectively. Appendix F presents the optimizations included in the *StaDyn* compiler, following the architecture presented in Figure 3.b.

Figure 3 shows how every optimization is based on the idea of replacing an AST with another AST that uses the DLR services. Figures 4 to 7 present the most significant inference rules used to optimize VB. An example of these transformations is replacing the program in the left-hand side of Figure 2 with the code in the right-hand side. This AST transformation is denoted by $\leadsto$, so that $e_1 \leadsto e_2$ represents that the AST of the expression e_1 is replaced with the AST of e_2.

The meta-variables e range over expressions; C, f, m and ω range over class, field, method and member names, respectively; and T ranges over types. $e{:}T$ denotes that the e expression has the T type. For the two architectures showed in Figure 3 (binary code transformation and compiler internals), our transformations can make use of the types of expressions. In the binary code transformation scenario, the CCI tool provides us this information (Section 4); for the compiler approach, we obtain expression types from the annotated AST [21]. $C \times T_1 \times \ldots \times T_n \rightarrow T_r$ represents the type of a (instance or `static`) method of the C class, receiving n parameters of $T_1, \ldots, T_n$ types, and returning T_r. $T_{L-built-in}$ represents the built-in types of the L language[2], and we use the *dynamic* type to indicate that an expression is dynamically typed (although VB represent dynamic types by removing type annotations –as shown in Figure 1).

Figure 4 shows the proposed optimizations for arithmetic expressions. BinaryOp optimizes binary expressions when at least one of the operands is dynamically typed; similarly, UnaryOp optimizes unary dynamically typed expressions. In both cases, a fresh `CallSite` object is created for each expression, passing the operator as an argument ($\oplus$ and $\ominus$ represent the VB binary and unary operators, respectively). Then, original dynamically typed expressions are replaced with an invocation to the `Target` method of the new

[2]For VB, the types in $T_{\text{VB}-built-in}$ are `Boolean`, `Byte`, `Char`, `Date`, `Decimal`, `Double`, `Integer`, `Long`, `SByte`, `Short`, `Single`, `String`, `UInteger`, `ULong` and `UShort`.

(BinaryOp)

$$\frac{\begin{array}{c} e_1 : \mathit{dynamic} \lor e_2 : \mathit{dynamic} \\ \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{Mod}, \texttt{==}, \texttt{<>}, \texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}, \texttt{And}, \texttt{Or}, \texttt{Xor}\} \\ \mathit{callsite} = \texttt{New CallSite(Binder.BinaryOperation(ExpressionType.}\oplus\texttt{)} \end{array}}{e_1 \oplus e_2 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}}$$

(UnaryOp)

$$\frac{\begin{array}{c} e : \mathit{dynamic} \qquad \ominus \in \{\texttt{Not}, \texttt{-}\} \\ \mathit{callsite} = \texttt{New CallSite(Binder.UnaryOperation(ExpressionType.}\ominus\texttt{)} \end{array}}{\ominus\, e \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

Figure 4: Transformation of common expressions.

`CallSite` object, passing the two operands as arguments.

Figure 5 shows different optimizations when a type conversion is required, using the `Convert` binder provided by the DLR [18]. CCast describes explicit type conversion (casting) for built-in types. In VB, the `CType` function explicitly converts the type of an expression[3]. When the expression is dynamically typed, we replace the operation with the appropriate `Convert` binder provided by the DLR.

When a dynamically typed expression is assigned to a statically typed one, CAssign replaces the dynamic type conversion with a DLR operation. As with CCast, this optimization is only performed when the type of the left-hand side expression is built-in. CFunction converts a dynamically typed argument into the built-in type of the corresponding parameter. In CFunction, e represents any expression evaluated as a method, since VB provides methods as first class entities (the so-called delegates) [23].

The conversion of a dynamically typed expression into a non-built-in type is done by VB with just one `castclass` instruction of the IL assembly language [24]. Since the implementation of that instruction is so efficient, the DLR does not provide any optimization for non-built-in type conversions (Table A.1). Therefore, the explicit conversions in CCast, CAssign and

[3]Although VB provides additional conversion functions (`CBool`, `CByte`, `CChar`, `CDate`, `CDec`, `CDbl`, `CInt`, `CLng`, `CSByte`, `CShort`, `CSng`, `CStr`, `CUInt`, `CULng` and `CUShort`), all of them can be expressed with `CType`.

(CCast)

$$\frac{T \in T_{\text{VB}-\mathit{built-in}} \qquad \begin{array}{c} e : \mathit{dynamic} \\ \mathit{callsite} = \texttt{New CallSite(Binder.Convert(}T\texttt{))} \end{array}}{\texttt{CType(}e, T\texttt{)} \leadsto \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

(CAssign)

$$\frac{e_2 : \mathit{dynamic} \qquad \begin{array}{c} e_1 : T \qquad T \in T_{\text{VB}-\mathit{built-in}} \\ \mathit{callsite} = \texttt{New CallSite(Binder.Convert(}T\texttt{))} \end{array}}{e_1\texttt{=}e_2 \leadsto e_1\texttt{=}\mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_2\texttt{)}}$$

(CFunction)

$$\frac{\begin{array}{c} e_i : \mathit{dynamic} \\ T_i \in T_{\text{VB}-\mathit{built-in}} \end{array} \qquad \begin{array}{c} e : C \times T_1 \times \ldots \times T_i \times \ldots \times T_n \to T_r \\ \mathit{callsite} = \texttt{New CallSite(Binder.Convert(}T_i\texttt{))} \end{array}}{e\texttt{(}e_1, \ldots, e_i, \ldots, e_n\texttt{)} \leadsto e\texttt{(}e_1, \ldots, \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_i\texttt{)}, \ldots, e_n\texttt{)}}$$

(CIf)

$$\frac{e : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{New CallSite(Binder.Convert(Boolean))}}{\begin{array}{c} \texttt{If } e \texttt{ Then } \mathit{stmt}^{+}_{\text{if}} \ (\texttt{Else } \mathit{stmt}^{+}_{\text{else}})^{?} \texttt{ End If} \leadsto \\ \texttt{If } \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{) Then } \mathit{stmt}^{+}_{\text{if}} \ (\texttt{Else } \mathit{stmt}^{+}_{\text{else}})^{?} \texttt{ End If} \end{array}}$$

Figure 5: Transformation of common type conversions.

CFunction are only applied to built-in types.

VB requires the type of the expression in a conditional statement to be `Boolean`. CIf performs this type conversion when the condition is dynamically typed. Similar inference rules for typical *do-while* (CDWhile and CRWhile) and *repeat-until* (CDUntil and CRUntil) loops are detailed in Appendix B. Likewise, CIndex in Appendix B performs the same optimization for array indexing expressions, converting the index to `Integer`.

Figure 6 shows the optimization of array indexing operations, when arrays are dynamically typed. In VB, parentheses are used for both array indexing and method invocation. However, the CCI generates different ASTs for each kind of operations, facilitating us the transformation of programs. VB provides the indexing operation not only for arrays, but also for other types such as dictionaries, lists and strings (i.e., any type that implements *indexer* properties [23]). When the collection is dynamically typed, the `GetIndex` binder is used for reading operations and `SetIndex` for writing.

$$\textsc{(GetIndex)}\quad \frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.GetIndex))}}{e_1\texttt{(}e_2\texttt{)} \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2\texttt{)}}$$

$$\textsc{(SetIndex)}\quad \frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.SetIndex))}}{e_1\texttt{(}e_2\texttt{)=}e_3 \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2, e_3\texttt{)}}$$

Figure 6: Transformation of indexing operations.

IOIMethod in Figure 7 shows the optimization of instance method invocation, when the method may be overloaded and one of the arguments (e_i) is dynamically typed. Method overloading is represented with intersection types: the type of an overloaded method is an intersection type holding all the types of its different implementations [25]. The $\exists\, T_i^j\ .\ T_i^j \neq \textit{dynamic}^{\ j\,\in\,1\ldots m}$ condition checks that at least one method implementation declares a statically typed i^{th} parameter. Otherwise, the argument would not need to be type checked (it is already dynamic). Unlike CFunction in Figure 5, the generated `InvokeMember` call-site receives the object and all the parameters to resolve method overloading at runtime [26]. Appendix B describes how we optimize overloaded class methods.

IMethod optimizes an instance method invocation when the object that receives the message is dynamically typed. The number of parameters must be greater than zero, because field access and zero-argument method invocation is performed with the same low-level operation in VB (parenthesis are not required to invoke a method with no arguments). LateGet represents this special case scenario. Since we do not have enough information to know if the expression is either a zero-argument method invocation or a member access, we perform additional runtime checks. We statically create two different call-sites for each alternative: `GetMember` and `InvokeMember`. Then, the `HandleCallSiteCall` method of the `LateGet_Utils` class calls the appropriate call-site depending on the dynamic type of ω (field or method). Our implementation of `HandleCallSiteCall` includes a runtime cache storing the type of each member [27]. Appendix B describes how we optimize the assignment of fields and properties.

(IOIMethod)

$$\frac{\begin{array}{c} m : C \times T_1^1 \times \ldots \times T_i^1 \times \ldots \times T_n^1 \rightarrow T_r^1 \wedge \ldots \wedge C \times T_1^m \times \ldots \times T_i^m \times \ldots \times T_n^m \rightarrow T_r^m \\ e : C \qquad e_i : \mathit{dynamic} \qquad \exists\, T_i^j \,.\, T_i^j \neq \mathit{dynamic}^{\,j \,\in\, 1\ldots m} \\ \mathit{callsite} = \texttt{New CallSite(Binder.InvokeMember(}m\texttt{))} \end{array}}{e\texttt{.}m\texttt{(}e_1, \ldots, e_i, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e, e_1, \ldots, e_i, \ldots, e_n\texttt{)}}$$

(IMethod)

$$\frac{\begin{array}{c} e : \mathit{dynamic} \\ n > 0 \qquad \mathit{callsite} = \texttt{New CallSite(Binder.InvokeMember(}m\texttt{))} \end{array}}{e\texttt{.}m\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e, e_1, \ldots, e_n\texttt{)}}$$

(LateGet)

$$\frac{\begin{array}{c} e : \mathit{dynamic} \qquad \mathit{callsite}_1 = \texttt{New CallSite(Binder.GetMember(}\omega\texttt{))} \\ \mathit{callsite}_2 = \texttt{New CallSite(Binder.InvokeMember(}\omega\texttt{))} \end{array}}{e\texttt{.}\omega \rightsquigarrow \texttt{LateGet\_Utils.HandleCallSiteCall(}e, \omega, \mathit{callsite}_1, \mathit{callsite}_2\texttt{)}}$$

Figure 7: Transformation of method invocation and field access.

4. Implementation

4.1. Binary Program Transformation

As mentioned, our .NET binary transformation tool has been developed using the Microsoft Common Compiler Infrastructure (CCI). The CCI libraries offer services for building, analyzing and modifying .NET assemblies [19]. Figure 8 shows the design class diagram of the binary optimization tool (classes provided by the CCI are represented with the `CCI` stereotype). First, our `DLROptimizer` class uses a CCI `PEReader` to read each program assembly, returning an `IAssembly` instance. Each `IAssembly` object represents an AST. The second step is transforming the ASTs into optimized ones, following the Visitor design pattern [28]. Finally, the modified ASTs are saved as new assemblies with `PEWriter`.

In the general process described above, the most complex task is the AST transformation algorithm, which is divided in three different phases. First, the dynamically typed expressions to be optimized are identified, traversing the AST. For each language, we implement a Visitor class (e.g., `VBCode-Visitor` and `BooCodeVisitor`) that identifies the expressions to be optimized, following the specific language optimization rules described in this paper. For each expression, the corresponding call-site pattern is stored in a `CallSiteContainer` object. Second, the code that instantiates the `CallSite`s is generated. As shown in Figure 2 and Table A.1, an instance of the DLR `CallSite` class must be created for each optimized expression collected in `CallSiteContainer`. The code that creates these call-site instances is generated by the `DLROptimizer`, using the CodeDOM API [29]. Finally, the `OptimizerCodeRewriter` class traverses the original `IAssembly` AST, returning the optimized one, where the dynamically typed expressions are replaced with appropriate invocations to the call-sites created.

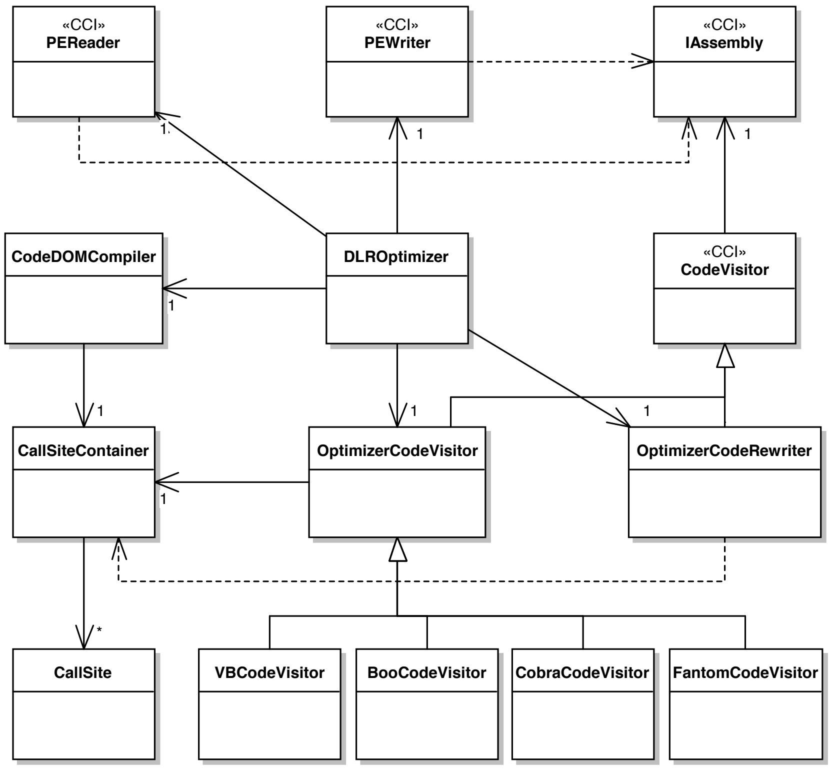


Figure 8: Class diagram of the binary program transformation tool.

4.2. Compiler Optimization Phase

The optimization of *StaDyn* programs have been implemented as part of the compiler internals. After lexical and syntax analysis, the *StaDyn* compiler performs type inference in 5 phases [21]. Code generation is performed afterwards, traversing the type-annotated AST and following the Visitor design pattern [28]. Originally, the existing code generator produced .NET assemblies for the CLR (Figure 3.b). We have added code generation for the DLR using the Parallel Hierarchies design pattern [30]. The optimizations proposed are applied when the `server` command-line option is passed to the compiler. The code generation templates of dynamically typed expressions are detailed in Appendix F.

5. Evaluation

In this section, we evaluate the runtime performance gains of the proposed optimizations. We measure the execution time and memory consumption of the original programs, and compare them with the optimized versions. We measure different benchmarks executed in all the existing hybrid static and dynamic programming languages for the .NET platform.

5.1. Methodology

This section comprises a description of the languages and the benchmark suites used in the evaluation, together with a description of how data is measured and analyzed.

5.1.1. Selected Languages

We have considered the existing hybrid typing languages for the .NET platform, excluding C# that already uses the DLR:

– Visual Basic 11. The VB programming language supports hybrid typing [23]. 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.

– Boo 0.9.4.9. An object-oriented programming language for the CLI with Python inspired syntax. It is statically typed, but also provides dynamic typing by using its special `duck` type [31]. Boo has been used to create views in the Brail view engine of the MonoRail Web framework [32], to program the Specter object-behavior specification framework [33], in the implementation of the Binsor domain-specific language for the Windsor Inversion of Control container for .NET [34], and in the development of games and mobile apps with Unity [35].

– Cobra 0.9.6. A hybrid statically and dynamically typed programming language. It is object-oriented and provides compile-time type inference [36]. As C#, dynamic typing is provided with a distinctive `dynamic` type. Cobra has been used to develop small projects and to teach programming following the test-driven development and the design by contract approaches [36].

– Fantom 1.0.64. Fantom is an object-oriented programming language than generates code to the Java VM, the .NET platform, and JavaScript. It is statically typed, but provides the dynamic invocation of methods with the specific `->` message-passing operator [37]. The Fantom language provides an API that abstracts away the differences between the Java and .NET platforms. Fantom has been used to develop some projects such as the Kloudo integrated business organizer [38], the SkySpark analytics software [39], and the netColarDB object-relational mapping database [40].

– *StaDyn*. *StaDyn* is a hybrid static and dynamic typing object-oriented language for the .NET Framework, created as an extension of C# [20]. It supports implicitly (and explicitly) typed references. Unlike C#, the *StaDyn* compiler gathers type information for `dynamic` references, improving compile-time error detection and runtime performance [21].

5.1.2. Selected Benchmarks

We have used different benchmark suites to evaluate the performance gain of our implementations:

– Pybench. A Python benchmark designed to measure the performance of standard Python implementations [41]. Pybench is composed of a collection of 52 tests that measure different aspects of the Python dynamic language.

– Pystone. This benchmark is the Python version of the Dhrystone benchmark [42], which is commonly used to compare different implementations of the Python programming language. Pystone is included in the standard Python distribution.

– A subset of the statically typed Java Grande benchmark implemented in C# [43], including large scale applications:

    - Section 2 (Kernels). FFT, one-dimensional forward transformation of n complex numbers; Heapsort, the heap sort algorithm over arrays of integers; and Sparse, management of an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure.
    - Section 3 (Large Scale Applications). RayTracer, a 3D ray tracer of scenes that contain 64 spheres, and are rendered at a resolution of 25×25 pixels.

- Points. A hybrid static and dynamic typing program designed to measure the performance of hybrid typing languages [22]. It computes different properties of two- and three-dimensional points.

We have taken Python (Pybench and Pystone) and C# (Java Grande and Points) programs, and manually translated them into the rest of languages. Although this translation might introduce a bias in the runtime performance of the translated programs, we have thoroughly checked that the same operations were executed in all the implementations. We have verified that the benchmarks compute the same results in all the programs.

Those tests that use a specific language feature not provided by the other languages (i.e., tuples, dynamic code evaluation, and Python-specific built-in functions) have not been considered. We have not included those that use any input/output interaction either. Therefore, 31 tests of the 52 programs of the Pybench benchmark have been measured [27]. All the references in the programs have been declared as dynamically typed.

5.1.3. Data Analysis

We have followed the methodology proposed in [44] to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT-compilation. In this methodology, two approaches are considered: 1) *start-up* performance is how quickly a system can run a relatively short-running application; 2) *steady-state* performance concerns long-running applications, where start-up JIT compilation does not involve a significant variability in the total running time.

For start-up, we followed the two-step methodology defined to evaluate short-running applications:

1. We measure the elapsed execution time of running multiple times the same program. This results in p (we have taken $p = 30$) measurements x_i with $1 \leq i \leq p$.
2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is calculated using the *Student's t*-distribution because we took $p = 30$ [45]. Therefore, we compute the confidence interval $[c_1, c_2]$ as:

   $$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\tfrac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\tfrac{s}{\sqrt{p}}$$

   Where $\overline{x}$ is the arithmetic mean of the x_i measurements; $\alpha = 0.05(95\%)$; s is the standard deviation of the x_i measurements; and $t_{1-\alpha/2;p-1}$ is defined such that a random variable T, which follows the *Student's t*-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1-\alpha/2$. In the subsequent figures, we show the mean of the confidence interval plus the width of the confidence interval relative to the mean (bar whiskers). If two confidence intervals do not overlap, we can conclude that there is a statistically significant difference with a 95% (1 - α) probability [44].

The steady-state methodology comprises the following four steps:

1. Each application (program) is executed p times ($p = 30$), and each execution performs at least k ($k = 10$) different iterations of benchmark invocations, measuring each invocation separately. We refer x_{ij} as the measurement of the j^{th} benchmark iteration of the i^{th} application execution.
2. For each i invocation of the benchmark, we determine the s_i iteration where steady-state performance is reached. The execution reaches this state when the coefficient of variation (CoV, defined as the standard deviation divided by the mean) of the last k iterations (from s_{i-k+1} to s_i) falls below a threshold (2%).

   To avoid an influence of the previous benchmark execution, a full heap garbage collection is done before performing every benchmark invocation. Garbage collection may still occur at benchmark execution, and it is included in the measurement. However, this method reduces the non-determinism across multiple invocations due to garbage collection kicking in at different times across different executions.

3. For each application execution, we compute the $\overline{x_i}$ mean of the k benchmark iterations under steady state:

$$\overline{x_i} = \frac{\sum\limits_{j=s_{i-k+1}}^{s_i} x_{ij}}{k}$$

4. Finally, we compute the confidence interval for a given confidence level (95%) across the computed means from the different application invocations using the *Student's t*-statistic described above. The overall mean is computed as $\overline{x} = \sum_{i=1}^{p} \overline{x_i}/p$. The confidence interval is computed over the $\overline{x_i}$ measurements.

5.1.4. Data Measurement

To measure the execution time of each benchmark invocation, we have instrumented the applications with code that registers the value of high-precision time counters provided by the Windows operating system. This instrumentation calls the native function `QueryPerformanceCounter` of the `kernel32.dll` library. This function returns the execution time measured by the Performance and Reliability Monitor of the operating system [46]. We measure the difference between the beginning and the end of each benchmark invocation to obtain the execution time of each benchmark run.

The memory consumption has been also measured following the same methodology to determine the memory used by the whole process. For that purpose, we have used the maximum size of working set memory employed by the process since it was started (the `PeakWorkingSet` property). The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to be used without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including those from the process modules and the system libraries. The `PeakWorkingSet` has been measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [47].

All the tests were carried out on a 3.30 GHz Intel Core i7-4500U system with 8 GB of RAM, running an updated 64-bit version of Windows 8.1 and the .NET Framework 4.5.1 for 32 bits. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating

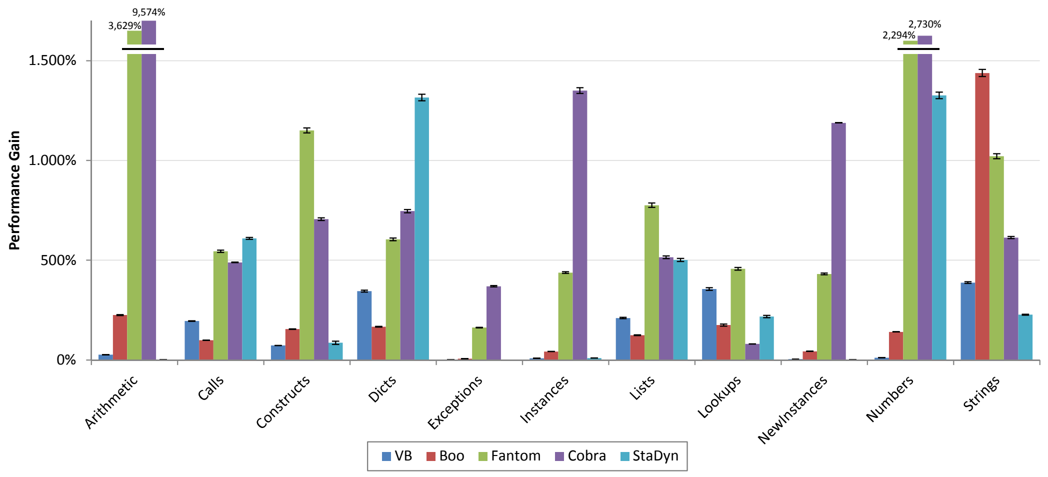


Figure 9: Start-up performance improvement for Pybench.

system to be loaded.

If the P_1 and P_2 programs run the same benchmark in T and $2.5 \times T$ milliseconds, respectively, we say that runtime performance of P_1 is 150% (or 2.5 times) higher than P_2, P_1 is 150% (or 2.5 times) faster, P_2 requires 150% (or 2.5 times) more execution time than P_1, or the performance benefit of P_1 compared to P_2 is 150% –the same for memory consumption. To compute average percentages, factors and orders of magnitude, we use the geometric mean.

5.2. Start-up Performance

Figures 9 and 10 show the start-up performance gains obtained with our optimizations, relative to the original program. First, we analyze the results of the Pybench micro-benchmark (Figure 9) to examine how the optimizations introduced may improve the runtime performance of each language feature. Afterwards, we analyze more realistic applications in Figure 10.

The average runtime performance gains in Pybench range from the 141% improvement for VB up to the 891% benefit obtained for the Fantom language. The proposed optimizations speed up the average execution of Boo, *StaDyn* and Cobra programming languages in 190%, 252% and 772%, respectively.

Figure 10 shows the start-up performance improvements for all the programs –average results for Pybench are included. Our optimizations show the best performance gains for Fantom, presenting a 915% average speedup.

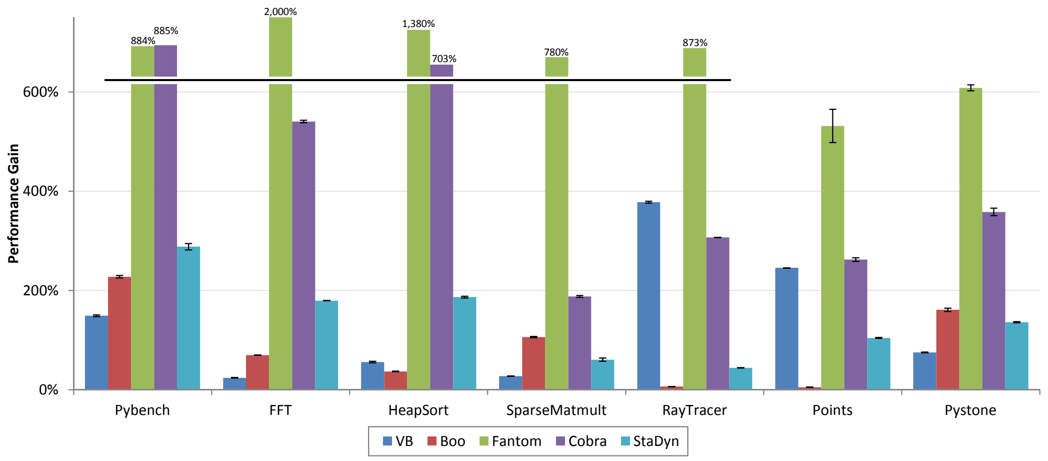


Figure 10: Start-up performance improvement.

For Cobra, *StaDyn*, VB and Boo, the average performance improvements are 406%, 120.5%, 87.4% and 44.6%, respectively.

5.2.1. Discussion

Analyzing the previous start-up performances, we can identify different discussions. Considering the different kind of operations in Figure 9, Boo, Fantom and Cobra obtain the highest performance improvements when running the programs that perform arithmetic and comparison computation, and string manipulations (arithmetic, numbers and strings). For these operations, the three languages use reflection, which is highly optimized by the DLR cache [16]. Thus, the DLR provides important performance benefits for introspective operations.

For arithmetic operations, VB and *StaDyn* show little improvement compared to the rest of languages (Figure 9). Both languages already support an optimization based on nested dynamic type inspections, avoiding the use of reflection [21] –unlike *StaDyn*, VB also provides this optimization for number comparisons (the numbers test). Fantom, Cobra and *StaDyn* do not provide any runtime cache for dynamically typed method invocation (calls), and vector (lists) and map (dicts) indexing, causing high performance gains –VB and Boo show lower improvements because they implement their own caches. So, when the language implementation provides other runtime optimizations to avoid the use of reflection, the performance gains of using the DLR are decreased.

Exceptions, instances and new instances are the programs for which our optimizations show the lowest performance gains. This inferior performance edge is because almost no dynamically typed reference is used in these tests. For example, the exceptions test has the loop counter as the only dynamically typed variable (for Fantom and Cobra, the benefit is higher than for the rest of languages because their runtimes do not implement a cache for dynamic types). Therefore, the DLR provides little performance improvement when just a few dynamically typed references are used.

In the execution of the RayTracer and Points programs (Figure 10), the performance gains for Boo are just 6.84% and 5.12%, respectively. These two programs execute a low number of DLR call-sites, and hence the DLR cache does not provide significant performance improvements. The initialization of the cache, together with the dynamic code generation technique used to generate the cache entries [18], incur a performance penalty that reduces the global performance gain. As we analyze in the following subsection, for long-running applications (steady-state methodology) this performance cost is almost negligible.

5.3. Steady-State Performance

We have executed the same programs following the steady-state methodology described in Section 5.1.3. Figure 11 shows the runtime performance improvements for all the programs. In this scenario, the performance gains for every language are higher than those measured with the start-up methodology. The lowest average improvement is 244% for VB; the greatest one is 1113%, for Cobra. We speed up Boo, *StaDyn* and Fantom in 322%, 368% and 1083%, respectively.

5.3.1. Discussion

Table 1 compares the performance improvements of short- and long-running applications (start-up and steady-state). It shows how the proposed optimizations provide higher performance gains for long-running applications than for sort-running ones, in all the benchmarks.

Boo and VB are the two languages that show the highest performance difference depending on the methodology used. Average steady-state performance improvements are 758% (Boo) and 442% (VB) higher than the start-up ones. This dependency is because both languages implement their own dynamic type cache, reducing the benefits of the DLR optimizations in start-up. As the number of DLR cache hits increases in steady-state, the

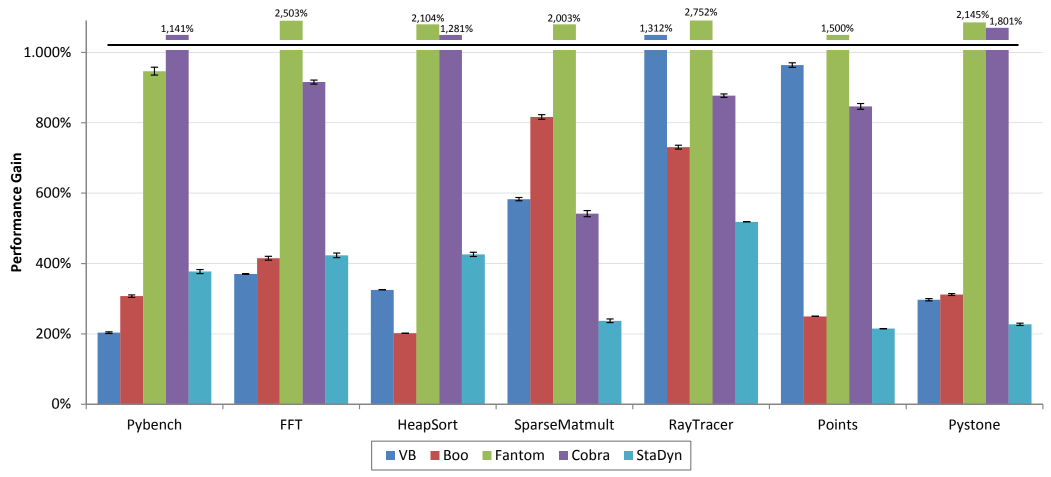


Figure 11: Steady-state performance improvement.

performance edge is also improved. Therefore, the DLR increases the steady-state performance gains of languages that provide their own type cache, compared to start-up.

Table 1 shows how Fantom is the language with the smallest steady-state performance gain compared to the start-up one. The average steady-state benefit (1897%) is 107% higher than the start-up one (915%). In the Fantom language, every dynamically typed operation generates the same type of call-site: `InvokeMember` (detailed in Appendix E). Since the DLR creates a different cache for each type of call-site [18], the optimized code for Fantom incurs lower performance penalties caused by cache initialization in start-up. Therefore, in languages that use the same type of call-site for many different operations, the start-up performances may be closer to the steady-state ones.

When analyzing the performance gains per application, Pybench shows the lowest performance improvements across methodologies (Table 1). The synthetic programs of the Pybench benchmark perform many iterations over the same code (i.e., call-sites). This causes many cache hits, bringing the steady-state performance gains closer to the start-up ones. So, the important steady-state performance improvements are applicable not only to long-running applications, but also to short-running ones that perform many iterations over the same code.

Benchmark		VB	Boo	Fantom	Cobra	*StaDyn*
Pybench	(startup)	149%	228%	884%	885%	288%
	(steady)	203%	307%	947%	1,141%	377%
FFT	(startup)	24%	70%	2,000%	540%	179%
	(steady)	370%	415%	2,503%	916%	423%
HeapSort	(startup)	56%	37%	1,380%	703%	187%
	(steady)	325%	202%	2,104%	1,281%	426%
Sparse Matmult	(startup)	27%	106%	781%	188%	61%
	(steady)	583%	817%	2,003%	542%	237%
RayTracer	(startup)	378%	7%	873%	307%	44%
	(steady)	1,312%	731%	2,752%	877%	518%
Points	(startup)	246%	5%	531%	262%	104%
	(steady)	964%	250%	1,500%	847%	215%
Pystone	(startup)	75%	161%	608%	358%	136%
	(steady)	297%	312%	2,155%	1,801%	227%

Table 1: Performance benefits for both start-up and steady-state methodologies.

5.4. Memory Consumption

Figure 12 (and Table 2) shows the memory consumption increase introduced by our performance optimizations. For each language and application, we present the memory resources used by the optimized programs (DLR), relative to the original ones (CLR). Optimized Fantom, Boo, *StaDyn*, Cobra and VB programs consume 6.42%, 45.32%, 53.75%, 57.67% and 64.48% more memory resources than the original applications.

5.4.1. Discussion

We compare the memory consumption increase caused by the DLR (Figure 12) with the corresponding performance gains (Figures 10 and 11). In both start-up and steady-state scenarios, performance benefits are significantly higher than the corresponding memory increase, for all the languages measured.

Fantom is the language with the smallest memory increase. Table 2 shows how Fantom is the language that originally requires more memory resources, hence reducing the relative memory increase value. Additionally, in the previous section we mentioned that Fantom uses the same type of DLR call-site for every dynamic operation. Since the DLR has a shared cache for each type of call-site [18], Fantom does not consume the additional resources of the rest of call-sites. Therefore, the memory increase introduced by the DLR may depend on the number of services used.

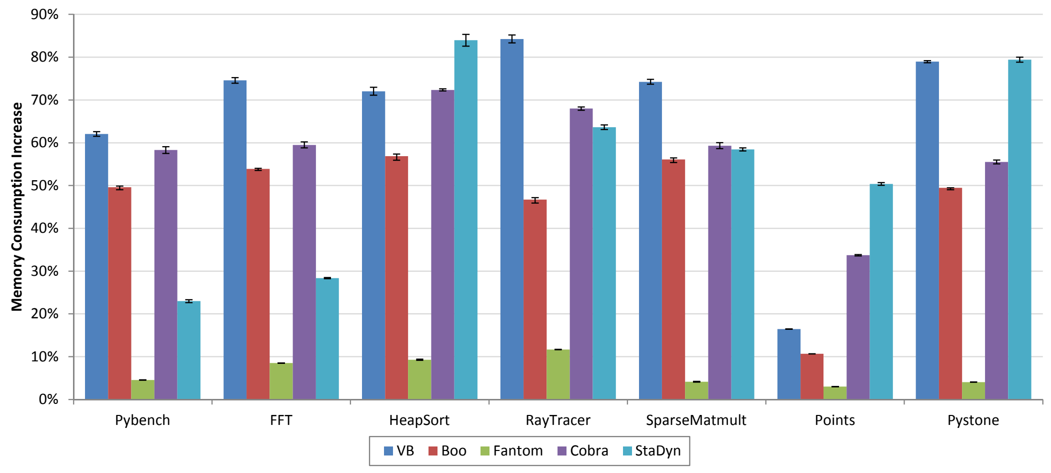


Figure 12: Memory consumption increase.

	VB		**Boo**		**Fantom**		**Cobra**		***StaDyn***	
	CLR	DLR	CLR	DLR	CLR	DLR	CLR	DLR	CLR	DLR
Pybench	13.93	22.58	14.03	20.99	22.29	23.30	13.67	21.65	19.06	23.43
FFT	15.00	26.18	15.02	23.12	22.94	24.89	15.54	24.79	17.66	22.67
HeapSort	14.31	24.61	14.67	23.01	22.23	24.29	14.10	24.30	11.94	21.97
RayTracer	14.87	27.40	16.96	24.88	23.73	26.50	15.68	26.35	13.78	22.54
SparseMatmult	14.47	25.21	14.79	23.09	23.34	24.31	15.43	24.58	14.07	22.29
Points	19.72	22.97	20.59	22.79	23.42	24.13	17.26	23.09	14.35	21.58
Pystone	14.65	26.21	15.55	23.24	23.36	24.31	16.05	24.95	12.27	22.01

Table 2: Memory consumption expressed in MBs.

Analyzing the applications in Figure 12, the Points program shows the lowest average memory increase. This application also presents the smallest average start-up and steady-state performance gains (Sections 5.2 and 5.3). As discussed in the previous paragraph, Points is the application that executes the smallest number of DLR call-sites, causing the lowest performance and memory increases.

6. Related Work

The optimization of hybrid static and dynamic typing code has been faced in different ways. We first describe the works related to the optimization of hybrid typing languages. Afterwards, we identify research aimed at optimizing dynamically typed code at the virtual machine level. Finally, some particular optimizations of dynamically typed languages are discussed.

6.1. Hybrid Static and Dynamic Typing Languages

There are different works aimed at optimizing hybrid static and dynamic typing languages. The theoretical works of quasi-static typing [48], hybrid typing [49] and gradual typing [50] perform implicit conversions between dynamically and statically typed code, employing the subtyping relation in the case of quasi-static and hybrid typing, and a consistency relation in gradual typing. The gradual type system for the $\lambda^{?}_{\rightarrow}$ functional calculus provides the flexibility of dynamic typing when type annotations are omitted by the programmer, and the benefits of static typing when all the function parameters are annotated [50]. Gradual typing has also been defined for object-based languages, showing that gradual typing and subtyping are orthogonal and can be combined [51]. The gradually typed lambda calculus $\lambda^{?}_{\rightarrow}$ was also extended with type variables, integrating unification-based type inference and gradual typing to aid programmers in adding types to their programs [52].

Thorn is a programming language that allows the combination of dynamically and statically typed code [53]. Thorn offers `like` types, an intermediate point between static and dynamic types [54]. Occurrences of `like` types variables are checked statically within their scope but, as they may be bound to dynamic values, their usage must be still checked at runtime. `like` types increase the robustness of the Thorn programming language, and programs developed using `like` types have been assessed to be about 3x and 6x faster than using dynamic [54].

C# 4.0 added the `dynamic` type to its static type system, supporting the safe combination of dynamically and statically typed code. In C#, type checking of the references defined as `dynamic` is deferred until runtime [15]. This hybrid type system was formalized by Bierman *et al.*, defining a core fragment of C# that is translated to a simplification of the DLR [15]. The operational semantics of the target language reuse the compile-time typing and resolution rules, implying that the dynamic code fragments are type-checked and resolved using the same rules as the statically typed code [15]. The cache implemented by the DLR provides significant runtime performance benefits compared to the use of reflection [26].

There exist some other hybrid static and dynamic typing languages such as Boo, Visual Basic, Cobra, Dylan, Strongtalk, Groovy 2, Fantom, *StaDyn* and Objective-C. Some programming languages have taken the approach of adding a new dynamic type as proposed in [55] (`dynamic` in *StaDyn* and Cobra, `duck` in Boo, and `id` in Objective-C), whereas others represent dynamic types by removing type annotations in variable declarations (VB, Dylan and Groovy 2) [23]. Fantom and Objective-C provide the `->` and `[ ]` operators, respectively, to allow passing any message to an object, postponing type checking until runtime. Strongtalk follows a completely different approach based on the concept of *pluggable* type systems [56]. In these languages, dynamic types are implicitly coerced to static ones following the approach defined in [48] and [51], opposite to the explicit use of a conversion instruction like the `typecase` operator proposed by [55]. Since these implicit coercions may fail at runtime, a dynamic type-check is inserted in the generated code as described in [49]. The *StaDyn* compiler gathers type information of dynamically typed references to optimize the generated code [57].

6.2. Dynamically Typed Virtual Machines

Other research works are aimed at optimizing some specific features of dynamic languages at the virtual machine level. Smalltalk is a class-based dynamically typed programming language [58]. Although the initial implementations were based on byte-code interpreters, some later versions included JIT compilation to native code (e.g., VisualWorks, VisualAge and Digital) [59]. JIT compilation provided important performance benefits, making VisualWorks to be, on average, 3 times faster than GNU Smalltalk [60].

Self is a dynamic prototype-based object-oriented language supported by a JIT-compiler virtual machine [61]. When a dynamic method is executed, runtime type information is gathered to perform type specialization of method invocations, using the specific types inferred for each argument [62]. The overhead of dynamically bound message passing is reduced by means of inline caches [59], introducing polymorphic inline caches (PIC) for polymorphic invocations [63]. Some other adaptive optimization strategies where implemented to improve the performance of hotspot functions while the program is running [64].

These JIT-compiler adaptive optimizations have been recently added to JavaScript virtual machines. V8 is the Google JavaScript engine used in Chrome, which can run standalone and embedded into C++ applications [65]. V8 uses a quick response JIT compiler to generate native code. For hotspot functions detected at runtime, a high performance JIT compiler applies aggressive optimizations. These optimizations include inline caches, type feedback, customization, control flow graph optimizations and dead code elimination [65].

SpiderMonkey is the new JavaScript engine of Mozilla, currently included in the Firefox Web browser and the GNOME 3 desktop [66]. It uses three optimization levels: an interpreter, the baseline JIT-compiler, and the IonMonkey compiler for more powerful optimizations. The slow interpretation collects profiling and runtime type information. The baseline compiler generates binary code dynamically, collecting more accurate type information and applying basic optimizations. Finally, IonMonkey is only triggered for hotspot functions, providing optimizations such as type specialization, function inlining, linear-scan register allocation, dead code elimination, and loop-invariant code motion [66].

ЯRotor is an extension of the .NET SSCLI virtual machine implementation that provides JIT-compilation of the structural reflective primitives provided by dynamic languages [60]. A hybrid class- and prototype-based object-oriented model is formally described, and then implemented as part of a shared source release of the .NET CLI [16]. On average, ЯRotor performs 4 times better than the DLR, consuming 65% less memory resources [67].

The work of Würthinger *et al.* modifies an implementation of the Java Virtual Machine to allow arbitrary changes to the definition of loaded classes, providing dynamic inheritance [68]. The static type checking of Java is maintained; and the dynamic verification of the current state of the program ensures the type safety of the changes in the class hierarchy. Runtime performance after code evolution implies an approximate performance penalty of 15%, but the slowdown of the next run after code evolution was measured to be only about 3% [69]. This system is currently the reference implementation of the hot-swapping feature (JSR 292) of the Da Vinci Machine project [70].

6.3. Dynamic Typing Programming Languages

There are also some other works based on gathering static type information of dynamically typed code to optimize its execution. Brian Hackett and Shu-yu Guo defined a hybrid static and dynamic type inference algorithm for JavaScript based on points-to analysis [71]. They propose a constraint-based type system to unsoundly infer type information statically. Type information is extended with runtime semantic triggers to generate sound type information at runtime, as well as type barriers to efficiently handle polymorphic code. The proposed system was implemented and integrated in the

JavaScript JIT compiler inside Firefox. The performance improvement on major benchmarks and JavaScript-heavy websites was up to 50% [71].

PyPy is an alternative implementation of Python that provides JIT compilation, memory usage optimizations, and full compatibility with CPython [72]. PyPy implements a tracing JIT compiler to optimize program execution at runtime, generating dynamically optimized machine code for the hot code paths of commonly executed loops [72]. The optimization techniques implemented have made PyPy outperform the rest of Python implementation in many different benchmarks [17]. PyPy is designed to be language agnostic, allowing the Just-in-Time (JIT) compilation of any dynamic language.

7. Conclusions

These are the key findings of this work:

– The dynamically typed operations of hybrid typing languages for the .NET platform can be significantly optimized by using the runtime cache implemented by the DLR.

– Performance gains for long-running applications (from 224% to 1113%) are higher than those for short-running ones (from 44.6% to 406%). Higher performance gains are also present in short-running applications that perform many iterations over the same code.

– For those languages that use reflection, the performance gains are significantly higher than for those that avoid reflection by providing their own runtime cache.

– The runtime performance gains depend on the number of dynamically typed references in the program.

– The initialization of the DLR cache may incur a performance penalty, reducing the global performance gain (especially in start-up scenarios).

– The memory consumption increase (from 6.2% to 64.5%) is considerably lower than the runtime performance gains (from 44.6% to 1113%).

– Both memory and performance increases depend on the types of the DLR call-sites used in the optimizations.

– The proposed optimizations are effective when implemented as binary transformations and as a compiler optimization phase.

Future work will be focused on including DLR optimizations in dynamic languages for the .NET platform where the DLR is not used yet. Adding these optimizations as a new compilation option would be convenient for long-running application where the DLR cache benefits are more effective.

The binaries and source code of the optimization tool, the optimizing *StaDyn* compiler, the benchmarks used in this paper, and all the execution time and memory consumption tables can be downloaded from: `http://www.reflection.uniovi.es/stadyn/download/2015/jss`

Acknowledgments

This work was partially funded by Microsoft Research to develop the project entitled *Extending Dynamic Features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution* Request for Proposals. This work was also funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*. We have also received funds from the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100).

Appendix A. DLR Call-Sites

Table A.1 shows the list of dynamically typed expressions that can be represented with DLR `CallSite`s. In this case, we use C# instead of VB because some of the DLR call-sites cannot be used from VB (e.g., the `InvokeConstructor` binder for overloaded constructors). There is one row for each binder. The column in the middle shows C# fragments where dynamically typed expressions are used. The corresponding C# code that uses the DLR call-sites is detailed in the last column –in fact, that code was obtained by decompiling the binary assemblies. For the sake of legibility, the code shown is simplified the following way: 1) `CallSite` type definitions are shortened, 2) the lazy initialization for `CallSite`s has been replaced by initializations in the declaration; and 3) arguments of `CallSiteBinder`s have been omitted.

Binder name	Dynamically typed expressions	Explicit use of the DLR services

Binary
Operation
	dynamic Add(dynamic a,
	            dynamic b) {
	  return a + b;
	}
		static CallSite<...> p_Site1 = CallSite<...>.Create(
		        Binder.BinaryOperation(ExpressionType.Add));
		dynamic Add(dynamic a, dynamic b) {
		  return p_Site1.Target(p_Site1, a, b);
		}

Unary
Operation
	dynamic Negation(dynamic a) {
	  return -a;
	}
		static CallSite<...> p_Site2 = CallSite<...>.Create(
		       Binder.UnaryOperation(ExpressionType.Negate));
		dynamic Negation(dynamic a){
		  return p_Site2.Target(p_Site2, a);
		}

Convert	T CastToType<T>(dynamic obj) {
	  return (T)obj;
	}
		static CallSite<...> p_Site3=CallSite<...>.Create(
		                         Binder.Convert(typeof(T));
		T CastToType<T>(dynamic obj) {
		  return p_Site3.Target(p_Site3, obj);
		}

GetIndex	dynamic GetPosition(dynamic v,
	                  dynamic i) {
	  return v[i];
	}
		static CallSite<..> p_Site4=CallSite<...>.Create(
		                               Binder.GetIndex());
		dynamic GetPosition(dynamic v, dynamic i) {
		  return p_Site4.Target(p_Site4, v, i);
		}

SetIndex	void SetPosition(dynamic v,
	                 dynamic i,
	                 dynamic val) {
	  v[i] = val;
	}
		static CallSite<...> p_Site5 = CallSite<...>.Create(
		                                  Binder.SetIndex());
		void SetPosition(dynamic v, dynamic i, dynamic val) {
		  p_Site5.Target(p_Site5, v, i, val);
		}

GetMember	dynamic GetName(dynamic obj) {
	  return obj.Name;
	}
		static CallSite<...> p_Site6=CallSite<...>.Create(
		                         Binder.GetMember("Name"));
		dynamic GetName(dynamic obj) {
		  return p_Site6.Target(p_Site6, obj);
		}

SetMember	void SetName(dynamic obj,
	             dynamic val) {
	  obj.Name = val;
	}
		static CallSite<...> p_Site7 = CallSite<...>.Create(
		                            Binder.SetMember("Name"));
		static void SetName(dynamic obj, dynamic val) {
		  p_Site7.Target(p_Site7, obj, val);
		}

Invoke	dynamic Invoke(dynamic fun,
	               dynamic a,
	               dynamic b) {
	  return fun(a, b);
	}
		static CallSite<...> p_Site8=CallSite<...>.Create(
		                                  Binder.Invoke());
		dynamic Invoke(dynamic fun, dynamic a, dynamic b) {
		  return p_Site8.Target(p_Site8, fun, a, b);
		}

Invoke
Constructor
	Decimal DecimalFactory(
	           dynamic argument) {
	  return new Decimal(argument);
	}
		static CallSite<...> p_Site9=CallSite<...>.Create(
		                       Binder.InvokeConstructor());
		decimal DecimalFactory(dynamic argument) {
		  return p_Site9.Target(p_Site9, typeof(decimal),
		                              argument);
		}

Invoke
Member
	dynamic InvokePrint(dynamic o,
	               dynamic arg) {
	  return o.Print(arg);
	}
		static CallSite<...> p_Site10=CallSite<...>.Create(
		                     Binder.InvokeMember("Print"));
		dynamic InvokePrint(dynamic o, dynamic arg) {
		  return p_Site10.Target(p_Site10, o, arg);
		}

Table A.1: Call-sites provided by the DLR (coded in C#).

Appendix B. Additional VB Rules

This section completes the VB conversion rules shown in Section 3.1. In Figure B.1, the CDWhile, CRWhile, CDUntil and CRUntil rules show the implicit conversions to `Boolean`, in the different loop statements provided by VB.

CIndex performs the optimization of array indexing expressions, when the index is dynamically typed. The type of the index expression is converted to `Integer` at runtime by the DLR. A premise in the inference rule requires the array not to be dynamically typed, since otherwise the expression would be optimized by the GetIndex rule in Figure 6.

IOCMethod provides the optimization of class methods (i.e., `shared` in VB, or `static` in C# and Java), when the method may be overloaded and one of the arguments (e_i) is dynamically typed. This rule is quite similar to IOIMethod in Figure 7. In this case, the second parameter of the `Target` method is `Nothing`, indicating that there is no implicit object, since the method is `shared`.

SetMember in Figure B.1 optimizes the assignment of fields and properties, when the object is dynamically typed. A `SetMember` call-site binder is created for this purpose.

Appendix C. Optimization Rules for Boo

Figures C.1 and C.2 show the transformation rules implemented to optimize Boo programs. Although the Boo language provides the `duck` keyword for dynamically typed variables, we keep using the *dynamic* type for consistency. $\oplus_{\mathrm{Boo}}$ and $\ominus_{\mathrm{Boo}}$ represent, respectively, the optimized binary and unary Boo operators. We also transform explicit (CCast$_{\mathrm{Boo}}$) and implicit (CAssign$_{\mathrm{Boo}}$, CFunction$_{\mathrm{Boo}}$ and CMethod$_{\mathrm{Boo}}$) type conversions. CMethod$_{\mathrm{Boo}}$ optimizes the type conversion of methods. When the method is overloaded, the types of the i^{th} parameter must be equal to be able to perform the type conversion. In case it is *dynamic*, no conversion is required.

As for VB, we also optimize indexing operations (GetIndex$_{\mathrm{Boo}}$ and SetIndex$_{\mathrm{Boo}}$ in Figure C.2), method invocations (IMethod$_{\mathrm{Boo}}$) and member accesses (GetMember$_{\mathrm{Boo}}$ and SetMember$_{\mathrm{Boo}}$). In Boo, we add two optimizations not implemented in VB. The first one, IDelegate$_{\mathrm{Boo}}$, is the use of dynamically typed delegates; i.e., methods and functions variables. In this language, function and methods are first-class entities, and they can also

(CDWhile)

$$\frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.Convert(Boolean))}}{\texttt{Do While}\ e\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop} \rightsquigarrow \texttt{Do While}\ \textit{callsite}\texttt{.Target(}\textit{callsite}, e\texttt{)}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop}}$$

(CRWhile)

$$\frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.Convert(Boolean))}}{\texttt{Do}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop While}\ e \rightsquigarrow \texttt{Do}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop While}\ \textit{callsite}\texttt{.Target(}\textit{callsite}, e\texttt{)}}$$

(CDUntil)

$$\frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.Convert(Boolean))}}{\texttt{Do Until}\ e\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop} \rightsquigarrow \texttt{Do Until}\ \textit{callsite}\texttt{.Target(}\textit{callsite}, e\texttt{)}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop}}$$

(CRUntil)

$$\frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.Convert(Boolean))}}{\texttt{Do}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop Until}\ e \rightsquigarrow \texttt{Do}\ \textit{stmt}^{+}_{\text{do}}\ \texttt{Loop Until}\ \textit{callsite}\texttt{.Target(}\textit{callsite}, e\texttt{)}}$$

(CIndex)

$$\frac{\begin{array}{c} e_1 : T_1 \qquad T_1 \neq \textit{dynamic} \\ e_2 : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.Convert(Integer))} \end{array}}{e_1\texttt{(}e_2\texttt{)} \rightsquigarrow e_1\texttt{(}\textit{callsite}\texttt{.Target(}\textit{callsite}, e_2\texttt{))}}$$

(IOCMethod)

$$\frac{\begin{array}{c} m : C \times T_1^1 \times \ldots \times T_i^1 \times \ldots \times T_n^1 \to T_r^1 \wedge \ldots \wedge C \times T_1^m \times \ldots \times T_i^m \times \ldots \times T_n^m \to T_r^m \\ e_i : \textit{dynamic} \qquad \exists\, T_i^j \,.\, T_i^j \neq \textit{dynamic}^{\,j \,\in\, 1 \ldots m} \\ \textit{callsite} = \texttt{New CallSite(Binder.InvokeMember(}m\texttt{))} \end{array}}{C\,\texttt{.}\,m\texttt{(}e_1, \ldots, e_i, \ldots, e_n\texttt{)} \rightsquigarrow \textit{callsite}\texttt{.Target(}\textit{callsite}, \texttt{Nothing}, e_1, \ldots, e_i, \ldots, e_n\texttt{)}}$$

(SetMember)

$$\frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{New CallSite(Binder.SetMember(}f\texttt{))}}{e_1\,\texttt{.}\,f\ \texttt{=}\ e_2 \rightsquigarrow \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2\texttt{)}}$$

Figure B.1: Additional optimization rules for VB.

be dynamically typed –in VB, they must be called with the `invoke` method. The second new optimization is the invocation of constructors with (at least) one of its parameters dynamically typed (IConstructor$_{\text{Boo}}$). In that case, the expression is replaced with an `InvokeConstructor` call-site.

Appendix D. Optimization Rules for Cobra

Figure D.1 shows the optimization rules for Cobra. This programming language does not support type conversion for dynamically typed expressions. As for Boo, IConstructor$_{\text{Cobra}}$ optimizes constructor invocation when one of the arguments is dynamically typed.

Appendix E. Optimization Rules for Fantom

In Fantom, all the optimizations are done when the `->` operator is used. This operator sends a message to an object, but no static type checking is performed. Therefore, Fantom does not define a *dynamic* type. All the dynamically typed expressions are expressed with the `->` operator. Consequently, the Fantom optimizations transform method invocation expressions into `InvokeMember` call-sites (Figure E.1).

Fantom represents language operators as methods, so that the `1->plus(2)` dynamically typed expression corresponds to the `1+2` statically typed one. Consequently, IMethod$_{\text{Fantom}}$ optimizes both methods and operators. However, when the method represents the operator of a built-in type (e.g., `1->plus(2)`), Fantom calls a class method that performs nested type inspections that cannot be optimized by the DLR [26]. To detect this special case, the premise $m \in M_{operators} \Rightarrow T \notin T_{\text{Fantom}-built-in}$ checks that, when m is an operator, T must not be a built-in type.

(BinaryOp$_{\text{Boo}}$)

$$\frac{\begin{array}{c} e_1 : \mathit{dynamic} \vee e_2 : \mathit{dynamic} \\ \oplus_{\text{Boo}} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{==}, \texttt{!=}, \texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}, \texttt{<<}, \texttt{>>}, \texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{and}, \texttt{or}\} \\ \mathit{callsite} = \texttt{CallSite(Binder.BinaryOperation(ExpressionType.}\oplus_{\text{Boo}}\texttt{)} \end{array}}{e_1 \oplus_{\text{Boo}} e_2 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}}$$

(UnaryOp$_{\text{Boo}}$)

$$\frac{\begin{array}{c} e : \mathit{dynamic} \qquad \ominus_{\text{Boo}} \in \{\texttt{not}, \texttt{-}\} \\ \mathit{callsite} = \texttt{CallSite(Binder.UnaryOperation(ExpressionType.}\ominus_{\text{Boo}}\texttt{)} \end{array}}{\ominus_{\text{Boo}}\, e \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

(CCast$_{\text{Boo}}$)

$$\frac{e : \mathit{dynamic} \qquad T \neq \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.Convert(}T\texttt{))}}{e\ \texttt{cast}\ T \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

(CAssign$_{\text{Boo}}$)

$$\frac{\begin{array}{c} \qquad\qquad\qquad\qquad e_1 : T \\ T \neq \mathit{dynamic} \qquad e_2 : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.Convert(}T\texttt{))} \end{array}}{e_1\texttt{=}e_2 \rightsquigarrow e_1\texttt{=}\mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_2\texttt{)}}$$

(CFunction$_{\text{Boo}}$)

$$\frac{\begin{array}{c} e : T_1^1 \times \ldots \times T_i^1 \times \ldots \times T_n^1 \rightarrow T_r^1 \wedge \ldots \wedge T_1^j \times \ldots \times T_i^j \times \ldots \times T_n^j \rightarrow T_r^j \wedge \ldots \\ \ldots \wedge T_1^m \times \ldots \times T_i^m \times \ldots \times T_n^m \rightarrow T_r^m \qquad e_i : \mathit{dynamic} \\ T_i^1 = \ldots = T_i^m = T \neq \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.Convert(}T\texttt{))} \end{array}}{e\texttt{(}e_1, \ldots, e_i, \ldots, e_n\texttt{)} \rightsquigarrow e\texttt{(}e_1, \ldots, \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_i\texttt{)}, \ldots, e_n\texttt{)}}$$

(CMethod$_{\text{Boo}}$)

$$\frac{\begin{array}{c} e : C \qquad e_i : \mathit{dynamic} \\ m : C \times T_1^1 \times \ldots \times T_i^1 \times \ldots \times T_n^1 \rightarrow T_r^1 \wedge \ldots \wedge C \times T_1^j \times \ldots \times T_i^j \times \ldots \times T_n^j \rightarrow T_r^j \wedge \ldots \\ \ldots C \times \wedge T_1^m \times \ldots \times T_i^m \times \ldots \times T_n^m \rightarrow T_r^m \\ T_i^1 = \ldots = T_i^m = T \neq \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.Convert(}T\texttt{))} \end{array}}{e.m\texttt{(}e_1, \ldots, e_i, \ldots, e_n\texttt{)} \rightsquigarrow e.m\texttt{(}e_1, \ldots, \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_i\texttt{)}, \ldots, e_n\texttt{)}}$$

Figure C.1: Optimization of Boo basic expressions and type conversions.

$$\textsc{(GetIndex}_{\text{Boo}}\textsc{)}\quad \frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.GetIndex())}}{e_1\texttt{[}e_2\texttt{]} \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2\texttt{)}}$$

$$\textsc{(SetIndex}_{\text{Boo}}\textsc{)}\quad \frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.SetIndex())}}{e_1\texttt{[}e_2\texttt{]=}e_3 \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2, e_3\texttt{)}}$$

$$\textsc{(IMethod}_{\text{Boo}}\textsc{)}\quad \frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.InvokeMember(}m\texttt{))}}{e.m\texttt{(}e_1, \ldots, e_n\texttt{)} \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e, e_1, \ldots, e_n\texttt{)}}$$

$$\textsc{(IDelegate}_{\text{Boo}}\textsc{)}\quad \frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.Invoke())}}{e\texttt{(}e_1, \ldots, e_n\texttt{)} \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e, e_1, \ldots, e_n\texttt{)}}$$

$$\textsc{(IConstructor}_{\text{Boo}}\textsc{)}\quad \frac{\begin{array}{c}\exists\, e_i \,.\, e_i : \textit{dynamic}^{\; i \in 1 \ldots n} \\ \textit{callsite} = \texttt{CallSite(Binder.InvokeConstructor())}\end{array}}{C\texttt{(}e_1, \ldots, e_n\texttt{)} \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, C, e_1, \ldots, e_n\texttt{)}}$$

$$\textsc{(GetMember}_{\text{Boo}}\textsc{)}\quad \frac{e : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.GetMember(}\omega\texttt{))}}{e\texttt{.}\omega \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e\texttt{)}}$$

$$\textsc{(SetMember}_{\text{Boo}}\textsc{)}\quad \frac{e_1 : \textit{dynamic} \qquad \textit{callsite} = \texttt{CallSite(Binder.SetMember(}\omega\texttt{))}}{e_1\texttt{.}\omega \texttt{ = } e_2 \leadsto \textit{callsite}\texttt{.Target(}\textit{callsite}, e_1, e_2\texttt{)}}$$

Figure C.2: Optimization of Boo invocations, indexing and member access.

(Binary$\text{Op}_{\textsc{Cobra}}$)

$$\frac{\begin{array}{c} e_1 : \mathit{dynamic} \vee e_2 : \mathit{dynamic} \\ \oplus_{\text{Cobra}} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{==}, \texttt{<>}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{+=}, \texttt{-=}, \texttt{*=}, \texttt{/=}, \texttt{\%=}, \texttt{\&=}, \texttt{|=}, \texttt{\^{}=}\} \\ \mathit{callsite} = \texttt{CallSite(Binder.BinaryOperation(ExpressionType.}\oplus_{\text{Cobra}}\texttt{)} \end{array}}{e_1 \oplus_{\text{Cobra}} e_2 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}}$$

(UnaryOp$_{\textsc{Cobra}}$)

$$\frac{\begin{array}{c} e : \mathit{dynamic} \qquad \ominus_{\text{Cobra}} \in \{\texttt{not}, \texttt{\~{}}\} \\ \mathit{callsite} = \texttt{CallSite(Binder.UnaryOperation(ExpressionType.}\ominus_{\text{Cobra}}\texttt{)} \end{array}}{\ominus_{\text{Cobra}}\ e \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

(GetIndex$_{\textsc{Cobra}}$)

$$\frac{e_1 : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.GetIndex())}}{e_1\texttt{[}e_2\texttt{]} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}}$$

(SetIndex$_{\textsc{Cobra}}$)

$$\frac{e_1 : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.SetIndex())}}{e_1\texttt{[}e_2\texttt{]=}e_3 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2, e_3\texttt{)}}$$

(IMethod$_{\textsc{Cobra}}$)

$$\frac{e : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.InvokeMember(}m\texttt{))}}{e.m\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e, e_1, \ldots, e_n\texttt{)}}$$

(IConstructor$_{\textsc{Cobra}}$)

$$\frac{\begin{array}{c} \exists\ e_i\ .\ e_i : \mathit{dynamic}\ ^{i \in 1 \ldots n} \\ \mathit{callsite} = \texttt{CallSite(Binder.InvokeConstructor())} \end{array}}{C\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, C, e_1, \ldots, e_n\texttt{)}}$$

(GetMember$_{\textsc{Cobra}}$)

$$\frac{e : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.GetMember(}f\texttt{))}}{e\texttt{.}f \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}}$$

(SetMember$_{\textsc{Cobra}}$)

$$\frac{e_1 : \mathit{dynamic} \qquad \mathit{callsite} = \texttt{CallSite(Binder.SetMember(}f\texttt{))}}{e_1\texttt{.}f \texttt{ = } e_2 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}}$$

Figure D.1: Cobra optimization rules.

$$\frac{\begin{array}{c}m \in M_{operators} \Rightarrow T \notin T_{\text{Fantom}-built-in} \\ callsite = \texttt{CallSite(Binder.InvokeMember(}m\texttt{))}\end{array}}{e\texttt{->}m\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow callsite\texttt{.Target(}callsite, e_1, \ldots, e_n\texttt{)}} \quad (\text{IMethod}_{\text{Fantom}})$$

where $M_{operators} \in$ { `negate`, `increment`, `decrement`, `toFloat`, `toDecimal`, `upper`, `lower`, `toStr`, `chars`, `size`, `typeof`, `sqrt`, `tan`, `sin`, `plus`, `minus`, `mult`, `div`, `mod`, `div`, `mod`, `pow`, `compare`, `equals`, `getRange`, `removeAt`, `size`, `get`, `set` } *and*
$T_{\text{Fantom}-built-in}$ = { `Bool`, `Long`, `Double`, `BigDecimal` }

Figure E.1: Fantom optimization rules.

Appendix F. Optimization Rules for *StaDyn*

Figure F.1 shows the optimization rules included in the *StaDyn* compiler. *StaDyn* infers type information of all the dynamically typed references but method arguments. Therefore, the expressions in our formalization are *dynamic* only when they are built from a `dynamic` argument. We optimize method (IMETHOD$_{\text{StaDyn}}$) and constructor (ICONSTRUCTOR$_{\text{StaDyn}}$) invocations, and field accesses (GETMEMBER$_{\text{StaDyn}}$ and SETMEMBER$_{\text{StaDyn}}$). The rest of transformations are not applicable to *StaDyn* because it already optimizes the generated code by implementing the type system rules in the generated code [21].

$$\frac{\begin{array}{c}(\textsc{IMethod}_{StaDyn})\\ e : dynamic \qquad callsite = \texttt{new CallSite(Binder.InvokeMember(}m\texttt{))}\end{array}}{e.m(e_1, \ldots, e_n) \rightsquigarrow callsite\texttt{.Target(}callsite, e, e_1, \ldots, e_n\texttt{)}}$$

$$\frac{\begin{array}{c}(\textsc{IConstructor}_{StaDyn})\\ \exists\ e_i\ .\ e_i : dynamic\ ^{i\,\in\,1\ldots n}\\ callsite = \texttt{new CallSite(Binder.InvokeConstructor())}\end{array}}{\texttt{new}\ C(e_1, \ldots, e_n) \rightsquigarrow callsite\texttt{.Target(}callsite, C, e_1, \ldots, e_n\texttt{)}}$$

$$\frac{\begin{array}{c}(\textsc{GetMember}_{StaDyn})\\ e : dynamic \qquad callsite = \texttt{new CallSite(Binder.GetMember(}f\texttt{))}\end{array}}{e\,.\,f \rightsquigarrow callsite\texttt{.Target(}callsite, e\texttt{)}}$$

$$\frac{\begin{array}{c}(\textsc{SetMember}_{StaDyn})\\ e_1 : dynamic \qquad callsite = \texttt{new CallSite(Binder.SetMember(}f\texttt{))}\end{array}}{e_1\,.\,f\ \texttt{=}\ e_2 \rightsquigarrow callsite\texttt{.Target(}callsite, e_1, e_2\texttt{)}}$$

Figure F.1: *StaDyn* optimization rules.

References

[1] F. Ortin, J. M. Cueva, Dynamic adaptation of application aspects, Journal of Systems and Software (2004) 229–243.

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

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

[4] A. Hunt, D. Thomas, The pragmatic programmer: from journeyman to master, Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999.

[5] ECMA-357, ECMAScript for XML (E4X) Specification, 2nd edition, European Computer Manufacturers Association, Geneva, Switzerland, 2005.

[6] D. Crane, E. Pascarello, D. James, AJAX in Action, Manning Publications, Greenwich, 2005.

[7] G. van Rossum, L. Fred, J. Drake, The Python Language Reference Manual, Network Theory, United Kingdom, 2003.

[8] A. Latteier, M. Pelletier, C. McDonough, P. Sabaini, The Zope book, `http://www.zope.org/Documentation/Books/ZopeBook/` (2008).

[9] Django Software Foundation, Django, the web framework for perfectionists with deadlines, `https://www.djangoproject.com` (2015).

[10] E. Meijer, P. Drayton, Static typing where possible dynamic typing when needed: The end of the cold war between programming languages, in: Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004, pp. 1–6.

[11] B. C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, Massachusetts, 2002.

[12] F. Ortin, M. Garcia, J. M. Redondo, J. Quiroga, Combining static and dynamic typing to achieve multiple dispatch, Information – An International Interdisciplinary Journal 16 (12) (2013) 8731–8750.

[13] F. Ortin, P. Conde, D. F. Lanvin, R. Izquierdo, Runtime performance of `invokedynamic`: an evaluation with a Java library, IEEE Software 31 (4) (2014) 82–90.

[14] J. Strachan, Groovy 2.0 release notes, `http://groovy.codehaus.org/Groovy+2.0+release+notes` (2014).

[15] G. Bierman, E. Meijer, M. Torgersen, Adding dynamic types to C#, in: Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP'10, Springer-Verlag, Maribor, Slovenia, 2010, pp. 76–100.

[16] F. Ortin, M. A. Labrador, J. M. Redondo, A hybrid class- and prototype-based object model to support language-neutral structural intercession, Information and Software Technology 44 (1) (2014) 199–219.

[17] J. M. Redondo, F. Ortin, A comprehensive evaluation of widespread python implementations, IEEE Software 34 (4) (2015) 76–84.

[18] B. Chiles, A. Turner, Dynamic Language Runtime, `http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512` (2012).

[19] M. Barnett, Microsoft Research Common Compiler Infrastructure, http://research.microsoft.com/en-us/projects/cci/ (2015).

[20] F. Ortin, D. Zapico, J. Perez-Schofield, M. García, Including both static and dynamic typing in the same programming language, IET Software 4 (4) (2010) 268–282.

[21] M. Garcia, F. Ortin, J. Quiroga, Design and implementation of an efficient hybrid dynamic and static typing language, Software: Practice and Experience (to be published). `doi:10.1002/spe.2291`.

[22] F. Ortin, Type inference to optimize a hybrid statically and dynamically typed language, Computer Journal 54 (11) (2011) 1901–1924.

[23] P. Vick, The Microsoft Visual Basic language specification, Microsoft Corporation, Redmond, Washington, 2007.

[24] ECMA-335, Common Language Infrastructure (CLI), European Computer Manufacturers Association, Geneva, Switzerland, 2012.

[25] B. C. Pierce, Programming with intersection types and bounded polymorphism, Tech. Rep. CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA (1992).

[26] F. Ortin, J. Quiroga, J. M. Redondo, M. Garcia, Attaining multiple dispatch in widespread object-oriented languages, Dyna 81 (186) (2014) 242–250.

[27] J. Quiroga, F. Ortin, Optimizing runtime performance of hybrid dynamically and statically typed languages for the .NET platform (Web page), `http://www.reflection.uniovi.es/stadyn/download/2015/jss` (2015).

[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, 1995.

[29] Microsoft Developer Network, Dynamic source code generation and compilation, http://msdn.microsoft.com/en-us/library/650ax5cx(v=vs.110).aspx (2015).

[30] F. Ortin, M. Garcia, Modularizing different responsibilities into separate parallel hierarchies, Communications in Computer and Information Science 275 (2013) 16–31.

[31] R. B. De Oliveira, The Boo programming language, http://boo.codehaus.org (2014).

[32] P. McEvoy, Brail, a view engine for MonoRail, http://docs.castleproject.org/MonoRail.Brail.ashx (2015).

[33] A. Davey, C. Vivier, Specter framework, a behaviour-driven development framework for .NET and Mono, http://specter.sourceforge.net (2015).

[34] K. Kozmic, Castle windsor, mature inversion of control container for .NET and Silverlight, http://docs.castleproject.org/Windsor.MainPage.ashx (2015).

[35] Unity Technologies, Unity3d, http://unity3d.com (2015).

[36] J. Siegel, D. Frantz, H. Mirsky, R. Hudli, P. de Jong, A. Klein, B. Wilkins, A. Thomas, W. Coles, S. Baker, M. Balick, COBRA fundamentals and programming, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[37] B. Frank, A. Frank, Fantom, the language formerly known as Fan, http://fantom.org (2015).

[38] NetVed Technologies, Kloudo, the simplest way to get your business organized, http://www.kloudo.com (2015).

[39] SkyFoundry, Skyspark, analytics software for a world of smart devices, http://skyfoundry.com/skyspark (2015).

[40] T. Colar, NetColarDB, ORM features on top of Fantom's SQL package, https://bitbucket.org/tcolar/fantomutils/src/tip/netColarDb (2015).

[41] P. S. Foundation, Pybench benchmark project trunk page, http://svn.python.org/projects/python/trunk/Tools/pybench (2015).

[42] R. P. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM 27 (10) (1984) 1013–1030.

[43] Krintz, Chandra, A collection of phoenix-compatible C# benchmarks, http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks (2015).

[44] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, ACM SIGPLAN Notices 42 (10) (2007) 57–76.

[45] D. J. Lilja, Measuring computer performance: a practitioner's guide, Cambridge University Press, 2005.

[46] MicrosoftTechnet, Windows server techcenter: Windows performance monitor, http://technet.microsoft.com/en-us/library/cc749249.aspx (2015).

[47] Microsoft, Windows management instrumentation, http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx (2015).

[48] S. Thatte, Quasi-static typing, in: Proceedings of the 17th symposium on Principles of programming languages (POPL), ACM, San Francisco, California, United States, 1990, pp. 367–381.

[49] C. Flanagan, S. N. Freund, A. Tomb, Hybrid types, invariants, and refinements for imperative objects, in: Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL), ACM, San Antonio, Texas, 2006, pp. 1–11.

[50] J. G. Siek, W. Taha, Gradual typing for functional languages, in: Scheme and Functional Programming Workshop, 2006, pp. 1–12.

[51] J. G. Siek, W. Taha, Gradual typing for objects, in: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Berlin, Germany, 2007, pp. 2–27.

[52] J. G. Siek, M. Vachharajani, Gradual typing with unification-based inference, in: Proceedings of the Dynamic Languages Symposium, ACM, Paphos, Cyprus, 2008, pp. 7:1–7:12.

[53] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, T. Wrigstad, Thorn—robust, concurrent, extensible scripting on the JVM, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), ACM, Orlando, Florida, 2009, pp. 117–136.

[54] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, J. Vitek, Integrating typed and untyped code in a scripting language, in: Proceedings of the 37th annual symposium on Principles of Programming Languages (POPL), POPL'10, ACM, New York, NY, USA, 2010, pp. 377–388.

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

[56] G. Bracha, Pluggable type systems, in: Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004, pp. 1–6.

[57] F. Ortin, M. Garcia, Union and intersection types to support both dynamic and static typing., Information Processing Letters 111 (6) (2011) 278–286.

[58] A. Goldberg, D. Robson, Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[59] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, POPL'84, ACM, New York, NY, USA, 1984, pp. 297–302.

[60] F. Ortin, J. M. Redondo, J. B. G. Perez-Schofield, Efficient virtual machine support of runtime structural reflection, Science of Computer Programming 74 (2009) 836–860.

[61] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, OOPSLA'87, ACM, New York, NY, USA, 1987, pp. 227–242.

[62] C. Chambers, D. Ungar, Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language, in: Conference on Programming language design and implementation (PLDI), 1989, pp. 146–160.

[63] U. Hölzle, C. Chambers, D. Ungar, Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, in: ECOOP'91 European Conference on Object-Oriented Programming, Springer, 1991, pp. 21–38.

[64] U. Hölzle, D. Ungar, Reconciling responsiveness with performance in pure object-oriented languages, ACM Transactions on Programming Languages and Systems (TOPLAS) 18 (4) (1996) 355–400.

[65] Google Inc., The V8 JavaScript engine, `https://code.google.com/p/v8` (2015).

[66] Mozilla, The SpiderMonkey JavaScript engine, `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey` (2015).

[67] J. M. Redondo, F. Ortin, Efficient support of dynamic inheritance for class- and prototype-based languages, Journal of Systems and Software 86 (2) (2013) 278–301.

[68] T. Würthinger, C. Wimmer, L. Stadler, Dynamic code evolution for Java, in: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ'10, ACM, New York, NY, USA, 2010, pp. 10–19.

[69] T. Würthinger, C. Wimmerb, L. Stadler, Unrestricted and safe dynamic code evolution for Java, Science of Computer Programming 78 (5) (2013) 481–498.

[70] Oracle, The Da Vinci Machine, a multi-language renaissance for the Java virtual machine architecture, `http://openjdk.java.net/projects/mlvm` (2012).

[71] B. Hackett, S.-y. Guo, Fast and precise hybrid type inference for javascript, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'12, ACM, New York, NY, USA, 2012, pp. 239–250.

[72] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS'09, ACM, New York, NY, USA, 2009, pp. 18–25.