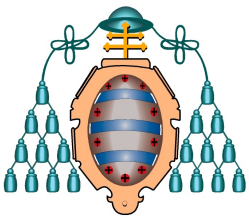
Optimizing Runtime Performance of Dynamically Typed Code



Jose Quiroga Alvarez

PhD Supervisor

Dr. Francisco Ortín Soler

Department of Computer Science

University of Oviedo

A thesis submitted for the degree of

Doctor of Philosophy

Oviedo, Spain

April 2016

Acknowledgements

This work has been partially funded by the Spanish Department of Science and Technology, under the National Program for Research, Development and Innovation. The main project was *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages* (TIN2011-25978). The work is also part of the project entitled *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software* (TIN2008-00276).

I was awarded a FPI grant by the Spanish Department of Science and Technology. The objective of these grants is to support graduate students wishing to pursue a PhD degree associated to a specific research project. This PhD dissertation is associated to the project TIN2011-25978 (previous paragraph).

This work has also been funded by Microsoft Research, under the project entitled *Extending dynamic features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution Request for Proposals.*

Part of the research discussed in this dissertation has also been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Innovation Plan (grant GRUPIN14-100).

Abstract

Dynamic languages are widely used for different kinds of applications including rapid prototyping, Web development and programs that require a high level of runtime adaptiveness. However, the lack of compile-time type information involves fewer opportunities for compiler optimizations, and no detection of type errors at compile time. In order to provide the benefits of static and dynamic typing, hybrid typing languages provide both typing approaches in the very same programming language. Nevertheless, dynamically typed code in this languages still shows lower performance and lacks early type error detection.

The main objective of this PhD dissertation is to optimize runtime performance of dynamically typed code. For this purpose, we have defined three optimizations applicable to both dynamic and hybrid typing languages. The proposed optimizations have been included in an existing compiler to measure the runtime performance benefits.

The first optimization is performed at runtime. It is based on the idea that the dynamic type of a reference barely changes at runtime. Therefore, if the dynamic type is cached, we can generate specialized code for that precise type. When there is a cache hit, the program will perform close to its statically typed version. For this purpose, we have used the DLR of the .Net framework to optimize all the existing hybrid typing languages for that platform. The optimizations are provided as a binary optimization tool, and included in an existing compiler. Performance benefits range from 44.6% to 11 factors.

The second optimization is aimed at improving the performance of dynamic variables holding different types in the same scope. We have defined a modification of the classical SSA transformations to improve the task of type inference. Due to the proposed algorithms, we infer one single type for each local variable. This makes the generated code to be significantly faster, since type casts are avoided. When a reference has a flow sensitive type, we use union types and nested runtime type inspections. Since we avoid the use of reflection, execution time is significantly faster than existing approaches. Average performance improvements range from 6.4 to 21.7 factors. Besides, the optimized code consumes fewer memory resources.

The third optimization is focused on the improvement of multiple dispatch for object-oriented languages. One typical way of providing multiple dispatch is resolving method overload at runtime: depend-

ing on the dynamic types of the arguments, the appropriate method implementation is selected. We propose a multiple dispatch mechanism based on the type information of the arguments gathered by the compiler. With this information, a particular specialization of the method is generated, making the code to run significantly faster than reflection or nested type inspection. This approach has other benefits such as better maintainability and readability, lower code size, parameter generalization, early type error detection and fewer memory resources.

Keywords

Dynamic typing, runtime performance, optimization, hybrid dynamic and static typing, Dynamic Language Runtime, Static Single Assignment, SSA Form, multiple dispatch, multi-method, union types, reflection, *StaDyn*, .NET

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

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 [2]. An important benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software. Some features provided by most dynamically typed languages are meta-programming, variables with different types in the same scope, high levels of reflection, code mobility, and dynamic reconfiguration and distribution [3].

For example, in the Web development scenario, Ruby [4] is used for the rapid development of database-backed Web applications with the Ruby on Rails framework [5]. This framework has confirmed the simplicity of implementing the DRY (Do not Repeat Yourself) [6] and the Convention over Configuration [5] principles in a dynamic language. Nowadays, JavaScript [7] is being widely employed to create interactive Web applications [8], while PHP is one of the most popular languages for developing Web-based views. Python [9] is used for many different purposes; two well-known examples are the Zope application server [10] (a framework for building content management systems, intranets and custom applications) and the Django Web application framework [11].

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 [12]. 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 [13]. Moreover, avoiding the runtime type inspection and type checking performed by dynamically typed languages commonly involve a runtime performance improvement [14, 15].

Since both approximations offer different benefits, some existing languages provide hybrid static and dynamic typing, such as Objective-C, Visual Basic,

```
class Triangle {
    internal int[] edges;
    public Triangle(int edge1, int edge2, int edge3) {
        this.edges = new int[] { edge1, edge2, edge3};
    }
}

class Square {
    internal int[] edges;
    public Square(int edge) {
        this.edges = new int[] { edge, edge, edge, edge};
    }
}

class Circumference {
    internal int radius;
    public Circumference(int radius) {
        this.radius = radius;
    }
}

class Program {
    static double TrianglePerimeter(Triangle poly) {
        double result = 0;
        foreach (var edge in poly.edges)
          result += edge;
        return result;
    }

    static double PolygonPerimeter(dynamic poly){
        double result = 0;
        foreach(var edge in poly.edges)
            result += edge;
        return result;
    }

    static void Main() {
        double perimeter;
        Triangle triangle = new Triangle(3, 4, 5);
        Square square = new Square(3);
        Circumference circ = new Circumference(4);

        perimeter = TrianglePerimeter(triangle);
        // compiler error
        perimeter = TrianglePerimeter(square);

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

        // runtime error
        perimeter = PolygonPerimeter(circ);
    }
}
```

Figure 1.1: Hybrid static and dynamic typing example in C#.

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) [16]. Likewise, the statically typed C# language has included the `dynamic` type in its version 4.0 [17], indicating the compiler to postpone type checks until runtime.

The example hybrid statically and dynamically typed C# code in Figure 1.1 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.

The `PolygonPerimeter` method implements the same algorithm but using dynamic typing. The `poly` parameter is declared as `dynamic`, meaning that type checking is postponed until runtime. The flexibility of dynamic typing supports duck typing [18]: 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 to this method 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. C#, and most hybrid languages for .NET and Java, implement this runtime type checking using introspection, causing a performance penalty [18]. In general, the runtime type checks implemented by dynamic languages generally cause runtime

performance costs [19]. To minimize the use of these introspection services, some techniques may be used.

In this dissertation, we propose some different techniques to optimize dynamically typed code. We discuss the use of runtime caches for optimizing typical dynamically typed operations, gathering type information of dynamically typed variables, using the Single Static Assignment (SSA) form to allow variables with different types in the same scope, and performing method specialization to optimize multi-methods. We have included these optimizations in the open source *StaDyn* programming language for the .NET platform.

1.2 Contributions

These are the major contributions of this dissertation:

1. Optimization of the common dynamically typed operations of dynamic typing languages using a runtime cache. This optimization is based on the idea that most of the times a dynamically typed variable holds the same dynamic type. If that is the case, we can cache the type and avoid the use of reflection, obtaining important performance gains.

2. Using SSA transformations to efficiently support variables with different types in the same scope. These transformations are implemented in the *StaDyn* compiler. Similar to the Java platform, .NET does not allow one variable to have different types in the same scope (`object` must be used). The code generated by our compiler performs significantly better than the use of `dynamic` in C#, avoiding unnecessary type conversions and reflective invocations.

3. Optimization of multiple dispatch methods by using information gathered from dynamically typed code. Multiple dispatch allows determining the actual method to be executed, depending on the dynamic types of its arguments. Using dynamic typing, the implementation of multiple dispatch is an easy task for those languages that provide method overload resolution at runtime. Instead of using introspection, we propose the generation of nested type method inspections depending on the possible types a dynamically typed variable may hold.

4. Including the optimizations in a full-fledged programming language. The proposed optimizations are included in the existing *StaDyn* programming language. This language is an extension of C# that gathers type information of `dynamic` references. This type information is used to perform some of the optimizations detailed in this PhD dissertation.

5. A tool to optimize binary .NET files compiled from the existing hybrid typing languages. That tool takes binary .NET code and produces new binary files with the same behavior and better runtime performance. The files to be optimized should be generated from one of the existing hybrid languages for the .NET platform.

6. Evaluation of runtime performance and memory consumption. A comparison of all the existing hybrid static and dynamic typing languages implemented for the .NET platform is presented. This comparison empirically shows the benefits and drawbacks of our optimizations.

1.3 Structure of the document

This PhD thesis is structured as follows. The next chapter presents related work, including a brief description of the *StaDyn* language (Section 2.1). Chapter 3 describes the first optimization, based on caching the common dynamically typed operations. Chapter 4 presents the second optimization, a set of SSA transformations to efficiently support variables with different types in the same scope. Chapter 5 describes the last optimization: multiple dispatch using information gathered for dynamic references. Different evaluations of runtime performance and memory consumption are presented in Sections 3.4, 4.3 and 5.3. Chapter 6 presents the conclusions and future work.

Appendixes A, B and C contain the complete tables of execution times and memory consumptions measured. Appendix D presents the list of publications derived from this PhD.

Chapter 2

Related Work

This section describes the existing research works related to this PhD dissertation. We start describing the *StaDyn* programming language, since that is the language implementation where we included all the proposed optimizations. Then, we describe the existing optimizations for hybrid static and dynamic languages (*StaDyn* is a hybrid typing language). Since we define optimizations performed at runtime, we also discuss the existing optimizations included in current dynamically typed platforms (Section 2.3). We then analyze the use of SSA form to obtain runtime performance optimizations. Finally, Section 2.5 describes the existing works in optimizing multi-methods.

2.1 The *StaDyn* programming language

The *StaDyn* programming language is an extension of C# 3.0 [20]. It extends the behavior of the `var` and `dynamic` keywords to provide both static and dynamic typing. We have used *StaDyn* because we know the internals of its implementation, but the work presented in this dissertation could be applied to any object-oriented hybrid typing language. In *StaDyn*, the type of references can be explicitly declared, while it is also possible to use the `var` keyword to declare implicitly typed references. *StaDyn* includes this keyword as a new type (it can be used to declare local variables, fields, method parameters and return types), whereas C# only provides its use in the declaration of initialized local references. Therefore, `var` references in *StaDyn* are more powerful than implicitly typed local variables in C#.

The dynamism of `var` references can be placed in a separate file (an XML document) [21]. The programmer does not need to manipulate these XML documents directly, leaving this task to the *StaDyn* IDE [22]. When the programmer (un)sets a reference as dynamic, the IDE transparently modifies the corresponding XML file. Depending on the dynamism of a `var` reference, type checking and type inference is performed pessimistically (for static references) or optimistically (for dynamic ones) –detailed in Section 2.1.3. Since the dynamism concern is not explicitly stated in the source code, *StaDyn* facilitates the conversion of dynamic

```
class Test {
    public static void Main() {
        var v;
        dynamic myObject;
        v = new int[10];
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            v[i] = i+1;
            sum += v[i]; // No compiler error
        }
        myObject = "StaDyn";
        System.Console.Write(myObject*2); // Compiler error
    }
}
```

Figure 2.1: Type inference of `var` and `dynamic` references.

references into static ones, and vice versa [23]. This separation facilitates the process of turning rapidly developed prototypes into final robust and efficient applications [24]. It is also possible to make parts of an application more adaptable, maintaining the robustness and runtime performance of the rest of the program.

C# 4.0 added the `dynamic` type to its static type system, supporting the safe combination of dynamically and statically typed code [17]. In C#, type checking of the references defined as `dynamic` is deferred until runtime. Following the C# approach, *StaDyn* also added `dynamic` to the language. The behavior is exactly the same as using a `var` variable set as dynamic in the XML document described in the previous paragraph.

2.1.1 Type inference

StaDyn provides type inference (type reconstruction) for `var` and `dynamic` variables. It defines an implicit parametric polymorphic type system [25], implementing the Hindley-Milner type inference algorithm to infer the types of local variables [26]. This algorithm was modified to perform type reconstruction of `var` and `dynamic` parameters and attributes (fields) [27].

The *StaDyn* program shown in Figure 2.1 is an example of this capability. The `v` variable is declared with no type, and the *StaDyn* compiler infers its type to `int[]`. Therefore, the use of `v` to compute `sum` produces no compiler error. Similarly, the type of `myObject` is inferred to `string`. Thus, the *StaDyn* compiler detects an error in the `myObject*2` expression, even though `myObject` was declared as `dynamic`.

2.1.2 Duck typing

Duck typing[1] [4] is a property of dynamic languages meaning that an object is interchangeable with any other object that implements the same dynamic inter-

[1] It receives its name from the idiom *if it walks like a duck and quacks like a duck, it must be a duck.*

```
var reference;
if (new Random().NextDouble() < 0.5)
  reference = new StringBuilder("A string builder");
else
  reference = "A string";
Console.WriteLine(reference.Length);
```

Figure 2.2: Static duck typing.

face, regardless of whether those objects have a related inheritance hierarchy or not. Duck typing is a powerful feature offered by most dynamic languages.

There exist statically typed programming languages such as Scala [28] or OCaml [29] that offer structural typing, providing part of the benefits of duck typing. However, the structural typing implementation of Scala is not implicit, forcing the programmer to explicitly declare part of the structure of types. In addition, intersection types should be used when more than one operation is applied to a variable, making programming more complicated. Although OCaml provides implicit structural typing, variables should only have one type in the same scope, and this type is the most general possible (principal) type [30]. Principal types are more restrictive than duck typing, because they do not consider all the possible (concrete) values a variable may hold.

The *StaDyn* programming language offers *static* duck typing. The benefit provided by *StaDyn* is not only that it supports (implicit) duck typing, but also that it is provided statically. Whenever a `var` or `dynamic` reference points to a potential set of objects that implement a public `m` method, the `m` message could be safely passed. These objects do not need to implement a common interface or a (abstract) class with the `m` method. Since this analysis is performed at compile time, the programmer benefits from both early type error detection and runtime performance.

The static duck typing of *StaDyn* makes its static type system *flow-sensitive*. This means that it takes into account the flow context of each `var` reference. It gathers *concrete* type information (opposite to classic *abstract* type systems) [31] knowing all the possible types a `var` or `dynamic` reference may hold. Instead of declaring a reference with an abstract type that embraces all the possible concrete values, the compiler infers the union of all possible concrete types a `var` reference may point to. Notice that different types depending on flow context could be inferred for the same reference, using the type inference mechanism mentioned above.

Code in Figure 2.2 shows this feature. `reference` may point to either a `StringBuilder` or a `String` object. Both objects have the `Length` property and, therefore, it is statically safe to access to this property. It is not necessary to define a common interface or class to pass this message.

The key technique used to obtain this concrete-type flow-sensitiveness is *union types* [32]. Concrete types are first obtained by the abovementioned unification algorithm (applied in assignments and method calls). Whenever a branch is detected, a union type is created with all the possible concrete types inferred.

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

Figure 2.3: Static `var` reference.

Type checking of union types depends on the dynamism concern (next section).

2.1.3 Dynamic and static typing

StaDyn permits the use of both statically and dynamically typed references. Explicitly, the programmer may use, respectively, the `var` and the `dynamic` keywords. Depending on their dynamism, type checking and type inference would be more pessimistic (static) or optimistic (dynamic), but the dynamic semantics of the programming language is not changed (i.e., program execution does not depend on its dynamism).

The source code in Figure 2.3 defines a `g` method, where `reference` may point to a `StringBuilder`, `String` or `Exception` object. However, even though `reference` is declared as `dynamic`, the compiler shows the following error message:

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

The error is produced because no public `Lenght` property (it is misspelled) is implemented in the `String`, `StringBuffer` or `Exception` classes. This message shows how type-checking is performed at compile time, even in dynamic scenarios, providing early type error detection. This feature improves the way most dynamic languages work. For example, the erroneous use of the `dynamic myObject` reference in Figure 2.1 is detected by the *StaDyn* compiler; whereas C# shows no type error at compile time, but the program shows a type error at runtime.

In *StaDyn*, setting a reference as `dynamic` does not imply that every message could be passed to that reference; static type-checking is still performed. The major change is that the type system is more optimistic when `dynamic` references are used. The dynamism concern implies a modification of type checking over union types. If the implicitly typed reference inferred with a union type is declared

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

Figure 2.4: Implicitly typed parameters.

as `var`, type checking is performed over all its possible concrete types. However, if the reference is `dynamic`, type checking is performed over those concrete types that do not produce a type error; if none exists, then a type error is shown –this semantics is formalized in [33].

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

The generated `g` function program will not produce any runtime type error because the random number that is generated will be either 1 or 2. However, if the programmer, once the prototype is tested, wants to compile the application with static typing, `dynamic` may be replaced with `var`. In this case, the compilation of the `g` method will produce an error message saying that `Length` is not a property of `Exception`. The programmer should then modify the source code to compile this program with the robustness and efficiency of a static type system, but without requiring to translate the source code to a new programming language since *StaDyn* provides both approaches.

2.1.4 Implicitly typed parameters

Concrete type reconstruction is not limited to local variables. *StaDyn* performs a global *flow-sensitive* analysis of implicit `var` references. The result is an implicit parametric polymorphism [25], more straightforward for the programmer than the one offered by Java, C# (F-bounded) and C++ (unbounded) [34].

Implicitly typed parameter references cannot be unified to a single concrete type. Since they represent any actual type of an argument, they cannot be inferred the same way as local references. This issue is shown in the source code of Figure 2.4. Both methods require the parameter to implement a specific method, returning its value. In the `getString` method, any object could be passed as a parameter because every object accepts the `ToString` message. In the `upper` method, the parameter should be any object implementing a `ToUpper` message. Depending on the type of the actual parameter, the *StaDyn* compiler generates the corresponding compilation error.

```
public class Node {
  private var data;
  private var next;
  public Node(var data, var next) {
    this.data = data;
    this.next = next;
  }
  public var getData() {
    return data;
  }
  public void setData(var data) {
    this.data = data;
  }
}
```

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

Figure 2.5: Implicitly typed attributes.

For this purpose the *StaDyn* type system was extended to be constraint-based [35]. Types of methods in *StaDyn* hold an ordered set of constraints specifying the set of restrictions that must be fulfilled by the parameters [36]. In our example, the type of the `upper` method is:

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

This means that the type of the parameter (α) should implement a public `ToUpper` method with no parameters, and the type returned by `ToUpper` (β) will be also returned by `upper`. Therefore, if an integer is passed to the `upper` method, a compiler error is shown. However, if a string is passed instead, the compiler not only reports no error, but also infers the resulting type as a string. Type constraint fulfillment is, thus, part of the type inference mechanism (the concrete algorithm can be consulted in [36]).

2.1.5 Implicitly typed attributes

StaDyn also provides the use of the `var` type in class fields (attributes). With implicitly typed attribute references, it is possible to create the generic `Node` class shown in Figure 2.5. The `Node` class can hold any `data` of any type. Each time the `setData` method is called, the new concrete type of the parameter is saved as the `data` field type. By using this mechanism, the two lines with comments report compilation errors. This coding style is polymorphic and it is more legible that the parametric polymorphism used in C++ and much more straightforward than the F-bounded polymorphism offered by Java and C#. At the same time, runtime performance is equivalent to explicit type declaration [1]. Since the possible concrete types of `var` and `dynamic` references are known at compile time, the compiler has more opportunities to optimize the generated code, improving runtime performance [1].

Implicitly typed attributes extend the constraint-based behavior of parameter references in the sense that the concrete type of the implicit object parameter (the object used in every non-static method invocation) could be modified on a method invocation expression. In our example, the type of the `data` attribute is

```
public class List {

  private var list;

  public List(Node node) {
    this.list = node;
  }
```

```
public static void Main() {
    Node node = new Node(true, 0);
    var aList = new List(node);
    bool b1 = aList.list.getData();
    node.setData(1);
    bool b2 = aList.list.getData(); // Error
    int n = aList.list.getData();
  }
}
```

Figure 2.6: Alias analysis.

modified each time the `setData` method (and the constructor) is invoked. This does not imply a modification of the whole `Node` type, only the type of the single `Node` object –due to the *concrete* type system employed.

For this purpose, a new kind of *assignment* constraint was added to the type system [36]. Each time a value is assigned to a `var` or `dynamic` field, an assignment constraint is added to the method being analyzed. This constraint postpones the unification of the concrete type of the attribute to be performed later, when an actual object is used in the invocation. Therefore, the unification algorithm is used to type-check method invocation expressions, using the concrete type of the actual object (a detailed description of the unification algorithm can be consulted in [36]).

2.1.6 Alias analysis for concrete type evolution

The problem of determining if a storage location may be accessed in more than one way is called *alias analysis* [37]. Two references are aliased if they point to the same object. Although alias analysis is mainly used for optimizations, *StaDyn* uses it to know the concrete types of the objects a reference may point to.

Code in Figure 2.6 uses the `Node` class previously shown in Figure 2.5. Initially, the `aList` reference points to a node whose data is a boolean. If we get the `data` inside the `Node` object inside the `List` object, we get a `bool`. Then, the `node` is modified to hold an integer value. Repeating the previous access to the `data` inside the `Node` object inside the `List` object, an `int` is then obtained.

The alias analysis algorithm implemented by *StaDyn* is type-based (uses type information to decide alias) [38], inter-procedural (makes use of inter-procedural flow information) [37], context-sensitive (differentiates between different calls to the same method) [39], and may-alias (detects all the objects a reference *may* point to; opposite to *must* point to) [40].

2.1.7 Implementation

The *StaDyn* programming language is implemented over the .NET Framework platform, using C#. The compiler is a multiple-pass language processor that fol-

lows the *Pipes and Filters* architectural pattern [41]. It uses the AntLR language processor tool to implement lexical and syntactic analysis [42]. Abstract Syntax Trees (ASTs) are implemented following the *Composite* design pattern [43] and each pass over the AST implements the *Visitor* design pattern [43].

The compiler implements the following AST visits: two visitors to load types into the types table; one visitor for symbol identification [44] and another one for type inference [45, 46]; and two visitors to generate code. The type system was implemented following the guidelines described in [47], and the code generation module follows the design in [24].

StaDyn generates .NET intermediate language and then assembles it to produce the binaries. At present, it uses the CLR 2.0 as the unique back-end. However, the code generator module follows the *Parallel Hierarchies* design pattern [24, 48] to add new back-ends, such as the DLR (Dynamic Language Runtime) [49] (Chapter 3) and the ЯRotor [50] platforms.

A brief description of the *StaDyn* programming language has been presented in this section. A formal specification of its type system is depicted in [36] and its semantics is presented in [27].

2.2 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* [51], *hybrid typing* [52] and *gradual typing* [53] 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 [53]. Gradual typing has also been defined for object-based languages, showing that gradual typing and subtyping are orthogonal and can be combined [54]. 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 [55].

Strongtalk was one of the first programming language implementation that included both dynamic and static typing in the same programming language. Strongtalk is a major re-thinking of the Smalltalk-80 programming language [56]. It retains the basic Smalltalk syntax and semantics [57], but a type system is added to provide more reliability and a better runtime performance. The Strongtalk type system is completely optional, following the *pluggable* type system approach [58]. The programmer selects the robustness and efficiency of a static type system, or the adaptiveness and expressiveness of dynamically typed code. This assumes that it is the programmer's responsibility to ensure that types are sound in regard to dynamic behavior. Type checking is performed at compile-time, but it does not guarantee an execution without type errors. Although, its

type system is not completely safe, it has been used to perform performance optimizations, implying a significant improvement.

Dylan is a high-level programming language, designed to allow efficient compilation of features commonly associated with dynamic languages [59]. Dylan permits both explicit and implicit variable declaration. It also supports two compilation scenarios: production and interactive. In the interactive mode, all the types are ignored and no static type checking is performed. This behavior is similar to the one offered by dynamic languages. When the production configuration is selected, explicitly typed variables are checked using a static type system. However, types of generic references (references without type declaration) are not inferred at compile time –they are always checked at runtime. The two modes of compilation proposed in Dylan are aimed at converting rapidly developed prototypes into robust and efficient production applications, reducing the changes to be done in the source code.

Boo is an object-oriented programming language that is both statically and dynamically typed, with a Python inspired syntax [60]. In Boo, references may be declared without specifying its type and the compiler performs type inference. Opposite to Python, references could only have one unique type in the same scope. In Boo, fields and parameters could not be declared without specifying its type. Boo offers dynamic type inference with a special type called `duck`. Any operation could be performed over a `duck` reference –no static typing is performed. Any dynamic reference is converted into a static one without a cast. The Boo compiler also provides a *ducky* option that interprets the `Object` type as if it was `duck`. This *ducky* option allows the programmer to test out the code more quickly, and makes coding in Boo feel much more like coding in a dynamic language. So, when the programmer has tested the application, he or she may wish to turn the *ducky* option back off and add various type declarations and casts.

Visual Basic for .NET also incorporates both dynamic and static typing [61]. Its dynamic type system supports duck typing, but no static type inference is performed over dynamic references. Every type can be converted to a dynamic one, and vice versa. Therefore, all the type checking of dynamic references is performed at runtime. At the same time, dynamic references do not produce any type error at compile time. Dynamic references are declared using the `Dim` reserved word and the variable identifier, omitting the `As` keyword and the variable type. Function parameters and class fields can also be declared as dynamic.

Objective-C is a general-purpose object-oriented extension of the C programming language [62]. It is commonly compiled into a native format, without requiring any virtual machine. Objective-C has recently grown in popularity due to its relation with the development of iOS and OS X applications. According to the Tiobe raking [63], in March 2016 Objective-C was the 15^{th} most used programming language; whereas it was the 45^{th} in March 2008. One of the main differences with C++ is that Objective-C is hybrid statically and dynamically typed. Method execution is based on message passing (between `[` and `]`) that performs no static type checking (duck typing). If the object to which the message is directed does not provide a suitable method, an `NSInvalidArgumentException`

is raised. Besides, Objective-C also provides an `id` type to postpone the static type checking until runtime.

Thorn is a programming language that allows the combination of dynamically and statically typed code [64]. Thorn offers `like` types, an intermediate point between static and dynamic types [65]. 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 [65].

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 [17]. 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 [17]. 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 [17]. The cache implemented by the DLR provides significant runtime performance benefits compared to the use of reflection [66].

Cobra is another hybrid static and dynamic typing programming language for the .NET platform [67]. The language is compiled to .NET assemblies. Although it is object oriented, it also supports functional features such as lambda expressions, closures, list comprehensions and generators. It provides first class support of unit tests and contracts. The way Cobra provides dynamic typing is similar to C# 4.0, offering a new `dynamic` type. Any expression is implicitly coerced to `dynamic` type, and the other way round.

The *Fantom* programming language generates both JVM and .NET code, providing a hybrid dynamic and static type system [68]. Instead of adding a new type, dynamic typing is provided with the `->` dynamic invocation operator. Unlike the dot operator, the dynamic invocation operator does not perform compile-time checking. In order to obtain duck typing over language operators, operators can be invoked as if they were methods. For instance, to evaluate `a+b` with dynamic typing, the Fantom programmer writes `a->plus(b)`. The returned type is the object top type (`Obj` in Fantom), so dynamically typed expressions are not implicitly converted into statically typed ones.

Groovy is a dynamically typed language for the Java platform. Groovy has included static typing in its version 2.0 [16]. The programmer can write explicit type annotations in Groovy 2.0, and force static type checking with the `@TypeChecked` and `@CompileStatic` annotations. If that is the case, some type errors are detected by the compiler, and significantly better runtime performance is obtained [69].

2.3 Optimizations of 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 [57]. 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) [70]. JIT compilation provided important performance benefits, making VisualWorks to be, on average, 3 times faster than GNU Smalltalk [3].

Self is a dynamic prototype-based object-oriented language supported by a JIT-compiler virtual machine [71]. When a dynamic method is executed, run-time type information is gathered to perform type specialization of method invocations, using the specific types inferred for each argument [72]. The overhead of dynamically bound message passing is reduced by means of inline caches [70], introducing polymorphic inline caches (PIC) for polymorphic invocations [73]. Some other adaptive optimization strategies where implemented to improve the performance of hotspot functions while the program is running [74].

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 [75]. 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 [75].

SpiderMonkey is the new JavaScript engine of Mozilla, currently included in the Firefox Web browser and the GNOME 3 desktop [76]. 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 [76].

ЯRotor is an extension of the .Net SSCLI virtual machine implementation that provides JIT-compilation of the structural reflective primitives provided by dynamic languages [3]. 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 [18]. On average, ЯRotor performs 4 times better than the DLR, consuming 65% fewer memory resources [77].

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 [78]. 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% [79]. This system is currently the reference implementation of the hot-swapping feature (JSR 292) of the Da Vinci Machine project [80].

2.4 Optimizations based on the SSA form

This PhD dissertation uses the Single Static Assignment (SSA) form to optimize the use of local variables with different types in the same scope (Chapter 4). The SSA form is a property of a program representation (commonly intermediate representations), which requires that each variable is assigned exactly once, and every variable is defined before it is used. SSA form was developed by Wegman, Zadeck, Alpern and Rosen for efficient computation of dataflow problems [81, 82]. The SSA form is used in global value numbering, congruence of variables, aggressive dead-code removal and constant propagation with conditional branches [83]. An efficient computation of the SSA form was developed by Ron Cytron *et al.* using dominance frontiers [84].

These popular optimizations have been included in both commercial and open-source compilers [85]. They use the SSA form as an intermediate representation during the optimization phases. Sometimes, some optimizations may introduce new variables, and hence additional transformations are performed to preserve the SSA form [86, 87].

The SSA form is also used in Just-in-time (JIT) compilation. In this case, the transformation to the SSA form is done at runtime [88, 89]. Examples of JIT compilers that use the SSA form are the V8 JavaScript Engine [88], the Java Virtual Machine (JVM) [90, 91], PyPy [92] and Lua JIT [89].

PyPy is an alternative implementation of Python that provides JIT compilation, memory usage optimizations, and full compatibility with CPython [93]. 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 [93]. The flow-graph generated in the object space is in SSA form [94]. The optimization techniques implemented have made PyPy outperform the rest of Python implementation in many different benchmarks [15].

Although SSA form was initially developed for optimizing imperative programs, other works apply SSA transformations to functional programming [95]. Richard A. Kelsey transforms Continuation Passing Style (CPS) functional programs into SSA form and vice versa [96]. He also provides a transformation for analyzing loops that are expressed as recursive procedures. This allows simplifying the optimizations and avoids interprocedural analysis.

There are also works on combining type systems and SSA form. *SafeTSA* extends the internal SSA representation used by JVM, adding type information [97]. This information is used to prevent malicious code and check referential integrity.

Matsuno and Ohori propose a type inference algorithm to produce SSA-equivalent type information [98]. Their type system allows type-directed optimizations without requiring an intermediate transformation of the original code.

Brian Hackett and Shu-yu Guo define a hybrid static and dynamic type inference algorithm for JavaScript based on points-to analysis [99]. 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% [99].

2.5 Multiple dispatch (multi-methods)

One of the optimizations proposed in this dissertation is aimed at improving multiple dispatch methods, also known as multi-methods [100]. This feature allows the runtime association of a message to a specific method, based on the runtime type of all its arguments. At runtime, the dynamic types of the arguments are inspected and the appropriate implementation of an overloaded method is invoked.

There exist some programming languages that provide multiple dispatch. *CLOS* [101] and *Clojure* [102] are examples of dynamically typed languages that include multi-methods in their semantics. Clojure has recently created a port for .NET that makes use of the DLR [103]. Clojure supports multiple dispatch on argument types and values. A Clojure multi-method is a combination of a dispatching function (defined with `defmulti`), and one or more method implementations (using `defmethod`). When a multi-method is called, the dispatch function is transparently invoked with the same arguments. The value returned by the dispatch function, called the *dispatch value*, is used to select the appropriate method implementation to be invoked. This approach is fully dynamic, detecting all the type errors at runtime.

Xtend is a Java extension that, among other features, provides statically typed multiple dispatch [104]. Method resolution and method binding in Xtend are done at compile time, as in Java. Dylan [105], Cecil [100] and Groovy 2 [16] are programming languages that provide both dynamic and static typing, and dynamically typed multi-methods (multiple dispatch).

Many different approaches exist to provide multiple dispatch to the Java platform. One of the first works is *Runabout*, a library to support two-argument dispatch (i.e., double dispatch) for Java [106]. Runabout is based on improving a previous reflective implementation of the *Visitor* pattern called *Walkabout* [107]. Double dispatch is achieved without modifying the existing classes (e.g., the *Visitor* pattern requires adding an `accept` method to a class hierarchy). The programmer specifies the different `visit` method implementations in a class, extending the provided `Runabout` class. The appropriate method implementation

is found via reflection, but method invocation is performed by generating Java bytecode at runtime. The generated bytecode does not use reflection and it is optimized by the just-in-time compiler just like the rest of the application, implying a significant runtime performance improvement compared to `Walkabout` [107].

Dynamic Dispatcher is a double-dispatch framework for Java [108]. Three different dispatch methods are provided: `SCDispatcherFactory`, which uses reflection to analyze the `visit` methods and writes a temporary Java class implementing a runtime type inspection dispatcher (using the `instanceof` operator); `BCDispatcherFactory`, similar to `SCDispatcherFactory` but generates Java bytecode; and `ReflectiveDispatcherFactory`, that uses reflection to invoke the appropriate method, without generating any code. Dynamic Dispatcher provides the generalization of multi-method parameters by means of polymorphism.

Sprintabout is another double-dispatch alternative for Java, provided as a library [109]. Sprintabout uses a naming convention to identify multi-methods: any abstract method whose name ends with `Appropriate` can be considered as a multi-method. The different concrete implementations of the multi-method are implemented using method overload (named with the `multi-method` identifier, removing `Appropriate`). An instance of a multi-method is built calling the `createVisitor` method, which dynamically generates a dispatch object implementing a runtime type inspection dispatch (using the `GetType` approach discussed in Section 5.1.2). The dispatch object implements a cache to efficiently obtain the different method implementations at runtime, avoiding the use of reflection. The current implementation of Sprintabout does not permit built-in types as arguments.

MultiJava is a backward-compatible extension of Java that supports any dispatch dimension (not just double dispatch) [110]. Argument types of multi-method parameters are declared as `StaticType@DynamicType` to extend the single dynamic dispatching semantics of Java. The left-hand side of the type denotes the static type of the argument, whereas the right-hand side indicates its dynamic type used for the dynamic method selection. Given a set of multi-method implementations, the MultiJava compiler produces a single Java dispatch method containing the bodies of the set of multi-method implementations. The generated dispatch method implements the runtime type inspection approach described in this dissertation, using the `instanceof` Java operator (`is` operator in C#).

The *Java Multi-Method Framework* (JMMF) uses reflection to provide multiple dispatch for Java [111]. Multi-methods can be defined in any class and with any name. JMMF is provided as a library; it proposes neither language extensions nor virtual machine modifications. It implements a two-step multiple dispatch algorithm. The first step is multi-method creation, which performs a reflection-based analysis computing several data structures to be used upon multi-method invocation. The second step is multi-method execution, which invokes the appropriate method depending on the actual type of the arguments. If no such method exists, an exception is thrown.

PolyD is aimed at providing a flexible multiple dispatch technique for Java [112].

PolyD generates Java bytecodes dynamically, and allows the user to define customized dispatching policies (e.g., those analyzed in this Chapter 5). PolyD uses Java 1.5 annotations to identify the selected dispatch mechanism (`@DispatchingPolicy`). No restriction on the number of arguments, the type of the return value, or the use of primitive types is imposed. Three standard dispatching policies are available in PolyD: multiple dispatching (cached `GetType` runtime type inspection), overloading (static method overload) and a 'non-subsumptive' policy (only calls a method if the classes of the arguments match exactly those of the method parameters; i.e. no parameter generalization). Moreover, it is possible to define personalized dispatching policies using its API.

Chapter 3

Optimizing Dynamically Typed Operations with a Type Cache

Dynamically typed code has become popular in scenarios where high flexibility and adaptability are important issues. For this reason, there has been an increase in the use of dynamic languages in the last years [113]. Statically typed code also provides important benefits such as earlier type error detection and, usually, better runtime performance. Therefore, hybrid statically and dynamically typed languages are aimed at providing 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 is type checked at runtime [114]. The lack of compile-time type information involves fewer opportunities for compiler optimizations, and the extra run-time type checking commonly implies performance costs [70]. In addition, dynamically typed code for .NET and Java commonly employs the introspective services of the platforms, causing significant performance penalties [18]. The additional information kept around at runtime to enable type checking can also increase the memory resources required at runtime [113].

In this chapter, we propose a set of optimizations for the common dynamically typed operations of hybrid typing languages for the .NET platform using the Dynamic Language Runtime (DLR). We evaluate 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. Our system has been used to optimize 37 programs in 5 different languages, obtaining significant runtime performance improvements. We have also included the proposed optimizations in the implementation of the open source *StaDyn* compiler, obtaining similar results.

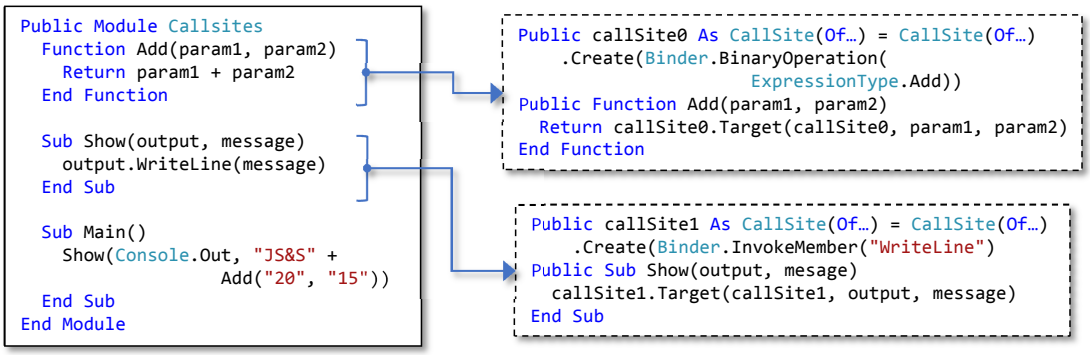


Figure 3.1: Example VB program with (right-hand side) and without (left-hand side) DLR optimizations.

3.1 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 [115]. 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 [115]. 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 [115].

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 3.1[1] shows two examples of dynamically typed operations executed with (right-hand side) and without (left-hand side) DLR `CallSites`.

Figure 3.1 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 invocation). 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 [15, 115]. The types of the operands are stored in a cache implemented by the `CallSite`.

[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.

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 [115].

Table 3.1 shows the list of dynamically typed expressions that can be represented with DLR `CallSites` [115]. 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 `CallSites` has been replaced by initializations in the declaration; and 3) arguments of `CallSiteBinders` have been omitted.

We previously measured that the runtime cache provided by the DLR provides a significant performance improvement compared to the use of introspection [18]. 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.

3.2 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 different approaches shown in Figure 3.2: as an optimizer of .NET executable files (Figure 3.2.a), and as part of an open source compiler (Figure 3.2.b).

Figure 3.2.a shows the binary optimization approach implemented for programs coded in VB, Boo, Cobra and Fantom. Using the Microsoft Research Common Compiler Infrastructure (CCI) [116], 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 `CallSites`. Finally, the ASTs are saved as new optimized binary files that use the DLR.

The proposed optimizations have also been included in the *StaDyn* compiler (Figure 3.2.b) –*StaDyn* was described in Section 2.1. We have modified its existing implementation [117]. The *StaDyn* compiler performs type inference with 5 traversals of the AST [1]. 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

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 3.1: Call-sites provided by the DLR (coded in C#).

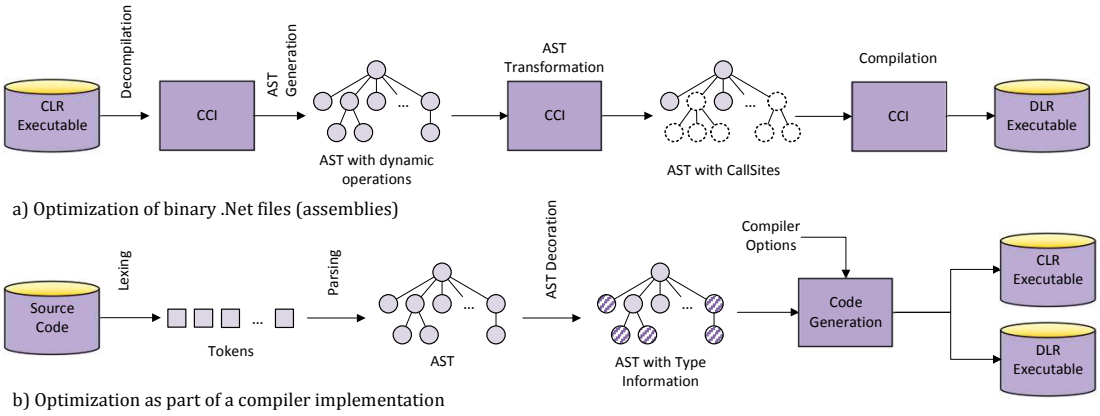


Figure 3.2: Architecture of the two optimization approaches.

typed references that the *StaDyn* compiler does not manage to infer: `dynamic` method arguments (3.2.5). 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 [27].

3.2.1 VB optimizations

In this section, we formalize the performance optimizations implemented for VB, which follow the .NET binary optimization approach presented in Figure 3.2.a. Sections 3.2.2, 3.2.3 and 3.2.4 detail the binary optimizations for Boo, Cobra and Fantom, respectively. Section 3.2.5 presents the optimizations included in the *StaDyn* compiler, following the architecture presented in Figure 3.2.b.

Figure 3.2 shows how every optimization is based on the idea of replacing an AST with another AST that uses the DLR services. Figures 3.3 to 3.6 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 3.1 with the code in the right-hand side. This AST transformation is denoted by $\rightsquigarrow$, so that $e_1 \rightsquigarrow e_2$ 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 3.2 (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 3.3); for the compiler approach, we obtain expression types from the annotated AST [1]. $C \times T_1 \times \ldots \times T_n \rightarrow T_r$ 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[1], and we use the *dynamic* type to

[1]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`.

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

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

Figure 3.3: Transformation of common expressions.

indicate that an expression is dynamically typed (although VB represent dynamic types by removing type annotations –as shown in Figure 3.1).

Figure 3.3 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 `CallSite` object, passing the two operands as arguments.

Figure 3.4 shows different optimizations when a type conversion is required, using the `Convert` binder provided by the DLR [115]. CCast describes explicit type conversion (casting) for built-in types. In VB, the `CType` function explicitly converts the type of an expression[1]. 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) [61].

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 [118]. Since the implementation of that instruction is so efficient, the DLR does not provide any optimization for non-built-in type conversions (Table 3.1). Therefore, the explicit conversions in CCast, CAssign and CFunction are only applied to built-in types.

[1] 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)

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

(CAssign)

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

(CFunction)

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

(CIf)

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

(CDWhile)

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

(CRWhile)

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

(CDUntil)

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

(CRUntil)

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

(CIndex)

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

Figure 3.4: Transformation of common type conversions.

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

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

Figure 3.5: Transformation of indexing operations.

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. Figure 3.4 also shows similar inference rules for typical *do-while* (CDWhile and CRWhile) and *repeat-until* (CDUntil and CRUntil) loops. Likewise, CIndex performs the same optimization for array indexing expressions, converting the index to `Integer`.

Figure 3.5 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 [61]). When the collection is dynamically typed, the `GetIndex` binder is used for reading operations and `SetIndex` for writing.

IOIMethod in Figure 3.6 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 [32]. The $\forall\, T_i^j \,.\, T_i^j \neq \mathit{dynamic}^{\,j \in 1 \ldots m}$ condition checks that at all the method implementations declare a statically typed i^{th} parameter. Otherwise, no optimization is done (the dynamically typed method overload is invoked). Unlike CFunction in Figure 3.4, the generated `InvokeMember` callsite receives the object and all the parameters to resolve method overloading at runtime [66].

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 3.6. In this case, the second parameter of the `Target` method is `Nothing`, indicating that there is no implicit object, since the method is `shared`.

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

(IOIMethod)

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

(IOCMethod)

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

(IMethod)

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

(LateGet)

$$\frac{\begin{array}{c} e : dynamic \qquad callsite_1 = \texttt{New CallSite(Binder.GetMember(}\omega\texttt{))} \\ callsite_2 = \texttt{New CallSite(Binder.InvokeMember(}\omega\texttt{))} \end{array}}{e\,.\,\omega \rightsquigarrow \texttt{LateGet\_Utils.HandleCallSiteCall(}e, \omega, callsite_1, callsite_2\texttt{)}}$$

(SetMember)

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

Figure 3.6: Transformation of method invocation and field access.

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 [119].

SetMember in Figure 3.6 optimizes the assignment of fields and properties, when the object is dynamically typed. A `SetMember` call-site binder is created for this purpose.

3.2.2 Boo optimizations

Figures 3.7 and 3.8 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. $\oplus_{\text{Boo}}$ and $\ominus_{\text{Boo}}$ represent, respectively, the optimized binary and unary Boo operators. We also transform explicit (CCast$_{\text{Boo}}$) and implicit (CAssign$_{\text{Boo}}$, CFunction$_{\text{Boo}}$ and CMethod$_{\text{Boo}}$) type conversions. CMethod$_{\text{Boo}}$ 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$_{\text{Boo}}$ and SetIndex$_{\text{Boo}}$ in Figure 3.8), method invocations (IMethod$_{\text{Boo}}$) and member accesses (GetMember$_{\text{Boo}}$ and SetMember$_{\text{Boo}}$). In Boo, we add two optimizations not implemented in VB. The first one, IDelegate$_{\text{Boo}}$, 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 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.

3.2.3 Cobra optimizations

Figure 3.9 shows the optimization rules for Cobra. This programming language does not support type conversions for dynamically typed expressions. As for Boo, IConstructor$_{\text{Cobra}}$ optimizes constructor invocation when one of the arguments is dynamically typed.

3.2.4 Fantom optimizations

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.

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

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

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

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

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

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

Figure 3.7: Optimization of Boo basic expressions and type conversions.

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

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

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

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

$$\frac{\exists\, e_i \,.\, e_i : \mathit{dynamic}^{\ i \in 1\ldots n} \qquad \mathit{callsite} = \texttt{CallSite(Binder.InvokeConstructor())}}{C\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, C, e_1, \ldots, e_n\texttt{)}} \ (\textsc{IConstructor}_{\textsc{Boo}})$$

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

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

Figure 3.8: Optimization of Boo invocations, indexing and member access.

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

$$\frac{e : \mathit{dynamic} \quad \ominus_{\text{Cobra}} \in \{\texttt{not}, \texttt{\~{}}\} \quad \mathit{callsite} = \texttt{CallSite(Binder.UnaryOperation(ExpressionType.}\ominus_{\text{Cobra}}\texttt{)}}{\ominus_{\text{Cobra}}\, e \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}} \quad (\textsc{UnaryOp}_{\text{Cobra}})$$

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

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

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

$$\frac{\exists\, e_i \,.\, e_i : \mathit{dynamic}^{\;i \in 1\ldots n} \quad \mathit{callsite} = \texttt{CallSite(Binder.InvokeConstructor())}}{C\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, C, e_1, \ldots, e_n\texttt{)}} \quad (\textsc{IConstructor}_{\text{Cobra}})$$

$$\frac{e : \mathit{dynamic} \quad \mathit{callsite} = \texttt{CallSite(Binder.GetMember(}f\texttt{))}}{e\,.\,f \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e\texttt{)}} \quad (\textsc{GetMember}_{\text{Cobra}})$$

$$\frac{e_1 : \mathit{dynamic} \quad \mathit{callsite} = \texttt{CallSite(Binder.SetMember(}f\texttt{))}}{e_1\,.\,f\texttt{ = }e_2 \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, e_1, e_2\texttt{)}} \quad (\textsc{SetMember}_{\text{Cobra}})$$

Figure 3.9: Cobra optimization rules.

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

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 3.10: Fantom optimization rules.

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 3.10).

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 [66]. 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.

3.2.5 *StaDyn* optimizations

Figure 3.11 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 [1].

3.3 Implementation

3.3.1 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 [116]. Figure 3.12

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

$$\frac{\exists\, e_i \,.\, e_i : \mathit{dynamic}^{\;i \in 1\ldots n} \qquad \mathit{callsite} = \texttt{new CallSite(Binder.InvokeConstructor())}}{\texttt{new } C\texttt{(}e_1, \ldots, e_n\texttt{)} \rightsquigarrow \mathit{callsite}\texttt{.Target(}\mathit{callsite}, C, e_1, \ldots, e_n\texttt{)}} \quad (\text{IConstructor}_{StaDyn})$$

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

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

Figure 3.11: *StaDyn* optimization rules.

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 [43]. 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., `VBCodeVisitor` and `BooCodeVisitor`) that identifies the expressions to be optimized, following the specific language optimization rules described in this dissertation. For each expression, the corresponding call-site pattern is stored in a `CallSiteContainer` object. Second, the code that instantiates the `CallSites` is generated. As shown in Figure 3.1 and Table 3.1, 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 [120]. 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.

3.3.2 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 per-

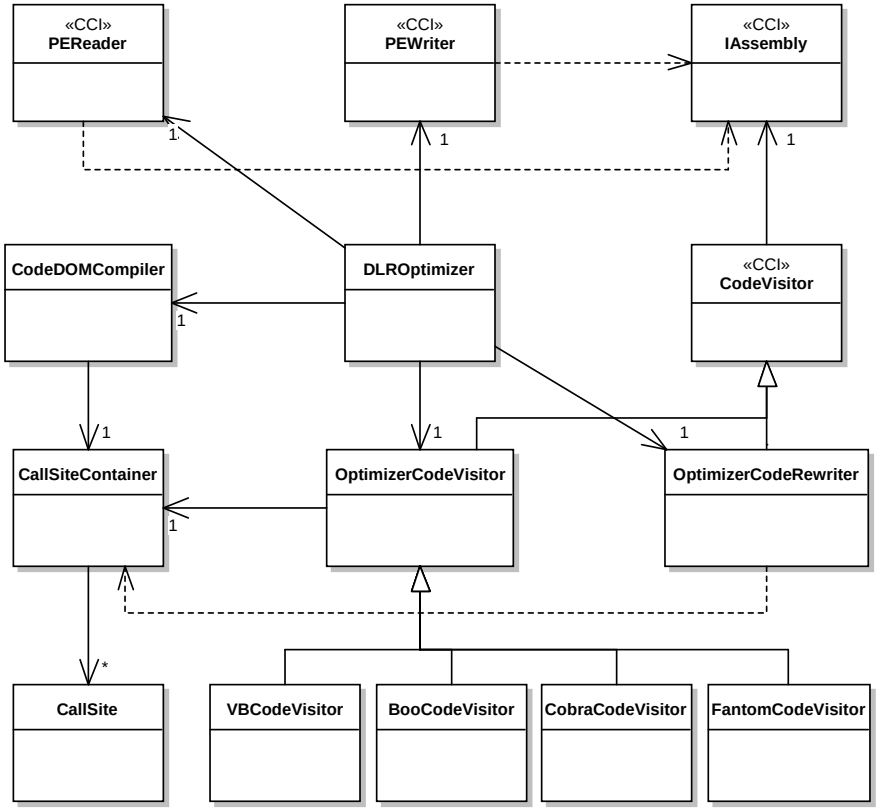


Figure 3.12: Class diagram of the binary program transformation tool.

forms type inference in 5 phases [1]. Code generation is performed afterwards, traversing the type-annotated AST and following the Visitor design pattern [43]. Originally, the existing code generator produced .Net assemblies for the CLR (Figure 3.2.b). We have added code generation for the DLR using the Parallel Hierarchies design pattern [24]. 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 3.2.5.

3.4 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.

3.4.1 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. Many elements of the methodology described here will be used for evaluating the optimizations presented in the two following chapters.

3.4.1.1 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 [61]. 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 [60]. Boo has been used to create views in the Brail view engine of the MonoRail Web framework [121], to program the Specter object-behavior specification framework [122], in the implementation of the Binsor domain-specific language for the Windsor Inversion of Control container for .NET [123], and in the development of games and mobile apps with Unity [124].
- Cobra 0.9.6. A hybrid statically and dynamically typed programming language. It is object-oriented and provides compile-time type inference [67]. 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 [67].
- 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 [68]. 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 [125], the SkySpark analytics software [126], and the netColarDB object-relational mapping database [127].
- *StaDyn*. The hybrid static and dynamic typing object-oriented language for .NET described in Section 2.1.

3.4.1.2 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 [128]. 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 [129], 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# [130], 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 [27]. 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 [119]. All the references in the programs have been declared as dynamically typed.

3.4.1.3 Data analysis

We have followed the methodology proposed in [131] 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 $p = 30$ [132]. Therefore, we compute the confidence interval $[c_1, c_2]$ as:

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

   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 [131].

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:

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

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.

3.4.1.4 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 [133]. 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 [134].

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 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 1.5 times) higher than P_2, P_1 is 150% (or 1.5 times) faster, P_2 requires 150% (or 1.5 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.

All the data discussed in the following subsections are detailed in Appendix A.

3.4.2 Start-up performance

Figures 3.13 and 3.14 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 3.13) to examine how the optimizations introduced may improve the runtime performance of each language feature. Afterwards, we analyze more realistic applications in Figure 3.14.

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.

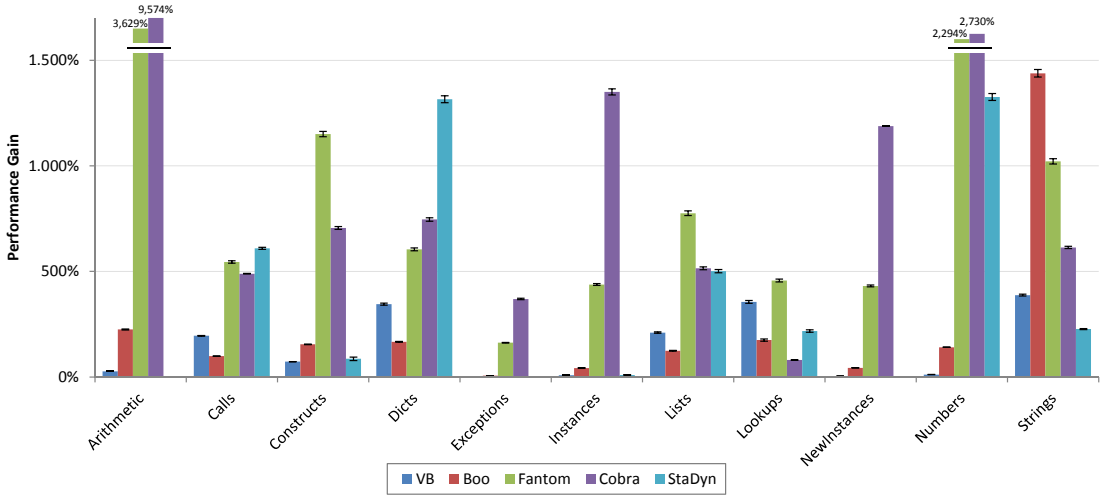


Figure 3.13: Start-up performance improvement for Pybench.

Figure 3.14 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. For Cobra, *StaDyn*, VB and Boo, the average performance improvements are 406%, 120.5%, 87.4% and 44.6%, respectively.

3.4.2.1 Discussion

Analyzing the previous start-up performances, we can identify different discussions. Considering the different kind of operations in Figure 3.13, 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 [18]. 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 3.13). Both languages already support an optimization based on nested dynamic type inspections, avoiding the use of reflection [1] –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

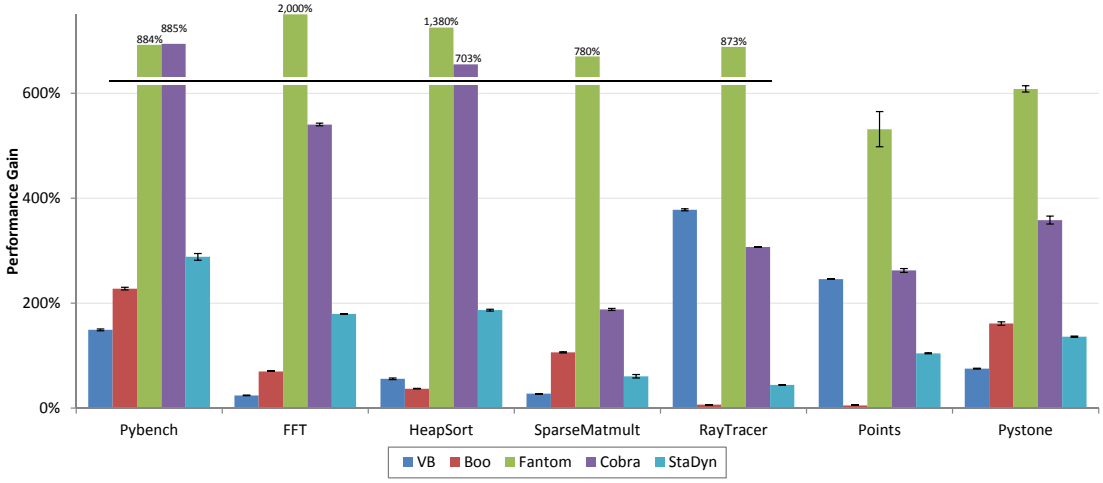


Figure 3.14: Start-up performance improvement.

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 3.14), 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 [115], 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.

3.4.3 Steady-State performance

We have executed the same programs following the steady-state methodology described in Section 3.4.1.3. Figure 3.15 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 1,113%, for Cobra. We speed up Boo, *StaDyn* and Fantom in 322%, 368% and 1,083%, respectively.

3.4.3.1 Discussion

Table 3.2 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

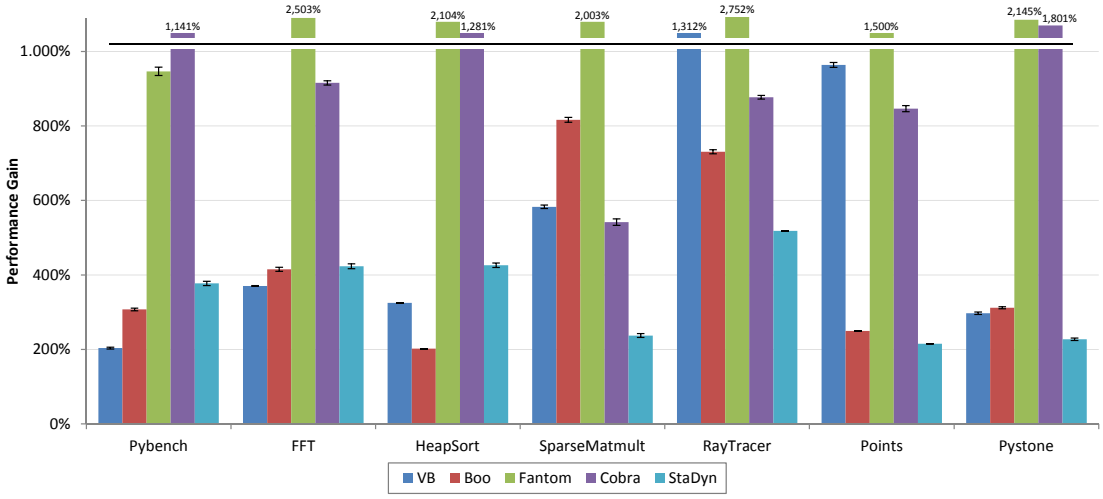


Figure 3.15: Steady-state performance improvement.

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 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 3.2 shows how Fantom is the language with the smallest steady-state performance gain compared to the start-up one. The average steady-state benefit (1,897%) 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 3.2.4). Since the DLR creates a different cache for each type of call-site [115], 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 3.2). 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.

3.4.4 Memory consumption

Figure 3.16 (and Table 3.3) 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

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

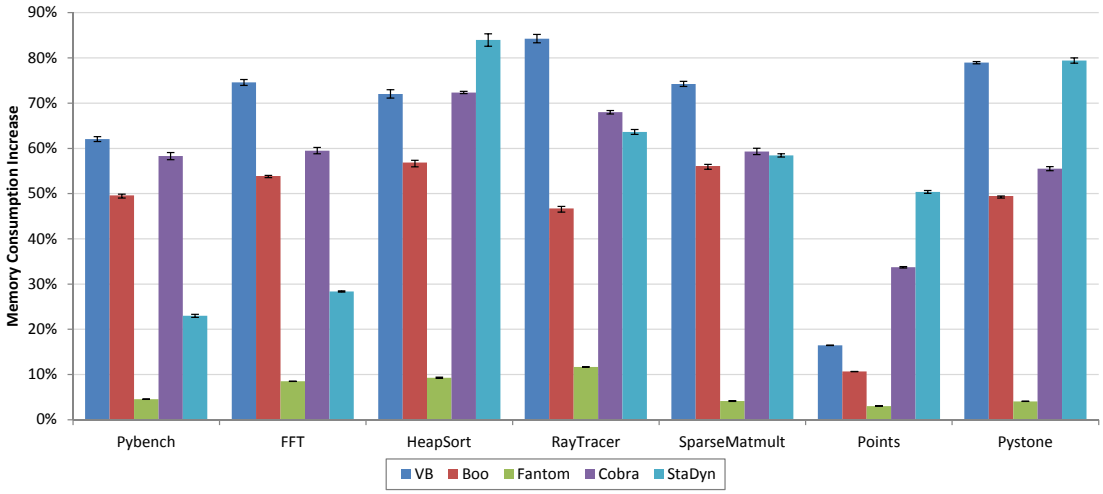


Figure 3.16: Memory consumption increase.

than the original applications.

3.4.4.1 Discussion

We compare the memory consumption increase caused by the DLR (Figure 3.16) with the corresponding performance gains (Figures 3.14 and 3.15). 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 3.3 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 [115], 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.

	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 3.3: Memory consumption expressed in MBs.

Analyzing the applications in Figure 3.16, the Points program shows the lowest average memory increase. This application also presents the smallest average start-up and steady-state performance gains (Sections 3.4.2 and 3.4.3). 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.

Chapter 4

Optimizations based on the SSA form

Most dynamic languages allow variables to have different types in the same scope. Figure 4.1 shows an example C# program where the dynamically typed variable `number` has different types in its scope [17]. First, a `string` is assigned to `number` (line 2), representing a real number in the scientific format; then, the `string` is converted into a `double` (line 3); and it is finally converted into a floating-point format `string` with the appropriate number of decimals (line 5).

Unlike dynamic languages, most statically typed languages force a variable to have the same type within its scope. Even languages with static-type inference (type reconstruction) such as ML [135] and Haskell [136] do not permit the assignment of different types to the same reference in the same scope. This also happens in the Java and .NET platforms. At the virtual machine level, assembly variables should be defined with a single type in their scope. If we want a variable to hold different types (as dynamic languages do), the general `Object` type should be used.

Different issues appear when dynamically typed variables are declared as `object` (Figure 4.2 shows an example). The compiler must generate conversion operations (casts) to change the type of the expression (from `object` to the expected type) [54]. Lines 3, 4 and 5 in Figure 4.2, show how casts should be added when the `number` variable is used. If the casts are not added, the expressions cannot be executed by the virtual machine. For instance, the `ToString` message passed to `number` in line 5 cannot be invoked without the cast, because `Object` does not provide a `ToString` method receiving the `string` format as a parameter.

```
01: Console.Write("Scientific format: ");
02: dynamic number = Console.ReadLine();
03: number = Double.Parse(number);
04: int decimalDigits = NumberOfDecimalDigits(number);
05: number = number.ToString("F" + decimalDigits);
06: Console.WriteLine("Fixed point format: {0}.", number);
```

Figure 4.1: The dynamically typed reference `number` holds different types in the same scope.

```
01: Console.Write("Scientific format: ");
02: object number = Console.ReadLine();
03: number = Double.Parse((string)number);
04: int decimalDigits = NumberOfDecimalDigits((double)number);
05: number = ((double)number).ToString("F" + decimalDigits);
06: Console.WriteLine("Fixed point format: {0}.", number);
```

Figure 4.2: Type conversions must be added when `number` is declared as `object`.

```
01: Console.Write("Scientific format: ");
    // number₀ is inferred to string
02: dynamic number₀ = Console.ReadLine();
    // number₁ is inferred to double
03: dynamic number₁ = Double.Parse(number₀);
04: int decimalDigits = NumberOfDecimalDigits(number₁);
    // number₂ is inferred to string
05: dynamic number₂ = number₁.ToString("F" + decimalDigits);
06: Console.WriteLine("Fixed point format: {0}.", number₂);
```

Figure 4.3: A SSA transformation of the code in Figure 4.1.

Another issue with `object` references is that the type conversions imply an important runtime performance penalty [137]. A cast operation checks the dynamic type of an expression, analyzing whether the runtime conversion is feasible. This runtime type inspection consumes significant execution time in both the Java [15] and .NET [66] platforms. Sometimes, the compiler does not infer the type of a reference (e.g., a `dynamic` parameter). In these cases, reflection is used and the performance penalty is even higher [15, 138, 139].

Therefore, we propose an alternative approach to compile dynamically typed local references, avoiding the performance cost of casts and reflection. Programs are transformed so that each dynamically typed reference is statically assigned at most once, as shown in Figure 4.3. Three different `number` variables with three different types are declared. The Abstract Syntax Tree (AST) is modified and passed to the type inference phase of an existing compiler, before generating binary code for the .NET platform. The generated code avoids type casts and reflective calls, providing better runtime performance (Section 4.3). The AST transformations proposed are a modification of the classical Static Single Assignment (SSA) transformation used for compiler optimizations [84].

Figure 4.3 shows a simple case, where the execution flow is sequential. However, the transformation into SSA form must also consider conditional and iterative control flow structures, where dynamically typed variables may have different types depending on the execution flow [22] –detailed in Section 4.2.

In this chapter, we use SSA transformations to efficiently support variables with different types in the same scope, as dynamic languages do. These transformations have been include in the *StaDyn* programming language implementation, which generates code for the .NET framework. Similar to the Java platform, .NET does not allow one variable to have different types in the same scope (`object` must be used). The code generated by our compiler performs significantly better than the use of `dynamic` in C#, avoiding unnecessary type conversions and reflective invocations.

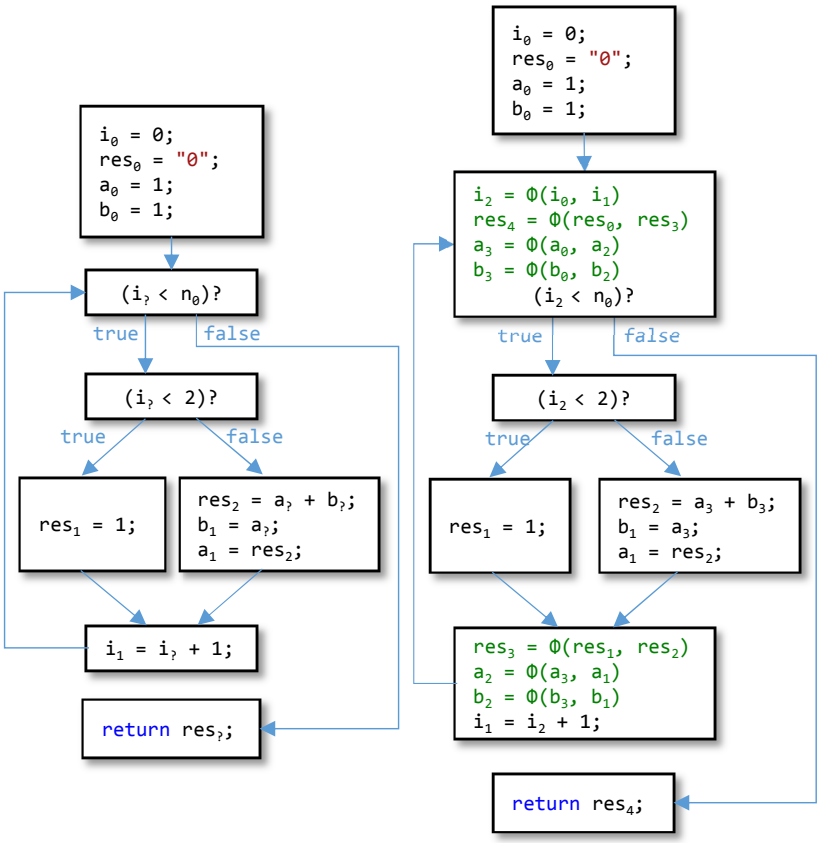


Figure 4.4: CFG for the code in Figure 4.5 (left) and its SSA form (right).

4.1 SSA form

A program is in SSA form if every variable is statically assigned at most once [84]. The SSA form is used in modern compilers to facilitate code analysis and optimizations. Examples of such optimizations are elimination of partial redundancies [82], constant propagation [83] and increase of parallelism in imperative programs [140].

In code within a basic block (a straight-line code sequence with no branches), the transformation into SSA form is quite simple. First, a new variable is created when an expression is assigned to it. The code in Figure 4.1 is transformed to the one in Figure 4.3, creating new $\texttt{number}_\texttt{i}$ variables in lines 2, 3 and 5. Second, the use of each variable is renamed to the last "version" of that variable. For instance, the use of `number` in line 4 of Figure 4.1 is replaced with $\texttt{number}_1$ in Figure 4.3.

This simple algorithm cannot be applied to code with branches. Conditional and loop statements define different execution flow paths, making it more difficult to decide which variable version must be used. This is shown in the left-hand side of Figure 4.4, the Control Flow Graph (CFG) of the program in Figure 4.5[1].

[1]The aim of assigning an initial `"0"` string value to `res` is to later explain flow-sensitive types: how a variable may have two different types (`string` and `int`) depending on the execution flow.

```
01: dynamic Fibonacci(dynamic n) {
02:   dynamic i = 0, res = "0";
03:   dynamic a = 1, b = 1;
04:   while (i < n) {
05:     if(i < 2)
06:       res = 1;
07:     else {
08:       res = a + b;
09:       b = a;
10:       a = res;
11:     }
12:     i = i + 1;
13:   }
14:   return res;
15: }
```

Figure 4.5: An iterative Fibonacci function using dynamically typed variables.

Variables `i`, `a` and `b` have different versions, and their use in some expressions depends on the execution flow (represented as $\texttt{i}_?$, $\texttt{a}_?$ and $\texttt{b}_?$ in the left-hand side of Figure 4.4). For instance, the use of the `i` variable in the `while` condition may be referring to the initial $\texttt{i}_0$ variable or to $\texttt{i}_1$ defined at the end of the loop.

To solve this problem, the CFG of the program (left-hand side of Figure 4.4) is processed, inserting invocations to a fictitious Φ-function at the beginning of join nodes (right-hand side of Figure 4.4). The assignment $\texttt{i}_2=\Phi(\texttt{i}_0,\texttt{i}_1)$ generates a new definition for $\texttt{i}_2$ by choosing either $\texttt{i}_0$ or $\texttt{i}_1$ depending on the execution path taken (control comes from the first basic block or the loop, respectively)[1]. The following accesses of the `i` variable will use the $\texttt{i}_2$ version (in the `while` and `if` conditions), until a new assignment is done (the last line in the loop).

4.2 SSA form to allow multiple types in the same scope

As mentioned, the SSA form is commonly used to optimize programs performing intermediate code transformations. In this dissertation, we adapt the SSA transformations to allow `dynamic` variables to have multiple types in the same scope. The SSA form facilitates the inference of a single type for each variable version, representing flow sensitive types with union types [33]. We first describe the SSA transformation for basic blocks (Section 4.2.1), and then for conditional (Section 4.2.2) and iterative (Section 4.2.3) statements. Flow sensitive types are discussed in Section 4.2.4 and Section 4.2.5 describes the implementation.

[1]The Φ-function is a notational fiction used for type inference purposes (Section 4.2.4). To implement such a function that knows which execution path is taken at runtime, we add additional *move* statements in the transformed program (Section 4.2.2).

4.2.1 Basic blocks

A basic block is a straight-line code sequence with no branches. Statements in a basic block are executed sequentially, following a unique execution path in the CFG. Since no jump occurs in a basic block, no Φ-function is needed in its SSA form.

Figure 4.6 shows the algorithm proposed to transform a sequence of statements into its SSA form. The first parameter is the sequence of statements to be transformed (its Abstract Syntax Tree, AST). The second parameter holds the last version of each variable: a map that associates to each `dynamic` local variable in the statements (including the function parameters) one integer representing its last version number. The transformed AST and one map with the new variable versions are returned.

The meta-variables *var* range over variables; i, j, k and n range over integers; and *exp* and *stmt* range over expressions and statements, respectively. Maps (association lists) are represented as $\{key_1 \mapsto content_1, \ldots, key_n \mapsto content_n\}$, where n is the number of pairs in the map. The empty map is represented through $\{\}$. If m is one map, then $m[var \mapsto i]$ is a new map identical to m except that *var* is overridden/added by i. The $m[var]$ expression represents the lookup of the value associated with the *var* key. Fixed-length font code (e.g., `while` ***exp*** ***block***) represent ASTs. The term $[stmt \mapsto stmt_{out}]$***block*** denotes the AST obtained by replacing all the occurrences of *stmt* in ***block*** by $stmt_{out}$.

SSA_{stmts} in Figure 4.6 calls SSA_{if} (Section 4.2.2) or SSA_{while} (Section 4.2.3) when, respectively, an `if-else` or `while` statement is analyzed. Otherwise, the SSA_{stmt} function is called for the rest of statements. The transformed statement ($stmt_{out}$) replaces the previous statement (*stmt*) in the returned AST ($block_{out}$).

Figure 4.7 shows the SSA transformation of any statement but `if-else` and `while`. If a `dynamic` variable is defined, the 0 version of that variable is added to the output map. In the returned SSA form, the variable is replaced with its 0 version. If the declaration has an initialization expression, that expression must also be transformed. An example is shown in line 2 of Figure 4.1, where `number` declaration is replaced by $\texttt{number}_0$.

The statement in SSA_{stmt} may not be a variable definition (Figure 4.7). In that case, all the expressions in the statement are transformed by SSA_{exp}. Then, all the occurrences of the original expression (*exp*) in the statement ($stmt_{out}$) are replaced by the transformed ones (exp_{out}). This is the case of line 3 in Figure 4.1, replacing the argument `number` by $\texttt{number}_0$.

SSA_{exp} in Figure 4.8 transforms the expressions. When the expression is an assignment and the left-hand side is a `dynamic` variable (it was included in map_{in} by SSA_{stmt}), the variable is replaced with a new version ($i + 1$). The right-hand side of the assignment is replaced with its SSA form (exp_{out}). For example, in line 5 of Figure 4.1 `number` on the right is replaced with $\texttt{number}_1$, and a new version is set to the variable on the left ($\texttt{number}_2$). For the rest of expressions, SSA_{exp} replaces variables with their last version.

$\mathbf{SSA_{stmts}}(block_{in}, map_{in}) \rightarrow block, map$
  $map_{out} \leftarrow map_{in}$
  $block_{out} \leftarrow block_{in}$
  for all *stmt* **in** $block_{out}$ **do**
    if *stmt* **is** `if` *exp* `block` $_{true}$ (`else` `block` $_{false}$)$^?$ **then**
      $stmt_{out}, map_{out} \leftarrow SSA_{if}(exp, block_{true}, block_{false}, map_{out})$
    else if *stmt* **is** `while` *exp* `block` **then**
      $stmt_{out}, map_{out} \leftarrow SSA_{while}(exp, block, map_{out})$
    else
      $stmt_{out}, map_{out} \leftarrow SSA_{stmt}(stmt, map_{out})$
    end if
    $block_{out} \leftarrow [stmt \mapsto stmt_{out}]block_{out}$
  end for
  return $block_{out}, map_{out}$
end

Figure 4.6: SSA transformation of a sequence of statements.

$\mathbf{SSA_{stmt}}(stmt_{in}, map_{in}) \rightarrow stmt, map$
  if *stmt* **is** `dynamic` *var*(`=`*exp*)$^?$ **then**
    $exp_{out}, map_{out} \leftarrow SSA_{exp}(exp, map_{in})$
    $map_{out} \leftarrow map_{out}[var \mapsto 0]$
    return `dynamic` *var*$_0$(`=`*exp* $_{out}$)$^?, map_{out}$
  else
    $map_{out} \leftarrow map_{in}$
    $stmt_{out} \leftarrow stmt_{in}$
    for all *exp* **in** $stmt_{in}$ **do**
      $exp_{out}, map_{out} \leftarrow SSA_{exp}(exp, map_{out})$
      $map_{out} \leftarrow [exp \mapsto exp_{out}]stmt_{out}$
    end for
    return $stmt_{out}, map_{out}$
  end if
end

Figure 4.7: SSA transformation of statements.

$\mathbf{SSA_{exp}}(exp_{in}, map_{in}) \rightarrow exp, map$
  if exp_{in} **is** *var*`=`*exp* $_1$ **and** $map_{in} = \{\ldots, var \mapsto j, \ldots\}$ **then**
    $exp_{out}, map_{out} \leftarrow SSA_{exp}(exp_1, map_{in})$
    $i \leftarrow map_{out}[var]$
    return *var* $_{i+1}$`=`*exp* $_{out}, map_{out}[var \mapsto i+1]$
  else
    $exp_{out} \leftarrow exp_{in}$
    for all $var \mapsto i$ **in** map_{in} **do**
      $exp_{out} \leftarrow [var \mapsto var_i]exp_{in}$
    end for
    return exp_{out}, map_{in}
  end if
end

Figure 4.8: SSA transformation of expressions.

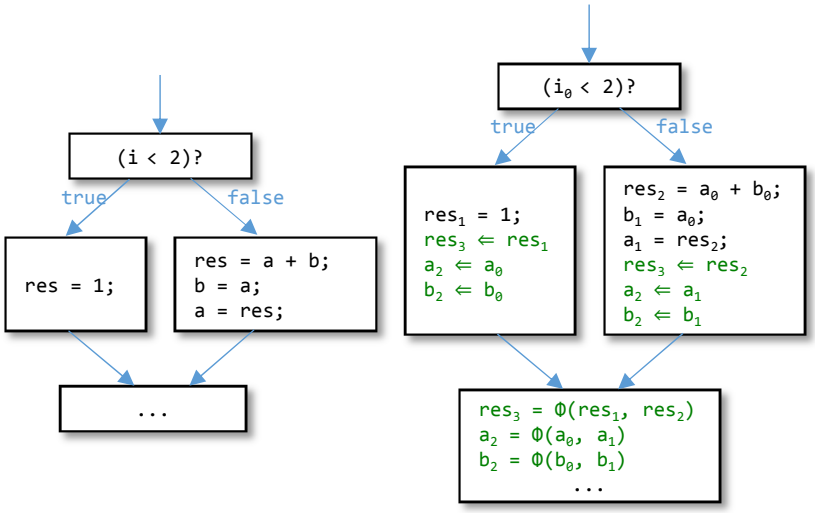


Figure 4.9: Original CFG of an `if-else` statement (left) and its SSA form (right).

4.2.2 Conditionals statements

As discussed in Section 4.1, conditional statements define different execution paths. The fictitious Φ-function knows which execution path is taken. The Φ-function may be implemented by adding *move* statements to each incoming edge [85], as shown in the right of Figure 4.9[1]. The $\texttt{res}_3 \Leftarrow \texttt{res}_1$ *move* statement means that the value of $\texttt{res}_1$ is stored in $\texttt{res}_3$. Then, if the `if` or the `else` block is executed, the respective $\texttt{res}_3 \Leftarrow \texttt{res}_1$ or $\texttt{res}_3 \Leftarrow \texttt{res}_2$ statement will be executed, making $\texttt{res}_3 = \Phi(\texttt{res}_1, \texttt{res}_2)$ assign the appropriate value to $\texttt{res}_3$.

Figure 4.10 shows the algorithm of the `if-else` statement transformation. The condition and `if` and `else` blocks are transformed. $block_3$ represents the join block in Figure 4.9, which is initialized to an empty list ([ ]). All the `dynamic` variables used in the statement (in map_{in}) are analyzed. If the variable version after the condition (i) is different to the one after the `if-else` statement (k), then that variable is assigned in the `if` or the `else` block. If so, the variable may be assigned in the `if` ($j = k$) or in the `else` block (otherwise).

When the variable is assigned in the `else` body, its version is incremented, and a new Φ-function is added to the join block ($block_3$) –*list* :: *stmt* represents a new list where *stmt* has been appended to *list*. The new version of the variable ($k + 1$) is computed from the one in the `if` (j) and the `else` (k) blocks using a Φ-function. In the example in Figure 4.9, $\texttt{res}_3 = \Phi(\texttt{res}_1, \texttt{res}_2)$ is added to the join block, since `res` is assigned in both `if` and `else` blocks. For the same reason, two move statements ($\texttt{res}_3 \Leftarrow \texttt{res}_1$ and $\texttt{res}_3 \Leftarrow \texttt{res}_2$) are added at the end of the `if` ($block_1$) and `else` ($block_2$) blocks. The case when one variable is only assigned in the `if` block (the case where $i = j$ and $j \neq k$ in Figure 4.10) is quite similar –there is no example of this case in Figure 4.9.

The SSA_{if} algorithm returns the transformed AST of the `if-else` statement and the variable versions in map_3. A new join block ($block_3$) is added after

[1]For the sake of brevity, the initialization of $\texttt{i}_0$, $\texttt{res}_0$, $\texttt{a}_0$ and $\texttt{b}_0$ is not shown.

SSA$_{\textbf{if}}$ $(exp_{cond}, block_{true}, block_{false}, map_{in}) \rightarrow stmt, map$
  $exp_{out}, map_1 \leftarrow SSA_{exp}(exp_{cond}, map_{in})$
  $block_1, map_2 \leftarrow SSA_{stmts}(block_{true}, map_1)$
  $block_2, map_3 \leftarrow SSA_{stmts}(block_{false}, map_2)$
  $block_3 \leftarrow [\,]$
  for all var **in** map_{in} **do**
    $i \leftarrow map_1[var]$
    $j \leftarrow map_2[var]$
    $k \leftarrow map_3[var]$
    if $i \neq k$ **then** // variable assigned in `if` or `else` block
      if $j = k$ **then** // variable assigned in `if` block but not in `else`
        $map_3 \leftarrow map_3[var \mapsto k+1]$
        $block_3 \leftarrow block_3$ :: var_{k+1}=Φ(var_j,var_i)
        $block_1 \leftarrow block_1$ :: $var_{k+1} \Leftarrow var_j$
        $block_2 \leftarrow block_2$ :: $var_{k+1} \Leftarrow var_i$
      else// variable assigned in `else` block
        $map_3 \leftarrow map_3[var \mapsto k+1]$
        $block_3 \leftarrow block_3$ :: var_{k+1}=Φ(var_j,var_k)
        $block_1 \leftarrow block_1$ :: $var_{k+1} \Leftarrow var_j$
        $block_2 \leftarrow block_2$ :: $var_{k+1} \Leftarrow var_k$
      end if
    end if
  end for
  return `if(`exp_{out}`)`$block_1$ `else` $block_2$`;`$block_3$, map_3
end

Figure 4.10: SSA transformation of `if-else` statements.

the transformed AST, holding the necessary Φ-function statements added by the SSA_{if} algorithm (bottom right block in Figure 4.9).

4.2.3 Loop statements

We define the SSA transformation for `while` statements –other loops follow a similar approach [141]. Figure 4.11 shows how the CFG has a join block at the beginning of the `while` statement. This block can be reached from the previous block outside the `while` loop (the first block in Figure 4.11), and from the block of the `while` body. Since there are two edges pointing to the block, Φ-functions must be added before the condition (right-hand side of Figure 4.11). Similarly, *move* statements must be placed at the end of the two blocks preceding the condition block (the first block and the `while` body in Figure 4.11).

Figure 4.12 details the algorithm for `while` statements. The condition and body blocks are first transformed into their SSA form. Then, all the `dynamic` variables are analyzed. If there is an assignment in the condition or the `while` body ($i \neq k$), a new version is created to hold the new value. When the variable is assigned in the `while` body ($i = j$), a Φ-function is added at the beginning of the condition block ($block_{cond_1}$). An example is the $\texttt{i}_2\texttt{=}\Phi\texttt{(i}_0\texttt{,i}_1\texttt{)}$ statement shown in Figure 4.11.

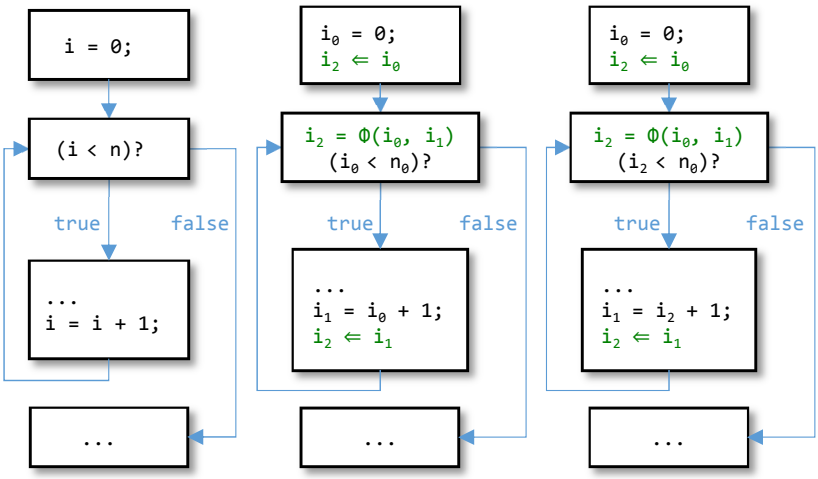


Figure 4.11: Original CFG of a `while` statement (left), and intermediate SSA representation (middle) and its final SSA form (right).

For the special case that one `dynamic` variable is assigned in the `while` condition, a Φ-function with 3 parameters is used: one for the version outside the loop (var_i); another one for the new version in the condition (var_j); and the last one, in case the variable is also assigned in the body block (var_k).

As mentioned, *move* statements must be placed at the end of the blocks preceding the Φ-functions. Therefore, if a variable is assigned in the condition or the body, one *move* is added at the end of the block before the `while` statement ($block_{before}$), and another one at the end of the `while` body ($block_{body}$)[1]. In Figure 4.11, these two *move* statements are $\texttt{i}_2 \Leftarrow \texttt{i}_0$ and $\texttt{i}_2 \Leftarrow \texttt{i}_1$, respectively.

We have seen how Φ-functions are added before the loop condition. In the middle of Figure 4.11, the new $\texttt{i}_2{=}\Phi(\texttt{i}_0,\texttt{i}_1)$ statement sets to $\texttt{i}_2$ the appropriate value of the `i` variable. Therefore, the subsequent uses of `i` in the condition and body must be replaced with $\texttt{i}_2$. However, the existing CFG (in the middle of Figure 4.11) uses $\texttt{i}_0$, whereas $\texttt{i}_2$ must be used instead (right-hand side of Figure 4.11). This behavior is defined in the two last assignments of the algorithm in Figure 4.12. In the AST of the condition (exp_{cond}), the original variable version (var_i) is substituted by the new one (var_{k+1}). The same substitution is applied to the `while` body ($block_{body}$).

4.2.4 Union types

Figure 4.13 shows the architecture of the *StaDyn* compiler. After the SSA transformation, a type inference phase annotates the AST with types. In *StaDyn*, one `dynamic` variable may have different types in the same scope. The SSA phase creates a new variable version for each assignment. Therefore, in basic blocks a different type is inferred for each `dynamic` variable. This is the case of the `i`, `a`

[1]For assignments in the condition, an additional *move* is required at the end of the condition block ($block_{cond_2}$).

$\mathbf{SSA_{while}}(exp_{cond}, block_{in}, map_{in}) \rightarrow stmt, map$
  $exp_{cond}, map_1 \leftarrow SSA_{exp}(exp_{in}, map_{in})$
  $block_{body}, map_2 \leftarrow SSA_{stmts}(block_{in}, map_1)$
  $block_{before} \leftarrow [\,]$
  $block_{cond_1} \leftarrow [\,]$
  $block_{cond_2} \leftarrow [\,]$
  for all $var \mapsto i$ **in** map_{in} **do**
    $j \leftarrow map_1[var]$
    $k \leftarrow map_2[var]$
    if $i \neq k$ **then** // variable assigned in the condition or the body
      $map_2 \leftarrow map_2[var \mapsto k+1]$
      if $i = j$ **then** // variable assigned in the body
        $block_{cond_1} \leftarrow block_{cond_1} :: \texttt{var}_{k+1}\texttt{=}\Phi\texttt{(var}_i\texttt{,var}_k\texttt{)}$
      else // variable assigned in the condition
        $block_{cond_1} \leftarrow block_{cond_1} :: \texttt{var}_{k+1}\texttt{=}\Phi\texttt{(var}_i\texttt{,var}_j\texttt{,var}_k\texttt{)}$
        $block_{cond_2} \leftarrow block_{cond_2} :: \texttt{var}_{k+1} \Leftarrow \texttt{var}_j$
      end if
      $block_{before} \leftarrow block_{before} :: \texttt{var}_{k+1} \Leftarrow \texttt{var}_i$
      $block_{body} \leftarrow block_{body} :: \texttt{var}_{k+1} \Leftarrow \texttt{var}_k$
      // the initial version is replaced with the LHS of the Φ-function
      $exp_{cond} \leftarrow [var_i \mapsto var_{k+1}]exp_{cond}$
      $block_{body} \leftarrow [var_i \mapsto var_{k+1}]block_{body}$
    end if
  end for
  return $\texttt{block}_{before}\texttt{;while(block}_{cond_1}\texttt{;exp}_{cond}\texttt{;block}_{cond_2}\texttt{)block}_{body}, map_2$
end

Figure 4.12: SSA transformation of `while` statements.

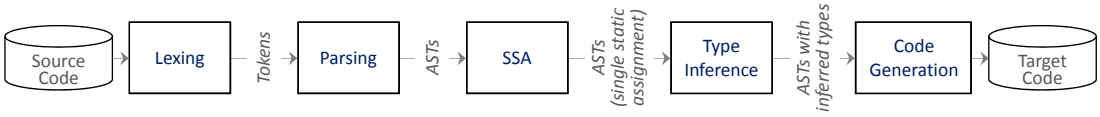


Figure 4.13: A simplification of the *StaDyn* compiler architecture [1].

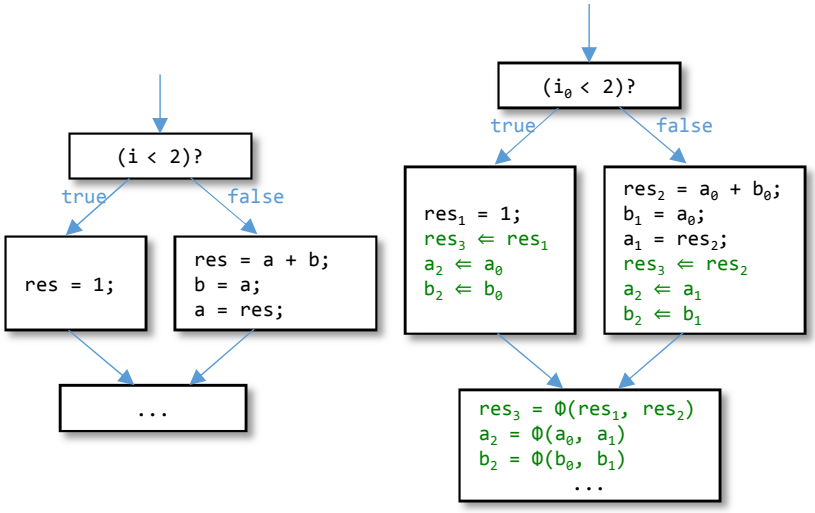


Figure 4.14: Type inference of the SSA form.

and `b` variables in Figure 4.14. This figure shows an extension of the SSA form with type annotations.

However, in conditional and loop statements, `dynamic` variables may have different types depending on the execution flow. In the example in Figure 4.14, if the first evaluation of the condition in the `while` loop is false, the returned value is `string`; otherwise, the result is an integer number. To represent these context sensitive types, we use union types [27].

A union type $T_1 \vee T_2$ denotes the ordinary union of the set of values belonging to T_1 and the set of values belonging to T_2 [142], representing the least upper bound of T_1 and T_2 [143]. A union type holds all the possible types a variable may have. The operations that can be applied to a union type are those accepted by every type in the union type. For instance, since $\texttt{res}_4$ in Figure 4.14 has the $\texttt{string} \vee \texttt{int}$ type, the + operator may be applied to it, but no the division [27].

The way union types are inferred is straightforward thanks to the Φ-function. Anytime a $var_1 = \Phi(var_2, var_3)$ statement is analyzed, the type of var_1 is inferred to a union type collecting the types of var_2 and var_3. In Figure 4.14, the type of $\texttt{res}_4$ is $\texttt{string} \vee \texttt{int}$, since the types of $\texttt{res}_0$ and $\texttt{res}_3$ are `string` and `int`, respectively –notice that $T \vee T = T$.

4.2.5 Implementation

The SSA transformations proposed in this article have been included in the *StaDyn* compiler [144] following the *Visitor* design pattern [43]. Each `visit`

method of the `SSAVisitor` class traverses one type of node in the AST, following the algorithms described in the previous subsections. The `visit` methods return the SSA form of the traversed AST node.

The unique parameter of the `visit` method is an instance of the `SSAMap` class, which provides the different services of the map abstraction used in our algorithms. The parameter is modified inside the `visit` method, so we clone it before each invocation to save its original state.

Variables in the AST were added an integer field representing its version. The `SSAVisitor` class modifies these versions accordingly to the proposed algorithms. In the code generation phase, a different variable is generated for each different variable version. We generate a `var__n` variable for var_n. In the generated code, most variables are declared with one single type because of the SSA transformation. Only those variables inferred as union types are declared as `object` and optimized with nested type inspections [1, 66].

A new `PhiStatement` was added to the AST. Its only purpose is to infer union types (Section 4.2.4) in the type inference phase. No code is generated for the `PhiStatement`. We also added a `MoveStatement`, which is translated into an assignment statement.

4.3 Evaluation

We evaluate the runtime performance benefit of using the SSA form to efficiently support variables with different types in the same scope. We measure the execution time and memory consumption of *StaDyn* programs compiled with and without the SSA phase, and compare it to C#. We also measure the compilation time consumed by the SSA algorithm.

4.3.1 Methodology

We followed the methodology described in Section 3.4.1 for data analysis (Section 3.4.1.3) and measurement (Section 3.4.1.4). We measured runtime execution, memory consumption and compilation time of the Pybench, Pystone, Points and the C# Java Grande benchmarks described in Section 3.4.1.2.

In this case, we measured the following language implementations:

- *StaDyn*. The statically and dynamically typed language described in Section 2.1. It implements the SSA transformation algorithms described in this chapter, gathering type information of `dynamic` references. This type information is used to improve compile-time error detection and runtime performance [141].
- *StaDyn* without SSA. This is the previous version of *StaDyn*, where the SSA transformations were not supported. In this case, `dynamic` variables are translated to `object` references. When arithmetic, comparison or logic

operators are used with `dynamic` variables, the compiler generates nested type inspections and casts [66]. For method invocation, field access and array indexing, introspection is used.

– C# 4.5.2. This version of C# combines static and dynamic typing. It generates code for the DLR, released as part of the .NET Framework 4+ [115]. The DLR optimizes the use of `dynamic` references implementing a three-level runtime cache [145].

We also measured the compilation time of the existing C# compilers, to compare them to the *StaDyn* compiler:

– CSC (CSharp Compiler) 4.5.2, the proprietary C# compiler developed by Microsoft and shipped with the .NET Framework.

– Roselyn, the code name for the .NET Compiler Platform [146]. It provides open-source compilation services and code analysis APIs for C# and Visual Basic .NET. As *StaDyn*, it was developed in C#, over the .NET Framework.

– Mono C# compiler 4.2, another open source C# compiler developed by Xamarin [147]. It was written in C# and it follows the ECMA 334 C# language specification [148].

– The *StaDyn* compiler, with and without the SSA phase.

All the measurements are detailed in the tables included in Appendix B.

4.3.2 Start-up performance

Figure 4.15 shows the start-up performance for the Pybench micro-benchmark. *StaDyn* is the language implementation that requires the lowest execution time for all the tests. If no SSA phase is used to infer the types of local variables, execution time is 13 times higher. This shows the impact of inferring the type of variables statically, instead of using type casts and introspection. The C# approach uses the DLR runtime type cache. This option requires, on average, 33 times more execution time than the *StaDyn* programs.

The DLR cache shows better performance than the use of `object` (*StaDyn* without SSA) in dynamically typed method invocations (calls) and map indexing operations (dicts). In these two programs, all the operations against `dynamic` references are translated to introspection calls, when the *StaDyn* compiler does not implement the SSA transformation. This shows how the DLR provides important performance benefits compared to reflection [145]. The constructs, lists, lookups and strings programs also utilize introspection. Figure 4.15 shows how, in these programs, the difference between C# and *StaDyn* without SSA is not as high as for those programs that just generate nested type inspections [14] (arithmetic and numbers).

On the contrary, the instances, new instances and exceptions C# programs show the worst relative performances. Since these programs have almost no

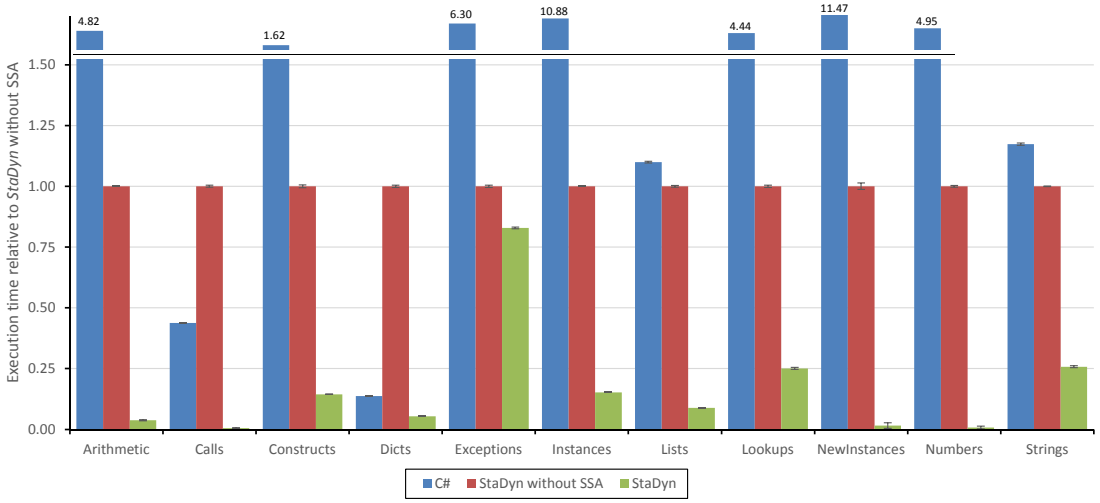


Figure 4.15: Start-up performance of Pybench, relative to *StaDyn* without SSA.

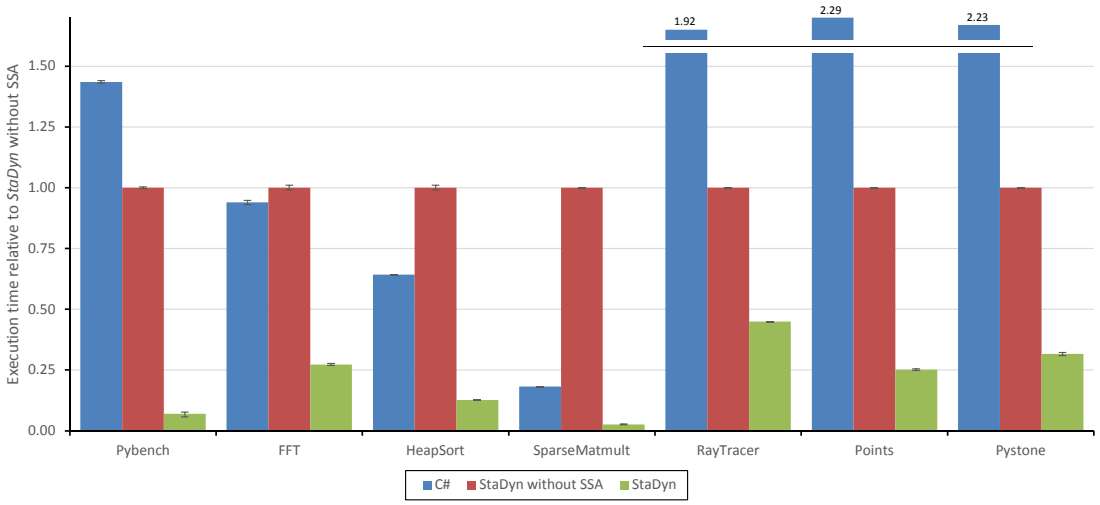


Figure 4.16: Start-up performance of all the benchmarks, relative to *StaDyn* without SSA.

`dynamic` references, the DLR initialization penalty is more significant (and the cache benefits are negligible).

Figure 4.16 shows the start-up performance for all the programs described in Section 3.4.1.2 –average results for Pybench are also included. As for Pybench, *StaDyn* obtains the best runtime performance in all the benchmarks. The *StaDyn* version without SSA requires 6.38 times more execution time. On average, C# is 6.8 factors slower than *StaDyn*.

FFT, HeapSort and SparseMatmult make intensive use of array operations. In this case, *StaDyn* without SSA uses reflective calls, performing worse than the DLR (C#). The rest of programs (RayTracer, Points, Pystone and Pybench) perform many arithmetical operations, offsetting the use of reflection. This common characteristic of these programs make the *StaDyn* version without SSA perform faster than C#.

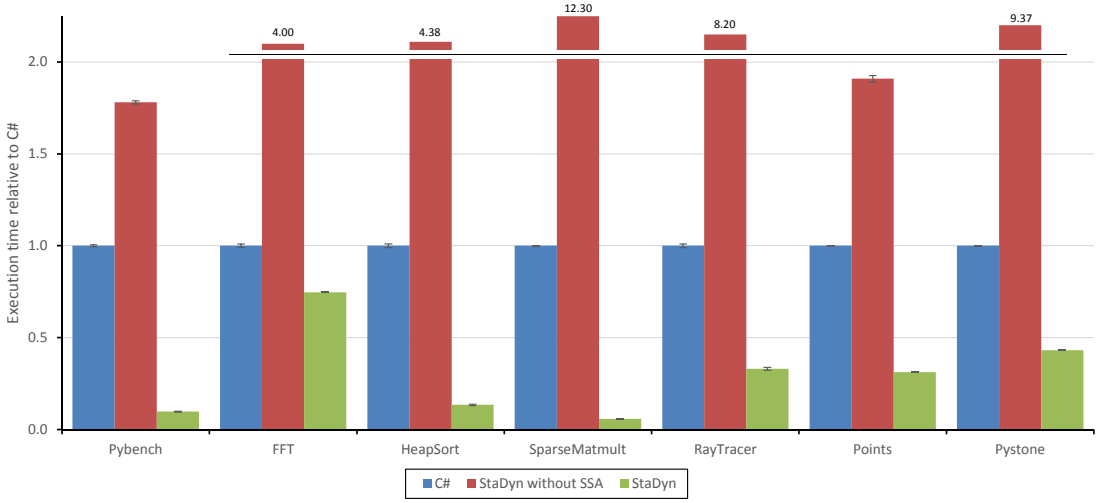


Figure 4.17: Steady-state performance of all the benchmarks, relative to C#.

4.3.3 Steady-state performance

We executed the same programs following the steady-state methodology described in Section 3.4.1.3. Figure 4.17 shows the runtime performance of all the benchmarks relative to the C# language. With this methodology, *StaDyn* is also the language with the best performance. It is on average 4.5 and 21.7 times faster than C# and *StaDyn* without SSA, respectively.

In steady state, the DLR cache used by C# shows a significant improvement. In this case, C# is faster than *StaDyn*, if the SSA transformation is not provided. With this methodology, the DLR cache is able to predict the dynamic type of many variables, since the same code is executed many times.

Table 4.1 shows the steady-state performance relative to the startup-up one. C# is the language that shows the best improvement due to its runtime cache. Steady-state programs in C# are from 130% to 2,410% faster than its start-up version. As discussed, the C# cache initialization penalty is obviated in the steady-state methodology.

StaDyn without SSA shows the lowest performance improvements, ranging from 2.59% to 59%. This language implementation has no initialization penalty, since it does not provide any cache. The slight steady-state improvement is caused by the runtime optimizations implemented by the CLR. *StaDyn* is in the middle of both approaches. It uses the runtime cache of the DLR for `dynamic` arguments; the exact type for `dynamic` local references with one type; and `object` with dynamic type inspections for union types (Section 4.2.4). Indeed, the two programs with more `dynamic` parameters (Pystone and Raytracer) are those with the highest performance gains.

	C#	*StaDyn* without SSA	*StaDyn*
Pybench	227.48%	28.15%	64.62%
FFT	487.76%	56.36%	128.57%
HeapSort	218.90%	13.20%	369.39%
RayTracer	130.13%	2.59%	482.22%
SparseMatmult	2,410.43%	59.01%	1,678.48%
Points	358.94%	4.83%	61.41%
Pystone	2,333.77%	16.69%	701.20%

Table 4.1: Steady-state runtime performance gain, relative to start-up.

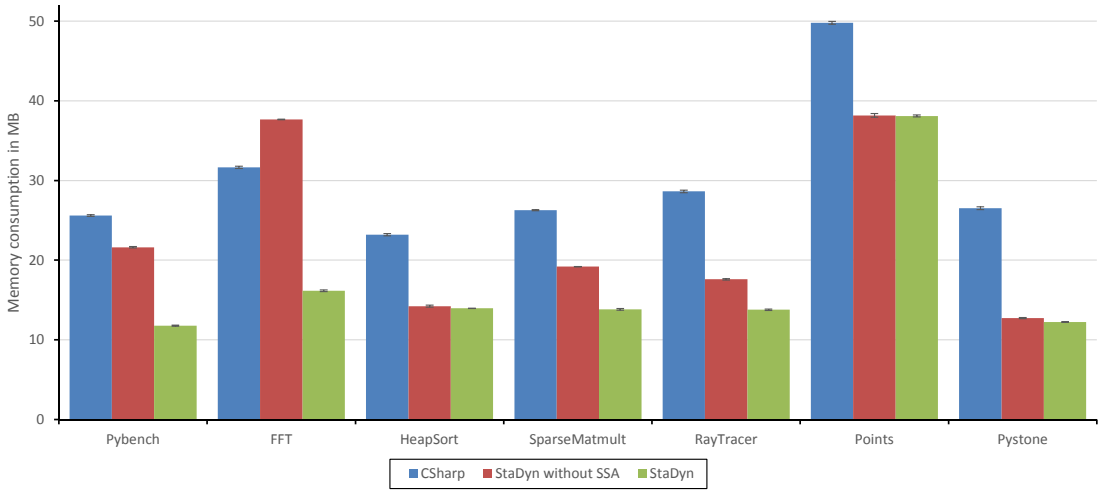


Figure 4.18: Memory consumption.

4.3.4 Memory consumption

Figure 4.18 shows the memory consumed by all the programs. The SSA transformation helps the compiler to infer the types of local variables, making *StaDyn* the language implementation with the lowest memory consumption. It avoids the use of reflection and runtime type inspections done by the previous implementation, which consumes 34.7% more memory. The DLR cache used by C# requires 86.8% more memory than *StaDyn*.

In the FFT program, C# consumes less memory resources than *StaDyn* without SSA. This program performs many computations over different variables. The nested type inspections and casts generated for these computations require significantly more code that the DLR approach. In fact, the executable file generated by *StaDyn* without SSA is 66.7% bigger than the generated by C#.

4.3.5 Compilation time

The *StaDyn* compiler provides runtime performance benefits by transforming programs into their SSA form, facilitating type inference of `dynamic` references. This process provides significant benefits in runtime performance (Sections 4.3.2 and 4.3.3), but requires more compilation time. To evaluate this cost, we measure

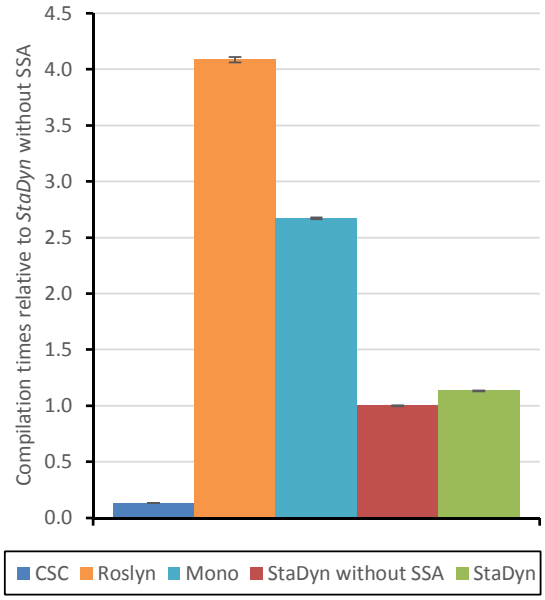


Figure 4.19: Compilation time relative to *StaDyn* without SSA.

compilation time of *StaDyn* with and without the SSA phase. We also measure the compilation time of the CSC commercial compiler developed in C, and its open source C# version (Roslyn). We use the start-up methodology described in Section 3.4.1.3.

Figure 4.19 shows the average compilation time for all the benchmarks. The *StaDyn* compiler requires 13% more compilation time when the SSA phase is enabled. This value is the compilation time cost of the SSA transformations proposed in this article. The native CSC requires 13% the compilation time used by *StaDyn*. When comparing the compilers implemented over the .NET framework (in C#), the *StaDyn* compiler is 308% and 167% faster than Mono and Roslyn, respectively.

Chapter 5

Optimizing Multimethods with Static Type Inference

Object-oriented programming languages provide dynamic binding as a mechanism to implement maintainable code. Dynamic binding is a dispatching technique that postpones until runtime the process of associating a message to a specific method. Therefore, when the `toString` message is passed to a Java object, the actual `toString` method called is that implemented by the dynamic type of the object, discovered by the virtual machine at runtime.

Although dynamic binding is a powerful tool, widespread languages such as Java, C# and C++ only support it as a single dispatch mechanism: the actual method to be invoked depends on the dynamic type of a *single* object. In these languages, multiple-dispatch is simulated by the programmer using specific design patterns, inspecting the dynamic type of objects, or using reflection.

In languages that support multiple-dispatch, a message can be dynamically associated to a specific method based on the runtime type of all its arguments. These multiple-dispatch methods are also called multi-methods [100]. For example, if we want to evaluate binary expressions of different types with different operators, multi-methods allow modularizing each operand-operator-operand combination in a single method. In the example C# code in Figure 5.1, each `Visit` method implements a different kind of operation for three concrete types, returning the appropriate value type. As shown in Figure 5.2, the values and operators implement the `Value` and `Operator` interface, respectively. Taking two `Value` operands and an `Operator`, a multi-method is able to receive these three parameters and dynamically select the appropriate `Visit` method to be called. It works like dynamic binding, but with multiple types. In our example, a triple dispatch mechanism is required (the appropriate `Visit` method to be called is determined by the dynamic type of its three parameters).

Polymorphism can be used to provide a default behavior if one combination of two expressions and one operator is not provided. Since `Value` and `Operator` are the base types of the parameters (Figure 5.2), the last `Visit` method in Figure 5.1 will be called by the multiple dispatcher when there is no other suitable `Visit` method with the concrete dynamic types of the arguments passed. An example

```
public class EvaluateExpression {

  // Addition
  Integer Visit(Integer op1, AddOp op, Integer op2) { return new Integer(op1.Value + op2.Value); }
  Double  Visit(Double op1,  AddOp op, Integer op2) { return new Double(op1.Value + op2.Value); }
  Double  Visit(Integer op1, AddOp op, Double op2)  { return new Double(op1.Value + op2.Value); }
  Double  Visit(Double op1,  AddOp op, Double op2)  { return new Double(op1.Value + op2.Value); }
  String  Visit(String op1,  AddOp op, String op2)  { return new String(op1.Value + op2.Value); }
  String  Visit(String op1,  AddOp op, Value op2)   { return new String(op1.Value + op2.ToString()); }
  String  Visit(Value op1,   AddOp op, String op2)  { return new String(op1.ToString() + op2.Value); }

  // EqualsTo
  Bool Visit(Integer op1, EqualToOp op, Integer op2) { return new Bool(op1.Value == op2.Value); }
  Bool Visit(Double op1,  EqualToOp op, Integer op2) { return new Bool((int)(op1.Value) == op2.Value); }
  Bool Visit(Integer op1, EqualToOp op, Double op2)  { return new Bool(op1.Value == ((int)op2.Value)); }
  Bool Visit(Double op1,  EqualToOp op, Double op2)  { return new Bool(op1.Value == op2.Value); }
  Bool Visit(Bool op1,    EqualToOp op, Bool op2)    { return new Bool(op1.Value == op2.Value); }
  Bool Visit(String op1,  EqualToOp op, String op2)  { return new Bool(op1.Value.Equals(op2.Value)); }

  // And
  Bool Visit(Bool op1, AndOp op, Bool op2) { return new Bool (op1.Value && op2.Value); }

  // The rest of combinations
  Value Visit(Value op1, Operator op, Value op2) { return null; }
}
```

Figure 5.1: Modularizing each operand and operator type combination.

is evaluating the addition (`AddOp`) of two Boolean (`Bool`) expressions.

In this chapter, we analyze the common approaches programmers use to simulate multiple dispatch in those widespread object-oriented languages that only provide single dispatch (e.g., Java, C# and C++) [66]. Afterwards, we propose an alternative approach, implemented as part of the *StaDyn* programming language [117]. All the alternatives are qualitatively compared considering factors such as software maintainability and readability, code size, parameter generalization, and compile-time type checking. A quantitative assessment of runtime performance and memory consumption is also presented. We also discuss the approach of hybrid dynamic and static typing languages, such as C#, Objective-C, Boo and Cobra [14].

5.1 Existing approaches

5.1.1 The Visitor design pattern

The *Visitor* design patter is a very common approach to obtain multiple dispatch in object-oriented languages than do not implement multi-methods [43]. By using method overloading, each combination of non-abstract types is implemented in a specific `Visit` method (Figure 5.1). Static type checking is used to modularize each operation in a different method. The compiler solves method overloading by selecting the appropriate implementation depending on the static types of the parameters. Suppose an n-dispatch scenario: a method with n polymorphic parameters, where each parameter should be dynamically dispatched considering its dynamic type (i.e., multiple dynamic binding). In this n-dispatch scenario, the n parameters belong to the H_1, H_2, ..., H_n hierarchies, respectively. Under these circumstances, there are potentially $\prod_{i=1}^{n} CC_i$ `Visit` methods, CC_i being the number of concrete (non-abstract) classes in the H_i hierarchy.

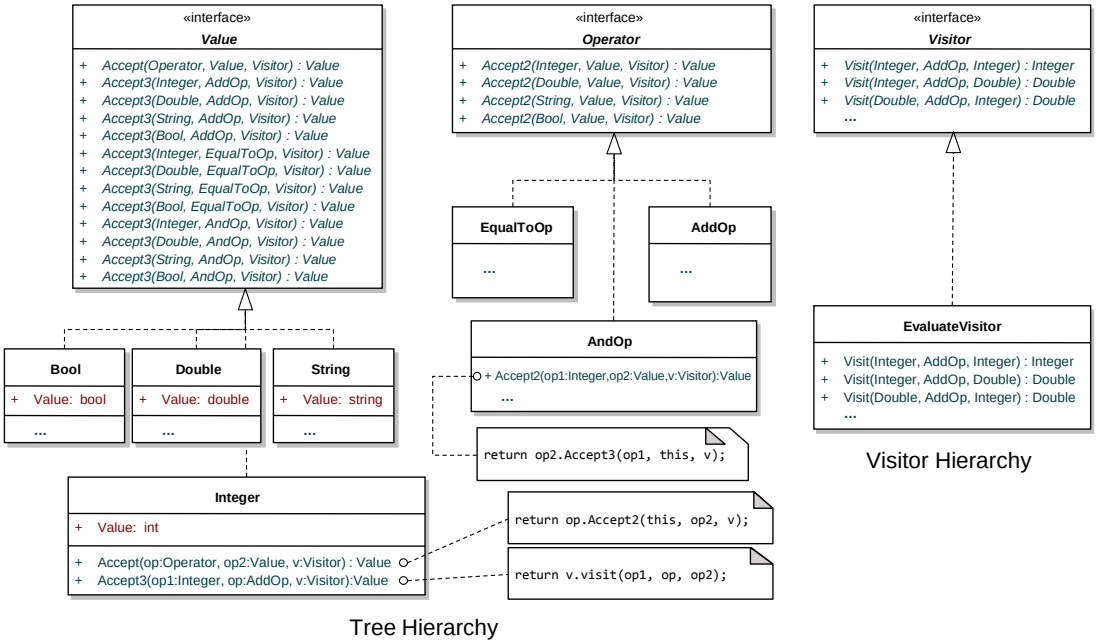


Figure 5.2: Multiple dispatch implementation with the statically typed approach (ellipsis obviates repeated members).

Using polymorphism, parameters can be generalized in groups of shared behavior (base classes or interfaces). An example of this generalization is the two last addition methods in Figure 5.1. They generalize the way strings are concatenated with any other `Value`. This feature that allows grouping implementations by means of polymorphism is the parameter generalization criterion mentioned in the previous section.

As shown in Figure 5.2, the *Visitor* pattern places the `Visit` methods in another class (or hierarchy) to avoid mixing the tree structures to be visited (`Value` and `Operator`) with the traversal algorithms (`Visitor`) [47]. The (single) dispatching mechanism used to select the correct `Visit` method is dynamic binding [43]. A polymorphic (virtual) method must be declared in the tree hierarchy, because that is the hierarchy the specific parameter types of the `Visit` methods belong to. In Figure 5.2, the `Accept` method in `Value` provides the multiple dispatch. When overriding this method in a concrete `Value` class, the type of `this` will be non-abstract, and hence the specific dynamic type of the first parameter of `Visit` will be known. Therefore, by using dynamic binding, the type of the first parameter is discovered. This process has to be repeated for every parameter of the `Visit` method. In our example (Figure 5.2), the type of the second operand is discovered with the `Accept2` method in `Operator`, and `Accept3` in `Value` discovers the type of the third parameter before calling the appropriate `Visit` method.

In this approach, the number of `AcceptX` method implementations grows geometrically relative to the dispatch dimensions (i.e., the n in n-dispatch, or the number of the `Visit` parameters). Namely, for H_1, H_2, ..., H_n hierarchies of the corresponding n parameters in `Visit`, the number of `Accept` methods are $1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} CC_j$. Therefore, the code size grows geometrically with the num-

ber of parameters in the multi-method. Additionally, declaring the signature of each single `Accept`*`X`* method is error-prone and reduces its readability.

Adding a new concrete class to the tree hierarchy requires adding more `Accept`*`X`* methods to the implementation (see the formula in the previous paragraph). This feature reduces the maintainability of this approach, causing the so-called *expression problem* [149]. This problem is produced when the addition of a new type to a type hierarchy involves changes in other classes.

The *Visitor* approach provides different advantages. First, the static type error detection provided by the compiler. Second, this approach provides the best runtime performance (see Section 5.3). Finally, parameter generalization, as mentioned, is also supported. A summary of the pros and cons of all the approaches is presented in Table 5.1, after analyzing all the alternatives.

5.1.2 Runtime type inspection

In the previous approach, the dispatcher is implemented by reducing multiple-dispatch to multiple cases of single dispatch. Its high dependence on the number of concrete classes makes it error-prone and reduces its maintainability. This second approach implements a dispatcher by consulting the dynamic type of each parameter in order to solve the specific `Visit` method to be called. This type inspection could be performed by either using an *is type of* operator (e.g., `is` in C# or `instanceof` in Java) or asking the type of an object at runtime (e.g., `GetType` in C# or `getClass` in Java). Figure 5.3 shows an example implementation in C# using the `is` operator. Notice that this single `Accept` method is part of the `EvaluateExpression` class in Figure 5.1 (it does not need to be added to the tree hierarchy).

Figure 5.3 shows the low readability of this approach for our triple dispatch example with seven concrete classes. The maintainability of the code is also low, because the dispatcher implementation is highly coupled with the number of both the parameters of the `Visit` method and the concrete classes in the tree hierarchy. At the same time, the code size of the dispatcher grows with the number of parameters and concrete classes.

The `is` operator approach makes extensive use of type casts. Since cast expressions perform type checks at runtime, this approximation loses the robustness of full compile-time type checking [150]. The `GetType` approach also has this limitation together with the use of strings for class names, which may cause runtime errors when the class name is not written correctly. Parameter generalization is provided by means of polymorphism. As discussed in Section 5.3, the runtime performance of these two approaches is not as good as that of the previous alternative.

```
public class EvaluateExpression {
  … // * Selects the appropriate Visit method in Figure 1
  public Value Accept(Value op1, Operator op, Value op2) {
    if (op is AndOp) {
      if (op1 is Bool) {
        if (op2 is Bool)         return Visit((Bool)op1, (AndOp)op, (Bool)op2);
        else if (op2 is String)  return Visit((Bool)op1, (AndOp)op, (String)op2);
        else if (op2 is Double)  return Visit((Bool)op1, (AndOp)op, (Double)op2);
        else if (op2 is Integer) return Visit((Bool)op1, (AndOp)op, (Integer)op2);
      }
      else if (op1 is String)    {  …  }
      else if (op1 is Double)    {  …  }
      else if (op1 is Integer)   {  …  }
    else if (op is EqualToOp) {  …  }
    else if (op is AddOp)     {  …  }
    Debug.Assert(false, String.Format("No implementation for op1={0}, op={1} and op2={2}",op1,
        op, op2));
    return null;
} }
```

Figure 5.3: Multiple dispatch implementation using runtime type inspection with the `is` operator (ellipsis is used to obviate repeating code).

5.1.3 Reflection

The objective of the reflection approach is to implement a dispatcher that does not depend on the number of concrete classes in the tree hierarchy. For this purpose, not only the types of the parameters but also the methods to be invoked are discovered at runtime. The mechanism used to obtain this objective is reflection, one of the main techniques used in meta-programming [151]. Reflection is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [152]. Using reflection, the self-representation of programs can be dynamically consulted and, sometimes, modified [77]. As shown in Figure 5.4, the dynamic type of an object can be obtained using reflection (`GetType`). It is also possible to retrieve the specific `Visit` method implemented by its dynamic type (`GetMethod`), passing the dynamic types of the parameters. It also provides the runtime invocation of dynamically discovered methods (`Invoke`).

The code size of this approach does not grow with the number of concrete classes. Moreover, the addition of another parameter does not involve important changes in the code. Consequently, as shown in Table 5.1, this approach is more maintainable than the previous ones. Although the reflective `Accept` method in Figure 5.4 may be somewhat atypical at first, we think its readability is certainly higher than the one in Figure 5.3.

The first drawback of this approach is that no static type checking is performed. If `Accept` invokes a nonexistent `Visit` method, an exception is thrown at runtime, but no compilation error is produced. Another limitation is that parameter generalization is not provided because reflection only looks for one specific `Visit` method. If an implementation with the exact signature specified does not exist, no other polymorphic implementation is searched (e.g., the last `Visit` method in Figure 5.1 is never called). Finally, this approach has showed the worst runtime performance in our evaluation (Section 5.3).

```
public class EvaluateExpression {
  … // * Selects the appropriate Visit method in Figure 1
  public Value Accept(Value op1, Operator op, Value op2) {
    MethodInfo method = this.GetType().GetMethod("Visit", BindingFlags.NonPublic | BindingFlags.Instance,
                                null, new Type[] { op1.GetType(), op.GetType(), op2.GetType() }, null);
    if (method == null) {
      Debug.Assert(false,String.Format("No implementation for op1={0}, op={1} and op2={2}",op1,op,op2));
      return null;
    }
    return (Value)method.Invoke(this, new object[] { op1, op, op2 });
} }
```

Figure 5.4: Multiple dispatch implementation using reflection.

5.1.4 Hybrid typing

Hybrid static and dynamic typing languages provide both typing approaches in the very same programming language. Programmers may use one alternative or the other depending on their interests, following the *static typing where possible, dynamic typing when needed* principle [12]. In the case of multiple dispatch, static typing can be used to modularize the implementation of each operand and operator type combination (`Visit` methods in Figure 5.1). Aside, dynamic typing can be used to implement multiple dispatchers that dynamically discover the suitable `Visit` method to be invoked [14].

In a hybrid typing language, its static typing rules are also applied at runtime when dynamic typing is selected. This means that, for instance, method overload is postponed until runtime, but the resolution algorithm stays the same [17]. This feature has been identified to implement a multiple dispatcher that discovers the correct `Visit` method to be invoked at runtime, using the overload resolution mechanism provided by the language [153]. At the same time, parameter generalization by means of polymorphism is also achieved.

Figure 5.5 shows an example of multiple dispatch implementation (`Accept` method) in C#. With `dynamic`, the programmer indicates that dynamic typing is preferred, postponing the overload resolution until runtime. The first maintainability benefit is that the dispatcher does not depend on the number of concrete classes in the tree hierarchy (the *expression problem*) [149]. Besides, another dispatching dimension can be provided by simply declaring one more parameter, and passing it as a new argument to `Visit`. The dispatcher consists in a single invocation to the overloaded `Visit` method, indicating which parameters require dynamic binding (multiple dispatch) with a cast to `dynamic`. If the programmer wants to avoid dynamic binding for a specific parameter, this cast to `dynamic` will not be used. This simplicity makes the code highly readable and reduces its size considerably (Table 5.1). At the same time, since the overload resolution mechanism is preserved, parameter generalization by means of polymorphism is also provided (i.e., polymorphic methods like the two last addition implementations for strings in Figure 5.1).

In C#, static type checking is disabled when the `dynamic` type is used, lacking the compile-time detection of type errors. Therefore, declaring the static types of the `Accept` parameters using polymorphism is helpful for restricting their types statically (e.g., `Value` and `Operator` in Figure 5.5). Exception handling is another mechanism that can be used to make the code more robust –notice that parameter

```
public class EvaluateExpression {
  … // * Selects the appropriate Visit method in Figure 1
  public Value Accept(Value op1, Operator op, Value op2) {
    try {
      return this.Visit((dynamic)op1, (dynamic)op, (dynamic)op2);
    } catch (RuntimeBinderException) {
      Debug.Assert(false, String.Format("No implementation for op1={0}, op={1}" +
         " and op2={2}",op1,op,op2));
    }
    return null;
} }
```

Figure 5.5: Multiple dispatch implementation with the hybrid typing approach.

generalization reduces the number of possible exceptions to be thrown, compared to the reflection approach.

Finally, this approach shows the second worst runtime performance (see Section 5.3). The DLR runtime type cache [3] improves runtime performance of the reflective approach [50], but it still significantly worse than the rest of approaches (Section 5.3).

5.2 Static type checking of dynamically typed code

We have seen how the hybrid typing approach provides important maintainability, readability, code size, and parameter generalization benefits. However, the use of dynamic typing also incurs in compile-time type checking, runtime performance and memory consumption penalties. We now propose an optimization of the dynamically typed code in the hybrid approach to avoid the limitations of dynamic typing, without losing the benefits of the statically typed code. This approach has been included as an optimization of the *StaDyn* programming language [144].

Our proposal is based on gathering type information for dynamically typed references, and use it to perform static type checking and performance optimizations. The `Accept` method of this approach is the simple implementation presented in Figure 5.6. As shown, it provides the maintainability, readability and code size of the hybrid typing approach.

When the method is called with three concrete types (first invocation in the `Main` method), the appropriate `Visit` method is invoked. If the specific method is not implemented for the particular types of the arguments (second invocation), the generalization of parameters takes place and `null` is returned. Therefore, parameter generalization is another benefit of this approach. When no `Visit` method is provided for the actual parameters (third invocation), a compiler error is shown. This is because type information is also gathered for `dynamic` references and statically checked by the compiler. This type information is used to provide early type error detection, better runtime performance and lower memory consumption (Table 5.1) –these two last variables are evaluated in Section 5.3.

```
public class EvaluateExpression {
  … // * Selects the appropriate Visit method in Figure 1
  public Value Accept(dynamic op1, dynamic op, dynamic op2) {
    return this.Visit(op1, op, op2);
  }
  … // * Invocation to Accept
  public void Main(string[] args) {
    Integer integer = new Integer(3);
    Double real = new Double(23.34);
    Bool boolean = new Bool(true);
    String str = new String("StaDyn");
    Accept(integer, new AddOp(), str);
    Accept(str, new AndOp(), real);
    Accept(boolean, real, str);
    dynamic union = args.Length>0 ? integer : real;
    Accept(union, new AddOp(), union);
  }
}
```

Figure 5.6: Multiple dispatch implementation with *StaDyn* approach.

The last invocation in `Main` requires a deeper explanation. In this case, the type of `union` may be `Integer` or `Double`. The compiler manages to detect that the invocation is correct, since there are four different implementations that provide the different combinations of the three parameters in the implementation of `Visit`. The generated code inspects the dynamic type of the actual parameter, calling the appropriate `Visit` method, following the runtime type inspection technique described in Section 5.1.2.

5.2.1 Method specialization

The existing implementation of *StaDyn* already performs type checking of `dynamic` parameters [27]. Therefore, the compile type checking benefit shown in Table 5.1 is a direct benefit of the existing language design. On the contrary, the `dynamic` parameters are translated into `object` references in the existing implementation. Therefore, execution time of applications increases when `dynamic` parameters are used [1].

To avoid this limitation, we have included in *StaDyn* a method specialization optimization using the type information inferred for the arguments. Specialization refers to translation (typically from a language into itself) of a program into a more specialized version of it, in the hope that the specialized version can be more efficient than the general one [154]. One form of program specialization is partial evaluation: it considers partial information about the variables and propagates them by abstractedly evaluating the program [154].

The *StaDyn* compiler gathers type information following an abstract interpretation process [117]. It starts analyzing the `Main` method, inferring the type of all the arguments before analyzing the invoked method. Then, for the method invocation expression, a specialized version for the particular types of the arguments is generated, and no type checking needs to be done at runtime –recursion is detected and handled as a special case [144]. When one parameter may hold

```
public class EvaluateExpression {
  public Value Accept_1(Integer op1, AddOp op, String op2) {
    return this.Visit(op1, op, op2);
  }
  public Value Accept_2(String op1, AndOp op, Double op2) {
    return this.Visit(op1, op, op2);
  }
  public Value Accept_3(object op1, AddOp op, object op2) {
    if (op1 is Integer) {
      if (op2 is Integer) return Visit((Integer)op1, op, (Integer)op2);
      else return Visit((Integer)op1, op, (Double)op2); // op2 is Double
    else // op1 is Double
      if (op2 is Integer) return Visit((Double)op1, op, (Integer)op2);
      else return Visit((Double)op1, op, (Double)op2); // op2 is Double
  }
  public Value Accept(object op1, object op, object op2) {
    return this.Visit((dynamic)op1, (dynamic)op, (dynamic)op2);
  }
  … // * Invocation to Accept
  public void Main(string[] args) {
    Integer integer = new Integer(3);
    Double real = new Double(23.34);
    Bool boolean = new Bool(true);
    String str = new String("StaDyn");
    Accept_1(integer, new AddOp(), str);
    Accept_2(str, new AndOp(), real);
    Accept(boolean, real, str);
    dynamic union = args.Length>0 ? integer : real;
    Accept_3(union, new AddOp(), union);
  }
}
```

Figure 5.7: *StaDyn* program specialized for the program in Figure 5.6.

more than one type, union types are used [33].

The code in Figure 5.7 is the specialized program generated by *StaDyn* for the input program in Figure 5.6 –actually, we generate assembly code, but we show high-level C# code for the sake of readability. This specialization is the optimization we introduced in the compiler. For the first and second invocations, the `Accept_1` and `Accept_2` specialized methods are created, receiving the three particular concrete types. When the arguments may hold more than one type (union types), another specialized method is generated receiving `object` types (`Accept_3`). In the method body, the different combinations of the possible types are checked, and cast operations are added to call the precise `Visit` method. It is worth noting than only those types in the union type are checked, differently to the runtime type inspection approach discussed in Section 5.1.2. Finally, a default implementation with `dynamic` parameters is kept in case the method is called from an external assembly written in another language (in that case, the *StaDyn* compiler cannot change the invocation for the appropriate specialized method).

This method specialization technique allows optimizing the generated code by using the type information gathered by the compiler. Particularly, it generates a specialized version of a method for the particular types of its arguments. If one argument has more than one possible type (a union type), the specialized method performs a runtime type checking analysis for only those types the argument may be holding. As discussed in Section 5.3, the only alternative to this approach that provides better runtime performance is the verbose *Visitor* design pattern, where

	Maintainability	Readability	Code Size	Parameter Generalization	Compile time type hecking	Runtime Performance	Memory Consumption
Visitor Pattern				✓	✓	✓	✓
`is` Operator				✓		1/2	✓
`GetType` Method				✓		1/2	✓
Reflection	✓	✓	✓				✓
Hybrid Typing	✓	✓	✓	✓			
StaDyn	✓	✓	✓	✓	✓	1/2	✓

Table 5.1: Qualitative evaluation of the approaches.

the programmer has to write much more error-prone code, difficult to maintain (Section 5.2.1). Furthermore, since a runtime type cache is not required, memory consumption is similar to the approaches requiring fewer memory resources (Section 5.3.3).

5.3 Evaluation

In this section, we measure execution time and memory consumption of the different approaches analyzed to justify the performance and memory assessment in the two last columns of Table 5.1. Detailed data is depicted in Appendix C.

5.3.1 Methodology

In order to compare the performance of all the approaches, we have developed a set of synthetic micro-benchmarks. These benchmarks measure the influence of the following variables on runtime performance and memory consumption:

- Dispatch dimensions. We have measured programs executing single, double and triple dispatch methods. These dispatch dimensions represent the number of parameters passed to the `Accept` method shown in Figures 5.3, 5.4, 5.5 and 5.6.
- Number of concrete classes. This variable is the number of concrete classes of each parameter of the `Accept` method. For each one, we define from 1 to 5 possible derived concrete classes. Therefore, the implemented dispatchers will have to select the correct `Visit` method out of up to 125 different implementations (5^3).
- Invocations. Each program is called an increasing number of times to analyze their performance in long-running scenarios (e.g., server applications).
- Approach. The same application is implemented using the following approaches: static typing (*Visitor* pattern), runtime type inspection (`is` and `GetType` alternatives), reflection, hybrid typing and the proposed optimizations included in the *StaDyn* language.

Each program implements a collection of `Visit` methods that simply increment a counter field. The idea is to measure the execution time of each dispatch

technique, avoiding additional significant computation –we have previously evaluated a more realistic application in [153]. Regarding the data analysis, we follow the start-up and steady-state methodologies described in Section 3.4.1.3.

5.3.2 Runtime performance

Figure 5.8 and 5.9 show the start-up and steady-state performances, respectively, of single, double and triple dispatch, when each parameter of the multi-method has five concrete derived types. Each `Visit` method is executed at least once. To analyze the influence of the number of invocations on the execution time, we invoke multi-methods in loops from 1 to 100,000 iterations. Figure 5.8 shows the average execution time for a 95% confidence level.

As can be seen in Figure 5.8, all the approaches tend to have a linear influence of the number of iterations on execution time when the number of iterations is bigger than 10,000. This trend is even clearer in the steady-state performance (Figure 5.9). With this methodology, the linear trend is shown from 100 iterations on.

The dispatch dimension (i.e., the number of parameters passed to the multi-method) of the analyzed approaches shows a different influence. For single dispatch, the hybrid typing is the slowest approach, when a few iterations are executed. Then, when the number of iterations increases, the DLR cache seems to provide the expected benefits, performing better than reflection. As the number of parameters increases, the benefits of the DLR are shown with a lower number of iterations. Similarly, steady-state performance shows this trend with a lower number of iterations, compared to start-up. When differences are linear, the fastest approach is the *Visitor* design pattern, followed by our optimization (134% more execution time), `is` (192%), `GetType` (926%), hybrid typing (6,852%) and reflection (40,693%).

Figures 5.10 and 5.11 shows the start-up and steady-state execution time, when the number of concrete classes that implement each multi-method parameter increases (for 100,000 fixed iterations). For each parameter, we increment (from 1 to 5) the number of its derived concrete classes. In the case of triple dispatch and five different concrete classes, the multiple dispatcher has to select the correct `Visit` method out of 125 (5^3) different implementations.

As in the previous case, differences between the different approaches tend to be linear when the number of concrete classes increases. There is no significant difference in the methodology (start-up or steady-state) or the number of concrete classes. The only exception is only one concrete class in steady-state. In that case, the DLR cache provides important benefits, making the hybrid approach perform better than reflection and `GetType`.

For 5 different classes, the *Visitor* approach is the only one that performs better (it is 25% faster) than the proposed optimization included in the *StaDyn* language. *StaDyn* is 22%, 467%, 1,600% and 14,312% faster than `is`, `GetType`, hybrid typing and reflection, respectively.

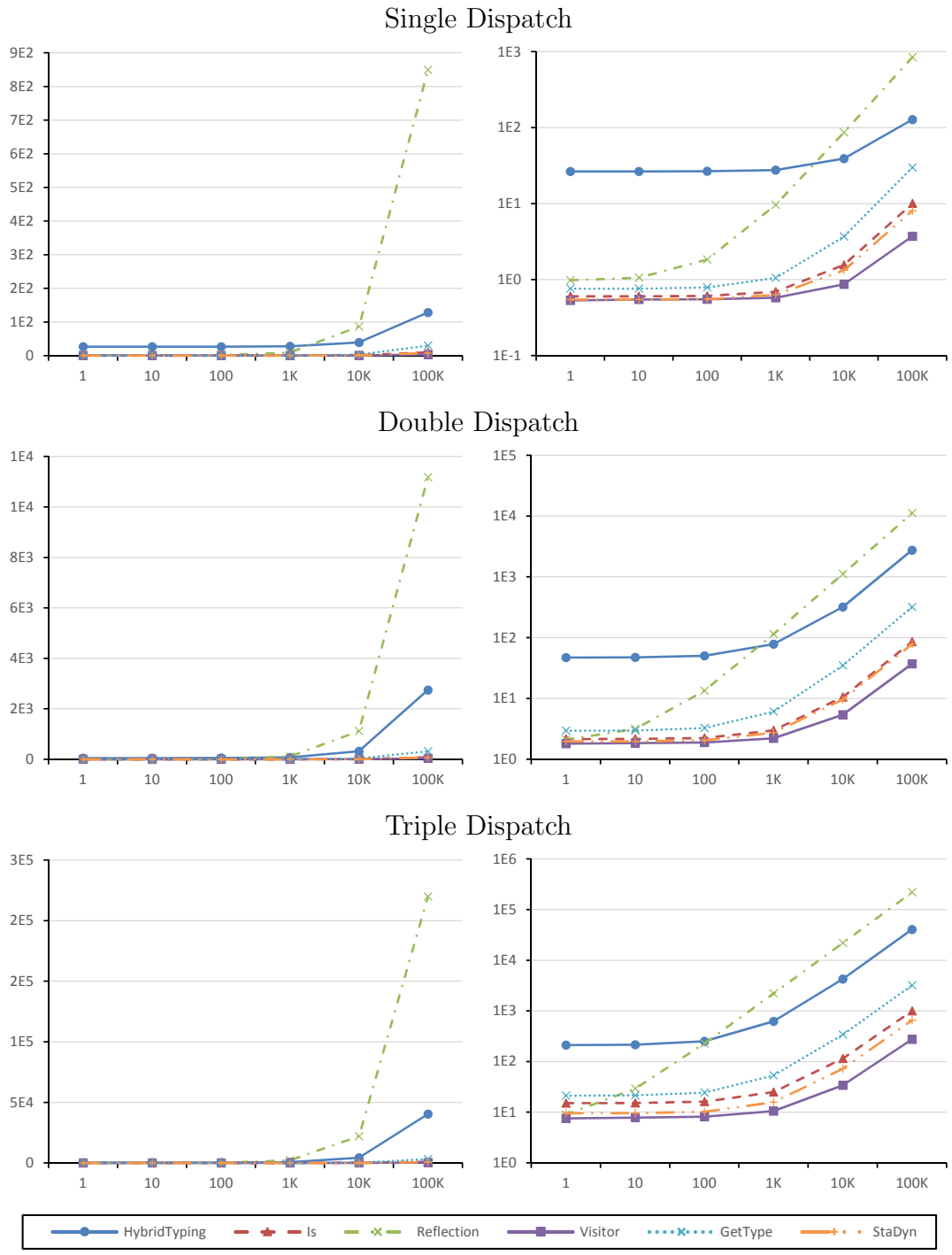


Figure 5.8: Start-up performance (in ms) for 5 different concrete classes, increasing the number of iterations; linear (left) and logarithmic (right) scales.

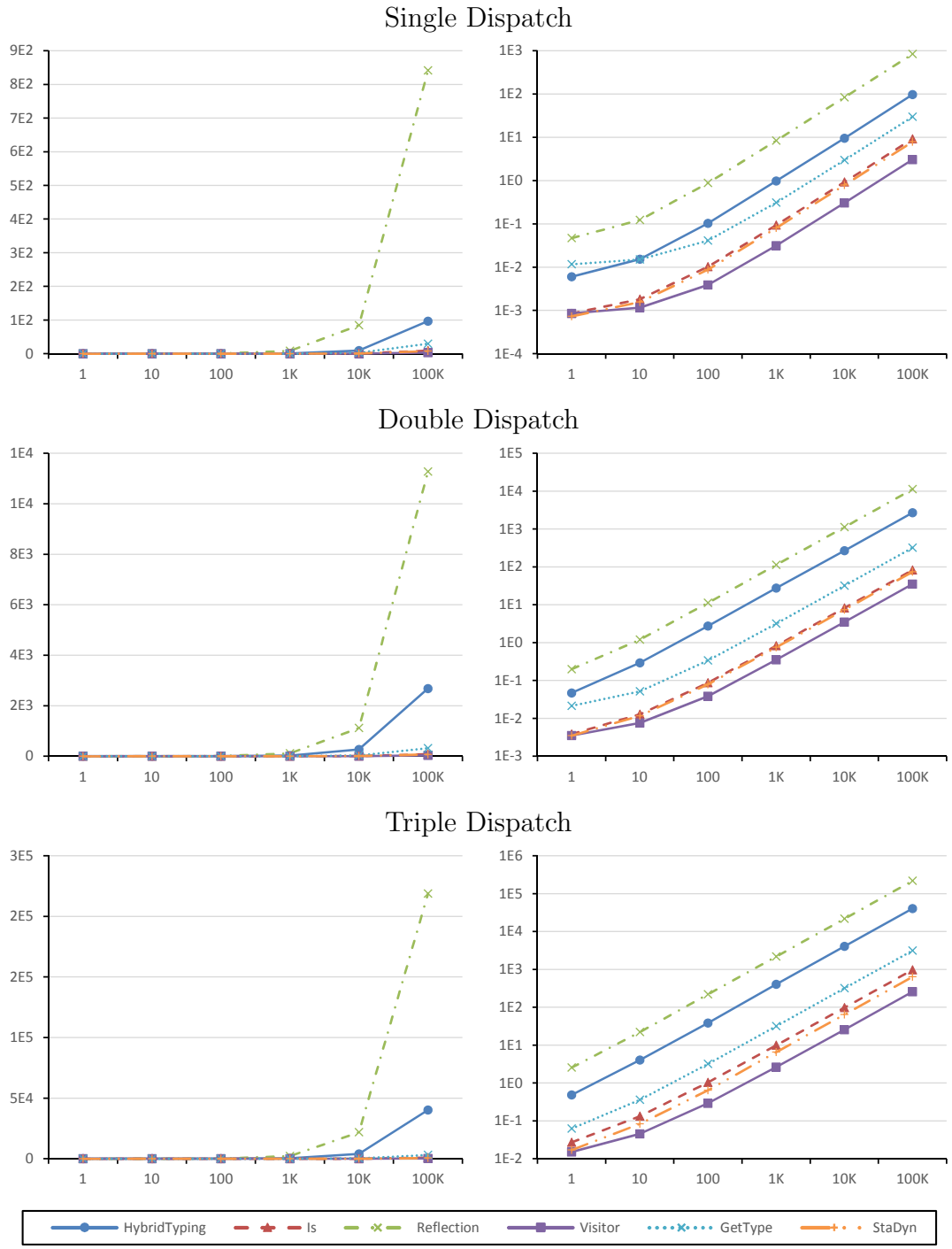


Figure 5.9: Steady-state performance (in ms) for 5 different concrete classes, increasing the number of iterations; linear (left) and logarithmic (right) scales.

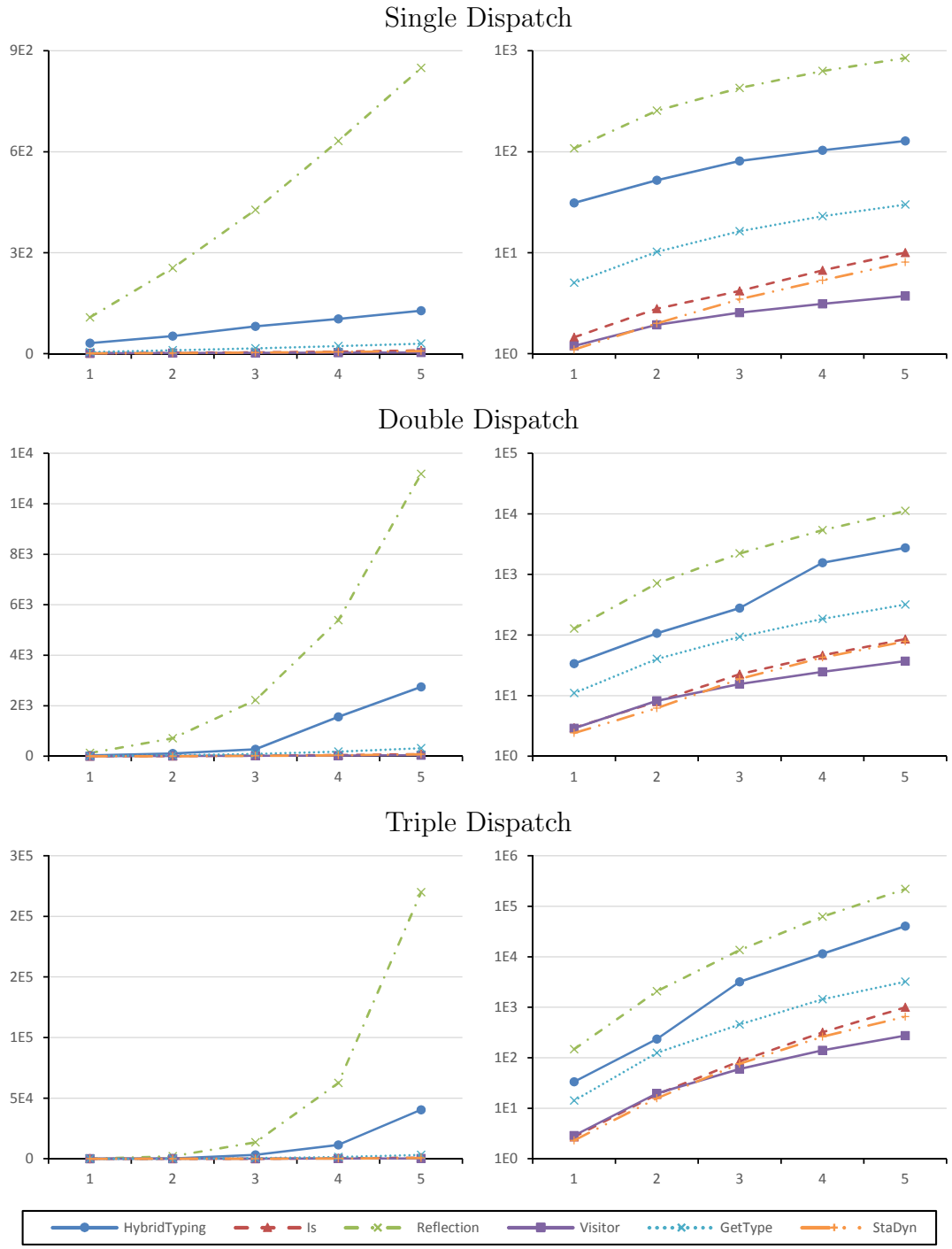


Figure 5.10: Start-up performance (in ms) for 100K iterations, increasing the number of concrete classes; linear (left) and logarithmic (right) scales.

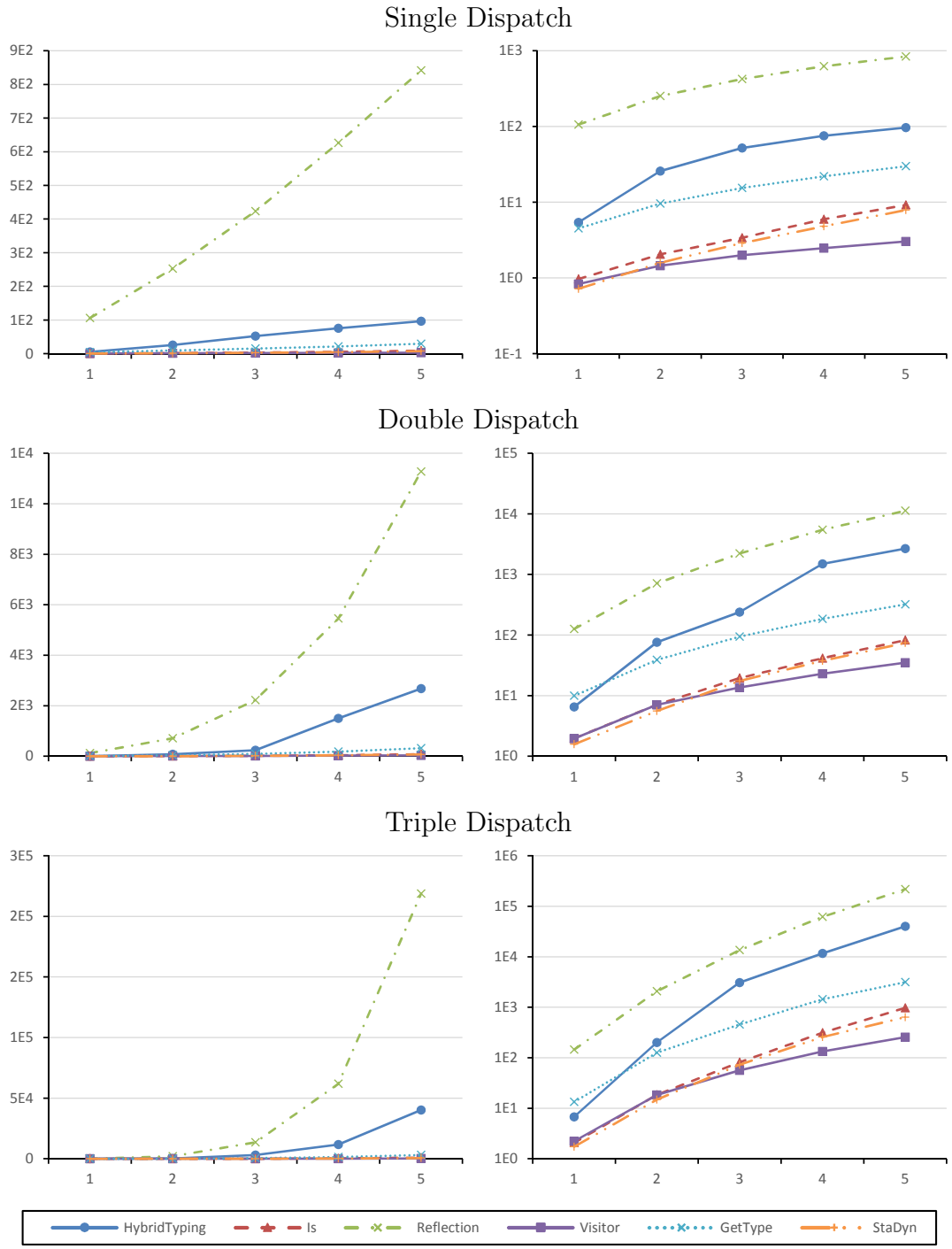


Figure 5.11: Steady-state performance (in ms) for 100K iterations, increasing the number of concrete classes; linear (left) and logarithmic (right) scales.

5.3.3 Memory consumption

We have measured memory consumption, analyzing all the variables mentioned in the Section 5.3.1. As shown in Figure 5.12, there is no influence of the dimensions of dispatch, or the number of concrete classes -although they are not shown, there is no influence of the number of iterations either. The hybrid approach involves an average increase of 31% compared with the rest of approaches. This difference is due to the DLR runtime cache [115]. The rest of alternatives, including ours, consume similar memory resources: the difference is 1%, lower than the error interval

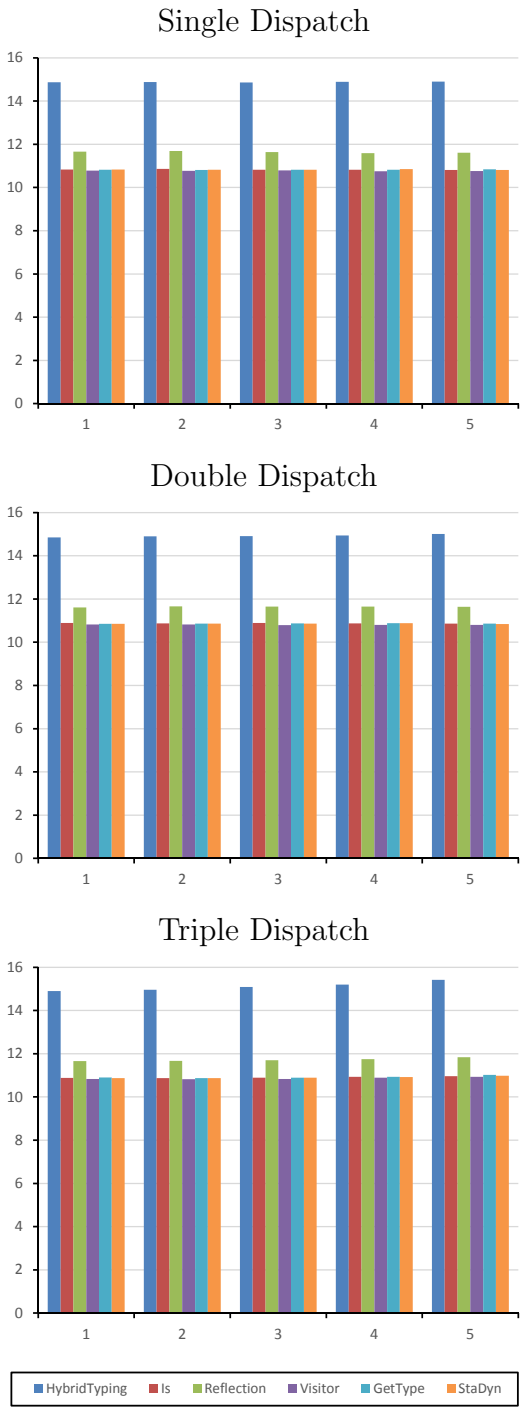


Figure 5.12: Memory consumption (in MB) for 100K iterations, increasing the number of concrete classes.

Chapter 6

Conclusions

This dissertation presents three optimization techniques for dynamically typed code, which can be applied to both dynamic and hybrid typing languages. One optimization is performed at runtime, and the other two use type information gathered by the compiler statically. The proposed optimizations are not language dependent, but we have included them in a hybrid static and dynamic typing language to measure the runtime performance benefits.

The first optimization is based on caching the dynamic types of objects at runtime. As the dynamic type of a reference barely changes, the second and subsequent uses of the same reference commonly produce a cache hit. We used the DLR of the .Net framework to optimize the existing implementations of the VB, Boo, Cobra, Fantom and *StaDyn* languages. For short-running programs, the performance gain is from 44.6% to 406%; while this benefit increases to the range of 224% to 1113% for long-running applications. Memory consumption also grows (from 6.2% to 64%), but significantly less than the performance gain.

The second optimization is based on SSA transformations, aimed at supporting variables with different types in the same scope. Since SSA transformation creates new versions of a variable in every new assignment, it facilitates the inference of a single type for each variable version. Union types are used when a variable may have different types depending on the execution flow. The average performance improvements of the SSA transformations range from 6.38 (start-up) to 21.7 (steady-state) factors. The generated code is from 4.5 to 6.8 times faster than C#. The SSA transformations also reduce memory consumption, but require 13% more compilation time.

Finally, the third optimization proposes the support of multiple dispatch using the static type information gathered by the compiler. Multiple dispatch is commonly supported with dynamic type checking, by choosing the correct method to invoke at runtime. In our approach, we use the type information gathered for the arguments. Then, a method specialization technique is used to select the correct method invocation at compile time. When the arguments are union types, nested type inspections are employed. Compared to the existing approaches, this combination of static and dynamic typing provides the highest evaluation rate in maintainability, readability, code size, parameter generalization, early type error

detection and memory consumption. Our approach is the second with the best runtime performance (out of six), requiring 25% more execution time than the type safe *Visitor* design pattern implementation.

6.1 Future Work

The first work to be done is the formalization of our multiple dispatch proposal (Chapter 5). By specifying the semantics and the type system of the language core, we can verify the properties of the proposed system [18]. This formal definition will make it easier to include our system in other programming languages. Additionally, it will be used to verify the correctness of the proposed optimization.

We also plan to add structural intercession to *StaDyn*. Structural intercession is the capability of dynamically adapting the structure of objects and types at runtime. C# only provides structural intercession for `ExpandoObjects` [115], postponing all the type checks until runtime. Besides, any existing class or object cannot be updated in C#. We plan to use the type-based alias analysis algorithm in *StaDyn* to provide structure evolution of both objects and classes [18]. In this way, many of the type errors could be detected at compile time, and significant performance optimizations could be applied [155].

After adding structural intercession to *StaDyn*, we can include the ЯRotor (Reflective Rotor) as a new back-end. ЯRotor is an extension of the Shared Source Common Language Infrastructure (SSCLI) that provides structural intercession as part of the JIT-compiler primitives [3]. It supports both the class- and prototype-based object-oriented models, allowing the adaptation of types and objects. The objective of using ЯRotor as a new back-end is to obtain better performance than using the DLR [77].

We also will add dynamic code evaluation to *StaDyn*. This means allowing the dynamic generation and evaluation of *StaDyn* code while the program is running (a typical feature in dynamically typed languages). This will make *StaDyn* to be closer to a dynamic language, but providing the robustness and performance of a statically typed language. In order to achieve this objective, we plan to include the compiler as part of the runtime, following the *compiler as a service* approach [156]. This way, the compiler and the runtime will share the same type system [17].

Another future line of work is providing a better battery of hybrid typing programs to measure runtime performance and memory consumption. The only program we used to measure hybrid typing code is the Points application (Section 3.4.1). We intend to take existing programs we developed in statically typed languages where we use introspection to simulate duck typing [157, 158, 159], and translate them to hybrid approaches such as C# and the combination of Java with `invokedynamic` [160].

Appendix A

Evaluation data of the DLR optimizations

This appendix details the data obtained when measuring the optimizations of dynamically typed code using the services of the DLR (Chapter 3).

Test Name	VB	VB-opt	Boo	Boo-opt	Fantom	Fantom-opt	Cobra	Cobra-opt	*StaDyn*	*StaDyn*-opt
Arith.SmplFloatArith	0.62 ±1.9%	0.49 ±2.0%	1.61 ±1.4%	0.49 ±2.0%	22.72 ±12.0%	0.61 ±0.0%	25.18 ±0.5%	0.45 ±0.0%	2.34 ±0.0%	2.34 ±0.0%
Arith.SmplIntegerArith	0.64 ±0.0%	0.50 ±0.0%	1.73 ±0.0%	0.51 ±2.0%	21.26 ±2.4%	0.56 ±0.0%	59.96 ±1.7%	0.47 ±1.9%	3.06 ±0.7%	3.05 ±0.3%
Arith.SmplIntFloatArith	0.64 ±1.9%	0.51 ±2.3%	1.65 ±1.3%	0.48 ±0.0%	20.70 ±0.0%	0.56 ±0.0%	60.25 ±0.5%	0.47 ±0.0%	0.05 ±0.0%	0.05 ±0.0%
Calls.FunctionCalls	0.25 ±0.0%	0.25 ±0.0%	0.15 ±3.9%	0.14 ±0.0%	0.41 ±0.0%	0.31 ±0.0%	1.36 ±0.0%	1.36 ±0.0%	5.70 ±1.1%	0.57 ±1.9%
Calls.MethodCalls	11.88 ±2.9%	0.71 ±2.0%	0.96 ±1.6%	0.41 ±0.0%	3.14 ±1.9%	0.30 ±2.7%	2.89 ±0.0%	0.73 ±0.0%	2.83 ±0.0%	0.39 ±0.0%
Calls.Recursion	0.58 ±0.0%	0.38 ±0.0%	1.70 ±0.0%	0.50 ±0.0%	8.71 ±1.9%	0.44 ±0.0%	21.05 ±1.7%	0.41 ±0.0%	0.77 ±1.8%	0.16 ±0.0%
Const.ForLoops	7.13 ±1.9%	4.93 ±0.7%	4.65 ±1.6%	1.73 ±1.5%	115.49 ±1.8%	4.82 ±0.8%	53.70 ±1.0%	1.63 ±1.3%	0.02 ±27.7%	0.02 ±27.0%
Const.IfThenElse	0.38 ±0.0%	0.30 ±2.0%	1.63 ±1.5%	0.59 ±1.9%	1.11 ±1.9%	0.29 ±2.0%	1.22 ±0.0%	1.17 ±1.9%	0.08 ±0.0%	0.08 ±0.0%
Const.NestedForLoops	26.62 ±0.5%	9.26 ±0.8%	3.29 ±1.1%	1.46 ±1.6%	104.70 ±0.1%	4.92 ±0.0%	34.85 ±1.4%	2.28 ±0.0%	25.01 ±1.3%	4.22 ±0.8%
Dicts.DictCreation	2.90 ±1.3%	1.01 ±1.8%	0.89 ±0.0%	0.89 ±0.0%	5.84 ±0.0%	5.69 ±0.6%	7.61 ±0.0%	0.95 ±1.8%	1.51 ±0.9%	0.09 ±0.0%
Dicts.DictWithFloatKeys	5.05 ±0.0%	1.17 ±1.8%	4.59 ±0.0%	1.56 ±1.8%	44.11 ±0.8%	4.02 ±1.9%	6.92 ±0.1%	0.86 ±1.9%	22.28 ±1.3%	1.53 ±1.9%
Dicts.DictWithIntKeys	6.75 ±1.9%	1.51 ±1.7%	5.67 ±0.0%	1.78 ±1.9%	18.49 ±0.6%	0.73 ±0.0%	9.09 ±1.5%	1.10 ±2.0%	29.21 ±0.4%	1.70 ±1.5%
Dicts.DictWithStrKeys	6.75 ±1.7%	1.53 ±0.0%	5.18 ±0.7%	1.22 ±0.0%	19.84 ±1.7%	2.46 ±1.7%	9.08 ±0.7%	0.95 ±0.0%	23.72 ±0.9%	1.93 ±3.0%
Dicts.SmplDictManip	19.85 ±1.3%	2.79 ±1.8%	6.71 ±1.1%	1.99 ±1.9%	20.87 ±1.9%	2.75 ±1.5%	16.23 ±0.9%	1.90 ±2.0%	23.36 ±0.9%	2.15 ±1.7%
Except.TryExcept	1.52 ±1.7%	1.47 ±1.5%	0.15 ±4.0%	0.10 ±5.6%	0.31 ±0.0%	0.05 ±0.0%	0.68 ±1.9%	0.03 ±0.0%	0.11 ±4.8%	0.11 ±4.8%
Except.TryRaiseExcept	146.01 ±0.6%	149.52 ±0.6%	8.45 ±1.5%	8.46 ±1.8%	1.18 ±1.6%	1.14 ±1.9%	0.81 ±2.0%	0.81 ±0.0%	0.72 ±0.0%	0.72 ±0.0%
Inst.CreateInstances	0.61 ±0.0%	0.56 ±0.0%	0.61 ±0.0%	0.42 ±2.0%	1.76 ±1.9%	0.33 ±0.0%	3.73 ±0.0%	0.26 ±2.2%	0.03 ±13.2%	0.03 ±0.0%
Lists.ListSlicing	1.22 ±1.8%	0.34 ±2.2%	0.96 ±1.6%	0.84 ±0.0%	0.44 ±0.0%	0.13 ±4.2%	0.30 ±0.0%	0.11 ±2.0%	0.21 ±2.6%	0.08 ±0.0%
Lists.SmplListManip	6.91 ±0.5%	2.55 ±1.4%	4.41 ±1.8%	0.94 ±1.9%	32.39 ±0.0%	1.37 ±1.6%	10.01 ±1.4%	0.72 ±1.8%	25.48 ±1.4%	1.92 ±1.9%
Lookups.ClassAttr	0.52 ±2.0%	0.48 ±3.6%	0.10 ±5.0%	0.17 ±0.0%	1.57 ±2.0%	1.38 ±1.8%	0.23 ±0.0%	0.23 ±0.0%	0.05 ±10.9%	0.05 ±0.0%
Lookups.InstanceAttr	35.37 ±2.0%	1.83 ±0.0%	3.45 ±0.0%	0.42 ±0.0%	11.47 ±1.8%	0.42 ±0.0%	5.49 ±1.7%	1.69 ±0.0%	3.58 ±0.0%	0.42 ±0.0%
NewInst.CreateNewInst	6.48 ±1.9%	6.24 ±0.6%	0.69 ±0.0%	0.47 ±0.0%	1.83 ±1.9%	0.34 ±0.0%	3.81 ±0.0%	0.30 ±0.0%	0.49 ±2.0%	0.48 ±0.0%
Num.CmpFloats	0.87 ±1.8%	0.80 ±0.0%	1.63 ±1.3%	0.70 ±1.9%	22.85 ±2.0%	1.14 ±0.1%	20.95 ±2.0%	0.79 ±2.0%	14.54 ±1.5%	0.80 ±0.0%
Num.CmpFloatsIntegers	0.99 ±2.0%	0.92 ±1.9%	1.25 ±1.6%	0.59 ±2.0%	30.43 ±1.9%	1.22 ±0.0%	16.91 ±0.0%	0.67 ±2.0%	12.10 ±1.6%	1.79 ±2.0%
Num.CmpIntegers	1.17 ±0.0%	0.99 ±2.0%	2.61 ±2.0%	0.88 ±2.3%	85.45 ±1.9%	3.11 ±1.2%	31.38 ±0.1%	0.92 ±1.0%	21.84 ±0.7%	0.93 ±1.8%
Str.CmpStrings	75.80 ±0.7%	4.89 ±1.9%	8.12 ±1.8%	4.34 ±0.8%	18.02 ±1.8%	0.77 ±0.1%	32.58 ±0.9%	4.36 ±0.0%	16.31 ±0.0%	4.82 ±0.9%
Str.ConcatStrings	2.27 ±0.0%	2.14 ±1.9%	1,079.55 ±1.8%	2.11 ±1.7%	19.32 ±2.0%	2.55 ±0.0%	22.21 ±0.6%	1.98 ±0.5%	10.98 ±0.8%	2.65 ±1.9%
Str.CreateStrWithConcat	1.05 ±1.8%	0.85 ±2.0%	1,052.01 ±1.7%	0.83 ±1.9%	17.38 ±1.5%	1.42 ±0.1%	20.50 ±1.4%	0.80 ±1.9%	0.69 ±0.0%	0.61 ±0.0%
Str.StringMappings	10.02 ±1.3%	1.67 ±0.5%	1.68 ±1.9%	1.19 ±1.9%	6.28 ±1.7%	3.81 ±1.0%	2.11 ±1.8%	1.62 ±1.8%	0.19 ±0.0%	0.12 ±1.7%
Str.StringPredicates	1.33 ±0.0%	0.31 ±0.0%	0.33 ±0.0%	0.22 ±0.0%	9.40 ±1.5%	0.30 ±0.0%	0.52 ±0.0%	0.13 ±0.0%	0.42 ±0.0%	0.13 ±4.2%
Str.StringSlicing	50.38 ±0.9%	1.95 ±1.9%	4.04 ±1.8%	0.78 ±0.0%	23.58 ±1.6%	1.35 ±3.5%	17.17 ±0.5%	1.49 ±1.7%	7.02 ±0.0%	0.46 ±1.9%

Table A.1: Start-up performance improvement for Pybench (Figure 3.13).

Test Name	VB	VB-opt	Boo	Boo-opt	Fantom	Fantom-opt	Cobra	Cobra-opt	*StaDyn*	*StaDyn*-opt
Pybench	3.24 ±1.1%	1.30 ±1.3%	2.48 ±1.4%	0.76 ±1.2%	9.05 ±1.7%	0.92 ±0.9%	6.92 ±0.8%	0.70 ±1.0%	1.74 ±2.5%	0.45 ±1.9%
FFTBench	0.54 ±2.0%	0.44 ±1.9%	0.61 ±0.0%	0.38 ±1.9%	3.28 ±0.0%	0.16 ±0.0%	4.10 ±0.9%	0.64 ±0.0%	0.61 ±0.0%	0.22 ±0.0%
HeapSortBench	0.28 ±1.8%	0.18 ±3.3%	0.13 ±0.0%	0.11 ±0.0%	1.16 ±0.0%	0.08 ±0.0%	1.75 ±0.0%	0.22 ±0.0%	0.45 ±2.0%	0.16 ±0.0%
RayTracerBench	3.58 ±1.0%	0.75 ±0.0%	0.52 ±0.0%	0.48 ±0.0%	2.28 ±1.9%	0.23 ±0.0%	2.80 ±0.0%	0.69 ±0.0%	0.39 ±0.0%	0.27 ±2.1%
SparseMatmultBench	0.22 ±0.0%	0.17 ±0.0%	0.30 ±0.0%	0.16 ±0.0%	0.50 ±0.0%	0.06 ±9.9%	0.58 ±2.0%	0.20 ±0.0%	0.12 ±4.6%	0.07 ±6.4%
points	0.48 ±0.0%	0.14 ±0.0%	0.13 ±0.0%	0.11 ±0.0%	0.35 ±1.9%	0.05 ±10.7%	0.50 ±0.0%	0.14 ±2.8%	0.42 ±2.0%	0.20 ±0.0%
pystone	0.71 ±1.9%	0.41 ±0.0%	1.19 ±2.0%	0.47 ±0.0%	0.55 ±2.0%	0.08 ±0.0%	0.97 ±1.6%	0.21 ±2.6%	0.66 ±1.9%	0.28 ±0.0%

Table A.2: Start-up performance improvement for all the benchmarks (Figure 3.14).

Test Name	VB		VB-opt		Boo		Boo-opt		Fantom		Fantom-opt		Cobra		Cobra-opt		*StaDyn*		*StaDyn*-opt	
Arith.SmplFloatArith	0.60	±0.7%	0.28	±0.6%	1.52	±1.9%	0.32	±2.4%	21.90	±2.0%	0.59	±1.1%	25.15	±0.8%	0.28	±1.3%	2.34	±1.2%	2.33	±0.4%
Arith.SmplIntegerArith	0.63	±0.7%	0.31	±2.9%	1.62	±0.0%	0.33	±1.3%	20.87	±1.7%	0.54	±1.0%	60.28	±1.5%	0.30	±1.2%	3.04	±0.4%	3.04	±0.5%
Arith.SmplIntFloatArith	0.63	±2.0%	0.30	±0.0%	1.72	±0.8%	0.33	±1.1%	21.57	±4.5%	0.55	±1.2%	60.16	±1.4%	0.31	±2.2%	0.03	±2.0%	0.03	±0.0%
Calls.FunctionCalls	0.24	±1.5%	0.24	±0.0%	0.14	±1.9%	0.14	±0.0%	0.40	±0.0%	0.28	±0.0%	1.35	±1.2%	1.35	±0.5%	5.68	±0.9%	0.42	±0.9%
Calls.MethodCalls	11.56	±1.6%	0.40	±1.9%	0.93	±1.5%	0.24	±2.2%	3.13	±2.0%	0.20	±1.9%	2.85	±0.6%	0.55	±0.0%	2.83	±1.9%	0.22	±2.0%
Calls.Recursion	0.57	±0.0%	0.34	±0.0%	1.64	±0.8%	0.48	±0.7%	8.76	±1.8%	0.44	±0.0%	20.92	±0.3%	0.37	±0.2%	0.76	±1.2%	0.11	±0.0%
Const.ForLoops	7.06	±0.8%	4.78	±0.6%	4.65	±0.6%	1.68	±0.9%	114.89	±1.9%	4.61	±0.9%	53.61	±0.3%	1.51	±0.9%	0.01	±13.4%	0.01	±13.4%
Const.IfThenElse	0.36	±0.0%	0.24	±1.9%	1.74	±2.0%	0.52	±0.7%	1.09	±3.1%	0.27	±1.5%	1.26	±0.1%	1.08	±1.2%	0.06	±0.0%	0.06	±0.0%
Const.NestedForLoops	26.98	±1.9%	9.22	±0.9%	3.29	±1.6%	1.43	±1.9%	105.95	±2.4%	4.90	±0.7%	34.89	±1.9%	2.31	±0.2%	25.12	±0.4%	4.19	±0.6%
Dicts.DictCreation	2.87	±1.1%	0.81	±0.1%	0.88	±0.0%	0.98	±0.1%	5.86	±0.5%	5.67	±0.2%	7.54	±0.8%	0.77	±0.7%	1.50	±0.8%	0.08	±0.9%
Dicts.DictWithFloatKeys	5.02	±0.1%	1.00	±1.9%	4.54	±0.3%	1.42	±1.9%	43.82	±0.7%	3.98	±0.7%	6.87	±0.4%	0.72	±1.9%	22.14	±1.7%	1.37	±1.4%
Dicts.DictWithIntKeys	6.70	±0.6%	1.36	±1.8%	5.58	±1.0%	1.66	±0.1%	18.02	±0.1%	0.70	±0.5%	9.01	±1.5%	0.95	±1.6%	29.13	±0.0%	1.54	±1.6%
Dicts.DictWithStrKeys	6.69	±0.4%	1.33	±1.0%	5.17	±1.6%	1.10	±0.0%	19.89	±1.0%	2.44	±1.6%	8.93	±0.2%	0.80	±1.5%	25.38	±2.0%	1.73	±2.1%
Dicts.SmplDictManip	19.81	±1.1%	2.53	±0.5%	6.64	±0.2%	1.80	±1.9%	20.77	±1.7%	2.69	±0.6%	16.16	±0.0%	1.67	±1.9%	23.24	±0.8%	1.93	±1.5%
Except.TryExcept	1.46	±1.9%	1.37	±0.0%	0.11	±0.0%	0.05	±2.0%	0.31	±2.6%	0.03	±0.0%	0.67	±1.9%	0.01	±0.0%	0.04	±0.0%	0.04	±2.1%
Except.TryRaiseExcept	146.94	±1.9%	149.30	±0.6%	8.40	±0.2%	8.47	±1.8%	1.18	±1.8%	1.13	±2.0%	0.81	±1.8%	0.81	±1.6%	0.71	±1.1%	0.71	±1.9%
Inst.CreateInstances	0.58	±0.6%	0.54	±1.7%	0.60	±0.6%	0.38	±1.8%	1.77	±1.8%	0.32	±0.0%	3.84	±1.1%	0.22	±1.6%	0.03	±2.3%	0.03	±2.4%
Lists.ListSlicing	1.20	±0.0%	0.27	±2.0%	0.95	±1.5%	0.88	±1.5%	0.43	±1.6%	0.10	±0.1%	0.28	±0.0%	0.05	±0.0%	0.19	±0.5%	0.02	±0.0%
Lists.SmplListManip	6.89	±0.2%	2.25	±2.0%	4.29	±0.9%	0.75	±1.0%	32.73	±1.9%	1.35	±1.2%	9.94	±0.1%	0.51	±2.4%	25.42	±0.0%	1.71	±0.8%
Lookups.ClassAttr	0.51	±0.6%	0.36	±3.1%	0.09	±0.0%	0.09	±1.7%	0.02	±0.1%	0.02	±0.1%	0.23	±0.0%	0.23	±1.9%	0.04	±2.0%	0.04	±1.8%
Lookups.InstanceAttr	35.12	±1.4%	1.57	±1.2%	3.41	±1.7%	0.24	±2.0%	11.45	±1.5%	0.31	±0.0%	5.44	±1.5%	1.50	±1.8%	3.50	±1.7%	0.22	±1.8%
NewInst.CreateNewInst	6.41	±0.4%	6.24	±0.7%	0.66	±0.9%	0.46	±0.8%	1.80	±1.6%	0.33	±0.0%	3.75	±1.5%	0.26	±0.3%	0.48	±0.8%	0.48	±0.8%
Num.CmpFloats	0.85	±1.8%	0.49	±1.6%	1.59	±1.6%	0.44	±0.9%	22.70	±0.7%	1.08	±1.3%	21.00	±1.1%	0.51	±1.4%	14.40	±1.4%	0.48	±1.9%
Num.CmpFloatsIntegers	0.98	±0.0%	0.64	±1.4%	1.22	±1.8%	0.36	±1.9%	30.38	±1.9%	1.15	±1.3%	16.96	±0.5%	0.39	±0.2%	10.99	±0.5%	0.71	±1.9%
Num.CmpIntegers	1.16	±1.2%	0.68	±0.0%	2.65	±0.5%	0.65	±3.0%	85.60	±1.6%	3.05	±1.6%	31.32	±0.7%	0.62	±0.1%	21.64	±0.3%	0.59	±1.3%
Str.CmpStrings	75.60	±1.7%	4.75	±1.9%	8.05	±0.7%	4.18	±1.7%	18.68	±1.4%	0.75	±1.9%	33.44	±1.9%	4.15	±0.9%	16.05	±0.3%	4.67	±0.4%
Str.ConcatStrings	2.25	±1.5%	1.94	±0.7%	1,080.73	±0.7%	2.10	±1.8%	19.14	±0.0%	2.54	±1.2%	22.14	±1.0%	1.87	±1.5%	10.78	±2.6%	2.21	±1.0%
Str.CreateStrWithConcat	1.05	±1.7%	0.72	±2.0%	1,056.77	±1.8%	0.71	±0.0%	17.74	±1.7%	1.42	±0.0%	20.57	±1.3%	0.69	±1.4%	0.67	±1.4%	0.57	±1.7%
Str.StringMappings	10.00	±1.0%	1.57	±1.8%	1.61	±0.2%	1.10	±1.2%	6.15	±2.0%	3.82	±1.5%	2.09	±0.6%	1.52	±0.9%	0.19	±1.6%	0.11	±1.2%
Str.StringPredicates	1.33	±1.7%	0.16	±1.7%	0.28	±0.0%	0.17	±1.8%	9.41	±1.4%	0.28	±0.1%	0.50	±0.0%	0.09	±1.7%	0.41	±0.9%	0.08	±2.0%
Str.StringSlicing	50.36	±0.9%	1.79	±1.9%	3.98	±1.1%	0.59	±1.1%	23.49	±1.6%	1.32	±0.6%	17.15	±1.2%	1.29	±1.9%	6.98	±0.2%	0.35	±1.1%

Table A.3: Steady-state performance improvement for Pybench.

Test Name	VB		VB-opt		Boo		Boo-opt		Fantom		Fantom-opt		Cobra		Cobra-opt		*StaDyn*		*StaDyn*-opt	
Pybench	3.24	±1.1%	1.30	±1.3%	2.48	±1.4%	0.76	±1.2%	9.05	±1.7%	0.92	±0.9%	6.92	±0.8%	0.70	±1.0%	1.74	±2.5%	0.45	±1.9%
FFTBench	0.51	±0.7%	0.11	±0.0%	0.55	±0.7%	0.13	±1.8%	3.25	±1.7%	0.13	±0.0%	4.05	±0.3%	0.40	±0.9%	0.56	±1.2%	0.11	±1.8%
HeapSortBench	0.26	±0.0%	0.06	±0.0%	0.09	±0.0%	0.04	±2.4%	1.17	±1.2%	0.05	±0.0%	1.74	±0.8%	0.13	±0.0%	0.43	±0.9%	0.08	±1.9%
RayTracerBench	3.39	±1.9%	0.24	±0.0%	0.44	±0.8%	0.10	±0.0%	2.23	±1.3%	0.08	±0.0%	2.78	±1.1%	0.28	±0.0%	0.27	±0.0%	0.04	±0.0%
SparseMatmultBench	0.20	±1.6%	0.03	±0.0%	0.23	±1.7%	0.04	±0.0%	0.50	±0.0%	0.02	±2.2%	0.55	±1.3%	0.09	±1.9%	0.11	±1.5%	0.03	±2.9%
points	0.46	±1.3%	0.04	±0.0%	0.08	±1.9%	0.03	±2.1%	0.33	±0.0%	0.02	±0.0%	0.48	±0.0%	0.05	±1.9%	0.38	±0.2%	0.12	±0.0%
pystone	0.68	±1.0%	0.17	±1.3%	1.11	±1.8%	0.29	±1.7%	0.53	±1.4%	0.02	±0.0%	0.93	±1.4%	0.05	±1.9%	0.61	±1.1%	0.19	±1.9%

Table A.4: Steady-state performance improvement for all the benchmarks 3.13.

Test Name	VB	VB-opt	Boo	Boo-opt	Fantom	Fantom-opt	Cobra	Cobra-opt	*StaDyn*	*StaDyn*-opt
Arith.SmplFloatArith	13.03 ±1.0%	22.68 ±0.0%	13.35 ±0.8%	22.16 ±1.0%	22.36 ±1.0%	23.35 ±0.2%	13.82 ±1.9%	22.44 ±1.8%	11.59 ±0.3%	11.59 ±1.6%
Arith.SmplIntegerArith	13.02 ±0.7%	22.63 ±1.6%	13.28 ±0.3%	22.08 ±1.6%	22.04 ±1.0%	20.71 ±1.2%	15.48 ±1.4%	22.36 ±0.3%	11.56 ±1.8%	11.53 ±0.0%
Arith.SmplIntFloatArith	13.03 ±1.2%	22.55 ±0.3%	13.29 ±0.6%	22.12 ±0.8%	22.08 ±0.8%	22.44 ±0.7%	15.45 ±0.6%	22.39 ±0.7%	11.49 ±2.0%	11.46 ±0.0%
Calls.FunctionCalls	12.15 ±1.2%	12.16 ±1.0%	13.11 ±1.6%	12.53 ±0.3%	22.09 ±1.2%	23.02 ±0.0%	13.55 ±0.0%	13.53 ±1.9%	14.47 ±0.5%	21.83 ±0.2%
Calls.MethodCalls	19.86 ±0.6%	25.19 ±1.0%	14.50 ±1.2%	22.38 ±1.9%	22.19 ±0.7%	24.84 ±1.0%	13.84 ±1.1%	24.08 ±0.7%	15.28 ±1.0%	21.82 ±0.8%
Calls.Recursion	12.89 ±0.9%	21.82 ±1.2%	13.29 ±0.7%	21.55 ±0.5%	22.14 ±0.8%	22.20 ±1.3%	14.02 ±1.7%	22.14 ±0.2%	11.76 ±1.3%	21.53 ±1.2%
Const.ForLoops	13.96 ±0.0%	23.31 ±0.8%	13.47 ±0.5%	21.67 ±0.7%	22.26 ±1.8%	23.07 ±0.2%	14.07 ±0.5%	22.73 ±0.2%	11.76 ±1.3%	11.76 ±1.3%
Const.IfThenElse	12.96 ±1.1%	21.86 ±0.7%	13.19 ±0.3%	21.78 ±0.6%	22.14 ±0.2%	22.40 ±1.3%	13.48 ±0.3%	22.14 ±0.8%	11.48 ±1.3%	11.47 ±1.6%
Const.NestedForLoops	13.83 ±0.8%	23.03 ±1.3%	13.44 ±2.0%	21.50 ±0.7%	22.30 ±0.9%	24.25 ±0.5%	14.11 ±1.2%	22.47 ±0.2%	11.86 ±1.5%	21.32 ±1.4%
Dicts.DictCreation	13.22 ±0.8%	22.86 ±1.1%	13.31 ±1.0%	12.72 ±0.6%	22.20 ±1.0%	23.17 ±0.8%	13.81 ±1.1%	22.88 ±0.7%	11.78 ±0.3%	21.77 ±0.7%
Dicts.DictWithFloatKeys	13.25 ±1.1%	22.96 ±0.7%	14.67 ±1.5%	22.62 ±1.1%	22.20 ±1.0%	23.27 ±0.0%	13.89 ±0.6%	23.05 ±1.0%	323.56 ±7.9%	335.92 ±8.5%
Dicts.DictWithIntKeys	13.13 ±1.0%	22.60 ±1.1%	14.61 ±1.8%	22.62 ±0.8%	22.06 ±0.0%	22.69 ±1.5%	13.91 ±0.0%	23.06 ±1.3%	14.56 ±0.5%	21.98 ±0.2%
Dicts.DictWithStrKeys	14.44 ±1.3%	25.26 ±0.3%	14.60 ±1.3%	22.65 ±1.4%	21.99 ±1.5%	22.60 ±0.6%	13.89 ±1.1%	23.11 ±0.3%	313.87 ±9.5%	340.29 ±8.0%
Dicts.SmplDictManip	13.80 ±1.1%	23.51 ±0.7%	14.74 ±1.2%	23.13 ±0.3%	22.04 ±1.8%	24.64 ±1.6%	14.16 ±1.3%	23.92 ±0.3%	18.06 ±1.2%	22.58 ±1.5%
Except.TryExcept	21.90 ±1.8%	30.52 ±1.9%	13.97 ±0.8%	21.61 ±1.5%	22.32 ±1.0%	23.10 ±0.8%	13.91 ±0.5%	22.10 ±0.2%	18.98 ±1.0%	18.97 ±0.2%
Except.TryRaiseExcept	13.08 ±1.0%	22.18 ±0.7%	13.64 ±0.9%	21.95 ±0.8%	22.13 ±0.8%	23.08 ±1.0%	13.51 ±0.8%	13.53 ±1.2%	11.66 ±1.3%	11.63 ±1.6%
Inst.CreateInstances	13.05 ±1.4%	21.76 ±0.7%	13.41 ±1.4%	21.54 ±1.0%	22.04 ±1.0%	23.01 ±0.3%	13.95 ±1.6%	22.11 ±0.5%	11.43 ±1.6%	11.44 ±1.9%
Lists.ListSlicing	14.82 ±1.7%	23.32 ±0.8%	14.62 ±1.8%	22.32 ±0.6%	22.55 ±0.8%	24.48 ±0.9%	14.17 ±0.9%	22.87 ±1.4%	12.91 ±0.9%	21.85 ±0.5%
Lists.SmplListManip	14.03 ±1.0%	27.01 ±0.1%	14.42 ±0.9%	24.66 ±2.0%	22.65 ±1.1%	23.70 ±0.9%	14.30 ±1.0%	24.77 ±1.9%	18.80 ±0.8%	25.10 ±0.6%
Lookups.ClassAttr	12.94 ±0.6%	22.54 ±0.7%	10.93 ±1.7%	10.36 ±1.4%	22.07 ±1.7%	22.62 ±1.3%	11.23 ±1.6%	11.25 ±0.8%	11.44 ±1.0%	11.43 ±1.0%
Lookups.InstanceAttr	13.63 ±1.6%	23.22 ±0.2%	14.47 ±1.3%	22.41 ±0.9%	22.49 ±1.1%	24.06 ±1.2%	11.56 ±0.8%	22.86 ±0.3%	20.37 ±0.5%	21.88 ±0.3%
NewInst.CreateNewInst	12.95 ±1.0%	12.94 ±0.8%	13.40 ±1.4%	21.57 ±0.9%	22.51 ±0.7%	23.43 ±0.2%	14.01 ±0.8%	22.10 ±1.0%	11.45 ±1.0%	11.44 ±0.6%
Num.CmpFloats	13.24 ±1.1%	23.25 ±1.1%	13.29 ±1.1%	22.82 ±0.8%	22.63 ±0.7%	23.07 ±1.4%	11.62 ±0.6%	23.27 ±0.0%	249.26 ±0.0%	249.34 ±0.0%
Num.CmpFloatsIntegers	13.18 ±1.3%	23.40 ±0.8%	13.34 ±1.2%	22.89 ±0.9%	22.61 ±0.3%	24.78 ±1.0%	13.92 ±1.9%	23.38 ±0.9%	249.27 ±0.0%	249.27 ±0.0%
Num.CmpIntegers	13.20 ±0.9%	23.19 ±1.1%	13.35 ±0.6%	22.64 ±0.6%	22.88 ±0.3%	26.03 ±1.4%	11.66 ±1.9%	23.17 ±0.3%	249.27 ±0.1%	249.27 ±0.0%
Str.CmpStrings	15.11 ±1.0%	23.42 ±0.9%	16.10 ±1.1%	22.67 ±1.9%	22.10 ±1.0%	23.01 ±0.6%	14.03 ±0.8%	23.20 ±1.9%	16.49 ±0.4%	22.26 ±1.5%
Str.ConcatStrings	14.38 ±0.5%	23.32 ±0.8%	15.70 ±1.1%	22.21 ±0.7%	22.31 ±0.2%	23.29 ±0.6%	13.83 ±0.8%	22.53 ±0.5%	48.49 ±0.6%	51.22 ±0.1%
Str.CreateStrWithConcat	13.09 ±2.0%	22.70 ±1.6%	15.74 ±0.9%	22.27 ±0.9%	22.32 ±1.8%	23.26 ±0.5%	13.82 ±1.6%	22.52 ±0.7%	11.88 ±1.5%	21.27 ±0.2%
Str.StringMappings	14.41 ±0.5%	24.77 ±0.9%	16.10 ±0.9%	22.32 ±1.6%	22.10 ±0.2%	23.08 ±0.3%	13.81 ±1.9%	24.08 ±1.1%	11.90 ±1.9%	22.18 ±0.8%
Str.StringPredicates	14.49 ±0.0%	24.88 ±0.6%	14.78 ±0.7%	22.23 ±1.3%	22.89 ±0.5%	23.09 ±1.6%	14.06 ±0.7%	22.80 ±0.7%	12.18 ±1.4%	21.87 ±0.8%
Str.StringSlicing	15.31 ±0.5%	23.91 ±0.5%	16.19 ±1.4%	22.83 ±0.3%	22.24 ±0.5%	23.22 ±0.5%	14.06 ±1.0%	23.32 ±0.6%	13.39 ±0.5%	22.55 ±1.3%

Table A.5: Memory consumption increase for Pybench.

Test Name	VB	VB-opt	Boo	Boo-opt	Fantom	Fantom-opt	Cobra	Cobra-opt	*StaDyn*	*StaDyn*-opt
Pybench	13.93 ±1.0%	22.58 ±0.8%	14.03 ±1.1%	20.99 ±1.0%	22.29 ±0.9%	23.30 ±0.8%	13.67 ±1.0%	21.65 ±0.8%	22.59 ±1.5%	28.20 ±1.2%
FFTBench	15.00 ±0.0%	26.18 ±0.6%	15.02 ±0.7%	23.12 ±0.2%	22.94 ±0.0%	24.89 ±0.6%	15.54 ±1.9%	24.79 ±0.3%	17.66 ±0.2%	22.67 ±0.5%
HeapSortBench	14.31 ±1.5%	24.61 ±0.8%	14.67 ±1.7%	23.01 ±1.2%	22.23 ±1.8%	24.29 ±0.2%	14.10 ±0.5%	24.30 ±1.7%	11.94 ±1.6%	21.97 ±1.3%
RayTracerBench	14.87 ±1.7%	27.40 ±1.7%	16.96 ±1.5%	24.88 ±0.6%	23.73 ±0.5%	26.50 ±0.4%	15.68 ±0.5%	26.35 ±0.4%	13.76 ±0.3%	22.61 ±1.0%
SparseMatmultBench	14.47 ±0.5%	25.21 ±1.0%	14.79 ±1.3%	23.09 ±1.8%	23.34 ±1.9%	24.31 ±0.8%	15.43 ±1.2%	24.58 ±0.1%	14.07 ±0.7%	22.29 ±1.7%
points	19.72 ±0.6%	22.97 ±0.5%	20.59 ±1.1%	22.79 ±1.9%	23.42 ±0.8%	24.13 ±0.2%	17.26 ±0.9%	23.09 ±0.6%	14.34 ±0.5%	21.64 ±0.6%
pystone	14.65 ±0.7%	26.21 ±1.0%	15.55 ±1.8%	23.24 ±1.3%	23.36 ±0.6%	24.31 ±1.8%	16.05 ±1.6%	24.95 ±1.2%	12.27 ±1.5%	22.01 ±0.5%

Table A.6: Memory consumption increase for all the benchmarks (Figure 3.16).

Appendix B

Evaluation data of the SSA optimizations

This appendix details the data obtained when measuring the cost and benefits of the SSA transformations for inferring the type of dynamically typed local variables (Chapter 4).

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Arith.SmplFloatArith	1,786.00	±2.0%	328.00	±0.0%	15.00	±0.0%
Arith.SmplIntegerArith	1,760.00	±2.0%	390.00	±0.0%	12.00	±17.3%
Arith.SmplIntFloatArith	1,781.00	±0.0%	390.00	±0.0%	15.00	±0.0%
Calls.FunctionCalls	1,703.00	±0.0%	12,382.50	±1.1%	6.50	±15.4%
Calls.MethodCalls	1,703.00	±0.0%	1,503.75	±1.4%	15.00	±0.0%
Calls.Recursion	1,699.00	±1.3%	3,156.00	±0.0%	109.00	±0.0%
Const.ForLoops	5,374.67	±1.2%	13,043.20	±1.5%	703.00	±0.0%
Const.IfThenElse	1,371.00	±1.6%	62.00	±0.0%	62.00	±0.0%
Const.NestedForLoops	4,447.67	±0.8%	9,499.67	±1.8%	533.14	±1.8%
Dicts.DictCreation	2,093.00	±0.0%	10,828.00	±0.0%	281.00	±0.0%
Dicts.DictWithFloatKeys	2,510.00	±1.4%	15,132.50	±0.9%	1,484.00	±0.0%
Dicts.DictWithIntKeys	2,531.00	±0.0%	20,281.00	±1.4%	437.00	±0.0%
Dicts.DictWithStrKeys	2,828.00	±0.0%	20,187.50	±1.4%	1,625.00	±0.0%
Dicts.SmplDictManip	1,718.00	±0.0%	19,695.00	±0.7%	2,078.00	±0.0%
Except.TryExcept	1,093.00	±0.0%	62.00	±0.0%	46.00	±0.0%
Except.TryRaiseExcept	2,140.00	±0.0%	950.57	±1.9%	880.00	±1.7%
Inst.CreateInstances	2,208.00	±1.6%	203.00	±0.0%	31.00	±0.0%
Lists.ListSlicing	1,534.75	±1.3%	203.00	±0.0%	62.00	±0.0%
Lists.SmplListManip	3,234.00	±0.0%	20,218.50	±1.4%	515.00	±0.0%
Lookups.ClassAttr	1,273.00	±2.0%	31.00	±0.0%	78.00	±0.0%
Lookups.InstanceAttr	1,640.00	±0.0%	3,413.67	±2.0%	85.50	±6.6%
NewInst.CreateNewInst	2,187.00	±0.0%	190.73	±2.6%	3.00	±19.4%
Num.CmpFloats	2,062.00	±0.0%	421.00	±0.0%	3.50	±18.4%
Num.CmpFloatsIntegers	1,911.00	±1.8%	368.60	±2.0%	2.50	±16.0%
Num.CmpIntegers	2,385.00	±1.5%	500.00	±0.0%	4.50	±11.1%
Str.CmpStrings	5,650.33	±1.3%	13,000.00	±0.0%	3,537.20	±1.5%
Str.ConcatStrings	3,406.00	±0.0%	3,614.33	±1.0%	1,843.25	±1.9%
Str.CreateStrWithConcat	2,005.00	±1.8%	812.00	±0.0%	630.63	±1.9%
Str.StringMappings	2,609.00	±0.0%	1,828.00	±0.0%	1,109.00	±0.0%
Str.StringPredicates	1,812.00	±0.0%	328.00	±0.0%	35.50	±14.5%
Str.StringSlicing	2,213.00	±1.6%	6,765.00	±0.0%	281.00	±0.0%

Table B.1: Start-up performance of Pybench (Figure 4.15).

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Pybench	2,167.36	±0.0%	1,510.10	±0.0%	116.42	±0.2%
FFTBench	550.80	±2.0%	586.33	±0.0%	196.60	±3.0%
HeapSortBench	234.00	±1.9%	364.05	±2.0%	160.00	±3.1%
RayTracerBench	1,203.00	±0.0%	625.00	±2.0%	46.00	±0.0%
SparseMatmultBench	359.00	±0.0%	1,968.00	±0.0%	281.00	±0.0%
points	1,791.00	±0.0%	781.00	±0.0%	52.40	±11.2%
pystone	937.00	±0.0%	421.00	±0.0%	133.00	±4.2%

Table B.2: Start-up performance of all the benchmarks (Figure 4.16).

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Arith.SmplFloatArith	542.40	±0.6%	176.15	±0.5%	8.95	±3.9%
Arith.SmplIntegerArith	513.00	±0.7%	244.80	±0.0%	9.00	±0.0%
Arith.SmplIntFloatArith	574.55	±0.2%	242.70	±1.4%	8.15	±11.6%
Calls.FunctionCalls	380.90	±0.0%	12,339.23	±0.6%	1.50	±0.0%
Calls.MethodCalls	340.52	±1.5%	1,475.20	±1.0%	1.50	±0.0%
Calls.Recursion	518.58	±1.2%	3,144.75	±1.8%	108.68	±1.5%
Const.ForLoops	4,008.95	±1.5%	12,934.38	±1.3%	683.27	±0.5%
Const.IfThenElse	332.93	±1.3%	34.00	±0.0%	52.40	±0.0%
Const.NestedForLoops	3,332.80	±1.9%	9,497.67	±1.0%	537.77	±1.8%
Dicts.DictCreation	830.10	±1.7%	10,806.90	±0.1%	281.70	±1.9%
Dicts.DictWithFloatKeys	1,093.20	±0.0%	15,117.90	±1.0%	447.60	±0.8%
Dicts.DictWithIntKeys	1,304.50	±1.4%	20,197.45	±0.9%	440.73	±1.9%
Dicts.DictWithStrKeys	1,367.50	±0.7%	20,113.15	±0.9%	561.10	±1.7%
Dicts.SmplDictManip	424.67	±1.5%	19,612.35	±1.4%	1,138.70	±0.0%
Except.TryExcept	7.50	±0.0%	6.00	±0.0%	1.50	±0.0%
Except.TryRaiseExcept	953.33	±0.8%	944.20	±1.5%	872.72	±2.0%
Inst.CreateInstances	992.24	±1.9%	204.50	±1.7%	27.80	±0.0%
Lists.ListSlicing	63.07	±1.8%	191.80	±0.0%	58.80	±0.0%
Lists.SmplListManip	1,806.87	±0.8%	20,131.85	±0.3%	506.00	±0.0%
Lookups.ClassAttr	200.94	±1.9%	20.01	±4.6%	79.63	±1.8%
Lookups.InstanceAttr	358.80	±0.0%	3,390.13	±0.9%	80.70	±1.8%
NewInst.CreateNewInst	986.96	±2.0%	187.00	±0.0%	1.50	±0.0%
Num.CmpFloats	780.13	±1.2%	196.10	±0.0%	1.50	±0.0%
Num.CmpFloatsIntegers	621.37	±1.0%	153.98	±1.6%	0.85	±19.3%
Num.CmpIntegers	1,125.40	±1.3%	282.97	±1.3%	1.85	±14.4%
Str.CmpStrings	4,168.03	±1.7%	12,966.05	±0.1%	3,518.20	±0.8%
Str.ConcatStrings	2,020.75	±0.7%	2,698.80	±1.4%	1,866.80	±0.0%
Str.CreateStrWithConcat	837.95	±1.6%	729.80	±1.4%	624.81	±1.9%
Str.StringMappings	1,255.00	±1.1%	1,818.40	±0.0%	1,096.20	±0.0%
Str.StringPredicates	418.20	±1.4%	331.00	±0.0%	32.20	±2.4%
Str.StringSlicing	726.00	±0.0%	6,674.50	±1.2%	292.90	±2.0%

Table B.3: Steady-state performance of Pybench.

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Pybench	661.84	±0.0%	1,178.36	±0.0%	79.80	±0.1%
FFTBench	93.71	±0.2%	375.00	±1.9%	121.80	±0.0%
HeapSortBench	73.38	±1.8%	321.60	±0.0%	70.00	±0.0%
RayTracerBench	47.92	±1.9%	393.07	±0.0%	9.80	±5.7%
SparseMatmultBench	156.00	±2.0%	1,918.25	±1.0%	15.80	±4.7%
points	390.25	±0.0%	745.03	±1.5%	9.00	±0.0%
pystone	38.50	±0.0%	360.80	±1.5%	16.60	±0.0%

Table B.4: Steady-state performance of all the benchmarks.

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Arith.SmplFloatArith	24.47	±0.6%	78.23	±0.1%	11.47	±1.8%
Arith.SmplIntegerArith	24.95	±1.2%	78.10	±0.3%	11.43	±1.6%
Arith.SmplIntFloatArith	24.39	±0.0%	78.14	±0.1%	11.47	±1.3%
Calls.FunctionCalls	24.96	±0.3%	11.84	±0.9%	10.35	±0.4%
Calls.MethodCalls	24.89	±0.6%	11.88	±0.3%	10.17	±1.3%
Calls.Recursion	24.51	±0.8%	11.86	±0.6%	11.40	±1.5%
Const.ForLoops	27.49	±0.5%	29.71	±0.7%	11.69	±0.0%
Const.IfThenElse	24.00	±0.5%	18.37	±0.4%	11.48	±1.9%
Const.NestedForLoops	24.74	±0.7%	12.00	±1.3%	11.60	±1.0%
Dicts.DictCreation	24.63	±0.3%	14.30	±0.8%	11.59	±1.0%
Dicts.DictWithFloatKeys	25.18	±1.3%	12.06	±1.2%	11.73	±1.2%
Dicts.DictWithIntKeys	24.75	±0.8%	12.37	±0.7%	11.54	±1.1%
Dicts.DictWithStrKeys	25.38	±1.9%	12.06	±1.3%	11.94	±0.6%
Dicts.SmplDictManip	25.04	±1.5%	14.48	±1.5%	11.98	±1.2%
Except.TryExcept	24.02	±1.3%	19.06	±1.2%	18.88	±0.8%
Except.TryRaiseExcept	24.61	±1.7%	11.73	±0.0%	11.71	±0.7%
Inst.CreateInstances	24.41	±0.7%	11.52	±1.5%	11.40	±1.0%
Lists.ListSlicing	25.15	±0.1%	12.18	±1.8%	11.67	±1.9%
Lists.SmplListManip	32.12	±0.8%	14.77	±0.0%	15.64	±0.6%
Lookups.ClassAttr	23.18	±0.5%	11.51	±1.0%	11.52	±0.7%
Lookups.InstanceAttr	25.08	±1.0%	12.57	±1.3%	12.70	±1.7%
NewInst.CreateNewInst	24.40	±0.5%	11.56	±1.2%	11.44	±1.7%
Num.CmpFloats	31.64	±0.5%	114.17	±0.8%	11.18	±1.3%
Num.CmpFloatsIntegers	31.67	±0.7%	114.13	±0.0%	10.75	±1.6%
Num.CmpIntegers	31.71	±0.9%	114.08	±0.8%	11.43	±1.0%
Str.CmpStrings	25.97	±1.4%	12.34	±0.3%	11.60	±1.6%
Str.ConcatStrings	25.45	±1.3%	110.44	±0.0%	11.64	±1.1%
Str.CreateStrWithConcat	24.41	±0.2%	23.52	±0.9%	11.58	±1.1%
Str.StringMappings	25.40	±1.4%	11.95	±1.1%	11.58	±1.6%
Str.StringPredicates	25.18	±0.9%	11.99	±1.8%	11.51	±1.9%
Str.StringSlicing	25.91	±0.6%	23.27	±0.0%	11.48	±0.9%

Table B.5: Memory consumption of Pybench.

Test Name	C#		*StaDyn* no SSA		*StaDyn*	
Pybench	25.59	±0.0%	21.62	±0.0%	11.78	±0.0%
FFTBench	31.80	±0.7%	37.68	±1.3%	38.10	±0.6%
HeapSortBench	23.32	±0.9%	14.22	±0.2%	16.15	±1.4%
RayTracerBench	28.79	±1.1%	17.59	±1.8%	13.95	±0.3%
SparseMatmultBench	26.35	±1.2%	19.19	±1.0%	13.77	±1.3%
points	49.96	±0.6%	38.17	±0.2%	13.82	±1.3%
pystone	26.70	±1.4%	12.71	±1.2%	12.22	±0.7%

Table B.6: Memory consumption of all the benchmarks (Figure 4.18).

Test Name	CSC	Roselyn	Mono	*StaDyn* no SSA	*StaDyn*
Arith.SmplFloatArith	0.215 ±12.6%	6.375 ±1.8%	4.201 ±1.9%	1.358 ±0.0%	1.545 ±1.2%
Arith.SmplIntegerArith	0.209 ±1.3%	6.356 ±0.1%	4.200 ±1.3%	1.362 ±0.0%	1.546 ±0.6%
Arith.SmplIntFloatArith	0.209 ±1.1%	6.378 ±0.8%	4.199 ±1.1%	1.372 ±0.0%	1.550 ±0.6%
Calls.FunctionCalls	0.209 ±1.3%	6.417 ±0.9%	4.210 ±0.6%	1.456 ±0.0%	1.629 ±0.6%
Calls.MethodCalls	0.209 ±1.3%	6.392 ±0.5%	4.208 ±0.4%	1.648 ±0.0%	1.804 ±0.5%
Calls.Recursion	0.207 ±1.3%	6.393 ±0.8%	4.208 ±0.9%	1.400 ±0.0%	1.573 ±1.1%
Const.ForLoops	0.209 ±1.1%	6.399 ±0.6%	4.219 ±0.6%	1.416 ±0.0%	1.599 ±1.1%
Const.IfThenElse	0.207 ±1.3%	6.395 ±0.7%	4.204 ±0.9%	1.517 ±0.0%	1.723 ±0.0%
Const.NestedForLoops	0.207 ±2.0%	6.402 ±0.0%	4.219 ±0.8%	1.400 ±0.0%	1.587 ±0.6%
Dicts.DictCreation	0.208 ±0.0%	6.420 ±0.7%	4.212 ±1.9%	1.377 ±0.0%	1.564 ±1.1%
Dicts.DictWithFloatKeys	0.208 ±0.0%	6.429 ±0.7%	4.251 ±1.1%	1.477 ±0.0%	1.674 ±1.1%
Dicts.DictWithIntKeys	0.207 ±1.1%	6.383 ±0.8%	4.211 ±0.9%	1.376 ±0.0%	1.545 ±0.6%
Dicts.DictWithStrKeys	0.208 ±1.1%	6.375 ±1.3%	4.241 ±0.0%	1.504 ±0.0%	1.707 ±0.5%
Dicts.SmplDictManip	0.208 ±2.0%	6.368 ±0.1%	4.253 ±0.6%	1.476 ±0.0%	1.650 ±1.1%
Except.TryExcept	0.210 ±0.0%	6.454 ±0.8%	4.248 ±0.6%	1.439 ±0.0%	1.580 ±0.6%
Except.TryRaiseExcept	0.206 ±2.0%	6.442 ±1.4%	4.275 ±0.4%	1.369 ±0.0%	1.553 ±1.2%
Inst.CreateInstances	0.207 ±1.3%	6.472 ±0.8%	4.207 ±1.1%	1.379 ±0.0%	1.565 ±1.1%
Lists.ListSlicing	0.207 ±1.1%	6.382 ±0.7%	4.207 ±0.4%	1.353 ±0.0%	1.540 ±1.8%
Lists.SmplListManip	0.210 ±1.9%	6.387 ±0.7%	4.213 ±0.4%	1.392 ±0.0%	1.566 ±1.1%
Lookups.ClassAttr	0.207 ±1.1%	6.378 ±1.4%	4.189 ±0.0%	1.366 ±0.0%	1.537 ±0.6%
Lookups.InstanceAttr	0.210 ±1.9%	6.396 ±1.0%	4.211 ±0.6%	1.397 ±0.0%	1.572 ±0.0%
NewInst.CreateNewInst	0.206 ±1.1%	6.464 ±0.6%	4.216 ±0.2%	1.448 ±0.0%	1.605 ±0.6%
Num.CmpFloats	0.209 ±1.9%	6.374 ±1.0%	4.204 ±1.7%	1.368 ±0.0%	1.552 ±0.5%
Num.CmpFloatsIntegers	0.209 ±1.1%	6.389 ±0.3%	4.200 ±0.6%	1.369 ±0.0%	1.549 ±0.8%
Num.CmpIntegers	0.209 ±1.1%	6.389 ±1.7%	4.201 ±1.7%	1.361 ±0.0%	1.549 ±0.5%
Str.CmpStrings	0.210 ±1.3%	6.392 ±1.3%	4.217 ±1.1%	1.378 ±0.0%	1.553 ±0.0%
Str.ConcatStrings	0.208 ±2.0%	6.397 ±0.5%	4.211 ±0.0%	1.367 ±0.0%	1.544 ±0.6%
Str.CreateStrWithConcat	0.206 ±1.1%	6.368 ±0.1%	4.189 ±0.2%	1.345 ±0.0%	1.518 ±1.2%
Str.StringMappings	0.208 ±2.0%	6.402 ±0.7%	4.206 ±1.1%	1.373 ±0.0%	1.543 ±1.2%
Str.StringPredicates	0.207 ±1.1%	6.381 ±1.0%	4.211 ±0.4%	1.363 ±0.0%	1.542 ±0.6%
Str.StringSlicing	0.209 ±1.1%	6.394 ±1.3%	4.212 ±0.2%	1.371 ±0.0%	1.551 ±0.6%

Table B.7: Compilation time of Pybench.

Test Name	CSC	Roselyn	Mono	*StaDyn* no SSA	*StaDyn*
Pybench	0.208 ±0.0%	6.398 ±0.0%	4.214 ±0.0%	1.404 ±0.0%	1.583 ±0.0%
FFTBench	0.214 ±1.3%	6.474 ±1.9%	4.286 ±0.5%	1.872 ±0.0%	2.097 ±0.0%
HeapSortBench	0.208 ±1.1%	6.629 ±0.7%	4.270 ±0.2%	1.518 ±0.0%	1.726 ±0.5%
RayTracerBench	0.226 ±1.0%	6.846 ±1.0%	4.340 ±1.7%	1.782 ±0.0%	2.007 ±1.3%
SparseMatmultBench	0.209 ±1.1%	6.418 ±1.3%	4.275 ±0.8%	1.502 ±0.0%	1.711 ±0.0%
points	0.209 ±1.3%	6.480 ±1.7%	4.239 ±0.2%	1.514 ±0.0%	1.724 ±1.0%
pystone	0.213 ±1.1%	6.533 ±1.2%	4.295 ±0.0%	1.663 ±0.0%	1.900 ±0.9%

Table B.8: Compilation time of all the benchmarks (Figure 4.19).

Appendix C

Evaluation data for the multiple dispatch optimizations

This appendix details the data obtained for the different approaches to implement multiple dispatch methods (Chapter 5).

Single Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	26.49 ±1.8%	0.60 ±0.8%	0.98 ±0.6%	0.53 ±1.3%	0.76 ±0.3%	0.55 ±1.4%
10	26.60 ±1.7%	0.60 ±1.6%	1.06 ±2.0%	0.55 ±1.1%	0.76 ±1.5%	0.55 ±2.0%
100	26.69 ±1.9%	0.61 ±1.1%	1.84 ±1.9%	0.55 ±1.2%	0.79 ±1.3%	0.55 ±0.9%
1K	27.61 ±0.6%	0.70 ±0.8%	9.66 ±1.3%	0.58 ±0.7%	1.06 ±0.7%	0.63 ±1.9%
10K	39.20 ±0.7%	1.57 ±0.6%	86.81 ±1.6%	0.87 ±1.1%	3.70 ±0.4%	1.33 ±1.0%
100K	127.90 ±2.0%	10.06 ±0.4%	849.03 ±1.5%	3.72 ±1.7%	29.99 ±3.1%	8.07 ±1.7%

Double Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	47.06 ±0.9%	2.15 ±1.8%	2.10 ±1.7%	1.80 ±1.4%	2.95 ±0.5%	1.97 ±1.4%
10	47.35 ±1.4%	2.15 ±0.0%	3.14 ±0.6%	1.83 ±1.8%	2.98 ±1.9%	1.97 ±0.3%
100	50.09 ±0.2%	2.24 ±0.5%	13.46 ±0.2%	1.88 ±1.8%	3.25 ±0.9%	2.05 ±0.9%
1K	78.09 ±1.5%	2.99 ±1.3%	115.03 ±2.0%	2.21 ±0.9%	6.11 ±1.6%	2.74 ±1.6%
10K	318.80 ±1.5%	10.53 ±0.1%	1,121.21 ±1.1%	5.37 ±2.0%	35.02 ±1.4%	9.63 ±1.4%
100K	2,744.87 ±0.0%	85.86 ±0.3%	11,182.68 ±1.2%	37.02 ±1.6%	319.69 ±1.9%	78.56 ±0.8%

Triple Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	211.41 ±1.6%	15.14 ±1.7%	8.98 ±1.1%	7.56 ±1.2%	21.18 ±0.8%	9.54 ±1.4%
10	214.67 ±0.9%	15.26 ±1.5%	29.18 ±0.9%	7.81 ±1.9%	21.51 ±1.7%	9.62 ±0.8%
100	251.25 ±1.8%	16.16 ±1.4%	229.16 ±1.2%	8.14 ±1.4%	24.39 ±1.8%	10.19 ±1.3%
1K	619.48 ±1.6%	24.99 ±0.6%	2,208.91 ±1.4%	10.50 ±1.8%	53.61 ±1.9%	15.75 ±1.6%
10K	4,246.95 ±1.7%	114.52 ±0.7%	22,093.66 ±2.0%	33.82 ±2.0%	340.66 ±1.8%	72.17 ±0.8%
100K	40,392.14 ±1.3%	999.10 ±0.7%	219,926.99 ±0.7%	273.07 ±1.6%	3,218.07 ±1.6%	654.22 ±1.6%

Table C.1: Start-up performance for 5 different concrete classes, increasing the number of iterations (Figure 5.8).

Single Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	0.0060±0.0%	0.0009±0.0%	0.0470 ±0.0%	0.0009±1.9%	0.0117±0.0%	0.0007 ±0.0%
10	0.0152±7.7%	0.0018±1.7%	0.1238±12.7%	0.0012±2.0%	0.0150±1.9%	0.0016 ±8.2%
100	0.1030±0.0%	0.0103±1.2%	0.8841 ±3.2%	0.0039±1.9%	0.0411±1.9%	0.0088 ±9.2%
1K	0.9817±0.0%	0.0937±0.6%	8.4381 ±0.0%	0.0312±1.3%	0.3093±0.4%	0.0804 ±0.0%
10K	9.5352±2.0%	0.9252±0.5%	84.4720 ±0.0%	0.3036±0.3%	2.9957±1.6%	0.7938 ±0.0%
100K	96.5241±1.7%	9.2149±0.7%	841.8626 ±1.7%	3.0382±0.3%	29.9008±1.7%	7.9067 ±1.9%

Double Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	0.0469±0.0%	0.0038±1.8%	0.1999 ±7.9%	0.0035±1.9%	0.0214±2.0%	0.0035 ±0.0%
10	0.2927±3.0%	0.0129±1.6%	1.2053 ±4.2%	0.0076±1.9%	0.0520±0.9%	0.0117 ±0.0%
100	2.7230±2.0%	0.0872±1.2%	11.2709 ±0.0%	0.0385±1.1%	0.3376±1.2%	0.0793 ±1.9%
1K	27.4890±1.8%	0.8309±1.8%	113.2678 ±1.1%	0.3513±1.9%	3.1836±0.2%	0.7550 ±1.9%
10K	266.3182±1.2%	8.2666±0.1%	1,123.3781 ±1.7%	3.4762±1.7%	32.1341±1.6%	7.4510 ±1.7%
100K	2,673.8649±0.7%	82.7348±1.9%	11,276.9048 ±2.0%	34.9245±1.9%	322.0433±1.7%	74.3367 ±1.7%

Triple Dispatch

Iterations	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	0.4829±4.3%	0.0273±1.8%	2.5601 ±3.0%	0.0151±1.8%	0.0628±3.1%	0.0171 ±3.8%
10	4.0252±3.2%	0.1339±0.1%	22.2924 ±2.0%	0.0454±1.9%	0.3598±6.3%	0.0838±10.1%
100	38.3173±0.4%	1.0343±1.6%	220.0359 ±2.4%	0.2916±1.9%	3.1822±0.6%	0.6473 ±1.7%
1K	401.9257±1.2%	9.9327±1.3%	2,193.3350 ±1.9%	2.6424±1.8%	31.7280±0.8%	6.5410 ±1.6%
10K	4,034.9813±0.9%	97.8698±0.5%	21,866.0455 ±1.9%	25.6201±1.9%	318.3810±2.0%	64.4123 ±1.6%
100K	40,152.6208±1.2%	976.8111±1.2%	218,843.6154 ±1.9%	255.9491±1.9%	3,147.0280±1.9%	638.6662 ±1.5%

Table C.2: Steady-state performance for 5 different concrete classes, increasing the number of iterations (Figure 5.9).

Single Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	31.12 ±0.8%	1.46 ±0.7%	108.14 ±1.3%	1.19 ±0.8%	5.05 ±0.8%	1.10 ±0.7%
2	52.21 ±2.0%	2.79 ±2.0%	254.78 ±2.0%	1.93 ±0.5%	10.22 ±1.4%	2.00 ±2.0%
3	80.97 ±0.2%	4.19 ±1.7%	427.55 ±2.0%	2.55 ±1.9%	16.33 ±1.9%	3.47 ±0.3%
4	103.34 ±1.9%	6.72 ±0.9%	631.98 ±0.4%	3.12 ±2.0%	23.04 ±0.5%	5.35 ±2.2%
5	127.90 ±2.0%	10.06 ±0.4%	849.03 ±1.5%	3.72 ±1.7%	29.99 ±3.1%	8.07 ±1.7%

Double Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	33.67 ±2.0%	2.96 ±0.3%	128.39 ±0.1%	2.90 ±1.2%	11.06 ±1.4%	2.42 ±1.0%
2	106.51 ±0.8%	7.97 ±1.0%	713.79 ±1.8%	8.23 ±1.8%	40.53 ±1.5%	6.22 ±0.8%
3	277.48 ±1.3%	22.61 ±0.4%	2,227.87 ±0.9%	15.56 ±1.4%	93.47 ±1.9%	18.85 ±1.3%
4	1,561.57 ±0.7%	46.46 ±0.8%	5,392.84 ±0.1%	24.78 ±0.6%	185.91 ±1.6%	43.00 ±0.4%
5	2,744.87 ±0.0%	85.86 ±0.3%	11,182.68 ±1.2%	37.02 ±1.6%	319.69 ±1.9%	78.56 ±0.8%

Triple Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	33.42 ±1.1%	2.70 ±1.8%	148.19 ±2.0%	2.87 ±1.9%	14.23 ±1.9%	2.31 ±0.6%
2	234.37 ±1.8%	18.78 ±0.4%	2,085.43 ±1.4%	19.58 ±1.9%	124.83 ±1.4%	15.85 ±1.8%
3	3,194.18 ±1.2%	85.68 ±0.2%	13,607.55 ±1.9%	59.69 ±1.9%	459.47 ±1.1%	76.48 ±1.4%
4	11,432.71 ±1.3%	321.54 ±0.5%	62,493.59 ±2.0%	139.89 ±1.9%	1,447.45 ±0.0%	262.98 ±1.1%
5	40,392.14 ±1.3%	999.10 ±0.7%	219,926.99 ±0.7%	273.07 ±1.6%	3,218.07 ±1.6%	654.22 ±1.6%

Table C.3: Start-up performance for 100K iterations, increasing the number of concrete classes (Figure 5.10).

Single Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	5.40 ±4.2%	0.97 ±1.1%	106.13 ±3.8%	0.83 ±0.3%	4.53 ±2.0%	0.72 ±0.0%
2	25.73 ±0.2%	2.06 ±1.1%	252.89 ±0.3%	1.45 ±2.0%	9.63 ±0.3%	1.59 ±1.9%
3	52.08 ±0.0%	3.40 ±0.5%	423.53 ±1.7%	2.00 ±0.8%	15.48 ±1.8%	2.90 ±0.1%
4	75.30 ±2.0%	5.98 ±1.4%	626.47 ±0.2%	2.48 ±1.8%	22.03 ±1.9%	4.83 ±1.9%
5	96.52 ±1.7%	9.21 ±0.7%	841.86 ±1.7%	3.04 ±0.3%	29.90 ±1.7%	7.91 ±1.9%

Double Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	6.53 ±0.7%	1.97 ±1.2%	125.85 ±1.8%	1.96 ±1.7%	9.95 ±1.8%	1.58 ±1.9%
2	75.70 ±1.5%	7.09 ±2.0%	714.80 ±1.8%	7.05 ±1.9%	39.09 ±1.9%	5.58 ±1.9%
3	239.50 ±0.4%	19.55 ±1.8%	2,222.01 ±1.2%	13.64 ±1.6%	94.42 ±1.8%	17.56 ±1.3%
4	1,495.38 ±1.4%	41.73 ±1.9%	5,459.93 ±0.5%	23.03 ±1.7%	185.82 ±1.6%	37.68 ±1.6%
5	2,673.86 ±0.7%	82.73 ±1.9%	11,276.90 ±2.0%	34.92 ±1.9%	322.04 ±1.7%	74.34 ±1.7%

Triple Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	6.74 ±2.6%	2.08 ±0.7%	145.32 ±1.9%	2.21 ±1.7%	13.37 ±0.1%	1.73 ±0.4%
2	200.59 ±1.9%	18.53 ±0.9%	2,080.53 ±0.3%	18.33 ±1.7%	126.67 ±1.9%	14.87 ±1.6%
3	3,090.45 ±1.8%	82.15 ±0.6%	13,592.43 ±1.6%	56.49 ±1.0%	457.29 ±1.9%	73.36 ±0.6%
4	11,663.43 ±0.4%	314.47 ±0.4%	61,845.14 ±2.0%	132.67 ±1.8%	1,444.32 ±1.9%	257.71 ±0.9%
5	40,152.62 ±1.2%	976.81 ±1.2%	218,843.62 ±1.9%	255.95 ±1.9%	3,147.03 ±1.9%	638.67 ±1.5%

Table C.4: Steady-state performance for 100K iterations, increasing the number of concrete classes (Figure 5.11).

Single Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	14.87 ±1.1%	10.83 ±0.0%	11.66 ±1.7%	10.78 ±1.2%	10.82 ±0.2%	10.83 ±0.6%
2	14.88 ±1.0%	10.86 ±1.8%	11.69 ±0.7%	10.77 ±1.0%	10.81 ±0.7%	10.82 ±1.3%
3	14.86 ±1.0%	10.82 ±1.7%	11.64 ±1.3%	10.79 ±1.0%	10.83 ±1.2%	10.82 ±1.0%
4	14.89 ±0.7%	10.82 ±0.7%	11.59 ±1.7%	10.75 ±1.3%	10.82 ±0.5%	10.85 ±2.0%
5	14.90 ±0.0%	10.81 ±0.0%	11.61 ±1.7%	10.76 ±1.7%	10.84 ±1.0%	10.82 ±0.0%

Double Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	14.85 ±1.3%	12.02 ±0.6%	11.61 ±1.2%	11.96 ±0.9%	11.99 ±0.7%	12.02 ±1.0%
2	14.90 ±0.5%	12.02 ±1.0%	11.65 ±0.9%	11.96 ±1.5%	12.01 ±0.0%	12.01 ±0.9%
3	14.91 ±0.7%	12.01 ±1.0%	11.65 ±1.5%	11.98 ±0.6%	12.03 ±0.6%	12.01 ±0.9%
4	14.94 ±0.3%	12.03 ±0.3%	11.65 ±0.3%	11.95 ±1.2%	12.05 ±0.5%	12.02 ±0.8%
5	15.01 ±0.6%	12.01 ±1.2%	11.64 ±0.3%	11.97 ±1.4%	12.03 ±0.9%	12.01 ±0.9%

Triple Dispatch

Number of classes	Hybrid Typing	Is	Reflection	Static Typing	GetType	*StaDyn*
1	14.90 ±1.0%	10.88 ±0.0%	11.66 ±0.3%	10.83 ±1.6%	10.91 ±1.0%	10.88 ±1.3%
2	14.96 ±0.7%	10.88 ±0.9%	11.67 ±0.0%	10.82 ±0.0%	10.87 ±1.9%	10.87 ±0.3%
3	15.09 ±0.3%	10.89 ±0.7%	11.70 ±0.7%	10.83 ±0.0%	10.90 ±1.4%	10.90 ±0.6%
4	15.19 ±0.3%	10.93 ±1.5%	11.75 ±0.8%	10.90 ±0.0%	10.93 ±1.7%	10.92 ±2.0%
5	15.42 ±0.3%	10.96 ±0.8%	11.84 ±0.9%	10.93 ±1.3%	11.02 ±1.9%	10.98 ±0.8%

Table C.5: Memory consumed for 100K iterations, increasing the number of concrete classes (Figure 5.12).

Appendix D

Publications

The research work of this PhD thesis has been published in different journals and conferences. The following publications are derived from this PhD.

– Articles published in journals included in the *Journal Citation Reports* at acceptance date:

1. Optimizing Runtime Performance of Hybrid Dynamically and Statically Typed Languages for the .Net Platform. Jose Quiroga, Francisco Ortin, David Llewellyn-Jones, Miguel Garcia. Journal of Systems and Software, volume 113, pp. 114-129, 2016.

2. Design and implementation of an efficient hybrid dynamic and static typing language. Miguel Garcia, Francisco Ortin, Jose Quiroga. Software: Practice and Experience, volume 46, issue 2, pp. 199-226, 2016.

3. Attaining Multiple Dispatch in Widespread Object-Oriented Languages. Francisco Ortin, Jose Quiroga, Jose M. Redondo, Miguel Garcia. Dyna, volume 186, pp. 242-250, 2014.

4. SSA Transformations to Efficiently Support Variables with Different Types in the Same Scope. Jose Quiroga, Francisco Ortin. The Computer Journal (under review).

5. Combining Static and Dynamic Typing to Achieve Multiple Dispatch. Francisco Ortin, Miguel Garcia, Jose M. Redondo, Jose Quiroga. Information – An International Interdisciplinary Journal, volume 16, issue 12(b), pp. 8731-8750, 2013.

– Articles published other journals:

1. Automatic Generation of Object-Oriented Type Checkers. Francisco Ortin, Daniel Zapico, Jose Quiroga, Miguel Garcia. Lecture Notes on Software Engineering, volume 2, issue 4, pp. 288-293. November 2014.

2. From UAProf towards a Universal Device Description Repository. Jose Quiroga, Ignacio Marin, Javier Rodriguez, Diego Berrueta, Nicanor Gutierrez, Antonio Campos. Lecture Notes of the Institute for Com-

puter Sciences, Social Informatics and Telecommunications Engineering, volume 85, pp. 263-282, 2011.

– Articles presented in conferences:

1. TyS - A Framework to Facilitate the Implementation of Object-Oriented Type Checkers. Francisco Ortin, Daniel Zapico, Jose Quiroga, Miguel Garcia. 26^{th} International Conference on Software Engineering and Knowledge Engineering (SEKE), Vancouver, British Columbia (Canada). July 2014.
2. Device Independence approach for ICT-based PFTL Solutions. Ignacio Marin, Antonio Campos, Jose Quiroga, Patricia Miravet, Francisco Ortin. International Conference on Paperless Freight Transport Logistics (e-Freight), Munich (Germany). May 2011.
3. Design of a Programming Paradigms Course Using One Single Programming Language. Francisco Ortin, Jose M. Redondo and Jose Quiroga. 4^{th} World Conference on Information Systems and Technologies (WorldCIST), Recife (Brazil). March 2016.

References

[1] Miguel Garcia, Francisco Ortin, and Jose Quiroga. Design and implementation of an efficient hybrid dynamic and static typing language. *Software: Practice and Experience*, 46:199–226, 2015. vii, 10, 22, 24, 33, 35, 40, 55, 56, 69

[2] Francisco Ortin and Juan Manuel Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71:229–243, May 2004. 1

[3] Francisco Ortin, Jose M. Redondo, and J. Baltasar G. Perez-Schofield. Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming*, 70(10):836–860, 2009. 1, 15, 68, 80

[4] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby*. Addison-Wesley Professional, Raleigh, North Carolina, 2nd edition, 2004. 1, 6

[5] Dave Thomas, David Heinemeier Hansson, Andreas Schwarz, Thomas Fuchs, Leon Breedt, and Mike Clark. *Agile Web Development with Rails. A Pragmatic Guide*. Pragmatic Bookshelf, Raleigh, North Carolina, 2005. 1

[6] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999. 1

[7] ECMA-357. *ECMAScript for XML (E4X) Specification, 2nd edition*. European Computer Manufacturers Association, Geneva, Switzerland, 2005. 1

[8] Dave Crane, Eric Pascarello, and Darren James. *AJAX in Action*. Manning Publications, Greenwich, Connecticut, 2005. 1

[9] Guido van Rossum and Fred L. Drake, Jr. *The Python Language Reference Manual*. Network Theory, United Kingdom, 2003. 1

[10] Amos Latteier, Michel Pelletier, Chris McDonough, and Peter Sabaini. The Zope book. `http://old.zope.org/Documentation/Books/ZopeBook/ZopeBook-2_6.pdf/file_view`, 2016. 1

[11] Django Software Foundation. Django, the web framework for perfectionists with deadlines. `http://openjdk.java.net/projects/mlvm`, 2016. 1

[12] Erik Meijer and Peter 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*, Vancouver, Canada, 24-28 October 2004. ACM. 1, 67

[13] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002. 1

[14] Francisco Ortin, Miguel Garcia, Jose M. Redondo, and Jose Quiroga. Combining static and dynamic typing to achieve multiple dispatch. *Information – An International Interdisciplinary Journal*, 16(12):8731–8750, Dec 2013. 1, 57, 63, 67

[15] Francisco Ortin, Patricia Conde, Daniel Fernandez-Lanvin, and Raul Izquierdo. Runtime performance of `invokedynamic`: an evaluation with a Java library. *IEEE Software*, 31(4):82–90, 2014. 1, 16, 21, 46

[16] James Strachan. Groovy 2.0 release notes. `http://groovy.codehaus.org/Groovy+2.0+release+notes`, 2016. 2, 14, 17

[17] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 76–100, Maribor, Slovenia, 21-25 June 2010. Springer-Verlag. 2, 6, 14, 45, 67, 80

[18] Francisco Ortin, Miguel A. Labrador, and Jose M. Redondo. A hybrid class- and prototype-based object model to support language-neutral structural intercession. *Information and Software Technology*, 44(1):199–219, feb 2014. 2, 15, 20, 22, 40, 80

[19] Jose M. Redondo and Francisco Ortin. A comprehensive evaluation of widespread Python implementations. *IEEE Software*, 34(4):76–84, 2015. 3

[20] Microsoft Corporation. The C# Programming Language. `http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc`, 2016. 5

[21] Francisco Ortin and Anton Morant. IDE support to facilitate the transition from rapid prototyping to robust software production. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, TOPI '11, pages 40–43, New York, NY, USA, 2011. ACM. 5

[22] Francisco Ortin, Francisco Moreno, and Anton Morant. Static type information to improve the ide features of hybrid dynamically and statically typed languages. *Journal of Visual Languages & Computing*, 25:346–362, 2014. 5, 46

[23] Francisco Ortin, Daniel Zapico, and Miguel Garcia. A programming language to facilitate the transition from rapid prototyping to efficient software

production. In *Proceedings of the Fifth International Conference on Software and Data Technologies, Volume 2, Athens, Greece*, pages 40–50, July 2010. 6

[24] Francisco Ortin and Miguel Garcia. Modularizing Different Responsibilities into Separate Parallel Hierarchies. *Communications in Computer and Information Science*, 275:16–31, January 2013. 6, 12, 35

[25] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. 6, 9

[26] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 6

[27] Francisco Ortin. Type inference to optimize a hybrid statically and dynamically typed language. *Computer Journal*, 54(11):1901–1924, November 2011. 6, 12, 24, 37, 55, 69

[28] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 201–224. Springer-Verlag, 2002. 7

[29] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. 7

[30] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. 7

[31] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 324–340, New York, NY, USA, 1994. ACM. 7

[32] Benjamin C. Pierce. Programming with intersection types and bounded polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA, 1992. 7, 27

[33] Francisco Ortin and Miguel Garcia. Union and intersection types to support both dynamic and static typing. *Information Processing Letters*, 111(6):278–286, 2011. 9, 48, 70

[34] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM. 9

[35] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997. 10

[36] Francisco Ortin and Miguel Garcia. Supporting dynamic and static typing by means of union and intersection types. In *Proceedings of the IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 993–999, Shanghai, China, 10-12 December 2010. IEEE. 10, 11, 12

[37] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM. 11

[38] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM. 11

[39] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM. 11

[40] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997. 11

[41] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996. 12

[42] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. 12

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995. 12, 34, 35, 55, 63, 64

[44] David Watt, Deryck Brown, and Robert W. Sebesta. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. 12

[45] Francisco Ortin, Daniel Zapico, Jose Quiroga, and Miguel Garcia. Automatic generation of object-oriented type checkers. *Lecture Notes on Software Engineering*, 2(4):288, 2014. 12

[46] Francisco Ortin, Daniel Zapico Palacio, Jose Quiroga, and Miguel Garcia. TyS – A framework to facilitate the implementation of object-oriented type checkers. In *IEEE 16th International Conference on Software Engineering and Knowlege Engineering (SEKE)*, pages 150–155, 2014. 12

[47] Francisco Ortin, Daniel Zapico, and Juan Manuel Cueva. Design patterns for teaching type checking in a compiler construction course. *IEEE Transactions on Educucation*, 50(3):273–283, August 2007. 12, 64

[48] Francisco Ortin and Miguel Garcia. Separating different responsibilities into parallel hierarchies. In *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*, C3S2E, pages 63–72, New York, NY, USA, 2011. ACM. 12

[49] Jim Hugunin. Just glue it! Ruby and the DLR in Silverlight. In *The MIX Conference*, Las Vegas, Nevada, 30 April - 7 May 2007. 12

[50] Jose M. Redondo and Francisco Ortin. Optimizing reflective primitives of dynamic languages. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):759–783, 2008. 12, 68

[51] Satish Thatte. Quasi-static typing. In *Proceedings of the 17th symposium on Principles of programming languages (POPL)*, pages 367–381, San Francisco, California, United States, January 1990. ACM. 12

[52] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)*, San Antonio, Texas, 23 January 2006. ACM. 12

[53] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 1–12, September 2006. 12

[54] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, Berlin, Germany, 30 July - 3 August 2007. Springer-Verlag. 12, 45

[55] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the Dynamic Languages Symposium*, pages 7:1–7:12, Paphos, Cyprus, 25 July 2008. ACM. 12

[56] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. *SIGPLAN Notices*, 28(10):215–230, October 1993. 12

[57] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. 12, 15

[58] Gilad Bracha. Pluggable Type Systems. In *Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, October 2004. ACM. 12

[59] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. 13

[60] Rodrigo B. de Oliveira. The Boo programming language. http://boo.codehaus.org, 2016. 13, 36

[61] Paul Vick. *The Microsoft Visual Basic Language Specification*. Microsoft Corporation, Redmond, Washington, 2007. 13, 25, 27, 36

[62] Stephen Kochan. *Programming in Objective-C 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 13

[63] TIOBE Software. The TIOBE programming community index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2016. 13

[64] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn – robust, concurrent, extensible scripting on the JVM. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 117–136, Orlando, Florida, 25-29 October 2009. ACM. 14

[65] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan 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, pages 377–388, New York, NY, USA, 17-23 January 2010. ACM. 14

[66] Francisco Ortin, Jose Quiroga, Jose M. Redondo, and Miguel Garcia. Attaining multiple dispatch in widespread object-oriented languages. *Dyna*, 81(186):242–250, 2014. 14, 27, 33, 46, 56, 57, 63

[67] Jon Siegel, Dan Frantz, Hal Mirsky, Raghu Hudli, Peter de Jong, Alan Klein, Brent Wilkins, Alex Thomas, Wilf Coles, Sean Baker, and Maurice Balick. *COBRA fundamentals and programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996. 14, 36

[68] Brian Frank and Andy Frank. Fantom, the language formerly known as Fan. http://fantom.org, 2016. 14, 36

[69] Mikhail Vorontsov. Static code compilation in Groovy 2.0. http://java-performance.info/static-code-compilation-groovy-2-0, 2016. 14

[70] L. Peter Deutsch and Allan 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, pages 297–302, New York, NY, USA, 1984. ACM. 15, 20

[71] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA'87, pages 227–242, New York, NY, USA, 1987. ACM. 15

[72] Craig Chambers and David Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Conference on Programming language design and implementation (PLDI)*, pages 146–160, 1989. 15

[73] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991. 15

[74] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, 1996. 15

[75] Google Inc. The V8 JavaScript engine. https://github.com/v8/v8/wiki, 2016. 15

[76] Mozilla. The SpiderMonkey JavaScript engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, 2016. 15

[77] Jose M. Redondo and Francisco Ortin. Efficient support of dynamic inheritance for class- and prototype-based languages. *Journal of Systems and Software*, 86(2):278–301, February 2013. 15, 66, 80

[78] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ'10, pages 10–19, New York, NY, USA, 2010. ACM. 15

[79] Thomas Würthinger, Christian Wimmerb, and Lukas Stadler. Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming*, 78(5):481–498, May 2013. 16

[80] Sun Microsystems OpenJDK. The Da Vinci Machine, a multi-language renaissance for the java virtual machine architecture. http://openjdk.java.net/projects/mlvm, 2016. 16

[81] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM, 1988. 16

[82] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. 16, 47

[83] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1988. 16, 47

[84] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. 16, 46, 47

[85] Andrew W. Appel. Modern compiler implementation in Java, 1998. 16, 51

[86] Free Software Foundation. Gnu compiler colletion (gcc) internals, 2016. 16

[87] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004. 16

[88] Jay Conrod. A tour of V8: Crankshaft, the optimizing compiler, 2013. 16

[89] Mike Pall. LuaJIT 2.0 SSA IR. `http://wiki.luajit.org/SSA-IR-2.0`, 2016. 16

[90] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7, 2008. 16

[91] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179. ACM, 2010. 16

[92] PyPy project. What's new in pypy 2.5.0, 2016. 16

[93] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin 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, pages 18–25, New York, NY, USA, 2009. ACM. 16

[94] Holger Krekel and Armin Rigo. PyPy, architecture overview. In *PyCon conference*, PyCon, 2006. 16

[95] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN notices*, 33(4):17–20, 1998. 16

[96] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, volume 30, pages 13–22. ACM, 1995. 16

[97] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. *SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form*, volume 36. ACM, 2001. 16

[98] Yutaka Matsuno and Atsushi Ohori. A type system equivalent to static single assignment. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 249–260. ACM, 2006. 17

[99] Brian Hackett and Shu-yu 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, pages 239–250, New York, NY, USA, 2012. ACM. 17

[100] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming (ECOOP), Utrecht, The Netherlands*, pages 33–56, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. 17, 62

[101] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp object system: An overview. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 201–220, Paris, France, 1987. 17

[102] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM. 17

[103] David Miller. Clojure CLR. `https://github.com/clojure/clojure-clr`, 2016. 17

[104] The Eclipse project. Xtend, Java 10 today! `http://www.eclipse.org/xtend`, 2016. 17

[105] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman, Boston, Massachusetts, 1996. 17

[106] Christian Grothoff. Walkabout revisited: The runabout. In Luca Cardelli, editor, *17th European Conference on Object Oriented Programming (ECOOP), Darmstadt, Germany*, pages 103–125, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 17

[107] Jens Palsberg and C. Barry Jay. The essence of the Visitor pattern. In *Computer Software and Applications Conference (COMPSAC)*, pages 9–15. IEEE Computer Society, 1998. 17, 18

[108] Fabian Büttner, Oliver Radfelder, Arne Lindow, and Martin Gogolla. Digging into the Visitor pattern. In *IEEE 16th International Conference on Software Engineering and Knowlege Engineering (SEKE), Los Alamitos (CA)*, pages 135–141, 2004. 18

[109] Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of Java extensions: The double-dispatch use-case. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 1409–1413, New York, NY, USA, 2005. ACM. 18

[110] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 130–145, Minneapolis, Minnesota, 25-29 October 2000. ACM. 18

[111] Rémi Forax, Étienne Duris, and Gilles Roussel. A reflective implementation of Java multi-methods. *IEEE Transactions on Software Engineering (TSE)*, 30(12):1055–1071, 2004. 18

[112] Antonio Cunei and Jan Vitek. An efficient and flexible toolkit for composing customized method dispatchers. *Software, Practice and Experience*, 38(1):33–73, 2008. 18

[113] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, Feb 2007. 20

[114] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009. 20

[115] Bill Chiles and Alex Turner. Dynamic Language Runtime. http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512, 2016. 21, 22, 25, 41, 42, 43, 57, 77, 80

[116] Mike Barnett. Microsoft Research Common Compiler Infrastructure. http://research.microsoft.com/en-us/projects/cci/, 2016. 22, 33

[117] Francisco Ortin, Daniel Zapico, J. Baltasar G. Perez-Schofield, and Miguel Garcia. Including both static and dynamic typing in the same programming language. *IET Software*, 4(4):268–282, 2010. 22, 63, 69

[118] ECMA-335. *Common Language Infrastructure (CLI)*. European Computer Manufacturers Association, Geneva, Switzerland, 2016. 25

[119] Jose Quiroga and Francisco 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, 2016. 29, 37

[120] Microsoft Developer Network. Dynamic source code generation and compilation. http://msdn.microsoft.com/en-us/library/650ax5cx(v=vs.110).aspx, 2016. 34

[121] Patrick McEvoy. Brail, a view engine for MonoRail. https://github.com/castleproject/MonoRail/blob/master/MR2/docs/brail.md, 2016. 36

[122] Andrew Davey and Cedric Vivier. Specter framework, a behaviour-driven development framework for .NET and Mono. http://specter.sourceforge.net, 2016. 36

[123] Krzysztof Koźmic. Castle Windsor, mature inversion of control container for .NET and Silverlight. https://github.com/castleproject/Windsor/blob/master/docs/README.md, 2016. 36

[124] Unity Technologies. Unity3D. http://unity3d.com, 2016. 36

[125] NetVed Technologies. Kloudo, the simplest way to get your business organized. http://www.kloudo.com, 2015. 36

[126] SkyFoundry. SkySpark, analytics software for a world of smart devices. http://skyfoundry.com/skyspark, 2016. 36

[127] Thibaut Colar. NetColarDB, ORM features on top of Fantom's SQL package. https://bitbucket.org/tcolar/fantomutils/src/tip/netColarDb, 2016. 36

[128] Python Software Foundation. Pybench benchmark project trunk page. http://svn.python.org/projects/python/trunk/Tools/pybench, 2016. 36

[129] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984. 37

[130] Chandra Krintz. A collection of phoenix-compatible C# benchmarks. http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks, 2016. 37

[131] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007. 37, 38

[132] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. 38

[133] MicrosoftTechnet. Windows server techcenter: Windows performance monitor. http://technet.microsoft.com/en-us/library/cc749249.aspx, 2016. 39

[134] Microsoft. Windows management instrumentation. http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx, 2016. 39

[135] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997. 45

[136] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5):1–164, 1992. 45

[137] Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *European Conference on Object-Oriented Programming*, 2015. 46

[138] Francisco Ortin and Diego Diez. Designing an adaptable heterogeneous abstract machine by means of reflection. *Information & Software Technology*, 47(2):81–94, 2005. 46

[139] Francisco Ortin and Juan Manuel Cueva. Implementing a real computational-environment jump in order to develop a runtime-adaptable reflective platform. *SIGPLAN Notices*, 37(8):35–44, 2002. 46

[140] Ron Cytron and Jeanne Ferrante. *What's in a name?-or-the value of renaming for parallelism detection and storage allocation.* IBM Thomas J. Watson Research Division, 1987. 47

[141] Jose Quiroga and Francisco Ortin. SSA transformations to efficiently support variables with different types in the same scope. http://www.reflection.uniovi.es/stadyn/download/2016/compj, 2016. 52, 56

[142] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119:202–230, June 1995. 55

[143] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, 9-11 June 1993. ACM Press. 55

[144] Francisco Ortin. The StaDyn programming language. http://www.reflection.uniovi.es/stadyn, 2016. 55, 68, 69

[145] Jose Quiroga, Francisco Ortin, David Llewellyn-Jones, and Miguel Garcia. Optimizing runtime performance of hybrid dynamically and statically typed languages for the .NET platform. *Journal of Systems and Software*, 113:114–129, 2016. 57

[146] Microsoft. The .NET compiler platform (Roslyn). https://github.com/dotnet/roslyn, 2016. 57

[147] Mono-Project. The Mono project. http://www.mono-project.com, 2016. 57

[148] ECMA. ECMA-334 standard: C# language specification 4th edition. http://www.ecma-international.org/publications/standards/Ecma-334.htm, 2016. 57

[149] Mads Torgersen. The expression problem revisited four new solutions using generics. In *In Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 123–143. Springer-Verlag, 2004. 65, 67

[150] Francisco Ortin, Luis Vinuesa, and Jose Manuel Felix. The DSAW aspect-oriented software development platform. *International Journal of Software Engineering and Knowledge Engineering*, 21(07):891–929, 2011. 65

[151] Francisco Ortin, Benjamin Lopez, and J. Baltasar G. Perez-Schofield. Separating adaptable persistence attributes through computational reflection. *IEEE Software*, 21(6):41–49, Nov 2004. 66

[152] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit, Brussels, May 1987. 66

[153] Francisco Ortin, Miguel Garcia, Jose M. Redondo, and Jose Quiroga. Achieving multiple dispatch in hybrid statically and dynamically typed languages. In *World Conference on Information Systems and Technologies*, WorldCIST, pages 1–11, 2013. 67, 72

[154] Armin Rigo. Representation-based just-in-time specialization, 2004. 69

[155] Francisco Ortin, Jose Baltasar Garcia Perez-Schofield, and Jose Manuel Redondo. Towards a static type checker for python. In *European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop*, STOP '15, pages 1–2, 2015. 80

[156] Joe Kunk. 10 Questions, 10 Answers on Roslyn. *VisualStudio Magazine*, 03/20/2012, 2012. 80

[157] Patricia Miravet, Ignacio Marin, Francisco Ortin, and Abel Rionda. DIMAG: A framework for automatic generation of mobile applications for multiple platforms. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*, Mobility '09, pages 23:1–23:8, New York, NY, USA, 2009. ACM. 80

[158] Ignacio Marin, Antonio Campos, Jose Quiroga, Patricia Miravet, and Francisco Ortin. Device independence approach for ict-based pftl solutions. In *International Conference on Paperless Freight Transport Logistics (e-Freight)*, 2011. 80

[159] Jose Quiroga, Ignacio Marin, Javier Rodriguez, Diego Berrueta, Nicanor Gutierrez, and Antonio Campos. From UAProf towards a universal device description repository. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (Mobile Computing, Applications, and Services)*, 95:263–282, 2012. 80

[160] Patricia Conde and Francisco Ortin. JINDY: A java library to support invokedynamic. *Computer Science and Information Systems*, 11(1):47–68, 2014. 80