International Doctoral Dissertation

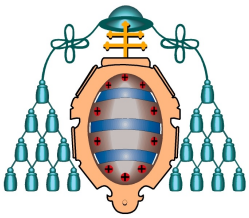
# Improving the Performance and Robustness of Hybrid Statically and Dynamically Typed Programming Languages

**Miguel García Rodríguez**

PhD Supervisor
Dr. Francisco Ortín Soler

Universidad de Oviedo

# Improving the Runtime Performance and Robustness of Hybrid Statically and Dynamically Typing Languages



Miguel García Rodríguez

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

June 2013

# Abstract

Dynamically typed languages have turned out to be suitable for different software development scenarios such as Web engineering, rapid prototyping, and the construction of applications where runtime adaptability is an important issue. In contrast, statically typed languages have undeniable advantages such as early type error detection and more opportunities for compiler optimizations. Since both approaches offer different benefits, hybrid statically and dynamically typed programming languages have emerged, and some statically typed languages have also incorporated dynamic typing capabilities. However, these languages do not perform static type inference on dynamically typed code, lacking the advantages provided for statically typed code.

This PhD dissertation presents *StaDyn*, a hybrid static and dynamic typing language that performs static type inference and type checking of both statically and dynamically typed references. *StaDyn* permits the straightforward development of adaptable software and rapid prototyping, offering early type error detection, improved runtime performance, and direct interoperation between dynamically and statically typed code. The programmer indicates whether high flexibility is required (dynamic typing) or stronger type checking (static) is preferred. It is also possible to combine both approaches, making parts of an application more flexible, whereas the rest of the program maintains its robustness and runtime performance.

The key features of the proposed hybrid static and dynamic type system are a new interpretation of union and intersection types, the combination of syntax-directed and constraint-based type-checking, type inference of implicitly-typed dynamic and static references, and flow-sensitive type-checking. The type system has been implemented as an extension of a real full-fledged programming language such as C# (*StaDyn*), obtaining the benefits of combining the .Net Framework and the proposed type system.

The runtime performance and memory consumption of *StaDyn* have been compared with the most widespread hybrid dynamic and static typing programming languages for the .Net Framework 4. The assessment has been done with an ample set of benchmarks. StaDyn has shown the best performance in all the programs that use at least one dynamic reference, being 150% and 500% faster executing dynamic and hybrid typing code, respectively. When no dynamic reference is used, the only language that performs better than StaDyn (2.5%) is the C# 4.0 production compiler, due to its static optimizations. Be-

sides, *StaDyn* has showed the lowest memory consumption in every scenario.

This dissertation presents how the static type information gathered for dynamically typed references can be used to effectively improve the runtime performance and robustness of hybrid static and dynamic typing languages. Unlike the rest of the analyzed languages, *StaDyn* continues collecting type information of dynamically typed code. The type information inferred by the compiler is used to detect type errors at compile time when dynamic references are used. Moreover, this information is also used to optimize the generated code without any runtime memory cost.

# Keywords

StaDyn, Hybrid Dynamic and Static Typing, Dynamic Languages, Union Types, Intersection Types, Runtime Performance, Robustness, .Net

# Resumen

Los lenguajes con comprobación dinámica de tipos son utilizados comúnmente en diversos escenarios dentro del desarrollo software, tales como la ingeniería Web, el desarrollo rápido de prototipos y la implementación de aquellas aplicaciones para las que la adaptabilidad dinámica sea un requisito importante. Por otro lado, los lenguajes con comprobación estática de tipos ofrecen innegables ventajas como la detección temprana de errores y un mayor número de optimizaciones por parte del compilador. Dado que ambos enfoques ofrecen diferentes beneficios, en los últimos años han surgido lenguajes de programación con sistemas de tipos híbridos, estáticos y dinámicos. Del mismo modo, algunos lenguajes con comprobación estática de tipos también han incorporado la posibilidad de incluir tipos dinámicos en su sistema de tipos. Sin embargo, estos lenguajes no realizan inferencia de tipos alguna en tiempo de compilación sobre código con comprobación dinámica de tipos, perdiendo así parte de su robustez y rendimiento.

En esta tesis presentamos *StaDyn*, un lenguaje con comprobación de tipos híbrida que realiza inferencia de tipos tanto en código declarado estático como dinámico. *StaDyn* facilita el desarrollo de software dinámicamente adaptable y la construcción rápida de prototipos, ofreciendo un mejor rendimiento y detección temprana de errores que los lenguajes con tipado dinámico. Adicionalmente, permite la interoperabilidad entre código con comprobación dinámica y estática de tipos, compartiendo un mismo sistema de tipos. El programador indica cuándo requiere alta flexibilidad (tipado dinámico) o la robustez de un sistema estático de tipos. También es posible combinar ambos enfoques haciendo partes de la aplicación más flexibles, mientras que el resto del programa mantiene su robustez y rendimiento en tiempo de ejecución.

Las principales características del sistema de tipos hibrido propuesto son una nueva interpretación de los tipos unión e intersección, la combinación de sistemas de tipos dirigidos por sintaxis y basados en restricciones, inferencia de tipos concretos frente a abstractos, y un sistema de tipos sensible al contexto. El sistema de tipos ha sido implementado como una extensión de un lenguaje de programación real como C#, obteniendo los beneficios de incorporar una investigación teórica a una plataforma real como el .Net Framework.

Hemos comparado el rendimiento en tiempo de ejecución y el consumo de memoria de *StaDyn* con los lenguajes híbridos existentes sobre el .Net Framework. La evaluación ha sido realizada con un amplio

conjunto de benchmarks. *StaDyn* ha mostrado el mejor rendimiento en todos los programas que usan al menos una referencia dinámica, siendo 150 % más rápido ejecutando código dinámico y un 500 % en código hibrido.

Hemos visto cómo la obtención de información de tipos de referencias dinámicamente tipadas puede ser utilizada para mejorar de forma efectiva el rendimiento y robustez de los lenguajes de programación híbridos. Al contrario que el resto de lenguajes analizados, *StaDyn* continúa obteniendo información de tipos del código declarado como dinámico. La información inferida por el compilador es usada para detectar errores de tipo en tiempo de compilación, incluso sobre referencias dinámicas. Adicionalmente, esta información también es usada para optimizar significativamente el código generado, sin representar ningún coste de memoria en tiempo de ejecución.

## Palabras Clave

StaDyn, Sistemas de Tipos Híbridos Estáticos y Dinámicos, Lenguajes Dinámicos, Tipos Unión, Tipos Intersección, Rendimiento, Robustez, .Net

# Acknowledgements

This work has been partially 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*. It has been also funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation with two projects: *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software* (TIN2008-00276) and *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages* (TIN2011-25978).

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Dynamic languages have recently turned out to be suitable for specific scenarios such as Web development, rapid prototyping, developing systems that interact with data that change unpredictably, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software. Common features of dynamic languages are meta-programming, reflection, mobility, and dynamic reconfiguration and distribution.

Taking Web engineering as an example, Ruby [1] has been successfully used together with the Ruby on Rails framework to create database-backed Web applications [2]. This framework has confirmed the simplicity of implementing the DRY (Don't Repeat Yourself) [3] and the Convention over Configuration [2] principles with this kind of languages. Nowadays, JavaScript [4] is being widely employed to create interactive Web applications with AJAX (Asynchronous JavaScript And XML) [5], while PHP (PHP Hypertext Preprocessor) is one of the most popular languages for developing Web-based views. Python [6] is used for many different purposes; two well-known examples are the Zope application server [7] (a framework for building content management systems, intranets and custom applications) and the Django Web application framework.

Due to the recent success of dynamic languages, statically typed languages (such as Java and C#) are gradually incorporating more dynamic features into their platforms. Taking Java as an example, the reflection API became part of the Java platform with its release 1.1. This API offers introspection services to examine the structures of objects and classes at runtime, plus object creation and method invocation –involving a substantial performance overhead. The dynamic proxy class API was added to Java 1.3. It allows defining a class at runtime that implements any interface, funneling all its method calls to an `InvocationHandler`. The Java `instrument` package (included in Java SE 1.5) provides services that allow Java agents to instrument programs running on the

JVM. This package has been used to implement JAsCo, a fast dynamic AOP platform [8]. Together with other tools such as BCEL [9] and Javassist [10], these agents have also been successfully used in the implementation of application servers such as Spring Java and JBoss, obtaining good runtime performance. The Java Scripting API added to Java 1.6 permits dynamic scripting programs to be executed from, and have access to, the Java platform [11]. Finally, the Java Specification Request 292 [12] has just been incorporated to the Java 1.7 Standard Edition. It adds the new `invokedynamic` opcode to the Java Virtual Machine (JVM) and the `java.lang.invoke` package to the platform [12], making it easier to implement dynamically typed languages in the Java virtual machine. Its main advantage is a user-defined linkage mechanism to postpone method call-sites resolution until runtime.

This trend has also been observed in the .NET platform. The Dynamic Language Runtime (DLR) has been included as part of the .NET framework 4.0 [13]. The DLR adds to the .NET platform a new layer that provides services to facilitate the implementation of dynamic languages over the platform [14]. Moreover, Microsoft has included the new `dynamic` type to C# 4.0, allowing the programmer to write dynamically typed code in a statically typed programming language. With this new characteristic, C# 4.0 offers direct access to code in IronPython, IronRuby and the JavaScript code in Silverlight, making use of the DLR services.

The great flexibility of dynamic languages is, however, counteracted by the limitations derived by the lack of static type checking. This deficiency implies two major drawbacks: no early detection of type errors, and few opportunities to perform runtime performance optimizations. Static typing offers the programmer the detection of type errors at compile time, making possible to fix them immediately rather than discovering them at runtime –when the programmer's efforts might be aimed at some other task, or even after the program has been deployed [15]. Moreover, since runtime adaptability of dynamic languages is mostly implemented with dynamic type systems, runtime type inspection and checking commonly involves a significant performance penalty.

Since both approximations offer different benefits, there have been former works on providing both typing approaches in the same language (see Chapter 2). Meijer and Drayton maintained that instead of providing programmers with a black or white choice between static or dynamic typing, it could be useful to strive for softer type systems [16]. Static typing allows earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance. Dynamic typing languages provide a solution to a kind of computational incompleteness inherent to statically-typed languages, offering, for example, storage of persistent data, inter-process communication, dynamic program behavior customization, or generative programming [17]. Therefore, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [18]. That is, *static typing where possible, dynamic typing when needed* [16].

This dissertation is aimed at breaking the programmers' black or white choice

between static and dynamic typing. We propose a programming language, called *StaDyn* [19], that supports both static and dynamic typing. This programming language combines the robustness and efficiency of a statically typed language with the flexibility and adaptiveness of dynamic typing. The programmer specifies those parts of the code where dynamic adaptability is required and those where common static typing rules should be applied. This separation facilitates turning rapidly developed prototypes into a final robust and efficient program. It is also possible to combine both approaches, making parts of an application more flexible, whereas the rest of the program maintains its robustness and runtime performance.

## 1.2 Contributions

These are the major contributions of this PhD dissertation:

1. A hybrid static and dynamic type system. There are previous works aimed at supporting static and dynamic typing in the very same language (§ 2.2), providing interoperability between these two kinds of code. We propose a new interpretation of union and intersection types (Chapter 4) to gather more type information than the existing approaches. This type information is used to provide better interoperation between statically and dynamically typed code.

2. Compile-time error detection of dynamically typed code. Existing approaches of hybrid static and dynamic typing languages postpone *all* the type checking of dynamically typed code until runtime. Our compiler gathers type information that allows detecting many type errors at compile time, even when dynamic references are used.

3. Runtime performance improvement. The type information gathered by the compiler can also be used to optimize the generated code. In particular, dynamic type checking is reduced and a consequent runtime performance improvement is achieved.

4. Separation of the *dynamism* concern. References are not explicitly declared as static or dynamic. They follow the same syntax and their dynamism is separated and controlled by the IDE. Consequently, the compiler provides different modes of compilation, processing the source language in different ways depending on the requirements of the application. With this approach, rapidly developed prototypes can be easily converted into robust production applications, minimizing the changes in the source code. Similarly, it facilitates making parts of an application more flexible, whereas the rest of the program maintains its robustness and runtime performance.

5. Implementation of a full-fledged programming language. The proposed type system is included in a real full-fledged programming language such as C#. C# has been extended with our type system, obtaining all the benefits of combining the proposed type system and the .NET Framework.

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

## 1.3 Structure of the Document

This dissertation is structured as follows. The next chapter presents related work. Chapter 3 provides an overview of the StaDyn programming language. Chapter 4 formally describes the abstract syntax (§ 4.2) and the type system (§ 4.3) of the StaDyn core. Chapter 5 presents its erasure semantics by translating it into C#. An evaluation of runtime performance and memory consumption is detailed in Chapter 6. Chapter 7 presents the conclusions and future work.

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

# Chapter 2

# Related Work

Since both dynamic and static typing offer important benefits, there have been previous approaches aimed at obtaining the advantages of both, following the philosophy of *static typing where possible, dynamic typing when needed* [16]. The existing language implementations are first described, and then the theoretical research. Finally, existing approaches to perform static typing in dynamically typed languages are also described.

## 2.1 Hybrid Programming Language Implementations

*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 [20]. It retains the basic Smalltalk syntax and semantics [21], 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 [22]. 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 [23]. 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 propsed 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 [24]. 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 [25]. 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. Visual Basic for .NET does not separate the dynamism concern: it forces the programmer to explicitly state in the source code which references are static and which ones are dynamic. Dynamic references are declared using the `Dim` reserved word and the variable identifier; `As` and the variable type are not stated. Function parameters and class fields can also be declared as dynamic.

*C#* includes in its version 4.0 the support of dynamically typed objects [26]. A new `dynamic` type has been added to the programming language. The compiler performs no static type checking over the `dynamic` references, postponing all the type verifications at runtime. However, the typing rules defined by the language are checked at runtime (e.g., method overload [27]) [28]. This new feature is supported by the Dynamic Language Runtime (DLR) [14], a new layer over the CLI (Common Language Infrastructure) that provides language services for several different dynamic languages [13]. The main objective of the new `dynamic` type is to offer direct access to dynamically typed code in IronPython, IronRuby and the JavaScript code in Silverlight.

*Objective-C* is a general-purpose object-oriented extension of the C programming language [29]. 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 [30], in March 2013 Objective-C was the $3^{rd}$ 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, a `NSInvalidArgumentException` is raised. Besides, Objective-C also provides an `id` type to postpone the static type checking until runtime.

*Cobra* is another hybrid static and dynamic typing programming language for the .NET platform [31]. 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 [32]. 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.

## 2.2 Hybrid Static and Dynamic Type Systems

One of the first works aimed at formalizing hybrid static and dynamic type system was *Soft Typing* [33]. Its main objective is to combine the advantages of static typing with the flexibility of dynamic typing. The soft type system is defined for a functional core of both ML and Scheme. Soft typing does not control which parts in a program are statically checked, and the static type information is not used to optimize the generated code either. They developed an algorithm as an extension of Hindley-Milner unification-based typing [34], including union types and recursive types.

The approach proposed by Abadi, Cardelli, Pierce and Plotkin [17] was the first one to add a new `Dynamic` type in order to support dynamic typing in a statically typed language. They extended a typed lambda calculus with this new type. `Dynamic` values are pairs of a value $v$ and a type tag $T$, where $v$ has the type denoted by $T$. The type tag $T$ is used in two conversion operations: `dynamic` to package $v$ and $T$, and `typecase` to inspect the $T$ type tag given a `Dynamic` expression. The resulting language results in a a verbose code deeply dependent on its dynamism.

The works of *Quasi-Static Typing* [35], *Hybrid Typing* [36] and *Gradual Typing* [37] perform implicit conversions between dynamic and static code. Quasi-static and hybrid typing modify the subtyping relation to consider static and dynamic typing interaction. However, gradual typing provides this interaction

by replacing type equality with type *consistency* [38]. The defined the $\mathbf{Ob}^{?}_{<:}$ object calculus showing how the type consistency relation can be naturally combined with subtyping. Gradual typing also identified unification-based constraint resolution as a suitable approach to integrate both dynamic and static typing [38]. However, with gradual typing a dynamic type is always implicitly converted into static without any static type-checking, because type inference is not performed over dynamic references. Gradual typing is the closest approach to the hybrid type system included in C# 4.0, where the main difference is that C# employs a nominal type system instead of a structural one [28].

The work developed by Wrigstad *et al.* allows the combination of dynamic and static typing in the *Thorn* programming language [39]. Thorn offers `like` types, an intermediate point between static and dynamic types [40]. 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 facilitate initial prototypes to smoothly evolve into efficient and robust applications. `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 types (`dyn`) in the same programming language [40].

Although the *Just* programming language [41] does not combine dynamic and static typing, it added implicit type reconstruction to an explicitly typed language such as Java to obtain statically checked duck typing. The combination of syntax-directed and constraint-based type-checking allows the programmer to write generic code without defining class hierarchies [42]. One major limitation of this approach is that they do not consider methods that generate constraints (polymorphic methods) to invoke other polymorphic methods.

## 2.3 Static Typing for Dynamically Typed Languages

There are also some works aimed at performing static type inference of dynamically typed languages to discover type errors before program execution. *Diamondback Ruby* (DRuby) is a tool that blends the Ruby dynamic type system with a static typing discipline [43]. DRuby was applied to a suite of benchmarks, finding several bugs that would cause run-time type errors. When possible, DRuby infers static types to discover type errors in Ruby programs. In many cases, the DRuby programmer must annotate programs with types in order to obtain compile-time type errors. Since DRuby trusts annotations to be correct, improperly annotated code may cause run-time type errors, and these errors may be misleading. DRuby does not use the statically inferred type information to optimize the generated code.

Anderson, Giannini and Drossopoulou formalized a subset of JavaScript ($JS_0$), defining a structural type inference algorithm that is sound with respect to a type system [44]. Therefore, programmers can benefit from the safety offered by the

type system, without the need to write explicitly types in their programs. Different features of the JavaScript programming language such as dynamic removal of members or dynamic code evaluation are not supported. As with DRuby, the type information gathered by the $JS_0$ is not used to perform code optimizations.

$\lambda_{JS}$ is a JavaScript core calculus structured as a small-step operational semantics [45]. The specified semantics was implemented with PLT Redex [46] and used to test $\lambda_{JS}$ for safety. A desugaring process was implemented and applied to a collection of JavaScript test suites, showing that they can be translated into $\lambda_{JS}$ (i.e., the core is adecuate to represent JavaScript). The formalization of the semantics was used to define a type system that disallows access to XMLHttpRequest, proving its safety.

Another approach to add static typing to a dynamically typed language such as JavaScript is using refinement and dependent types. The System D core calculus merges syntax and semantic reasoning into a single powerful mechanism of *nested refinement types*, wherein the typing relation is itself a predicate in the refinement logic [47]. System D coordinates SMT-based logical implication and syntactic subtyping to type-check dynamic language features such as runtime type tests, value-indexed dictionaries, polymorphism, and higher-order functions. System D permits dependent structural subtyping and a form of bounded quantification. Although type annotations are optional, the type system does not always infer them. The System D calculus has been scaled up to Dependent JavaScript (DJS), an explicitly typed dialect JavaScript [48]. DJS extends System D with imperative updates, prototype inheritance and arrays. The DJS type system is expressive enough to reason about a variety of JavaScript idioms found in small examples drawn from several sources, including the SunSpider benchmark suite. The type information is not used for performing optimizations to improve runtime performance of program execution.

# Chapter 3

# Overview of the *StaDyn* Programming Language

In this section an overview of the *StaDyn* programming language is described. Chapter 4 details its type system and Chapter 5 describes its semantics. The syntax of *StaDyn* is depicted in Appendix A.

## 3.1 Static and Dynamic Typing

### 3.1.1 Statically Typed Languages

A language is said to be *safe* if it produces no execution errors that go unnoticed and later cause arbitrary behavior [49], following the notion that well-typed programs should not go *wrong* (i.e., reach a *stuck* state on its execution) [15]. Statically typed languages ensure type safety of programs by means of static type systems. However, these type systems do not compile some expressions that do not produce any type error at runtime (e.g., in .NET and Java it is not possible to pass the `m` message to an `Object` reference, although the object actually implements a public `m` method). This happens because their static type systems require ensuring that compiled expressions do not generate any type error at runtime. Figure 3.1 illustrates this situation (the *not compilable and no runtime type error* region).

Static typing is focused on making sure that no type error is produced at runtime. This is the reason why languages with static typing employ a pessimistic policy regarding to program compilation. This pessimism causes compilation errors in programs that do not produce any runtime error. C# code shown in Figure 3.2 is an example program of this scenario. Although the program does not produce any error at runtime, the C# type system does not recognize it as a valid compilable program, showing a compilation error.

At the same time, static languages also permit the execution of programs that might cause an erroneous execution (e.g. array index out of bounds or null pointer

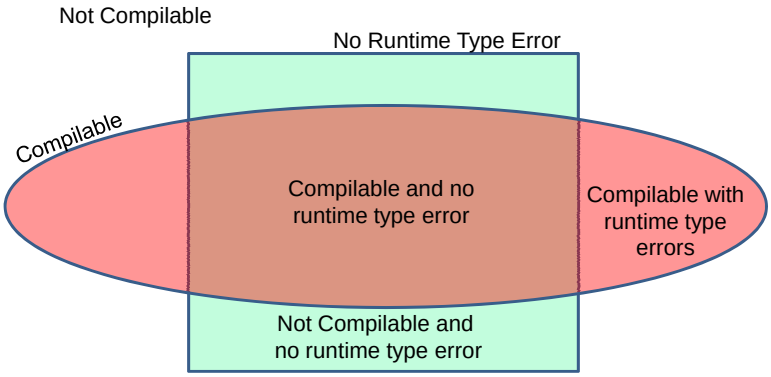



Figure 3.1: Program execution of statically typed languages.

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

Figure 3.2: Not compilable C# program that would not produce any runtime error.

access). This scenario is represented by the *compilable with runtime type errors* region in Figure 3.1.

### 3.1.2 Dynamically Typed Languages

The approach of dynamic languages is the opposite one. Instead of making sure that all valid expressions will be executed without any type error, they make *all* the syntactically valid programs compilable (Figure 3.3). This is a too optimistic approach that causes a high number of runtime type errors that might have been detected at compile time. This situation, where dynamic languages commonly throw runtime exceptions, is what is represented in Figure 3.3 as *compilable with runtime type errors*. This approach causes too many runtime type errors, compiling programs that might have been identified as erroneous statically. The C# source code in Figure 3.4 is an example of this too optimistic approach. This erroneous program is compiled without any error (the `dynamic` type in C# 4.0 postpones type checking until runtime), although a static type system might have detected the error before its execution.

### 3.1.3 Supporting both Approaches

The *StaDyn* programming language performs type inference at compile type, minimizing the *compilable with runtime type errors* region of dynamic languages (Figure 3.3) and the *not compilable and no runtime type error* area of static

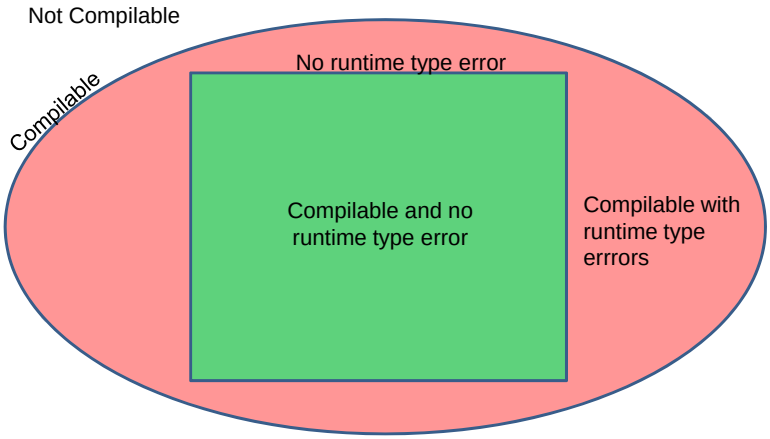



Figure 3.3: Program execution of dynamically typed languages.

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

Figure 3.4: Compilable dynamically typed C# program that generates runtime type errors.

languages (Figure 3.1). Consequently, *StaDyn* detects the compilation error of the dynamic program shown in Figure 3.4 (that C# does not detect) and compiles the valid statically typed code in Figure 3.2 (that C# does not compile) –using the appropriate *StaDyn* syntax.

For both typing approaches, the very same programming language is used, letting the programmer move from an optimistic, flexible and rapid development (dynamic) scenario to a more robust and efficient one (static). This transition can be done without changing the application source code, only modifying the compiler settings. Therefore, the *dynamism* concern (i.e., flexibility vs. robustness and performance) is separated from the functional requirements of the application (its source code).

## 3.2 The *StaDyn* Programming Language

The features of the *StaDyn* programming language are now presented, identifying (but not detailing) the techniques employed. A formal description of its type system is depicted in Chapter 4 and its semantics is presented in Chapter 5. A summary of the most relevant implementation issues is presented in § 3.3.

The *StaDyn* programming language is an extension of C# 3.0 [50]. Although the work presented in this dissertation could be applied to any object-oriented statically-typed programming language, C# has been used to extend the behavior of its implicitly typed local references. 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# 3.0 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# 3.0.

The dynamism of `var` references is placed in a separate file (an XML document). The programmer does not need to manipulate these XML documents directly, leaving this task to the *StaDyn* IDE [51]. 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). Since the dynamism concern is not explicitly stated in the source code, *StaDyn* facilitates the conversion of dynamic references into static ones, and vice versa. This separation facilitates the process of turning rapidly developed prototypes into final robust and efficient applications. It is also possible to make parts of an application more adaptable, maintaining the robustness and runtime performance of the rest of the program.

### 3.2.1 Multiple Types in the Same Scope

Existing statically typed languages force a variable of type $T$ to have the same type $T$ within the scope in which it is bound to a value. Even languages with static type inference (type reconstruction) such as ML [52] or Haskell [53] do not permit the assignment of different types to the same polymorphic reference in the same scope.

In contrast, dynamic languages provide the use of one reference to hold different types in the same scope. This is easily implemented at runtime with a dynamic type system. However, *StaDyn* offers this feature statically, taking into account the concrete type of each reference. The *StaDyn* program shown in Figure 3.5 is an example of this capability. The `number` reference has different types in the same scope. It is initially set to a string, and an `double` is later assigned to it. The static type inference mechanism implemented in *StaDyn* detects the error in the last line of code. Moreover, a better runtime performance is obtained because it is not necessary to use reflection to discover types at runtime (see Chapter 6).

In order to obtain this behavior, an implicit parametric polymorphic type system [54] that provides type reconstruction when a `var` reference is used has been developed. The Hindley-Milner type inference algorithm has been implemented to infer the types of local variables [34]. This algorithm has been modified to perform type reconstruction of `var` parameters and attributes (fields) –described in §§ 3.2.4 and 3.2.5.

The unification algorithm used in the Hindley-Milner type system provides parametric polymorphism, but it forces a reference to have the same static type

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

Figure 3.5: A reference with different types in the same scope.

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

Figure 3.6: Corresponding program after the SSA transformation.

in the scope it has been declared. To overcome this drawback a version of the SSA (*Single Static Assignment*) algorithm [55] has been developed. This algorithm guarantees that every reference is assigned exactly once by means of creating new temporary references. Since type inference is performed after the SSA algorithm, we have implemented it as a previous AST (*Abstract Syntax Tree*) transformation. The implementation of this algorithm follows the *Visitor* design pattern [56].

Figure 3.6 shows the corresponding program after applying the AST transformation to the source code in Figure 3.5. The AST represented by the source code in Figure 3.6 is the actual input to the type inference system. Each `number` reference is inferred to a single static type.

### 3.2.2 Duck Typing

Duck typing[1] [1] is a property of dynamic languages that means that an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether those objects have a related inheritance hierarchy or not. Duck typing is a powerful feature offered by most dynamic languages.

There exist statically typed programming languages such as Scala [57] or OCaml [58] that offer structural typing, providing part of the benefits of duck

[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 3.7: Static duck typing.

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 [59]. Principal types are more restrictive than duck typing, because they do not consider all the possible (concrete) values a variable may hold.

The *StaDyn* programming language offers *static* duck typing. The benefit provided by *StaDyn* is not only that it supports (implicit) duck typing, but also that it is provided statically. Whenever a `var` reference 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.

Static duck typing has been implemented, making the static type system of *StaDyn* *flow-sensitive*. This means that it takes into account the flow context of each `var` reference. It gathers *concrete* type information (opposite to classic *abstract* type systems) [60] knowing all the possible types a `var` reference may hold. Instead of declaring a reference with an abstract type that embraces all the possible concrete values, the compiler infers the union of all possible concrete types a `var` reference may point to. Notice that different types depending on flow context could be inferred for the same reference, using the type inference mechanism mentioned above.

Code in Figure 3.7 shows this feature. `reference` may point to either an `StringBuilder` or a `String` object. Both objects have the `Length` property and, therefore, it is statically safe to access to this property. It is not necessary to define a common interface or class to pass this message. Since the type inference system is *flow-sensitive* and uses *concrete* types, the programmer obtains a safe static duck-typing system.

The key technique we have used to obtain this concrete-type flow-sensitiveness is *union types* [61]. Concrete types are first obtained by the abovementioned unification algorithm (applied in assignments and method calls). Whenever a branch is detected, a union type is created with all the possible concrete types inferred. Type checking of union types depends on the dynamism concern (next section).

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

Figure 3.8: Static `var` reference.

### 3.2.3 Separation of the Dynamism Concern

*StaDyn* permits the use of both static and dynamic `var` references. Depending on their dynamism concern, type checking and type inference would be more pessimistic (static) or optimistic (dynamic), but the dynamic semantics of the programming language is not changed (i.e., program execution does not depend on its dynamism). This idea follows the *pluggable* type system approach described in [22] and [62]. Since the dynamism concern is not explicitly stated in the source code, it is possible to customize the trade-off between runtime flexibility of dynamic typing, and runtime performance and robustness of static typing. It is not necessary to modify the application source code to change its dynamism. Therefore, dynamic references could be converted into static ones, and vice versa, without changing the application source code.

The source code in Figure 3.8 defines a `g` method, where `reference` may point to a `StringBuilder`, `String` or `Exception` object. If we want to compile this code to rapidly develop a prototype, we can pass the compiler the *everything-Dynamic* option. However, although we are compiling the code in the optimistic configuration, the compiler shows the following error message:

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

The error is produced because no public `Lenght` property (it has been misspelled) is implemented in the `String`, `StringBuffer` or `Exception` classes. This message shows how type-checking is performed at compile time, even in dynamic scenarios, providing early type error detection. This feature improves the way most dynamic languages work. For example, in the erroneous program in Figure 3.4 that C# compiles without any error, *StaDyn* detects the error at compile time.

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

`var` references are used. The dynamism concern implies a modification of type checking over union types. If the implicitly typed `var` reference inferred with a union type is static, type checking is performed over all its possible concrete types. However, if the reference is dynamic, type checking is performed over those concrete types that do not produce a type error; if none exists, then a type error is shown –this semantics is formalized in § 4.3.4.

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

Actually, the programmer does not need to set all the `var` references in a compilation unit as dynamic. It is possible to specify the dynamism of each single reference by modifying the corresponding XML file. As discussed above, the programmer does not manipulate these XML documents directly, leaving this task to the *StaDyn* IDE. Each *StaDyn* source code file may have a corresponding XML document specifying its dynamism concern [51].

The generated `g` function program will not produce any runtime type error because the random number that is generated will always be 1 or 2. However, if the programmer, once the prototype has been tested, wants to compile the application with using static type system, he or she may use the *everythingStatic* option. When this option is used, no XML dynamism file is analyzed and static typing is performed over every `var` reference in that compilation unit. In this case, the compilation of the `g` method will produce an error message 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.

### 3.2.4 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 [54] more straightforward for the programmer than the one offered by Java, C# (F-bounded) and C++ (unbounded) [63].

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 3.9. Both methods require the parameter to implement a specific method, returning its value. In the `getString` method, any object could be

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

Figure 3.9: Implicitly typed parameters.

passed as a parameter because every object accepts the `ToString` message. In the `upper` method, the parameter should be any object capable of responding to the `ToUpper` message. Depending on the type of the actual parameter, the *StaDyn* compiler generates the corresponding compilation error.

For this purpose the *StaDyn* type system has been enhanced to be constraint-based [64]. Types of methods in our object-oriented language have an ordered set of constraints specifying the set of restrictions that must be fulfilled by the parameters. 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 ($\alpha$) should implement a public `ToUpper` method with no parameters, and the type returned by `ToUpper` ($\beta$) 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 [65]).

### 3.2.5 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 3.10. The `Node` class can hold any `data` of any type. Each time the `setData` method is called, the new concrete type of the parameter is saved as the `data` field type. By using this mechanism, the two lines with comments report compilation errors. This coding style is polymorphic and it is more legible that the parametric polymorphism used in C++ and much more straightforward than the F-bounded polymorphism offered by Java and C#. At the same time, runtime performance is equivalent to explicit type declaration (see Chapter 6). Since the possible concrete types of `var` references are known at compile time, the compiler has more opportunities to optimize the generated code, improving runtime performance.

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 modified each time the `setData` method (and the constructor) is invoked. This

```
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 3.10: Implicitly typed attributes.

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 has been added to the type system (Chapter 4). Each time a value is assigned to a `var` attribute, an assignment constraint is added to the method being analyzed. This constraint postpones the unification of the concrete type of the attribute to be performed later, when an actual object is used in the invocation. Therefore, the unification algorithm is used to type-check method invocation expressions, using the concrete type of the actual object (a detailed description of the unification algorithm can be consulted in [65]).

### 3.2.6 Interaction between Static and Dynamic Types

*StaDyn* performs static type checking of both dynamic and static `var` references. This makes possible the combination of static and dynamic code in the same application, because the compiler gathers type information in both scenarios. The source code in Figure 3.11 uses the `getString` and `upper` methods of Figure 3.9. `reference` may point to a string or integer. Therefore, it is safe to invoke to the `getString` method, but a dynamic type error might be obtained when the `upper` method is called.

Since type-checking of dynamic and static code is different, it is necessary to describe interoperation between both types of references. In case `reference` had been set as a dynamic, the question of whether or not it could have been passed as an argument to the `upper` or `getString` methods (Figure 3.9) arises. That is, how optimistic (dynamic) code could interoperate with pessimistic (static) one. An example is shown in Figure 3.11.

The first invocation is correct regardless of the dynamism of `parameter`. Being either optimistic or pessimistic, the argument responds to the `ToString` method correctly. However, it is not the same in the second scenario. By default, a compilation error is obtained, because the `reference` parameter is static and it may point to an integer, which does not implement a public `ToUpper` method.

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

Figure 3.11: Dynamic and static code interoperation.

```
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 3.12: Alias analysis.

However, if the parameter of the `upper` method is set as dynamic, the compilation will succeed.

This type checking is obtained taking into consideration the dynamism of references in the subtyping relation of the language (§ 4.3.4). A dynamic reference is a subtype of a static one when all the concrete types of the dynamic reference promote to the static one. Promotion of static references to dynamic ones is more flexible: static references should fulfill at least one constraint from the set of alternatives.

### 3.2.7 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* [66]. Two references are aliased if they point to the same object. Although alias analysis is mainly used for optimizations, we have used it to know the concrete types of the objects a reference may point to.

Code in Figure 3.12 uses the `Node` class previously shown in Figure 3.10. 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 is type-based (uses type information to decide alias) [67], inter-procedural (makes use of inter-procedural flow information) [66], context-sensitive (differentiates between different calls to the same method) [68], and may-alias (detects all the objects a reference *may* point to; opposite to *must* point to) [69].

## 3.3 Implementation

The *StaDyn* programming language has been implemented over the .Net Framework 4.0 platform, using C# 4.0. Our compiler is a multiple-pass language processor that follows the *Pipes and Filters* architectural pattern [70]. We have used the AntLR language processor tool to implement lexical and syntactic analysis [71] (Appendix A). Abstract Syntax Trees (ASTs) have been implemented following the *Composite* design pattern [56] and each pass over the AST implements the *Visitor* design pattern [56].

Currently we have developed the following AST visits: two visitors for the SSA algorithm; two visitors to load types into the types table; one visitor for symbol identification [72] and another one for type inference; and two visitors to generate code. Once the final compiler is finished (see § 7.1), the number of AST visits will be reduced to optimize the implementation. The type system has been implemented following the guidelines described in [73], and the code generation module follows the design in [74].

We generate .Net intermediate language and then assemble it to produce the binaries. At present, we use the CLR 2.0 as the unique compiler's back-end. However, we have designed the code generator module following the *Parallel Hierarchies* design pattern [74, 75] to add both the DLR (Dynamic Language Runtime) [14] and the ЯRotor [76] back-ends in the future (§ 7.1).

# Chapter 4

# The Hybrid Static and Dynamic Type System

After presenting an overview of the *StaDyn* programming language, this chapter reduces *StaDyn* to its minimal core in order to formalize its type system. The key features of the type system are a new interpretation of union and intersection types [77], the combination of syntax-directed and constraint-based type-checking, type inference of implicitly-typed dynamic and static references, and flow-sensitive type checking [78]. Chapter 5 specifies the erasure semantics of the *StaDyn* minimal core.

## 4.1 Informal Specification of the *StaDyn* Core

The *StaDyn* core specifies the minimal language features that allow the formalization of its static and dynamic semantics. These features are functions, objects (without methods), arrays, assignments, and integer and boolean expressions. Type variables are also included to offer implicit type reconstruction by means of extending the usage of the `var` reserved word added in C# 3.0 [50]. In the *StaDyn* core, `var` references can be set as static (by default) or dynamic, modifying how type checking is performed.

Figure 4.1 shows an example *StaDyn* core program that uses both static and dynamic typing. In the *StaDyn* core, dynamic `var` references are explicitly declared with the `dyn` reserved word. The major benefit of using *StaDyn* is that static type checking is performed even over dynamic references. For instance, the `positiveX` function statically checks that each `data` object in `list` provides a public `x` field. Unlike C#, the *StaDyn* core prompts a compilation error in line 70 (function invocation in Figure 4.1), if code in line 69 is commented out. The error indicates that one of the elements in the list (the integer) does not provide the `x` message. In contrast, C# 4.0 compiles the code and the error is produced at runtime [78].

Our compiler gathers type information at compile time in order to perform static type checking over dynamic references. One of the elements we have used

```
01: var createNode(var data, var next) {
02:   return new { data=data, next=next};
03: }
04: var createPoint(int dimensions, int x,int y,int z) {
05:   var point;
06:   if (dimensions == 2)
07:     point = new {x=x, y=y, dimensions=dimensions};
08:   else
09:     point = new {x=x, y=y, z=z, dimensions=3};
10:   return point;
11: }
12: var createPoints(int number) {
13:   int i;
14:   var list, point;
15:   i = 1;
16:   point = createPoint(3,0,0,0); // Last node (null)
17:   list = createNode(point, 0);
18:   while (i < number) {
19:     point = createPoint(i%2 + 2, number/2-i, i, i);
20:     list = createNode(point, list);
21:     i = i+1;
22:   }
23:   return list;
24: }
25: var positiveX(var list, int n) {
26:   int i;
27:   var l, result;
28:   result = i = 0;
29:   l = list;
30:   while (i < n) {
31:     if (l.data.x >= 0)
32:       result = createNode(l.data, result);
33:     l = l.next;
34:     i = i+1;
35:   }
36:   return result;
37: }
38: int distance3D(/*dyn*/ var point) {
39:   int value;
40:   value = 2147483647;
41:   // point.center; // Compiler error
42:   if (point.dimensions == 3)
43:     value = point.x*point.x + point.y*point.y
                              + point.z*point.z;
44:   return value;
45: }
46: var closestToOrigin3D(var list, int n) {
47:   int i, minDistance;
48:   var l, point3D;
49:   minDistance = 2147483647;
50:   l = list;
51:   i = 0;
52:   while (i < n) {
53:     if (distance3D(l.data) < minDistance) {
54:       minDistance = distance3D(l.data);
55:       point3D = l.data;
56:     }
57:     l = l.next;
58:     i = i+1;
59:   }
60:   return point3D;
61: }
62:
63:
64: void main() {
65:   int i, numberOfPoints;
66:   var list, positive, point;
67:   numberOfPoints = 10;
68:   list = createPoints(numberOfPoints);
69:   // list.data = 3; // Compiler error
70:   positive = positiveX(list, numberOfPoints);
71:   point = closestToOrigin3D(list, numberOfPoints);
72: }
```

Figure 4.1: Example coded in the minimal core of *StaDyn*.

for this purpose is union types [61]. 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 $T_2$ [79], representing the least upper bound of $T_1$ and $T_2$ [80]. A union type holds all the possible types a reference may have. The set of operations (e.g., addition, field access, assignment, invocation or indexing) that can be applied to a union type are those accepted by every type in the union type (inference rules of static union types are S-SUnionL and S-SUnionR in Figure 4.10). Union types were already included in object-oriented languages, in type systems where they were explicitly declared [81] or inferred from implicitly typed references [82].

In our example, the type inferred for `list` in line 68 (Figure 4.1) is a list of $\{x{:}\texttt{int}, y{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\} \vee \{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\}$, meaning two or three dimensional points. In the invocation of the `positiveX` function (line 70), it is statically checked that the argument is a list of objects that provide an `x` field. Since this condition is fulfilled statically, the program is compiled without errors (and the static type information is used to optimize its execution). However, if line 69 is uncomment, an error message will be shown.

The `closestToOrigin3D` function imposes more constraints to the `list` parameter. Objects in `list` must provide the `dimensions`, `x`, `y` and `z` fields because of the invocation to `distance3D`. We represent these constraints by means of intersection types [61]. $T_1 \wedge T_2$ denotes all the values belonging to both $T_1$ and $T_2$ [79], representing the greatest lower bound of $T_1$ and $T_2$ [80]. A type promotes to a static intersection type only if it is a subtype of all the types collected by the intersection type (S-SInterR rule in Figure 4.10).

In our example, the argument `list` of the `closestToOrigin3D` function must be a list of $X$ type, being $X \leq [\texttt{dimensions}{:}X_1] \wedge [\texttt{x}{:}X_2] \wedge [\texttt{y}{:}X_3] \wedge [\texttt{z}{:}X_4]$ (an object with all these four fields). However, the invocation in line 71 produces a compilation error because `list` holds a union type of both two and three dimensional points, and the former do not provide the `z` field. Our approach is to make the type system more lenient, without renouncing static type checking. The `point` parameter of the `distance3D` function can be declared as dynamic (uncommenting the `dyn` type qualification in line 38 of Figure 4.1). In this case, the promotion to intersection types is more permissive: the argument should be a subtype of at least one of the types in the intersection type (rule S-DInterR in Figure 4.10). Then, the program would generate no error because both types of points offer a public `dimensions` field. This relaxation of the subtyping relation when references are declared as dynamic is also applied to union types (S-DUnionL): the promotion should be fulfilled by at least one of the types in the union type.

It is worth noting that type checking is still performed at compile type even when the programmer uses dynamic references. As an example, if line 41 in Figure 4.1 is uncommented, an error is shown even though `point` has been declared as dynamic (the `center` message is not accepted by either of the two possible points); whereas C# compiles the code, producing the type error at runtime [78]. This first example informally shows the objective of *StaDyn*: to offer both the flexibility of dynamic typing and the robustness and efficiency of static typing.

## 4.2 Abstract Syntax of the *StaDyn* core

After an informal overview of the aim of the *StaDyn* core programming language, its syntax and type system are now described –Chapter 5 describes its erasure semantics by translating it into to C#. The first part of Figure 4.2 shows the abstract syntax of the minimal core (the second and third parts are, respectively, types and constraints). EBNF is used, where $^+$ means repetition of at least one element, $^*$ matches zero or more occurrences, $^?$ means optionally matching the previous element, and | represents alternative.

A program ($P$) is composed of a sequence of function declarations ($F^*$) followed by the local variable declarations ($D^*$) and statements ($S^+$) of the main function. Although the programmer may use the `return` statement the same way as in C#, it could only be placed as the last statement of the abstract syntax. This transformation is performed by the parser to facilitate type inference in conditional and iterative control structures (§ 4.3.6).

A statement can be any expression (including assignments), a conditional statement (`if`), or an iterative one (`while`). Since assignments are expressions, the parser annotates every expression node of the Abstract Syntax Tree (AST) with a boolean value (`lhsAssign`) that reveals whether or not it is a direct left child of an assignment. This value will be used by the type system for type-checking purposes.

| | | | |
|---|---|---|---|
| Program | $P$ | ::= | $F^*\ D^*\ S^+$ |
| Function | $F$ | ::= | $(\texttt{void} \mid ST)\ id\ \texttt{(}\ (ST\ id)^*\ \texttt{)}\ D^*\ S^*\ R^?$ |
| Declaration | $D$ | ::= | $ST\ id$ |
| Statement | $S$ | ::= | $E \mid \texttt{if}\ E\ S^+\ S^* \mid \texttt{while}\ E\ S^*$ |
| Return | $R$ | ::= | $\texttt{return}\ E$ |
| Expression | $E$ | ::= | $id \mid id\ \texttt{(}\ E^*\ \texttt{)} \mid E \oplus E \mid E \otimes E \mid E \,\#\, E \mid E\ \texttt{=}\ E \mid$ $E\ \texttt{.}\ id \mid E\ \texttt{[}\ E\ \texttt{]} \mid \texttt{new}\ ST\ \texttt{[}\ E\ \texttt{]}(\texttt{[}\ \texttt{]})^* \mid$ $\texttt{new}\ \{(\ id\ \texttt{=}\ E\ )^*\} \mid \texttt{true} \mid \texttt{false} \mid IntLiteral$ |
| Syntax types | $ST$ | ::= | $\texttt{int} \mid \texttt{bool} \mid \texttt{Array(}\ ST\ \texttt{)} \mid \{(id{:}ST)^*\} \mid TV$ |
| Type variable | $TV$ | ::= | $Dyn^?\ X_i$ |
| Dynamism | $Dyn$ | ::= | $\texttt{sta} \mid \texttt{dyn}$ |
| Internal types | $T$ | ::= | $ST \mid \texttt{[}\ (id : T)^*\ \texttt{]} \mid ST \times \ldots \times ST \to ST \parallel C^* \mid$ $\{(id : T)^*\} \mid \texttt{Array(}T\texttt{)} \mid$ $Dyn^?\ T \vee \ldots \vee T \mid Dyn^?\ T \wedge \ldots \wedge T$ |
| Constraints | $C$ | ::= | $IT \leq T \mid TV \leftarrow T$ |

Figure 4.2: Abstract Syntax of the *StaDyn* minimal core.

The $\oplus$ operator represents arithmetic operations, $\otimes$ logical ones and $\#$ symbolizes relational operators. Objects are created following the syntax of the C# 3.0 feature of anonymous types [50]: between curly braces, there is listed a sequence of field identifiers followed by the assignment operator and an expression representing their initial values. The `new` expression for arrays creates one-dimensional arrays. Multidimensional arrays should be built in loops repeating the construction of one-dimensional arrays.

## 4.3 Type System

Types used to describe the *StaDyn* minimal core type system are shown in the second part of Figure 4.2. Syntax types ($ST$) are those that may be directly written by the programmer, whereas internal types ($T$) are internally used by the type system without the knowledge of the programmer. The point of avoiding the direct use of internal types is to offer the programmer the greatest simplicity without losing the expressive power of the type system.

Object types are specified describing a collection of their fields between curly braces, not including methods[1]. Methods can be represented by functions where `this` is the first parameter. Although this representation does not support method overriding, it allows us to significantly reduce the *StaDyn* core type system. *StaDyn* (the whole programming language) does support method overriding by extending the behavior described in [81]: when a message is passed to a dynamic union type, it is checked that at least for one possible signature, the

[1] A class-based core like the one proposed in [83] would be more appropriate to formalize methods and overriding.

actual argument types are subtypes of the corresponding formal parameters; the type of the method invocation expression is the union of the return types declared by those methods that satisfy the previous condition.

Although the `var` keyword is part of the concrete syntax of type variables (included in C# 3.0 to allow avoiding type specification of initialized local variables [50]), the parser assigns them unique sequential numbers ($X_i$ metavariables range over type variables). Type variables can be declared as dynamic (`dyn`) or, by default, static (`sta`). Only intersection and union types can also be qualified as dynamic or static.

Member types (`[`$(id{:}IT)^*$`]`) represent the collection of fields an object may hold. Member types have been introduced in constraints to define structural width coercion of object types to member types (S-OMember rule in Figure 4.10), because objects in *StaDyn* do not define width subtyping (S-Object in Figure 4.10). This subtyping relation is used in the constraint resolution algorithm when function calls are type-checked (T-Inv in Figure 4.16).

Type inference is specified with the general judgment $\Gamma; \Omega \vdash E : T \parallel C; \Gamma'$, meaning that under constraints $C$, environment $\Gamma$, and context $\Omega$, expression $E$ has type $T$, producing the output environment $\Gamma'$. Environments ($\Gamma$) bind variables (identifiers) to types in the scope represented by $\Gamma$, and they also bind type variables to types (if type variables have been inferred). $\Gamma$ holds the environment before the scope of $E$, and $\Gamma'$ stores the environment after typing $E$. $\Gamma'$ might differ from $\Gamma$, containing inferred types of local variables and new types bound to type variables inferred in $E$. Output environments have already been used to define flow-sensitive type systems [84], because type variables may change their types depending on the control flow [43].

A context ($\Omega$) stores the information of the function being analyzed, in order to type-check its statements. $\Omega._{\text{params}}$ saves the parameter list of the current function, $\Omega._{\text{rt}}$ holds its declared return type, and $\Omega._{\text{tifp}}$ collects the types inferred from function parameters (see § 4.3.2).

Figure 4.3 shows another example program of our core language. Elements of the environment and the constraints generated are shown in the right part of the figure. For example, in the scope of the `main` function in Figure 4.3, $\Gamma$ holds the assumptions $\Gamma$(`increment`):`int`, $\Gamma$(`list1`):$X_{18}$, and, in line 16, $\Gamma(X_{18})$:{$data$:$X_{20}$, $next$:$X_{21}$}, $\Gamma(X_{20})$:`bool` and $\Gamma(X_{21})$:`int`. Since $\Gamma(X_{20})$:`bool` in line 16, the statement in line 17 is accepted by the type system. However in line 19 the type of the object `data` field is changed to `int` and, hence, line 20 compiles without any error, whereas line 21 is now erroneous. This example shows how a variable can hold different types in the same scope, depending on the execution flow. This is a common feature of dynamically typed languages, but *StaDyn* offers it in a statically typed way. This process has also been applied to control structures (§ 4.3.6).

We also define two kinds of constraints (the last part of Figure 4.2). Subtyping constraints ($T \leq T$) require the type on the left to be a subtype of (promote to) the type on the right. Assignment constraints ($TV \leftarrow T$) not only check that an

```
01: var createNode(var data, var next) {          Γ(data):X10, Γ(next):X11
02:   return new { data=data, next=next};
03: }                 Γ(createNode):X10×X11→X12 ‖ {data:X10,next:X11} ≤ X12
04: void setData(var                               Γ(node):X13, Γ(data):X14
    node, var data) {
05:   node.data = data;                          X13 ≤ [data:X15], X15 ← X14
06: }              Γ(setData): X13×X14→void ‖ X13 ≤ [data:X15], X15 ← X14
07: void clearList(var list, bool clear) {                    Γ(list):X16
08:   if (clear)
09:     list.next = 0;                         X16 ≤ [next:X17], Γ(X17)←int
10: }                       Γ(X17):X17∨int, Γ(clearList):X16xbool→void ‖
                                          X16 ≤ [next:X17], Γ(X17)←X17∨int
11: void main() {
12:   var list1;                                                Γ(list1):X18
13:   var list2;                                                Γ(list2):X19
14:   int increment;                                        Γ(increment):int
15:   bool boolean;                                          Γ(boolean):bool
16:   list1 = createNode(true, 0);             Γ(X18):{data:X20,next:X21},
                                                 Γ(X20):bool, Γ(X21):int
17:   boolean = list1.data;
18:   list2 = createNode(boolean, list1);      Γ(X19):{data:X22,next:X18},
                                                            Γ(X22):bool
19:   setData(list1, 3);                                      Γ(X20):int
20:   increment = list2.next.data + 1;
21:   boolean = list1.data; // Compiler error
22:   clearList(list2, false);            Γ(X19):{data:X22,next:X18∨int},
                                               Γ(X18):{data:X20,next:X21}
23: }
```

Figure 4.3: Example concrete *StaDyn* core program.

assignment could be performed, but also are used to infer types, binding a type variable to another type. Therefore, assignment constraints may modify type variable bindings in type environments, when function invocation expressions are checked. In line 5 of Figure 4.3, a subtyping constraint is generated for the `node` variable; it should be an object with a `data` field, i.e., a subtype of a member type: $X_{13} \leq [data{:}X_{15}]$. This constraint must be statically fulfilled wherever the function `setData` is called, e.g., line 19. Line 5 is also an example of an assignment constraint generation: $X_{15} \leftarrow X_{14}$. When the `setData` function is invoked, the `data` type of the `node` argument $X_{15}$ will be assigned the type of the `data` parameter $X_{14}$. This is the reason why $X_{20}$ is then bound to `int` in line 19.

### 4.3.1 Functions

We use $\diamond$ to denote well-formedness. Inference rules in Figure 4.4 not only check well-formedness, but also generate output environments and constraints that are used for type-checking subsequent expressions. As an example, T-Func adds the identifier of the function being declared to the output environment. This identifier type is now $T_1 \times \ldots \times T_n \rightarrow T \parallel C$, denoting that it is possible to type-check subsequent calls to it. Function types include the constraint set ($C$) that must be satisfied by the arguments at each invocation. These constraints are those produced by the statements within the function. For instance, the type expression of the `setData` function in Figure 4.3 (line 6) has the two constraints $X_{13} \leq [data{:}X_{15}]$ and $X_{15} \leftarrow X_{14}$. The rest of the rules in Figure 4.4 generate no constraint at all, and output environments become the input of the following

$$\frac{\begin{array}{c}\Omega.\text{params} = id_1...id_n, \Omega.\text{locals} = id_{n+1}...id_{n+m}, \Omega.\text{rt} = T, \Omega.\text{tifp} = T_1...T_n \\ id \notin dom(\Gamma) \qquad \Gamma; \Omega \vdash T_1\ id_1 : \diamond \parallel \emptyset; \Gamma_1 ...\ \Gamma_{n-1}; \Omega \vdash T_n\ id_n : \diamond \parallel \emptyset; \Gamma_n \\ \Gamma_n; \Omega \vdash T_{n+1}\ id_{n+1} : \diamond \| \emptyset; \Gamma_{n+1} ...\ \Gamma_{n+m-1}; \Omega \vdash T_{n+m}\ id_{n+m} : \diamond \| \emptyset; \Gamma_{n+m} \\ \Gamma_{n+m}; \Omega \vdash S_1 : \diamond \parallel C_1; \Gamma_{n+m+1} ...\ \Gamma_{n+m+l-1}; \Omega \vdash S_l : \diamond \parallel C_l; \Gamma_{n+m+l} \\ \Gamma_{n+m+l}; \Omega \vdash R : \diamond \parallel C_{l+1}; \Gamma_{n+m+l+1} \\ \Gamma' = \Gamma, id : T_1 \times \ldots \times T_n \to T \parallel C_1 \cup \ldots \cup C_{l+1}\end{array}}{\Gamma; \emptyset \vdash T\ id(T_1\ id_1 ... T_n\ id_n)\ T_{n+1}\ id_{n+1} ... T_{n+m}\ id_{n+m}\ S_1 ... S_l\ R : \diamond \parallel \emptyset; \Gamma'} \quad \text{(T-Func)}$$

$$\frac{id \notin dom(\Gamma) \qquad \Gamma' = \Gamma, id : T}{\Gamma; \Omega \vdash T\ id : \diamond \parallel \emptyset; \Gamma'} \quad \text{(T-Decl)}$$

$$\frac{\Gamma; \Omega \vdash D_1 : \diamond \parallel \emptyset; \Gamma_1 \ldots\ \Gamma_{n-1}; \Omega \vdash D_n : \diamond \parallel \emptyset; \Gamma_n}{\Gamma; \Omega \vdash D_1 \ldots D_n : \diamond \parallel \emptyset; \Gamma_n} \quad \text{(T-Decls)}$$

$$\frac{\Gamma; \Omega \vdash F_1 : \diamond \parallel \emptyset; \Gamma_1 \ldots\ \Gamma_{n-1}; \Omega \vdash F_n : \diamond \parallel \emptyset; \Gamma_n}{\Gamma; \Omega \vdash F_1 \ldots F_n : \diamond \parallel \emptyset; \Gamma_n} \quad \text{(T-Funcs)}$$

Figure 4.4: Program, declarations and functions.

terms, obtaining a flow-sensitive type checking.

### 4.3.2 Context

It is necessary to store information regarding a function in order to subsequently perform type checking of the terms in the function scope. This information is saved in the function context ($\Omega$) by means of T-Func (Figure 4.4) and the rules shown in Figure 4.5. At function declaration (T-Func), local variables are stored in $\Omega.\texttt{locals}$, parameters in $\Omega.\texttt{params}$, and $\Omega.\texttt{rt}$ saves the return type specified in the function declaration. The types inferred from the type parameters are stored in $\Omega.\texttt{tifp}$ (it will be described later why this information is necessary to perform type-checking of assignments, field accessing and array indexing). First, T-Func (Figure 4.4) adds parameter types to $\Omega.\texttt{tifp}$; in Figure 4.5, $\Omega.\texttt{TIFP}$-Field inserts field types in $\Omega.\texttt{tifp}$ whenever an object type is in $\Omega.\texttt{tifp}$; $\Omega.\texttt{TIFP}$-Array and $\Omega.\texttt{TIFP}$-Inv do the same with arrays and function calls, respectively.

Notice that not only type variables are inserted in $\Omega.\texttt{tifp}$. Objects are also added because they may indirectly hold type variables in their fields. The same happens with arrays, whose elements could be type variables.

### 4.3.3 Basic Expressions

This subsection describes the type-checking of variables, object field access, vector indexing, arithmetic, relational and logical expressions. Although assignments

$$\frac{\Gamma;\Omega \vdash E : \{id_1 : T_1, \ldots, id_n : T_n\} \parallel C; \Gamma' \qquad \{id_1 : T_1, \ldots, id_n : T_n\} \in \Omega._{\text{tifp}}}{\text{include } T_1 \ldots T_n \text{ in } \Omega._{\text{tifp}}} \quad (\Omega._{\text{tifp}}\text{-FIELD})$$

$$\frac{\Gamma;\Omega \vdash E : Array(T) \parallel C; \Gamma' \qquad Array(T) \in \Omega._{\text{tifp}}}{\text{include } T \text{ in } \Omega._{\text{tifp}}} \quad (\Omega._{\text{tifp}}\text{-ARRAY})$$

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1; \Gamma_1 \ldots \ \Gamma_{n-1}; \Omega \vdash E_n : T_n \parallel C_n; \Gamma_n \\ \Gamma_n; \Omega \vdash id(E_1, \ldots, E_n) : T \parallel C_{n+1}; \Gamma_{n+1} \qquad \exists\, i \in [1, n],\ T_i \in \Omega._{\text{tifp}}\end{array}}{\text{include } T \text{ in } \Omega._{\text{tifp}}} \quad (\Omega._{\text{tifp}}\text{-INV})$$

Figure 4.5: Inference rules for $\Omega$.

$$\frac{\Gamma(id) : T \qquad \neg tv(T)}{\Gamma;\Omega \vdash id : T \parallel \emptyset; \Gamma} \quad (\text{T-VAR}) \qquad \frac{\Gamma(id) : X \qquad \Gamma(X) : T}{\Gamma;\Omega \vdash id : T \parallel \emptyset; \Gamma} \quad (\text{T-BVAR})$$

$$\frac{\Gamma(id) : X \qquad X \in ftv(\Gamma) \qquad id \in \Omega._{\text{params}} \qquad \neg lhsAssign(id)}{\Gamma;\Omega \vdash id : X \parallel \emptyset; \Gamma} \quad (\text{T-PVAR})$$

$$\frac{\Gamma(id) : X \qquad X \in ftv(\Gamma) \qquad id \notin \Omega._{\text{params}} \qquad lhsAssign(id)}{\Gamma;\Omega \vdash id : X \parallel \emptyset; \Gamma} \quad (\text{T-AVAR})$$

Figure 4.6: Variables.

and function calls are also expressions, they will be described in §§ 4.3.5 and 4.3.7, respectively.

Figure 4.6 shows inference rules that type-check variables. The `tv` predicate tests whether a type is a type variable or not, and the `ftv` function returns the set of unbound type variables in an environment. T-VAR types a variable previously declared, when its type is not a type variable. When the type of an identifier is a type variable and it is bound to another type, T-BVAR types the identifier to the type bound to the type variable. This happens, for instance, in line 17 of Figure 4.3, where the type variable of `list1` ($X_{18}$) was previously bound to the object type $\{data{:}X_{20}, next{:}X_{21}\}$ in line 16.

Both T-PVAR and T-AVAR type-check identifiers when their types are free type variables (not bound to any other type). In the first case, the variable can be used when it is a parameter (thus, it has a value) and it is not the left-hand side of an assignment[1]. On the other hand, T-AVAR allows a free type variable

[1]The *StaDyn* core does not allow assigning values to function parameters in order to make type-checking easier. This feature could be obtained by a syntactical transformation where parameters are assigned to local variables, because parameters in C# are passed by value –by default, when no `ref` and `out` keywords are explicitly used.

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq \texttt{int} \parallel C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash E_2 : T_2 \parallel C_3;\Gamma''' \qquad \Gamma''' \vdash T_2 \leq \texttt{int} \parallel C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \oplus E_2 : \texttt{int} \parallel C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''} \text{(T-ARITH)}$$

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1;\Gamma_1 ... \; \Gamma_{n-1};\Omega \vdash E_n : T_n \parallel C_n;\Gamma_n}{\Gamma;\Omega \vdash \texttt{new}\ \{id_1{=}E_1, ..., id_n{=}E_n\}{:}\{id_1{:}T_1, ..., id_n{:}T_n\} \parallel C_1 \cup ... \cup C_n;\Gamma_n} \text{(T-NOBJECT)}$$

$$\frac{\Gamma;\Omega \vdash E : T_e \parallel C_1;\Gamma' \qquad \Gamma' \vdash T_e \leq \texttt{int} \parallel C_2;\Gamma''}{\Gamma;\Omega \vdash \texttt{new}\ T[E][\,]_1 ... [\,]_n : Array_1(...Array_n(Array(T))) \parallel C_1 \cup C_2;\Gamma''} \text{(T-NARRAY)}$$

$$\frac{\Gamma;\Omega \vdash E : T \parallel C_1;\Gamma' \qquad X \text{ fresh} \qquad \Gamma' \vdash T \leq [id : X] \parallel C_2;\Gamma'' \qquad X \in ftv(\Gamma'') \wedge X \notin \Omega._{\text{tifp}} \Rightarrow lhsAssign(E.id)}{\Gamma;\Omega \vdash E.id : X \parallel C_1 \cup C_2;\Gamma''} \text{(T-FIELD)}$$

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1;\Gamma' \qquad X \text{ fresh} \qquad \Gamma' \vdash T_1 \leq Array(X) \parallel C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash E_2 : T_2 \parallel C_3;\Gamma''' \qquad \Gamma''' \vdash T_2 \leq \texttt{int} \parallel C_4;\Gamma'''' \qquad X \in ftv(\Gamma'''') \wedge X \notin \Omega._{\text{tifp}} \Rightarrow lhsAssign(E_1[E_2])}{\Gamma;\Omega \vdash E_1[E_2] : X \parallel C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''} \text{(T-ARRAY)}$$

Figure 4.7: Basic expressions.

that is not a parameter to be used as an expression as long as it is the left-hand side of an assignment (because the type variable will be then bound to a type in the subsequent expression). For example, the utilization of the `data` parameter in line 2 of Figure 4.3 is allowed because, despite its type is a free type variable ($X_{10}$), it is contained in $\Omega._{\text{params}}$. `list1` can be used in line 16 because, although its type ($X_{18}$) is not in $\Omega._{\text{tifp}}$, it is in the left-hand side of an assignment.

Figure 4.7 shows the T-ARITH inference rule of arithmetic expressions (for the sake of brevity, relational and logical expressions are not shown). Operands of arithmetic and relational expressions must be subtypes of `int`; logical expressions should promote to `bool`. Output environments are used as the input environments of the subsequent expressions, the one returned by the whole expression being the last environment. The constraint set generated by each expression is the union of all the constraints produced by each of the four premise judgments.

The `new` object expression (T-NOBJECT) infers an object type comprising the field labels and types of the corresponding expressions. Line 2 in Figure 4.3 is an example of this inference rule, where the type of the expression is $\{data{:}X_{10}, next{:}X_{11}\}$. In a similar way, the T-NARRAY rule types the array construction expressions. The expression that specifies the array size must promote to integer. Only one-dimensional arrays can be constructed at a time, and the type returned is an array of the same dimensions as pairs of square brackets. The type of the `new` expression in line 11, Figure 4.8, is $Array(X_{34})$, $X_{34}$ being a new fresh type variable.

```
01: void vector(var[] w) {                                  Γ(w):Array(X30)
02:   var[] v;                                              Γ(v):Array(X31)
03:   int a;
04:   v = new var[2];                                              Γ(X31):X32
05:   a = v[3]; // Compiler error
06:   v[0] = w[0] = 0;                          Γ(X32):int, Γ(X30):X30∨int
07:   v[1] = w[1] = true;          Γ(X32):int∨bool, Γ(X30):X30∨int∨bool
08: }
09: void main() {
10:   var[] ve;                                             Γ(ve):Array(X33)
11:   ve = new var[3];                                             Γ(X33):X34
12:   ve[2] = new { attribute = 3 };             Γ(X34):{attribute:int}
13:   vector(ve);                       Γ(X34):{attribute:int}∨int∨bool
14: }
```

Figure 4.8: Example use of arrays in *StaDyn* core.

When accessing object fields (T-Field), the object should promote to the member type $[id : X]$, $X$ being a new fresh type variable. A member type is an internal type that denotes the set of fields an object should hold. Therefore, an object promotes to a member type following the same rules as structural subtyping for objects described in [85] (rule S-OMember in Figure 4.10). Moreover, if the object field is a free type variable not inferred from the parameters, i.e., not in $\Omega_{.\text{tifp}}$, it must be a direct left child of an assignment expression. Line 42 in Figure 4.1 is an example of a correct term. Although the `dimensions` field of the `point` object is a free type variable and it is not the left-hand side of an assignment, its type is in $\Omega_{.\text{tifp}}$.

T-Array requires the first expression to be a subtype of an array, and the index to be an integer. As with objects, if the calculated type is a free type variable, it should be the left-hand side of an assignment. This predicate generates a compilation error in line 5 of Figure 4.8. The type of the elements of the `v` array is the free type variable $X_{32}$, not inferred from the parameters (`v` is a local variable), producing a compilation error because no value has been assigned to it.

### 4.3.4 Subtyping

Judgments in subtyping rules ($\Gamma \vdash T_1 \leq T_2 \parallel C; \Gamma'$) require an input environment ($\Gamma$) and generate a set of constraints ($C$) and an output environment ($\Gamma'$). The input environment is used to know the type variables that might be bound to other types. In effect, the T-TVBS rule in Figure 4.9 types any expression to the type which is bound to the expression type variable.

The output environment is used to bind a type variable to a type when the type variable must be a subtype of a particular type. This is precisely the behavior of the S-FTVL and S-FTVR rules in Figure 4.10 that, in addition, generate a subtyping constraint. An example expression where the S-FTVL rule is applied is `node.data` in Figure 4.3, line 5. The T-Field rule requires the type of `node` (the $X_{13}$ free type variable), to be a subtype of the member type [`data`:$X_{15}$] –$X_{15}$ being a new fresh type variable. Then, the S-FTVL rule generates a

$$\frac{\Gamma;\Omega \vdash E : X \parallel C;\Gamma' \qquad \Gamma'(X) : T}{\Gamma;\Omega \vdash E : T \parallel C;\Gamma'} \quad \text{(T-TVBS)}$$

Figure 4.9: Type variable binding substitution.

new $X_{13} \leq$ [data:$X_{15}$] constraint and binds [data:$X_{15}$] to $X_{13}$ in the output environment $\Gamma'$.

S-FTVR offers the same functionality when a concrete type must promote to a free type variable. This rule is used in `return` statements inside functions that return a type variable (e.g., line 2 in Figure 4.3). When both type variables are not bound to any type, only a subtyping constraint is produced (S-FTVs in Figure 4.10).

The arrays (S-Array) and objects (S-Object) type constructors are invariant. $Array(T_1)$ is a subtype of $Array(T_2)$ when $T_1$ and $T_2$ are equivalent. $T_1$ and $T_2$ are equivalent under the subtype relation, when $T_1 \leq T_2$ and $T_2 \leq T_1$ (E-Types). An object promotes to another one when both have the same number of fields and equal field labels, and the corresponding types are equivalent (S-Object).

Member types were introduced for structural subtyping of objects. An object type is a subtype of a member type when the former has all the members of the latter, and their corresponding types are structurally equivalent (S-OMember). This rule is necessary in the fulfillment of subtyping constraints of function invocation (§ 4.3.7). As an example, the `setData` function in Figure 4.3 has the $X_{13} \leq [data : X_{15}]$ constraint (line 6). When the function is called in line 19 passing the $\{data : X_{20}, next : X_{21}\}$ object as the first argument, the S-OMember subtyping rule confirms that the argument promotes to the parameter.

Subtyping rules for union and intersection types are an enhancement of the ones defined by other authors such as [61] and [86] (S-SUnionL, S-SUnionR, S-SInterR and S-SInterL), adding new dynamic typing rules (S-DUnionL and S-DInterR) –[77] details this new interpretation of union and intersection types. If the type variable bound to a union type has been declared as static, the set of operations that can be applied to that union type are those accepted by every type in the union type (S-SUnionL). However, if the reference is dynamic, type-checking is more permissive. In that case, it is possible to perform an operation when it is accepted by at least one of the types in the union type (S-DUnionL) –in the conclusion of the rule, $\cup\Gamma_i$ and $\cup C_i$ represent the union of all the $\Gamma_i$ and $C_i$ that fulfill the predicate in the premise. If the operation cannot be applied to any type, a type error will be generated even if the reference is dynamic. This behavior can be seen in lines 18 and 19 of Figure 4.11. The type of both `sta` and `din` variables is `int`$\vee$`bool`, but `sta` is static whereas `din` is dynamic. This difference prevents the arithmetic operation in line 18 from compiling (the plus operator cannot be applied to a `bool`), while it is correct in line 19 (addition is defined for integers).

$$\frac{}{\Gamma \vdash \texttt{bool} \leq \texttt{bool} \parallel \emptyset; \Gamma} \quad \text{(S-Bool)} \qquad \frac{}{\Gamma \vdash \texttt{int} \leq \texttt{int} \parallel \emptyset; \Gamma} \quad \text{(S-Int)}$$

$$\frac{X \in ftv(\Gamma) \qquad T \notin ftv(\Gamma) \qquad C = X \leq T \qquad \Gamma' = \Gamma, X : T}{\Gamma \vdash X \leq T \parallel C; \Gamma'} \quad \text{(S-FTV)}$$

$$\frac{X \in ftv(\Gamma) \qquad T \notin ftv(\Gamma) \qquad C = T \leq X \qquad \Gamma' = \Gamma, X : T}{\Gamma \vdash T \leq X \parallel C; \Gamma'} \quad \text{(S-FTV)}$$

$$\frac{X_1 \in ftv(\Gamma) \qquad X_2 \in ftv(\Gamma) \qquad C = X_1 \leq X_2}{\Gamma \vdash X_1 \leq X_2 \parallel C; \Gamma} \quad \text{(S-FTVs)}$$

$$\frac{\Gamma \vdash T_1 \leq T_2 \parallel C_1; \Gamma' \qquad \Gamma' \vdash T_2 \leq T_1 \parallel C_2; \Gamma''}{\Gamma \vdash T_1 \equiv T_2 \parallel C_1 \cup C_2; \Gamma''} \quad \text{(E-Types)}$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 \parallel C; \Gamma'}{\Gamma \vdash Array(T_1) \leq Array(T_2) \parallel C; \Gamma'} \quad \text{(S-Array)}$$

$$\frac{\Gamma \vdash T_1 \equiv T_1' \parallel C_1; \Gamma_1 \ldots \qquad \Gamma_{n-1} \vdash T_n \equiv T_n' \parallel C_n; \Gamma_n}{\Gamma \vdash \{id_1{:}T_1, ..., id_n{:}T_n\} \leq \{id_1{:}T_1', ..., id_n{:}T_n'\} \parallel C_1 \cup \ldots \cup C_n; \Gamma_n} \quad \text{(S-Object)}$$

$$\frac{\forall\, i \in [1,m],\ \exists\, j \in [1,n],\ id_i' = id_j,\ \Gamma_{i-1} \vdash T_i' \equiv T_j \parallel C_i; \Gamma_i}{\Gamma_0 \vdash \{id_1{:}T_1, ..., id_n{:}T_n\} \leq [id_1'{:}T_1', ..., id_m'{:}T_m'] \parallel C_1 \cup \ldots \cup C_m; \Gamma_m} \quad \text{(S-OMember)}$$

$$\frac{\forall\, i \in [1,n], \Gamma \vdash T_i \leq T \parallel C_i; \Gamma_i}{\Gamma \vdash \texttt{sta}\ T_1 \vee \ldots \vee T_n \leq T \parallel C_1 \cup \ldots \cup C_n; \Gamma_1 \cup \ldots \cup \Gamma_n} \quad \text{(S-SUnionL)}$$

$$\Gamma \vdash\ T_i^{\,i \in 1..n} \leq \texttt{sta}\ T_1 \vee \ldots \vee T_n \parallel \emptyset; \Gamma \quad \text{(S-SUnionR)}$$

$$\frac{\exists\, i \in [1,n], \Gamma \vdash T_i \leq T \parallel C_i; \Gamma_i}{\Gamma \vdash \texttt{dyn}\ T_1 \vee \ldots \vee T_n \leq T \parallel\ \cup C_i; \cup \Gamma_i} \quad \text{(S-DUnionL)}$$

$$\Gamma \vdash \texttt{sta}\ T_1 \wedge \ldots \wedge T_n \leq\ T_i^{\,i \in 1..n} \parallel \emptyset; \Gamma \quad \text{(S-SInterL)}$$

$$\frac{\forall\, i \in [1,n], \Gamma \vdash T \leq T_i \parallel C_i; \Gamma_i}{\Gamma \vdash T \leq \texttt{sta}\ T_1 \wedge \ldots \wedge T_n \parallel C_1 \cup \ldots \cup C_n; \Gamma_1 \cup \ldots \cup \Gamma_n} \quad \text{(S-SInterR)}$$

$$\frac{\exists\, i \in [1,n], \Gamma \vdash T \leq T_i \parallel C_i; \Gamma_i}{\Gamma \vdash T \leq \texttt{dyn}\ T_1 \wedge \ldots \wedge T_n \parallel\ \cup C_i; \cup \Gamma_i} \quad \text{(S-DInterR)}$$

Figure 4.10: Subtyping and type equivalence.

```
01:var getElement(var list,
                  var fstOrSnd) {
02:  var element;
03:  if (fstOrSnd)
04:    element = list.data;
05:  else
06:    element = list.next.data;
07:  return element;
08:}
09:int increment(int value) {
10:  return value + 1;
11:}
12:void main() {
13:  int integer;
14:  var listOfTwo, sta;
15:  dyn var din;
16:  listOfTwo = createNode(1,createNode(true, 0));
17:  din = sta = getElement(listOfTwo, true);
18:  integer = sta + 1;// Compiler error
19:  integer = din + 1;
20:  increment(din); // Compiler error
21:}
```

Figure 4.11: Example use of dynamic and static references in *StaDyn* core.

In parallel, a type promotes to a static intersection type only if it is a subtype of all the types collected by the intersection type (rule S-SInterR). Similarly, we have defined the dynamic behavior to be more lenient, accepting the promotion when a type promotes to at least one of the types in the intersection type (rule S-DInterR). An example is the function call in line 71 of Figure 4.1: `data` of the `list` argument must be a subtype of $[dimensions{:}X_1] \wedge [x{:}X_2] \wedge [y{:}X_3] \wedge [z{:}X_4]$; the program is compiled only when the `point` parameter in line 38 is dynamic (two and three dimensional points provide `dimensions`, `x`, and `y` fields), producing an error in case it is static (the `z` field is not implemented by two dimensional points).

It is worth noting that the definition of subtyping is not complete for union and intersection types. We include inference rules for neither dynamic union types on the right-hand side (supertypes), nor dynamic intersection types on the left-hand side (subtypes). This is because the *StaDyn* core type system never type-checks whether a dynamic intersection type is a subtype of another type –they always appear in the right-hand side of subtyping constraints– or any type promotes to a dynamic union type –they are always checked to be subtypes of another type.

Since *StaDyn* does not yet support higher-order functions (*delegates* in C# terminology), we do not specify subtyping of the function type constructor. Subtyping is not defined between member types either, because they only appear in constraints. Therefore, the current definition of the subtyping relation is neither reflexive nor transitive.

### 4.3.5 Assignments

The abstract syntax in Figure 4.2 allows any expression to be the left-hand side of an assignment. The type system rejects all those expressions that cannot be used in that context. Only identifiers, array indexing and field access expressions can be the left-hand side of an assignment. For the sake of brevity, those rules are not shown.

The four inference rules in Figure 4.12 describe assignment expressions. T-Assign types assignment expressions when the left-hand side expression type is not a type variable. This straightforward rule only requires the right-hand side to be a subtype of the left-hand side. In case the left-hand side is a type variable,

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1;\Gamma' \qquad \neg tv(T_1) \qquad \Gamma';\Omega \vdash E_2 : T_2 \parallel C_2;\Gamma'' \qquad \Gamma'' \vdash \texttt{sta}\ T_2 \leq T_1 \parallel C_3;\Gamma'''}{\Gamma;\Omega \vdash E_1 = E_2 : T_1 \parallel C_1 \cup C_2 \cup C_3;\Gamma'''} \quad \text{(T-Assign)}$$

$$\frac{\Gamma;\Omega \vdash E_1 : X \parallel C_1;\Gamma' \qquad \Gamma';\Omega \vdash E_2 : T \parallel C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : T}{\Gamma;\Omega \vdash E_1 = E_2 : T \parallel C_1 \cup C_2;\Gamma'''} \quad \text{(T-TVAssign)}$$

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \parallel C_1;\Gamma' \qquad X \text{ fresh} \qquad \Gamma' \vdash T_1 \leq [id : X] \parallel C_2 \qquad \Gamma';\Omega \vdash E_2 : T_2 \parallel C_3;\Gamma'' \qquad \Gamma''' = \Gamma'', X : T_2 \qquad \texttt{if}\ T_1 \in \Omega_{.\text{tifp}}, \texttt{then}\ C_4 = X \leftarrow T_2, \texttt{else}\ C_4 = \emptyset}{\Gamma;\Omega \vdash E_1.id = E_2 : T_2 \parallel C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma'''} \quad \text{(T-FAssign)}$$

$$\frac{\Gamma;\Omega \vdash E_1[E_2] : X \parallel C_1;\Gamma' \qquad \Gamma';\Omega \vdash E_3 : T \parallel C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : \Gamma''(X) \vee T \qquad \texttt{if}\ X \in \Omega_{.\text{tifp}}, \texttt{then}\ C_3 = X \leftarrow X \vee \Gamma''(X) \vee T, \texttt{else}\ C_3 = \emptyset}{\Gamma;\Omega \vdash E_1[E_2] = E_3 : T \parallel C_1 \cup C_2 \cup C_3;\Gamma'''} \quad \text{(T-AAssign)}$$

Figure 4.12: Assignments.

it will from now on be bound to the type of the right-hand side expression (T-TVAssign).

T-FAssign types the assignment of an object field when it is a type variable. As with the T-Field rule, the object should be a subtype of a member type with the specific field label. The new field type will be the type of the right-hand side expression, regardless of its previous type. Finally, if the field type has been inferred from a parameter, a new assignment constraint will be generated. This constraint will cause changing the field type of the argument when the function is called. An example of this kind of assignment constraint generation is shown in the `setData` function in Figure 4.3 ($X_{15} \leftarrow X_{14}$). Calling this function with an object as the first argument (line 19) changes the type of the argument's `data` to the type of the second argument (`int`), making the statement in line 20 correct.

For an array type whose elements are type variables (T-AAssign), the new type of its elements will be a union type comprising its previous type and the type of the right-hand side. Therefore, the type of `v` in line 8 of Figure 4.8 is an array of integers or booleans (`int`$\vee$`bool`) because it holds both. If the type variable has been inferred from the function parameters, a new assignment constraint will be generated including the own type variable in the right-hand side of the assignment. This is the case of the `w` variable in the `vector` function (line 8 in Figure 4.8). Unlike the type of `v`, the type of `w` ($X_{30}$) is included in the union type ($X_{30}\vee$`int`$\vee$`bool`), denoting that the type of the actual parameter in the invocation will be included in the union type. Therefore, the type of the elements of `ve` when the function `vector` is called in line 13 is {`attribute:int`}$\vee$`int`$\vee$`bool`.

$$\frac{\Gamma;\Omega \vdash E : T \parallel C_1;\Gamma' \qquad \Gamma' \vdash T \leq \Omega.\text{rt} \parallel C_2;\Gamma''}{\Gamma;\Omega \vdash \texttt{return}\ \ E : \diamond \parallel C_1 \cup C_2;\Gamma''} \quad \text{(T-Return)}$$

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \parallel C';\Gamma' \qquad \Gamma' \vdash T \leq \texttt{bool} \parallel C'';\Gamma'' \\ \Gamma'';\Omega \vdash S_1 : \diamond \parallel C_1;\Gamma_1 \ldots\ \Gamma_{n-1};\Omega \vdash S_n : \diamond \parallel C_n;\Gamma_n \\ \Gamma'';\Omega \vdash S_{n+1} : \diamond \| C_{n+1};\Gamma_{n+1} \ldots\ \Gamma_{n+m-1};\Omega \vdash S_{n+m} : \diamond \| C_{n+m};\Gamma_{n+m}\end{array}}{\begin{array}{c}\Gamma;\Omega \vdash \texttt{if}\ E\ S_1 \ldots S_n\ S_{n+1} \ldots S_{n+m} : \diamond\ \parallel \\ C' \cup C'' \cup join(C_1 \cup \ldots \cup C_n, C_{n+1} \cup \ldots \cup C_{n+m}); join(\Gamma_n, \Gamma_{n+m})\end{array}} \quad \text{(T-If)}$$

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \parallel C';\Gamma' \qquad \Gamma' \vdash T \leq \texttt{bool} \parallel C'';\Gamma'' \\ \Gamma'';\Omega \vdash S_1 : \diamond \parallel C_1;\Gamma_1 \ldots\ \Gamma_{n-1};\Omega \vdash S_n : \diamond \parallel C_n;\Gamma_n\end{array}}{\Gamma;\Omega \vdash \texttt{while}\ E\ S_1 \ldots S_n : \diamond \parallel C' \cup C'' \cup join(C_1 \cup \ldots \cup C_n, \emptyset); join(\Gamma_n, \emptyset)} \quad \text{(T-While)}$$

Figure 4.13: Statements.

### 4.3.6 Statements

The minimal core includes the `return`, `if` and `while` statements (Figure 4.13). For the `return` statement, the expression type to be returned must be a subtype of the declared return type (T-Return).

Control-flow branches of `if` and `while` statements are taken into consideration to keep the flow-sensitiveness of our type system. The *join* of constraints and the union of type environments take into consideration this difficulty, taking the type information obtained on each execution path and combining both into a single constraint list and type environment. Each parameter represents the type information of an exclusive control flow. Since it might happen that the `while` body is not executed at runtime (it is not exclusive), the empty set is passed as its second argument (T-While). An example use of the *join* function is the type of the `point` variable ($\{x$:`int`, $y$:`int`, *dimensions*:`int`$\} \vee \{x$:`int`, $y$:`int`, $z$:`int`, *dimensions*:`int`$\}$) –line 10 in Figure 4.1– that is created from its types in lines 7 ($\{x$:`int`, $y$:`int`, *dimensions*:`int`$\}$) and 9 ($\{x$:`int`, $y$:`int`, $z$:`int`, *dimensions*:`int`$\}$). The same happens with constraints: the joined constraint for the `point` parameter of the `distance3D` function ($X_1 \leq [$*dimensions*:$X_2] \wedge [x$:$X_3] \wedge [y$:$X_4] \wedge [z$:$X_5]$) is obtained from the constraints generated in lines 42 ($X_1 \leq [$*dimensions*:$X_2]$) and 43 ($X_1 \leq [x$:$X_3] \wedge [y$:$X_4] \wedge [z$:$X_5]$).

The union of environments used in Figure 4.13 is also based on the *join* function described in Figure 4.14: variable bindings must be the same in both environments, and the resulting type variable binding set is the *join* of the type variable binding sets of each flow path.

Figure 4.14 shows the algorithm used to implement the *join* function. Each set holds either constraints (subtyping and assignment) or type variable bindings ($X$:$T$ in environments). The algorithm has been defined employing the `compare` and `union` operations defined by the axioms in Figure 4.15. The algorithm takes elements of both sets, adding new union and intersection types [61] to the return

$join(Set_1, Set_2) \equiv Set$ `in`
    $Set \leftarrow \emptyset$
    $\forall\, elem_1 \in Set_1$
        `if` $\exists\, elem_2 \in Set_2, compare(elem_1, elem_2)$
            $Set \leftarrow Set \cup union(elem_1, elem_2)$
        `else`
            $Set \leftarrow Set \cup union(elem_1)$
    $\forall\, elem \in Set_2 \div Set_1$
        $Set \leftarrow Set \cup union(elem)$

$Set_1 \div Set_2 \equiv Set$ `in`
    $Set \leftarrow \emptyset$
    $\forall\, elem_1 \in Set_1$
        `if` $\nexists\, elem_2 \in Set_2, compare(elem_1, elem_2)$
            $Set \leftarrow Set \cup elem_1$

Figure 4.14: The *join* algorithm.

set. It first processes the elements in the first set, and then those included in the second set but not in the first one ($\div$).

As Figure 4.15 shows, comparisons between constraints are based on the type in the constraint's left-hand side. This is because constraints are always generated with a free type variable in its left-hand side. Definitions of the `compare` and `union` operations in Figure 4.15 ensure that every constraint set will never have two different constraints with the same left-hand side type variable. The only statement that generates subtyping constraints with a particular type on the left-hand side is the `return` statement. However, this statement cannot appear in a control flow statement because of the AST transformation described in § 4.2.

Joins of assignment constraints and type variable bindings create a union type consisting of the two types in each execution path. However, subtyping constraints are joined in a new intersection type. If a static reference should promote to $T_1$ in one flow path and be a subtype of $T_2$ in the other, it must then be a subtype of both (subtype of the intersection type).

The `union` function is also defined for constraints or type variable bindings generated in only one of the optional execution paths (last four axioms in Figure 4.15). The union of a single static subtyping constraint is the own constraint, because static typing must check every possible flow of execution. However, if the type variable is dynamic, there is no resulting constraint because it has been produced in a single optional execution path and, since it is dynamic, the constraint fulfillment is not mandatory. In assignment constraints and type variable bindings, the type to be bound is included in the right-hand side of the assignment. This means that the type variable will be bound to a new union type including the type it was previously bound to, because a new type could be assigned to the existing one in an optional control flow. As an example, the `next` field of the

$$\text{(J-Compare)}\quad \frac{}{compare(X_1{\leftarrow}T_1, X_1{\leftarrow}T_2)}$$

$$\frac{}{compare(X_1{\leq}T_1, X_1{\leq}T_2)} \qquad \frac{}{compare(X_1{:}T_1, X_1{:}T_2)}$$

$$\text{(J-Union)}\quad \frac{}{union(X{\leftarrow}T_1, X{\leftarrow}T_2) = X \leftarrow T_1{\vee}T_2}$$

$$\frac{}{union(\texttt{sta}\ X{\leq}T_1, \texttt{sta}\ X{\leq}T_2) = X \leq \texttt{sta}\ T_1{\wedge}T_2}$$

$$\frac{}{union(\texttt{dyn}\ X{\leq}T_1, \texttt{dyn}\ X{\leq}T_2) = X \leq \texttt{dyn}\ T_1{\wedge}T_2}$$

$$\frac{}{union(X{:}T_1, X{:}T_2) = X : T_1{\vee}T_2} \qquad \frac{}{union(X{\leftarrow}T) = X{\leftarrow}X{\vee}T}$$

$$\frac{}{union(\texttt{sta}\ X{\leq}T) = \texttt{sta}\ X{\leq}T} \qquad \frac{}{union(\texttt{dyn}\ X{\leq}T) = \emptyset}$$

$$\frac{}{union(X{:}T) = X : X{\vee}T}$$

Figure 4.15: Comparison and union operations.

`list` variable ($X_{17}$) has the type $X_{17}{\vee}$`int` in line 10 of Figure 4.3. This implies that the type of `list2` in line 22 is converted from $\{data{:}X_{22}, next{:}X_{18}\}$ (being $\Gamma(X_{18}){:}\{data{:}X_{20}, next{:}X_{21}\}$) to $\{data{:}X_{22}, next{:}X_{18} \vee$ `int`$\}$. The result is that the `next` field is changed because one possible flow of execution in the `clearList` function may change its type to `int`.

### 4.3.7 Function Invocation

Figure 4.16 shows the two inference rules of function invocation. The difference is in the existence of free type variable arguments (T-Inv if there is no free type variable argument, and T-FTVInv otherwise). In both cases, the `shift` function takes a function type and returns an equivalent one, renaming the numbers of type variables to new fresh type variables. This process permits multiple invocations of the same function, creating new type variables for each function invocation. On each invocation, the types of the arguments are checked to be subtypes of the parameter types.

If no argument is a free type variable, constraints resolution is performed. The judgment $\Gamma_s \Vdash C, \Gamma$ means that under the $\Gamma$ input environment, $\Gamma_s$ is a *solution* for $C$; i.e., $\Gamma_s$ holds all the substitutions to fulfill $C$ under the $\Gamma$ environment. In that case, the type of the function call is the substitution $\Gamma_{2n+2}(T)$, where $\Gamma_{2n+2}$ is a solution for $C$. Under these circumstances, the $C$ constraint set is solved and,

$$\frac{\begin{array}{c} shift(\Gamma(id)) : Tp_1 \times \ldots \times Tp_n \to T \parallel C \\ \forall\, i \in [1,n], \Gamma_i \vdash E_i : T_i \parallel C_i; \Gamma_{i+1} \qquad \forall\, i \in [1,n], T_i \notin ftv(\Gamma_{i+1}) \\ \forall\, i \in [1,n], \Gamma_{n+i} \vdash \texttt{sta}\ T_i \leq Tp_i \parallel C_{n+i}; \Gamma_{n+i+1} \qquad \Gamma_{2n+2} \Vdash C; \Gamma_{2n+1} \end{array}}{\Gamma_1 \vdash id(E_1 \ldots E_n) : \Gamma_{2n+2}(T) \parallel C_1 \cup \ldots \cup C_{2n}; \Gamma_{2n+2}} \quad \text{(T-Inv)}$$

$$\frac{\begin{array}{c} shift(\Gamma(id)) : Tp_1 \times \ldots \times Tp_n \to T \parallel C \\ \forall\, i \in [1,n], \Gamma_i \vdash E_i : T_i \parallel C_i; \Gamma_{i+1} \qquad \exists\, i \in [1,n], T_i \in ftv(\Gamma_{i+1}) \\ \forall\, i \in [1,n], \Gamma_{n+i} \vdash \texttt{sta}\ T_i \leq Tp_i \parallel C_{n+i}; \Gamma_{n+i+1} \end{array}}{\Gamma_1 \vdash id(E_1 \ldots E_n) : T \parallel C \cup C_1 \cup \ldots \cup C_{2n}; \Gamma_{2n+1}} \quad \text{(T-FTVInv)}$$

Figure 4.16: Function invocation.

hence, it is not included in the constraints generated by the function call. This shows how constraint resolution is part of the type inference process (it is not global, i.e., it is not performed after traversing the whole AST). If any argument is a free type variable, $C$ is added to the constraint set produced by a function invocation expression (T-FTVInv).

The constraint resolution algorithm implemented is an adaptation of the algorithm defined by Aiken and Wimmers [80] that performs inclusion constraint resolution using union and intersection types. Its detailed description can be consulted in [65].

### 4.3.8 Converting Implicit into Explicit Types

Our language defines an automatic conversion of dynamic implicitly typed union types to explicit (particular) types (in assignments and function calls). When the union type is static, the subtyping rules described in § 4.3.4 require types in the union type to promote to the explicit type. However, if the implicit type is dynamic, the conversion is too lenient because only one single promotion is necessary to allow the conversion. This is why a static promotion is forced in both assignments (rule T-Assign in Figure 4.12) and function invocations (rules T-Inv and T-FTVInv in Figure 4.16). As an example, the `din` variable (typed `dyn int∨bool` in line 20 of Figure 4.11) is passed as an argument to the `increment` function that explicitly requires its `value` parameter to be `int`. Therefore, a compilation error is generated even though the argument is dynamic, because the `value` parameter is explicitly typed (and, hence, static).

# Chapter 5

# Erasure Semantics

The objective of this chapter is twofold. First, to describe the translation templates used to generate code for the .NET platform employing the static type information gathered by the compiler (Chapter 4). Second, based on the semantics of C# [87], to describe the erasure semantics of the minimal core of *StaDyn*.

The *StaDyn* core may be translated into C# following either of two implementation styles: first, by type-passing, augmenting the runtime system to carry information about type parameters; second, by erasure, removing all information about type parameters at runtime [83]. We have used the second approach, giving an erasure mapping from the *StaDyn* minimal core into C#. This style corresponds to the current implementation of *StaDyn*, which is compiled into the .NET platform by generating IL code (before the executable files), maintaining no information about type parameters at runtime –here the translation to C# is described for simplicity. Figure 5.1 shows an example translation that will be used throughout this chapter. The *StaDyn* core source code is shown on the left, while the corresponding output C# program is displayed on the right.

The translation is performed traversing the AST. This traversal is performed after type checking, where the AST nodes were annotated with their types and a copy of the state of the type environment ($\Gamma$) and context ($\Omega$) in the conclusion of each typing rule (written $\Gamma_{node}$ and $\Omega_{node}$).

## 5.1 Type Erasure

The erasure of a type in the *StaDyn* core is the corresponding C# type that will be used in the code generation process. Since type erasures depend on environments ($\Gamma$), we write $|T|_\Gamma$ for the erasure of the type $T$ with respect to the environment $\Gamma$. Translation rules insert type casts when necessary using the type information obtained by the compiler, and omitting them when it is trivially safe to do so, e.g., when the top type in C#, `object`, is the erased type that an expression should have.

Figure 5.2 shows type erasures of the *StaDyn* minimal core. Function and

```
dyn var point2D(int x, int y) {

  return new { x=x, y=y};
}
dyn var point3D(int x, int y,
                int z) {

  return new { x=x, y=y, z=z};
}
dyn var point(dyn var dim,
        dyn var x, dyn var y,
        dyn var z) {

  dyn var result;
  if (dim==2)
    result = point2D(x,y);
  else
    result = point3D(x,y,z);
  return result;
}
void main() {

  var sta;
  dyn var din;
  sta = point(2, 0, 4, 3);
  din = point(3, 0, 4, 3);
  sta.x +


          din.y



                * din.z;
}
```

StaDyn core (source)

```
01: using System
02: class AC_int_x_int_y {
03:   public int x;public int y;
04: }
05: class AC_int_x_int_y_int_z {
06:   public int x; public int y; public int z;
07: }
08: public class MainClass {
09:   static AC_int_x_int_y point2D(int x, int y) {
10:     object _temp;
11:     return new AC_int_x_int_y {x=x, y=y};
12:   }
13:   static AC_int_x_int_y_int_z point3D(int x, int y, int z) {

14:     object _temp;
15:     return new AC_int_x_int_y_int_z {x=x, y=y, z=z};
16:   }
17:   static object point(int dim, int x, int y, int z) {


18:     object _temp;
19:     object result;
20:     if (dim == 2)
21:       result = point2D(x, y);
22:     else
23:       result = point3D(x, y, z);
24:     return result;
25:   }
26:   public static void Main() {
27:     object _temp;
28:     object sta;
29:     object din;
30:     sta = point(2, 0, 4, 3);
31:     din = point(3, 0, 4, 3);
32:     _temp = ((_temp=sta) is AC_int_x_int_y ?
                    (int)(((AC_int_x_int_y)_temp).x) :
                (int)(((AC_int_x_int_y_int_z)_temp).x) ) +
33:             ((_temp=din) is AC_int_x_int_y ?
                    (int)(((AC_int_x_int_y)_temp).y) :
                _temp is AC_int_x_int_y_int_z ?
                    (int)(((AC_int_x_int_y_int_z)_temp).y) :
                (int)(_temp.GetType().GetField("y").GetValue(_temp)))*
34:             ((AC_int_x_int_y_int_z)din).z;
35:   }
36: }
```

C# (destination)

Figure 5.1: Example translation from *StaDyn* core to C#.

$$|\texttt{int}|_\Gamma = \texttt{int} \quad |\texttt{bool}|_\Gamma = \texttt{bool} \quad |\texttt{void}|_\Gamma = \texttt{void} \quad |Array(T)|_\Gamma = |T|_\Gamma\texttt{[]}$$

$$\frac{X \in ftv(\Gamma)}{|\texttt{sta}\ X|_\Gamma = |\texttt{dyn}\ X|_\Gamma = \texttt{object}} \qquad \frac{\Gamma(X) : T}{|\texttt{sta}\ X|_\Gamma = |\texttt{dyn}\ X|_\Gamma = |T|_\Gamma}$$

$$|\texttt{sta}\ T_1 \vee \ldots \vee T_n|_\Gamma = |\texttt{dyn}\ T_1 \vee \ldots \vee T_n|_\Gamma = \texttt{object}$$

$$|\texttt{sta}\ [id_1{:}T_1, \ldots, id_n{:}T_n]|_\Gamma = |\texttt{dyn}\ [id_1 : T_1, \ldots, id_n : T_n]|_\Gamma = \texttt{object}$$

$$|\texttt{sta}\ \{id_1{:}T_1, \ldots, id_n{:}T_n\}|_\Gamma = |\texttt{dyn}\ \{id_1{:}T_1, \ldots, id_n{:}T_n\}|_\Gamma = \texttt{AC\_}|T_1|_\Gamma\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_\Gamma\texttt{\_}id_n$$

where $id_1 \ldots id_n$ are lexicographically ordered, and
in $\texttt{AC\_}|T_1|_\Gamma\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_\Gamma\texttt{\_}id_n$, $T[\,]_1 \ldots [\,]_n$ is replaced with T_n

Figure 5.2: Type erasure definition.

intersection type erasures are not used in our translation rules, because our language does not support high-order functions, and intersection types only appear in constraints (no code is generated for them).

## 5.2 Anonymous Classes

As shown in Figure 5.2, an anonymous class ($\texttt{AC\_}|T_1|_\Gamma\_id_1\_\ldots\_|T_n|_\Gamma\_id_n$) is the type erasure of each different object structure. Since subtyping rules in our language require two objects to have the same structure (S-Object), we create a unique anonymous class for each object structure. To do so, the name of the anonymous class is the concatenation of each field name (lexicographically ordered) followed by its type erasure –arrays $T[\,]_1 \ldots [\,]_n$ are replaced with T_n because square brackets are not allowed in C# identifiers.

These anonymous classes are generated in the first traversal ($[\![\ ]\!]_{\text{AC}}$), after type-checking the AST. The visit of each AST node receives the set of classes that have already been declared. Starting from the AST root node ($P$), this set is passed from each node to their descendants. The only nodes that generate a new class declaration are the object type and the `new` object expression. The following translation template shows the anonymous class generation for the latter node.

$$\frac{\text{AC\_}|T_1|_\Gamma\_id_1\_\ldots\_|T_n|_\Gamma\_id_n \notin classes}{\begin{array}{l}[\![E = \texttt{new}\ \{id_1\texttt{=}E_1, \ldots, id_n\texttt{=}E_n\}]\!]_{\text{AC}}(classes) \triangleq \\ \quad classes \leftarrow classes \cup \text{AC\_}|T_1|_{\Gamma_E}\_id_1\_\ldots\_|T_n|_{\Gamma_E}\_id_n \\ \quad \texttt{class AC\_}|T_1|_{\Gamma_E}\_id_1\_\ldots\_|T_n|_{\Gamma_E}\_id_n\ \{ \\ \quad\quad \texttt{public}\ |T_1|_{\Gamma_E}\ |id_1|_{\Gamma_E}\ \texttt{;} \\ \quad\quad \ldots \\ \quad\quad \texttt{public}\ |T_n|_{\Gamma_E}\ |id_n|_{\Gamma_E}\ \texttt{;} \\ \quad \} \\ \text{where } \Gamma_{E_1};\Omega_{E_1} \vdash E_1 : T_1\ \ldots\ \Gamma_{E_n};\Omega_{E_n} \vdash E_n : T_n, \\ \quad id_1 \ldots id_n \text{ are lexicographically ordered, and} \\ \quad \text{in AC\_}|T_1|_\Gamma\_id_1\_\ldots\_|T_n|_\Gamma\_id_n, T[\,]_1 \ldots [\,]_n \text{ is replaced with T\_n}\end{array}}$$

Figure 5.1 shows how two anonymous classes (lines 2 to 7 of the C# code on the right) are created in the traversal of two `new` object nodes (lines 11 and 15 of the *StaDyn* core code on the left).

The second scenario where an anonymous class declaration is generated is when an object type is used and its class has not been previously declared.

$$\frac{\text{AC\_}|T_1|_\Gamma\_id_1\_\ldots\_|T_n|_\Gamma\_id_n \notin classes}{\begin{array}{l}[\![T = \{id_1{:}T_1, \ldots, id_n{:}T_n\}]\!]_{\text{AC}}(classes) \triangleq \\ \quad classes \leftarrow classes \cup \text{AC\_}|T_1|_{\Gamma_T}\_id_1\_\ldots\_|T_n|_{\Gamma_T}\_id_n \\ \quad \texttt{class AC\_}|T_1|_{\Gamma_T}\_id_1\_\ldots\_|T_n|_{\Gamma_T}\_id_n\ \{ \\ \quad\quad \texttt{public}\ |T_1|_{\Gamma_T}\ |id_1|_{\Gamma_T}\ \texttt{;} \\ \quad\quad \ldots \\ \quad\quad \texttt{public}\ |T_n|_{\Gamma_T}\ |id_n|_{\Gamma_T}\ \texttt{;} \\ \quad \} \\ \text{where } id_1 \ldots id_n \text{ are lexicographically ordered, and} \\ \quad \text{in AC\_}|T_1|_\Gamma\_id_1\_\ldots\_|T_n|_\Gamma\_id_n, T[\,]_1 \ldots [\,]_n \text{ is replaced with T\_n}\end{array}}$$

## 5.3 Translation of Programs

The translation of a program consists of the import of the main .NET namespace (`System`) followed by the declaration of anonymous classes ($[\![\ ]\!]_{\text{AC}}$) (passing an empty set of classes) and the final generation of code ($[\![\ ]\!]_{\text{CG}}$).

$$\begin{array}{lll}[\![P]\!]_{\text{program}} & \triangleq & \texttt{import System ;} \\ & & [\![P]\!]_{\text{AC}}(\emptyset) \\ & & [\![P]\!]_{\text{CG}}\end{array}$$

Code generated for a program ($[\![P]\!]_{\text{CG}}$) consists of a C# public class (`MainClass`) followed by two helper `_setValue` methods (explained in §§ 5.7 and 5.8). Each function is translated into a corresponding `static` C# method, and the main declarations and statements are placed inside the program's entry point (the C# `Main` method of the `MainClass`) –the example translation in Figure 5.1 omits the two `_setValue` methods.

$[\![P = F_1 \ldots F_n\ D_1 \ldots D_m\ S_1 \ldots S_l]\!]_{\text{CG}} \triangleq$

```
  public class MainClass {
  private static object _setValue(object obj, string id,
                                                  object value) {
    obj.GetType().GetField(id).SetValue(obj,value);
    return value;
  }
  private static object _setValue(Array array, object value,
                                                   int index) {
    array.SetValue(value, index);
    return value;
  }
```

$[\![F_1]\!]_{\text{CG}} \ldots\ [\![F_n]\!]_{\text{CG}}$

```
  public static void Main() {
    object _temp;
```

$[\![D_1]\!]_{\text{CG}} \ldots\ [\![D_m]\!]_{\text{CG}}$

$statement(S_1) \ldots\ statement(S_l)$

```
  }
}
```

Since not every single expression is a valid statement in C#, we define the *statement* function to generate an artificial assignment to a temporary reference (`_temp`), converting an expression into a valid C# statement when necessary.

**Definition 5.1** *Given a statement node $S$, we define:*

$$statement(S) \equiv \begin{cases} \texttt{\_temp=}[\![S]\!]_{\text{CG}}\texttt{;} & \text{if } S \text{ is } E \text{ and } S \neq E_1\texttt{=}E_2 \text{ and } S \neq id(E^*) \\ [\![S]\!]_{\text{CG}}\texttt{;} & \text{otherwise} \end{cases}$$

## 5.4 Declarations

The .NET platform forces the declaration of each single variable with a unique type. We could simply declare variables and function parameters with their type erasures. However, this would generate many unnecessary casts. As an example, if a free type variable parameter is always used as an integer it is better to declare it as `int` rather than as `object` –its type erasure– (examples are the `x`, `y` and `z` parameters of the `point` function in Figure 5.1). This involves a faster execution because no cast will be generated.

For this purpose, we define the $[\![\ ]\!]_{\text{types}}$ traversal of the AST that collects all the possible types which a local variable may have in a function scope. Notice that this type collection is not the output environment obtained after type checking every function body, because our type system is flow sensitive: types bound to type variables change while type checking is performed. The *types* traversal returns an environment with all the possible types a local variable may have in a specific function. If a variable has more than one type, a union type is then used

to represent its least upper bound.

**Definition 5.2** *Given two environments* $\Gamma_1$ *and* $\Gamma_2$*, we define:*

$$\begin{array}{lll} \Gamma_1 \vee \Gamma_2 \equiv \Gamma \text{ in} & \Gamma \leftarrow \Gamma_1 & \\ & \forall\, id{:}T \in \Gamma_2, & add(id, T, \Gamma) \\ & \forall\, X{:}T \in \Gamma_2, & add(X, T, \Gamma) \end{array}$$

**Definition 5.3** *Given a type variable or identifier* $x$*, a type* $T$*, and an environment* $\Gamma$*, we define:*

$$add(x, T, \Gamma) \equiv \begin{cases} \Gamma \leftarrow \Gamma, x : T & \text{if } x \notin dom(\Gamma) \\ \Gamma \leftarrow \Gamma, x : \Gamma(x) \vee T & \text{otherwise} \end{cases}$$

To obtain all the possible types of a local variable, it is also necessary to know the actual C# types of the generated *global* functions. As an example, the `x`, `y` and `z` parameters in the `point` function (Figure 5.1) are only used in function invocations (lines 21 and 23). Since parameters of both `point2D` and `point3D` were declared as `int` in the C# destination code, the three `point` function parameters should also be declared as integers. Consequently, we define the *types* traversal not only returning the $\Gamma$ of local variables, but also receiving the $\Gamma$ that holds the type of every *global* function.

Once we obtain all the possible types of each local variable, we can pass them as a parameter to the translation process in order to optimize the generated C# code. Therefore, the $[\![\ ]\!]_{\text{CG}}$ code generation function will from now on receive a $\Gamma$ parameter. This parameter contains all the possible types of each local variable in the current scope, plus the C# types of the previously declared functions. We should then extend the code generation template for a program, adding the following code to the translation scheme shown above (the *statement* function –Definition 5.1– has also been extended with the appropriate $\Gamma_{\text{local}}$ parameter):

$$\begin{array}{l} [\![P = F_1 \ldots F_n\ D_1 \ldots D_m\ S_1 \ldots S_l]\!]_{\text{CG}} \triangleq \\ \quad \Gamma_{\text{global}} \leftarrow \emptyset \\ \quad \Gamma_{\text{local}_1} \leftarrow [\![F_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \\ \quad [\![F_1]\!]_{\text{CG}}(\Gamma_{\text{local}_1} \vee \Gamma_{\text{global}}) \\ \quad \ldots \\ \quad \Gamma_{\text{local}_n} \leftarrow [\![F_n]\!]_{\text{types}}(\Gamma_{\text{global}}) \\ \quad [\![F_n]\!]_{\text{CG}}(\Gamma_{\text{local}_n} \vee \Gamma_{\text{global}}) \\ \quad \Gamma_{\text{local}_{\text{main}}} \leftarrow [\![D_1 \ldots D_m\ S_1 \ldots S_l]\!]_{\text{types}}(\Gamma_{\text{global}}) \\ \quad [\![D_1]\!]_{\text{CG}}(\Gamma_{\text{local}_{\text{main}}} \vee \Gamma_{\text{global}}) \\ \quad \ldots \\ \quad [\![D_m]\!]_{\text{CG}}(\Gamma_{\text{local}_{\text{main}}} \vee \Gamma_{\text{global}}) \\ \quad statement(S_1, \Gamma_{\text{local}_{\text{main}}} \vee \Gamma_{\text{global}}) \\ \quad \ldots \\ \quad statement(S_l, \Gamma_{\text{local}_{\text{main}}} \vee \Gamma_{\text{global}}) \end{array}$$

We now define how types of local variables are obtained, i.e., the $[\![\ ]\!]_{\text{types}}$ traver-

sal. Local types in declarations and statements are the union (Definition 5.2) of the local environments they return.

$$
\begin{array}{l}
[\![D_1 \ldots D_m\ S_1 \ldots S_l\ R]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq \\
\quad \text{return } [\![D_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee [\![D_m]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \\
\qquad\quad [\![S_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee [\![S_l]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \\
\qquad\quad [\![R]\!]_{\text{types}}(\Gamma_{\text{global}})
\end{array}
$$

For functions, the union of their parameters, declarations and statements are added to the local environment. Besides, the function type is added to the global environment, taking its parameter types (and return type) from the local environment. That is, the function type added to $\Gamma_{\text{global}}$ holds the generated C# type –not the one inferred by the compiler.

$$
\begin{array}{l}
[\![F{=}ST\ id(ST_1\ id_1 ... ST_n\ id_n)\ D_1 ... D_m\ S_1 ... S_l\ R]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq \\
\quad \Gamma_{\text{local}} \leftarrow id_1{:}T_1 ... id_n{:}T_n \vee [\![D_1 ... D_m\ S_1 ... S_l\ R]\!]_{\text{types}}(\Gamma_{\text{global}}) \\
\quad \Gamma_{\text{global}} \leftarrow \Gamma_{\text{global}} \vee id{:}T'_1 \times \ldots \times T'_n \to T' \\
\quad \text{return } \Gamma_{\text{local}} \\
\text{where } \Gamma_{\text{F}}; \Omega_{\text{F}} \vdash id : T_1 \times \ldots \times T_n \to T, \\
\quad \Gamma_{\text{local}}; \Omega_{\text{F}} \vdash id_1{:}T'_1,\ \ldots\ \ \Gamma_{\text{local}}; \Omega_{\text{F}} \vdash id_n{:}T'_n, \text{ and} \\
\quad T' = \begin{cases} \Gamma_{\text{local}}(T) & \text{if } T \in dom(\Gamma_{\text{local}}) \\ T & \text{otherwise} \end{cases}
\end{array}
$$

The rest of the code generation templates follow the same structure, returning the union of the Γs returned by its descendants. In addition, if one expression must have a specific type (e.g. integer in arithmetic expressions) and the type inferred is a type variable, that specific type is then added to a union type bound to the type variable.

$$
\begin{array}{l}
[\![\texttt{if}\ E\ S_1 \ldots S_n\ S_{n+1} \ldots S_{n+m}]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq \\
\quad \Gamma_{\text{local}} \leftarrow [\![E]\!]_{\text{types}}(\Gamma_{\text{global}}) \\
\quad \text{if } \Gamma_{\text{E}}; \Omega_{\text{E}} \vdash E : X \\
\qquad add(X, \text{bool}, \Gamma_{\text{local}}) \\
\quad \text{return } \Gamma_{\text{local}} \vee [\![S_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee\ [\![S_n]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \\
\qquad\quad [\![S_{n+1}]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee\ [\![S_{n+m}]\!]_{\text{types}}(\Gamma_{\text{global}})
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{while}\ E\ S_1 \ldots S_n]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq \\
\quad \Gamma_{\text{local}} \leftarrow [\![E]\!]_{\text{types}}(\Gamma_{\text{global}}) \\
\quad \text{if } \Gamma_{\text{E}}; \Omega_{\text{E}} \vdash E : X \\
\qquad add(X, \text{bool}, \Gamma_{\text{local}}) \\
\quad \text{return } \Gamma_{\text{local}} \vee [\![S_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee ... \vee\ [\![S_n]\!]_{\text{types}}(\Gamma_{\text{global}})
\end{array}
$$

$[\![\texttt{return}\ E]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\quad$ if $tv(\Omega_{\text{E.rt}})$
$\quad\quad add(\Omega_{\text{E.rt}}, T, \Gamma_{\text{local}})$
$\quad$ return $\Gamma_{\text{local}}$

$[\![ST\ id]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$ return $id{:}ST$

$[\![id]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$ return $\emptyset$

$[\![E_1 \oplus E_2]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee [\![E_2]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\quad$ if $\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : X_1$
$\quad\quad add(X_1, \text{int}, \Gamma_{\text{local}})$
$\quad$ if $\Gamma_{E_2}; \Omega_{E_2} \vdash E_2 : X_2$
$\quad\quad add(X_2, \text{int}, \Gamma_{\text{local}})$
$\quad$ return $\Gamma_{\text{local}}$

$[\![E_1\texttt{=}E_2]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee [\![E_2]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\quad$ if $\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : X_1$
$\quad\quad add(X_1, T_2, \Gamma_{\text{local}})$
$\quad$ return $\Gamma_{\text{local}}$
where $\Gamma_{E_2}; \Omega_{E_2} \vdash E_2 : T_2$

$[\![E_1\,\texttt{[}E_2\texttt{]}]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee [\![E_2]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\quad$ if $\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : X_1$
$\quad\quad add(X_1, Array(T), \Gamma_{\text{local}})$
$\quad$ if $\Gamma_{E_2}; \Omega_{E_2} \vdash E_2 : X_2$
$\quad\quad add(X_2, \text{int}, \Gamma_{\text{local}})$
$\quad$ return $\Gamma_{\text{local}}$
where $\Gamma_{E_1[E_2]}; \Omega_{E_1[E_2]} \vdash E_1[E_2] : T$

$[\![E\,.\,id]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\quad$ if $\Gamma_E; \Omega_E \vdash E : X$
$\quad\quad add(X, [id{:}T], \Gamma_{\text{local}})$
$\quad$ return $\Gamma_{\text{local}}$

$[\![\texttt{new}\ ST\ \texttt{[}E\texttt{]}(\texttt{[]})^*]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$ return $[\![E]\!]_{\text{types}}(\Gamma_{\text{global}})$

$[\![\texttt{new}\ \{id_1\texttt{=}E_1, \ldots, id_n\texttt{=}E_n\}]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad$ return $[\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee [\![E_n]\!]_{\text{types}}(\Gamma_{\text{global}})$

$[\![\texttt{true}]\!]_{\text{types}}(\Gamma_{\text{global}}) = [\![\texttt{false}]\!]_{\text{types}}(\Gamma_{\text{global}}) = [\![IntLiteral]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$ return $\emptyset$

In the invocation expression, the function type is taken from $\Gamma_{\text{global}}$ rather than from the inferred type, reducing the number of casts in the generated code.

$$
\begin{array}{l}
[\![id(E_1 \ldots E_n)]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq \\
\quad \Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee [\![E_n]\!]_{\text{types}}(\Gamma_{\text{global}}) \\
\quad \forall\, i \in [1, n] \\
\qquad \text{if } \Gamma_{E_i}; \Omega_{E_i} \vdash E_i : X_i \\
\qquad\quad add(X_i, T_i, \Gamma_{\text{local}}) \\
\quad \text{return } \Gamma_{\text{local}} \\
\text{where } \Gamma_{\text{global}}(id) : T_1 \times \ldots \times T_n \to T
\end{array}
$$

Finally, we can now define the $[\![\ ]\!]_{\text{CG}}$ template for local variable declarations, using the type erasures of the types inferred in the local scope.

$$
\begin{array}{l}
[\![D = ST\ id]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq |T|_{\Gamma_{\text{local}}}\ id\ ; \\
\quad \text{where } \Gamma_{\text{local}}; \Omega_D \vdash id : T
\end{array}
$$

Following the same process, each function is translated to a private `static` method in C#. The return type and the types of the parameters are the erasures of the types held in the local $\Gamma$. The return statement is the last one to be generated.

$$
\begin{array}{l}
[\![F{=}ST\ id(ST_1\ id_1 ... ST_n\ id_n)\ D_1 ... D_m\ S_1 ... S_l\ R]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq \\
\quad \texttt{static}\ |T'|_{\Gamma_{\text{local}}}\ id(|T'_1|_{\Gamma_{\text{local}}}\ id_1, \ldots, |T'_n|_{\Gamma_{\text{local}}}\ id_n)\{ \\
\qquad \texttt{object \_temp;} \\
\qquad [\![D_1]\!]_{\text{CG}}(\Gamma_{\text{local}}) \ldots\ [\![D_m]\!]_{\text{CG}}(\Gamma_{\text{local}}) \\
\qquad statement(S_1, \Gamma_{\text{local}}) \ldots\ statement(S_l, \Gamma_{\text{local}}) \\
\qquad [\![R]\!]_{\text{CG}}(\Gamma_{\text{local}}) \\
\quad \} \\
\text{where} \quad \Gamma_{\text{local}}; \Omega_F \vdash id_1 : T'_1, \ldots, \Gamma_{\text{local}}; \Omega_F \vdash id_n : T'_n, \text{ and} \\
\qquad \Gamma_{\text{local}}; \Omega_F \vdash id : T_{p_1} \times \ldots \times T_{p_n} \to T'
\end{array}
$$

## 5.5 Basic Expressions

To optimize runtime performance of the generated code, we define the $[\![\ ]\!]_{\text{CG}}$ traversal for expressions returning the type erasure of the generated expression. This makes it easier to reduce the number of unnecessary casts. Following this scheme, code generation of basic expressions is defined as follows:

$$[\![\texttt{true}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \begin{array}{l}\texttt{true}\\ \text{return bool}\end{array}$$

$$[\![\texttt{false}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \begin{array}{l}\texttt{false}\\ \text{return bool}\end{array}$$

$$[\![\mathit{IntLiteral}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \begin{array}{l}\mathit{IntLiteral}\\ \text{return int}\end{array}$$

$$[\![id]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \begin{array}{l}id\\ \text{return } |T|_{\Gamma_{\mathrm{local}}}\end{array}$$
where $\Gamma_{\mathrm{local}}; \Omega_{id} \vdash id : T$

Notice that the type erasure of the identifier is taken from the local environment, returning the least upper bound of all its possible C# types in the local scope.

**Definition 5.4** *Given two type erasures $T_1$ and $T_2$, an expression node $E$, and an environment $\Gamma$, we define:*

$$\mathit{cast}(T_1, T_2, E, \Gamma) \equiv \begin{cases} \texttt{((}T_2\texttt{)}[\![E]\!]_{\mathrm{CG}}(\Gamma)\texttt{)} & \text{if } T_1 \neq T_2 \text{ and } T_2 \neq \text{object and} \\ & \quad \text{not}(T_2{=}\mathit{Array} \text{ and } T_1{=}T([\,])^+) \\ \texttt{(}[\![E]\!]_{\mathrm{CG}}(\Gamma)\texttt{)} & \text{otherwise} \end{cases}$$

The *cast* function generates code for the $E$ expression including a cast when necessary. In case the types are the same, or the destination is `object`, or an array type is cast to the .NET `Array` type, the cast will not be generated.

To avoid generating unnecessary `object` type erasures for the types inferred by the compiler, we use the following properties of union types:

$$T \vee T \longrightarrow T$$
$$T_1 \vee T_2 \equiv T_2 \vee T_1$$
$$(T_1 \vee T_2) \vee T_3 \equiv T_1 \vee (T_2 \vee T_3) \longrightarrow T_1 \vee T_2 \vee T_3$$
$$\mathit{Array}(T_1) \vee \mathit{Array}(T_2) \longrightarrow \mathit{Array}(T_1 \vee T_2)$$
$$\{id_1{:}T_1, ..., id_n{:}T_n\} \vee [id_i{:}T_i]^{i\in[1,n]} \longrightarrow \{id_1{:}T_1, ..., id_n{:}T_n\}$$
$$[id_1{:}T_1, ..., id_n{:}T_n] \vee [id_i{:}T_i]^{i\in[1,n]} \longrightarrow [id_1{:}T_1, ..., id_n{:}T_n]$$

If a type already exists in a union type, it is not added. Union types are commutative, and nesting is avoided. A union of arrays is represented with an array of unions; this way, the union of objects will not be erased to the `object` type. If all the field labels in a member type exist in an object type and the corresponding types are equal, the member type can be deleted from the union

type. The previous property also holds for member types.

We now define the generation of arithmetic expressions (logical and relational ones are similar). The first operand is translated to C# and, if necessary, a cast to integer is inserted. If the type of one of the operands is dynamic and it is not a subtype of `int`, an `InvalidCastException` will be thrown by the CLR at runtime. The generated code does not perform extra type checking at runtime because it is already done by the CLR.

$$
\begin{array}{ll}
& [\![E_1 \oplus E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\
& \qquad cast(T_1, \mathrm{int}, E_1, \Gamma_{\mathrm{local}})\ \mathtt{op}_{\oplus}\ cast(T_2, \mathrm{int}, E_2, \Gamma_{\mathrm{local}}) \\
& \qquad \text{return int} \\
\text{where} & \quad T_1 = [\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}),\ T_2 = [\![E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}), \\
& \quad \mathtt{op}_{+} = \mathtt{+},\ \mathtt{op}_{-} = \mathtt{-},\ \mathtt{op}_{*} = \mathtt{*},\ \text{and}\ \mathtt{op}_{/} = \mathtt{/}
\end{array}
$$

At function invocation, each argument is converted to the corresponding parameter type. These parameter types are taken from the environment parameter ($\Gamma_{\mathrm{local}}$). Therefore, the arguments may be cast to the actual C# types of the declared function. For instance, although the type erasure of the four parameters of the `point` function (Figure 5.1) is `object`, all of them were declared as integers. Therefore, arguments of any `point` function call should be cast to `int`, when necessary. The return type erasure follows the same process.

$$
\begin{array}{ll}
& [\![id\mathtt{(}E_1 \ldots E_n\mathtt{)}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\
& \qquad id\mathtt{(}cast(T_1, |T_{p_1}|_{\Gamma_{\mathrm{local}}}, E_1, \Gamma_{\mathrm{local}})\mathtt{,}\ \ldots\ \mathtt{,}cast(T_n, |T_{p_n}|_{\Gamma_{\mathrm{local}}}, E_n, \Gamma_{\mathrm{local}})\mathtt{)} \\
& \qquad \text{return } |T|_{\Gamma_{\mathrm{local}}} \\
\text{where } T_1 = [\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}), \ldots,\ T_n = [\![E_n]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}), & \text{and} \\
\quad \Gamma_{\mathrm{local}}(id) = T_{p_1} \times \ldots \times T_{p_n} \to T &
\end{array}
$$

In assignments, the type erasure of the right-hand side must be converted to the type erasure of the left-hand side.

$$
\begin{array}{ll}
& [\![E_1\mathtt{=}E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\
& \qquad [\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})\ \mathtt{=}\ cast(T_2, T_1, E_2, \Gamma_{\mathrm{local}}) \\
& \qquad \text{return } T_1 \\
\text{where } T_1 = [\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}),\ T_2 = [\![E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) &
\end{array}
$$

Objects are created by calling the default constructors of their corresponding anonymous classes, and arrays allocation is translated into its analogous C# syntax.

$[\![E = \texttt{new}\{id_1\texttt{=}E_1, \ldots, id_n\texttt{=}E_n\}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  `new AC_`$|T_1|_{\Gamma_E}$`_`$id_1$`_`$\ldots$`_`$|T_n|_{\Gamma_E}$`_`$id_n$ {
    $id_1$`=`$[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}), \ldots, id_n$`=`$[\![E_n]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$
  }
  return AC_$|T_1|_{\Gamma_E}$_$id_1$_$\ldots$_$|T_n|_{\Gamma_E}$_$id_n$
where $id_1 \ldots id_n$ are lexicographically ordered, and
  in AC_$|T_1|_{\Gamma}$_$id_1$_$\ldots$_$|T_n|_{\Gamma}$_$id_n$, $T[\,]_1 \ldots [\,]_n$ is replaced with T_n

$[\![E = \texttt{new}\ ST\texttt{[}E_1\texttt{][ ]}\ldots\texttt{[ ]}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  `new` $|ST|_{\Gamma_E}$ `[`$[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$`][ ]`$\ldots$`[ ]`
  return $|ST|_{\Gamma_E}[\,][\,]\ldots[\,]$

## 5.6 Statements

In the `if` and `while` statements, the condition expression is checked to be `bool`. The rest of the translation process is similar to the code in functions.

$[\![\texttt{if}\ E\ S_1 \ldots S_n\ S_{n+1} \ldots S_{n+m}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  `if (` $cast(T, \mathrm{bool}, E, \Gamma_{\mathrm{local}})$ `) {`
    $statement(S_1, \Gamma_{\mathrm{local}}) \ldots\ statement(S_n, \Gamma_{\mathrm{local}})$
  }
  `else {`
    $statement(S_{n+1}, \Gamma_{\mathrm{local}}) \ldots\ statement(S_{n+m}, \Gamma_{\mathrm{local}})$
  }
where $T = [\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$

$[\![\texttt{while}\ E\ S_1 \ldots S_n]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  `while (` $cast(T, \mathrm{bool}, E, \Gamma_{\mathrm{local}})$ `) {`
    $statement(S_1, \Gamma_{\mathrm{local}}) \ldots\ statement(S_n, \Gamma_{\mathrm{local}})$
  }
where $T = [\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$

$[\![\texttt{return}\ E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$ `return` $[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$

## 5.7 Field Access

In the first scenario, the expression is an object type and the field can be obtained directly.

$$\frac{\Gamma_E;\Omega_E \vdash E : \{id_1{:}T_1,\ldots,id_n{:}T_n\}}{\begin{array}{l}[\![E\,.\,id_i]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\ \quad cast(T,|\{id_1{:}T_1,\ldots,id_n{:}T_n\}|_{\Gamma_E},E,\Gamma_{\mathrm{local}})\ .\ id_i \\ \quad \text{return } |T_i|_{\Gamma_E} \\ \text{where } T = [\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})\end{array}}$$

In case no type information has been gathered by the compiler, the field value is obtained using reflection. The same happens when it is only known that it is an object with the appropriate field, not knowing its specific type, i.e., it is a member type.

$$\frac{\Gamma_E;\Omega_E \vdash E : T \qquad T \in ftv(\Gamma_E) \text{ or } T = [...,id_i{:}T_i,...]}{\begin{array}{l}[\![E\,.\,id_i]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\ \quad \texttt{(\_temp=}[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})\texttt{).GetType().GetField("}id_i\texttt{").GetValue(\_temp)} \\ \quad \text{return object}\end{array}}$$

Under the same circumstances, if a field value is modified with the assignment operator, the `_setValue` helper method is used. The `_setValue` method simply returns the field value after the assignment. This method is necessary for generating a valid C# expression, because the `SetValue` method of the .NET's reflection API does not return any value.

$$\frac{\Gamma_{E_1};\Omega_{E_1} \vdash E_1 : T \qquad T{\in}ftv(\Gamma_{E_1}) \text{ or } T{=}[...,id_i{:}T_i,...]}{\begin{array}{l}[\![E_1\,.\,id_i \texttt{ = } E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\ \quad \texttt{\_setValue(}[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})\texttt{,"}id_i\texttt{",}[\![E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{global}})\texttt{)} \\ \quad \text{return object}\end{array}}$$

In the case of static union types, the generated code is optimized using the type information gathered statically. We use the ternary conditional operator to dynamically check the actual type from all the possible ones inferred by the compiler[1]. At runtime, this conditional code is significantly faster than reflection, which is the implementation of dynamic typing for both C# and Visual Basic [76, 88]. If the union type holds one (or more) free type variables, the last alternative in the conditional expression obtains the field value using reflection. Since this is the slowest alternative, we generate it as the last option in order to optimize runtime performance of the generated code.

[1]We use reflection when the number of types in the union type is greater than 120. We have measured that reflection is faster when the number of elements in a union type is more than 146.

$\Gamma_E;\Omega_E \vdash E : \texttt{sta}\ T_1 \vee ... \vee T_n \qquad \Gamma_{E.id};\Omega_{E.id} \vdash E.id : T$

---

$[\![E\,.id]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  $\forall\, i \in [1,n], T_i \notin ftv(\Gamma_E)$
    if *it is not the last iteration* or $\exists\, j{\in}[1,n], T_j{\in}ftv(\Gamma_E)$
      $\begin{cases} \texttt{\_temp=}[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) & \text{if } \textit{it is the first iteration} \\ \texttt{: \_temp} & \text{otherwise} \end{cases}$
        $\texttt{is}\ |T_i|_{\Gamma_E}\ \texttt{?(}|T|_{\Gamma_{E.id}}\texttt{)((}|T_i|_{\Gamma_E}\texttt{)\_temp).}id$
  if $\exists\, i \in [1,n], T_i \in ftv(\Gamma_E)$
    $\texttt{:(}|T|_{\Gamma_{E.id}}\texttt{)( \_temp.GetType().GetField("}id\texttt{").GetValue(\_temp) )}$
  return $|T|_{\Gamma_{E.id}}$

An example of the previous code generation template can be seen in line 32 of Figure 5.1. The type of `sta` is $\{x{:}\texttt{int}, y{:}\texttt{int}\} \vee \{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}\}$. In the first iteration, the object expression (`sta`) is assigned to `_temp` and it is checked whether it is $\{x{:}\texttt{int}, y{:}\texttt{int}\}$. If so, a cast is performed and the `x` field is obtained. The second condition is similar, but asking for the $\{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}\}$ type. Since the union type does not hold any free type variable, reflection is not used in another last condition.

When the expression type is dynamic, it should be taken into consideration that there may be types in the union type that do not provide the expected field. The first optimization consists in generating code only for those types that accept the specific field access operation, using the ternary conditional operator. A performance benefit is obtained because the generated code only checks for those types that are applicable. The last alternative generated is reflection. At runtime, if the field is still not found, a runtime exception will be thrown. This may happen when dynamic references are used, because it is not guaranteed that the field actually exists. Another final optimization is implemented when only one possible type fulfills the condition. In this case, a direct access to the field is generated (an `InvalidCastException` could be raised by the CLR).

$\Gamma_E;\Omega_E \vdash E : \texttt{dyn}\ T_1 \vee ... \vee T_n \qquad \Gamma_{E.id};\Omega_{E.id} \vdash E.id : T$

---

$[\![E\,.id]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
  if *only one* $T_i^{\,i\in[1,n]}$ *fullfils* $\Gamma_E;\Omega_E \vdash T_i{\leq}[id{:}T_i']$ ($T_i'$ fresh) and $T_i \notin ftv(\Gamma_{E.id})$
    $cast(T_E, |T_i|_{\Gamma_E}, E, \Gamma_{\mathrm{local}})\ .\ id$
    return $|T|_{\Gamma_{E.id}}$
  else
    $\forall\, i \in [1,n],\ \Gamma_E;\Omega_E \vdash T_i \leq [id : T_i']$ ($T_i'$ fresh) and $T_i \notin ftv(\Gamma_E)$
      $\begin{cases} \texttt{\_temp=}[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) & \text{if } \textit{it is the first iteration} \\ \texttt{\_temp} & \text{otherwise} \end{cases}$
        $\texttt{is}\ |T_i|_{\Gamma_E}\ \texttt{?(}|T|_{\Gamma_{E.id}}\texttt{)((}|T_i|_{\Gamma_E}\texttt{)\_temp).}id\ \texttt{:}$
      $\texttt{(}|T|_{\Gamma_{E.id}}\texttt{)( \_temp.GetType().GetField("}id\texttt{").GetValue(\_temp) )}$
      return $|T|_{\Gamma_{E.id}}$
  where $T_E = [\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$

Line 33 in Figure 5.1 is an example of accessing the `y` field of a dynamic union type. The ternary operator is the same as the previous field access (`sta.x`), but reflection is used in the last condition. We use reflection because the dynamic `din` reference may point to an object that does not implement the `y` field (it is a dynamic union type). Finally, line 34 generates faster code generating a direct cast because only one possible type in the union type ($\{x$:`int`, $y$:`int`, $z$:`int`$\}$) offers the `z` field.

Two special generation templates were specified to translate assignments of field access expressions when the object is a union type. Since they imply a simple modification of the two previous translation rules, we do not depict them.

## 5.8 Array Indexing

In the first scenario, the expression is an array type.

$$\frac{\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : Array(T)}{\begin{array}{l}[\![E_1\texttt{[}E_2\texttt{]}]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq \\ \quad cast(T_1, T[\,], E_1, \Gamma_{\text{local}})\ \texttt{[}\ cast(T_2, \text{int}, E_2, \Gamma_{\text{local}})\ \texttt{]} \\ \quad \text{return } |T|_{\Gamma_{E_1[E_2]}} \\ \text{where } T_1 = [\![E_1]\!](\Gamma_{\text{local}}), \text{ and } T_2 = [\![E_2]\!](\Gamma_{\text{local}})\end{array}}$$

If the first expression is not an array, reflection is used (the `GetValue` method of the .NET's `Array` class). Notice that it cannot be a union of arrays because of the way we create union types (§ 5.5). In that case, the type would be an array of union types.

$$\frac{\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : T \qquad T \neq Array}{\begin{array}{l}[\![E_1\texttt{[}E_2\texttt{]}]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq \\ \quad cast(T_1, \text{Array}, E_1, \Gamma_{\text{local}})\texttt{.GetValue(}cast(T_2, \text{int}, E_2, \Gamma_{\text{local}})\texttt{)} \\ \quad \text{return object} \\ \text{where } T_1 = [\![E_1]\!](\Gamma_{\text{local}}), \text{ and } T_2 = [\![E_2]\!](\Gamma_{\text{local}})\end{array}}$$

We have overloaded the `_setValue` method because the .NET `SetValue` method does not return the assigned value. It assigns values to an array element by means of reflection.

$$\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : T \qquad\qquad T \neq Array$$

$[\![E_1\texttt{[}E_2\texttt{]=}E_3]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
`_setValue(`$cast(T_1, \mathrm{Array}, E_1, \Gamma_{\mathrm{local}})$`,` $[\![E_3]\!](\Gamma_{\mathrm{local}})$`,` $cast(T_2, \mathrm{int}, E_2, \Gamma_{\mathrm{local}})$`)`
return object
where $T_1 = [\![E_1]\!](\Gamma_{\mathrm{local}})$, $T_2 = [\![E_2]\!](\Gamma_{\mathrm{local}})$, and$T_3 = [\![E_3]\!](\Gamma_{\mathrm{local}})$

# Chapter 6

# Evaluation

In this section, an assessment of the proposed hybrid static and dynamic type system is presented. The first subsection outlines the experimental methodology employed, programming languages and benchmarks used. The evaluation has been divided into four scenarios: micro-benchmark, dynamically typed code, hybrid dynamic and static typing code, and explicitly typed code. For each scenario, data of the runtime performance and memory consumption for different languages running a set of applications are presented and discussed (the whole evaluation data can be consulted in Appendices B and C).

## 6.1 Methodology

In order to assess the *StaDyn* programming language presented in this PhD thesis, its runtime performance and memory consumption have been compared with the most widely-used hybrid and dynamic programming languages for the .Net Framework 4.0, compiled with their maximum optimization options. Our evaluation is focused on .Net languages to avoid the introduction of a bias caused by the use of different platforms (such as Java or native applications). These are the languages we have used in the evaluation:

- C# 4.0. The latest version of C# combines static and dynamic typing with its new `dynamic` type (see Chapter 2). Its back-end is the DLR, released as part of the .Net Framework 4.0. The DLR is a new layer over the CLR to provide a set of services to facilitate the implementation of dynamic languages [13].
- IronPython 2.7.3 It is an open-source implementation of the Python programming language which is tightly integrated with the .Net Framework, targeting the DLR. It compiles Python programs into IL (Intermediate Language) bytecodes [89].
- Visual Basic (VB) 10: The VB programming language also supports dynamic typing [25]. 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, and a special focus on language and compiler extensibility. It is statically typed, but also provides duck typing by using its special type `duck` [24].
- Cobra 0.9.1: A hybrid statically and dynamically typed programming language. It is object-oriented and provides compile-time type inference [31]. As C#, dynamic typing is provided with a distinctive `dynamic` type.
- Fantom 1.0.63: Fantom is an object-oriented programming language than generates code to the Java VM, .NET platform, and JavaScript. It is statically typed, but provides dynamic invocation of methods with a specific message-passing operator [32].
- *StaDyn*. The same programs coded in C# 4.0 are simply translated into *StaDyn* by replacing the `dynamic` reserved word with `var`.

All these languages compile code to the .NET framework, facilitating the comparison of performance results. This way, the measurements obtained show the performance improvement of gathering type information of dynamic references at compile time. For each language, a version of the following applications using different kinds of typing has been measured:

- Micro-benchmark. We have developed a synthetic micro-benchmark to evaluate the influence of static type information gathered by the compiler.
- *Pybench* [90]. A Python benchmark designed to measure the performance of standard Python implementations. *Pybench* is composed of a collection of 52 tests that measure different aspects of the Python programming language.
- *Pystone*. This benchmark is the Python version of the Dhrystone benchmark [91] and is commonly used to compare different implementations of the Python programming language. *Pystone* is included in the standard CPython distribution.
- A subset of the *Java Grande* benchmark [92]:
  - 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*. We have extended the *StaDyn* core program in Figure 4.1, filling the list with 10,000 random two and three dimensional points. The two

`positiveX` and `closestToOrigin3D` functions are called passing the list reference as an argument.

Regarding the data analysis, we followed the methodology proposed in [93] to evaluate the performance of virtual machines that provide JIT-compilation. We followed the two step methodology defined to evaluate non-server 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 computed using the *Student's* $t$-distribution because we took $p = 30$ [94]. Therefore, we compute the confidence interval $[c_1, c_2]$ as:

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

Being $\overline{x}$ the arithmetic mean of the $x_i$ measurements, $\alpha = 0.05(95\%)$, $s$ the standard deviation of the $x_i$ measurements, and $t_{1-\alpha/2;p-1}$ defined such that a random variable $T$, that follows the *Student's* $t$-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1 - \alpha/2$.

The data provided (Appendices B and C) is the mean of the confidence interval plus a percentage indicating the width of the confidence interval relative to the mean.

To measure 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 operating system Performance and Reliability Monitor [95]. 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 [96].

All the tests were carried out on a 2.13 GHz Intel Core 2 Duo P7450 system with 4 GB of RAM running an updated 64-bit version of Windows 7 Home Premium SP1. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded. If the $L_1$ and $L_2$ languages run the same benchmark in $T$ and $2.5 \times T$ milliseconds, respectively, we say that runtime performance of $L_1$ is 150% (or 1.5 times) higher than $L_2$, $L_1$ is 150% (or 1.5 times) faster, $L_2$ requires 150% (or 1.5 times) more execution time than $L_1$, or the performance benefit of $L_1$ compared to $L_2$ is 150% –the same for memory consumption. To compute average percentages, factors and orders of magnitude, we use the geometric mean.

## 6.2 Micro-benchmark

To evaluate the influence of static type information gathered by the compiler, we have developed a synthetic micro-benchmark that takes the following scenarios into account:

- Explicit static type declaration. No `var` references are used at all, explicitly stating the type of every variable.
- Implicit dynamic `var` references declaration, when the compiler manages to infer different possible types. Figure 4.1 is a basic example of this kind of type inference, where the compiler infers two possible types. For this scenario, we measure code where 1, 5, 10, 50 and 100 possible types could be inferred statically.
- Implicit dynamic type reference declaration, when the compiler does not infer any type at all. In this scenario, dynamic `var` references are used as parameters. The argument reference randomly holds an object from 100 different types.

We measure the invocation to a polymorphic method that performs basic arithmetical operations in a loop of 100,000 iterations. Its implementation depends on the type of its parameters, local variables, object fields and the object itself (it is polymorphic). Since the difference of our approach is the type information gathered by the compiler, the primitives to measure are not really significant because, excluding reflection services, most low-level operations in .NET are statically typed. We have coded the described program with the hybrid programming languages appointed in § 6.1 (all but IronPython).

Table B.1 and Figure 6.1 show the execution time elapsed to run 1,000 iterations of each test, expressed in microseconds. Table B.1 also includes a percentage indicating the width of the 95% confidence interval.

The first test only uses explicit type declaration, and hence dynamic typing is not used at all. C# offers the best runtime performance, being 10.2% faster than *StaDyn*, the second fastest language. This difference is caused by the number of optimizations that the C# production compiler performs in relation to our

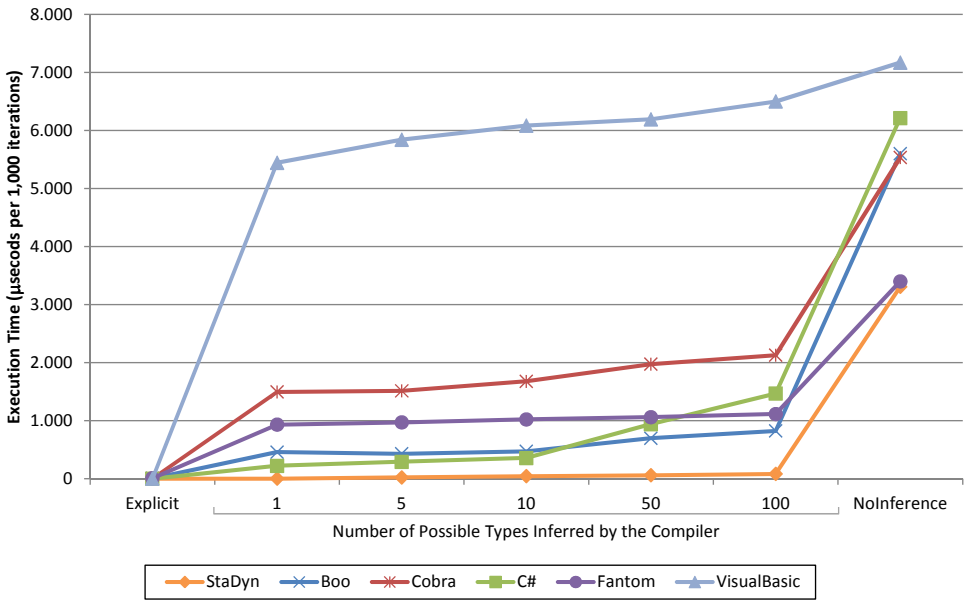



Figure 6.1: Execution time of the micro-benchmark.

implementation. However, *StaDyn* is 1.82%, 21.37%, 143% and 294% faster than Cobra, VB, Boo, and Fantom, respectively, when types are explicitly declared.

The runtime performance of *StaDyn* when the exact single type of a `var` reference is inferred shows the repercussion of our approach. Runtime performance is roughly the same as when using explicitly typed references (in fact, the code generated is exactly the same). In this special scenario, *StaDyn* shows a huge performance improvement. If the compiler infers the exact type of `var` references, *StaDyn* is more than 92 times faster than C#. For the rest of programming languages, *StaDyn* is 2.28, 2.59, 2.79 and 3.36 orders of magnitude faster than Boo, Fantom, Cobra and VB, respectively. This vast difference is caused by the lack of static type inference of these languages for this test. When a reference is declared as dynamic, every operation on that reference is performed at runtime using reflection. The big difference between C# and VB may be caused by the benefits of using the reflective services of the DLR (C#), compared to the reflective functions of the CLR (VB) [13]. The usage of reflective operations in the .Net platform involves an important performance cost [88].

Figure 6.1 shows the progression of execution times when the compiler infers 1, 5, 10, 50 and 100 possible types. In order to test the association between the number of possible types and the execution time, we carried out a regression analysis for linear relationship between these two variables, conducting an analysis of variance (ANOVA). Excluding VB, all the values of the Pearson coefficient were greater than 0.8 (0.728 for VB), with levels of significance ($p$ values) below 0.0386. *StaDyn* showed the lowest slope coefficient, 0.66, compared with the 1.58, 4.05, 6.41, 8.01, and 12.67 values exhibited by Fantom, Boo, Cobra, VB and C#, respectively. The slope coefficient represents the increase of execution time given an increase of one possible type inferred by the compiler. Therefore, as shown in Figure 6.1, *StaDyn* not only outperforms the rest of languages when the compiler manages to infer different possible types, but also shows the smallest growth of

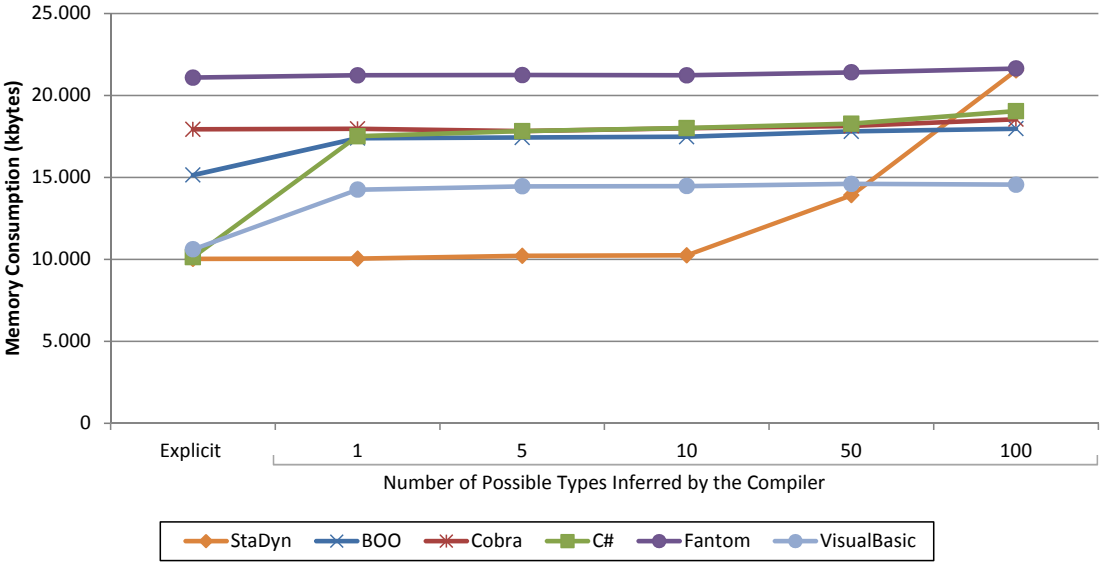



Figure 6.2: Memory consumption of the micro-benchmark.

execution time relative to the number of possible types inferred.

The final comparison to be made is when the compiler gathers no static type information at all. In this case, runtime performance is the worst in all languages, because dynamic operations (e.g., method invocation) are performed using reflection. In this scenario, *StaDyn* is the fastest language, showing only 20.82% better performance than Fantom, but being 67.3%, 69.3%, 87.9% and 116.7% faster than Cobra, Boo, C# and VB, respectively.

### 6.2.1 Memory Consumption

Figure 6.2 shows the memory consumption for each language running the micro-benchmark tests, expressed in Kbytes (Table C.1 shows the detailed results). Using explicit type declaration, *StaDyn* and C# are the languages with the lowest memory consumption values (10.256.384 and 10.374.144 Kbytes, respectively). However VB, Boo, Cobra and Fantom require 6%, 51%, 80% and 110% more memory than *StaDyn*. The runtime implemented by these languages require more memory resources.

As in the performance evaluation, when *StaDyn* infers the exact single type of a `var` reference, memory consumption is the same as when using explicit typed references. Boo and VB require 14% and 34% more memory than using explicit type declaration, respectively. C# presents the most significant increase, consuming 72.2% more memory, because, when running dynamically typed code, it uses the DLR. Finally, Fantom and Cobra (which present the worst memory consumption) hardly change it.

The line chart shows the progression of memory consumption when the compiler infers 1, 5, 10, 50 and 100 possible types. Excluding *StaDyn*, the number of possible types inferred by the compiler does not involve a notable change in their memory consumptions. *StaDyn* does not follow this pattern, requiring more

memory resources as the number of possible types increases. This is caused by the fact that the SSA algorithm uses a greater number of temporary variables (see § 3.2.1), implying a higher memory consumption.

The differences between our approach and the rest of programming languages are justified by the type information gathered by the compiler. Unlike the rest of tested languages, *StaDyn* continues collecting type information when references are set as dynamic. This is the reason why *StaDyn* offers the same runtime performance with explicit type declaration and with inference of one exact single type, involving a remarkable performance improvement. Only when the number of possible types of a `var` reference is higher than 50, the memory resources consumed by *StaDyn* are not the lowest ones.

## 6.3 Dynamically Typed Code

Different benchmarks of dynamically typed languages have been evaluated, comparing all the selected languages. *StaDyn* is run with all the references declared as `var`, and compiling the code with the `everythingDynamic` (§ 3.2.3) option. Consequently, the efficiency of executing dynamically typed code in each language implementation is evaluated.

For this scenario we have taken two well-known benchmarks for the Python programming language: *Pystone* and *Pybench* (§ 6.1). We have suppressed those that employ particular features of Python not provided by the other languages (i.e., tuples, dynamic code evaluation, and Python-specific built-in functions); and those that use any input/output interaction. Therefore 31 tests of the *Pybench* benchmark were measured (see Table B.5 for a detailed list of the selected tests in the benchmark).

In addition, an explicitly typed benchmark (Java Grande) and a hybrid static and dynamic typing program (the *Points* code showed in Figure 4.1) have also been considered. In this section, the source code of these programs have been modified to make them fully dynamically typed (declaring all the references as dynamic). No explicit type declarations are used at all. In the following sections their code will be changed to make them hybrid and explicitly typed, measuring and comparing the different scenarios for the same applications.

The default number of iterations and data for all benchmarks have been used, except in the case of *JG.RayTracer* (Java Grande, section 3). For this benchmark 25 pixels instead of 150 pixels were used, due to significant execution time that some languages use to execute it using dynamic typing.

Figure 6.3 displays the average execution time relative to *StaDyn* for the *Pybench* benchmark (Table B.5 shows the complete list of execution times). *StaDyn* offers the best runtime performance in all test, on average is 3.71, 3.78, 11.61, 14.39, 45.54 and 73.82 times faster than IronPython, C#, Boo, VB, Fantom and Cobra, respectively. Differences between C# and IronPython and the rest of the languages (Boo, VB, Fantom and Cobra) may be due to the optimizations imple-

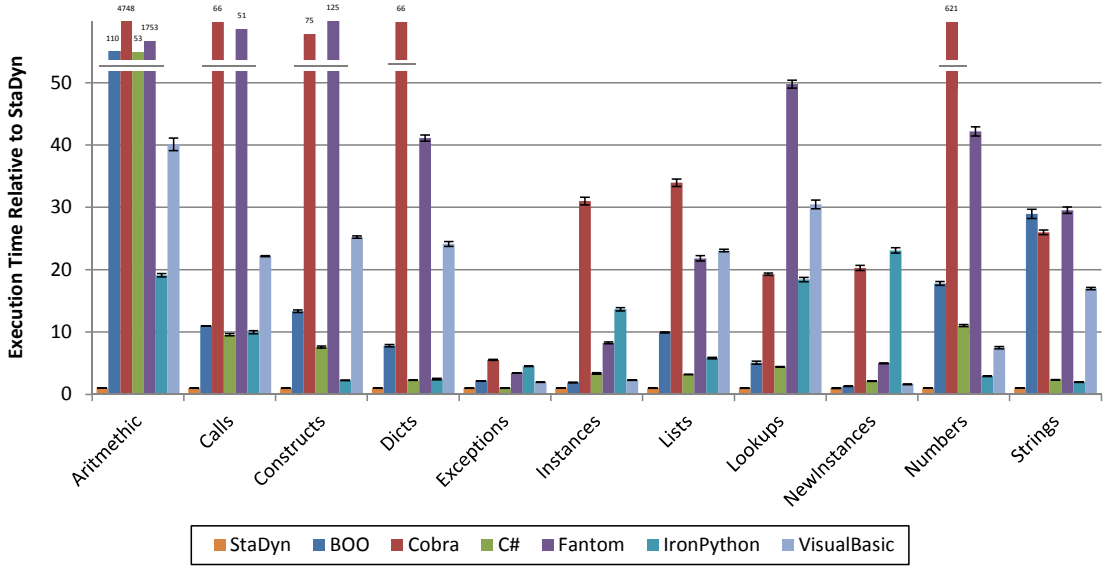



Figure 6.3: Execution time of the *Pybench* benchmark.

mented by the DLR (opposed to the CLR) when dynamically typed benchmark is executed.

The highest performance benefit obtained by *StaDyn* running dynamically typed code is with the *Arithmetic* test of *Pybench* (*StaDyn* is at least 180 times faster than the average measurements of the rest of languages). The second fastest language is IronPython, which requires 18 times more execution time than *StaDyn*; and the worst languages are Fantom and Cobra, that require 3.24 and 3.67 orders of magnitude more than *StaDyn*, respectively. This difference is caused by how each language performs the arithmetic operations over dynamically typed operands. Most languages use reflection to execute the operation at runtime, introducing an important performance cost [88]. The DLR implements a cache to reduce this penalty (IronPython and C#). Finally, *StaDyn* infers the types of the operands at compile time, optimizing the generated code.

Test *Calls* (which evaluates method and function invocation § 3.2.2) is the second test with the biggest differences in the execution time (the runtime performance of *StaDyn* is 20 times higher than the average measurements of the rest of languages). The three closest languages are C#, IronPython and Boo, which require at least one order of magnitude more execution time. As in the *Arithmetic* test, *StaDyn* manages to infer the object types used in the method invocations, casing a significant performance improvement –the rest of languages perform every type-checking operation at runtime.

In contrast, the *Exceptions* and *NewInstances* test present the lowest performance differences. The lowest performance benefits of *StaDyn* are 0.3% in the former test (C#) and 30% in the latter (Boo). Both tests do not evaluate any dynamic language property, but general language features such as exception handling or instance creation. The dynamically typed code used in these tests only assigns dynamically typed variables, which are never accessed. For this reason, the type information gathered by *StaDyn* does not imply a significant perfor-

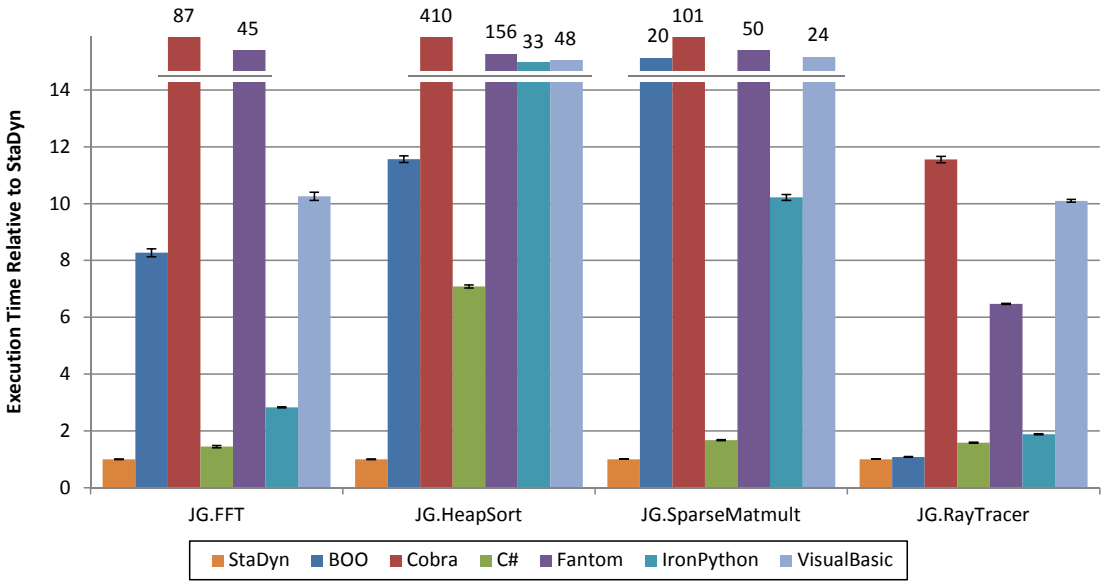



Figure 6.4: Execution time of the Java Grande benchmarks using dynamic typing.

mance improvement.

Figure 6.4 shows the average performance relative to *StaDyn* for the four evaluated tests of the Java Grande benchmark (§ 6.1). As in the *Pybench* benchmark, *StaDyn* is the fastest language in all the tests. On average, the runtime performance improvement of *StaDyn* is approximately one factor compared to C#, around 7 times faster than IronPython and Boo, and at least one order of magnitude than VB, Fantom and Cobra (Table B.2 details the execution time results).

In the execution of the *JG.FFT*, *JG.HeapSort* and *JG.SparseMatmult* tests (Java Grande, section 2) the measured execution times followed the same order: *StaDyn*, C#, IronPython, Boo, VB, Fantom and Cobra. This is because these algorithms, despite being different, use pretty similar language features. All of them are small programs with a lot of arithmetic operations and multiple array accesses using dynamically typed variables.

The execution time of *JG.HeapSort* shows a remarkable performance difference. For this test, *StaDyn* is 1.65 orders of magnitude faster than the average measurements of the rest of languages (in *JG.FFT* and *JG.SparseMatmult* this value is 1.02 and 1.27 orders of magnitude, respectively). This difference is because this algorithm makes extensive use of an array class field, declared as dynamic. Since *StaDyn* infers the attribute type, it can use specific instructions to access the array elements. On the other hand, the other tested languages use reflection to access the array, implying an important performance penalty.

In the *JG.SparseMatmult* and *JG.FFT* programs, both use methods with dynamically typed parameters. As we have already mentioned (§ 6.2), *StaDyn* does not infer type information for dynamically typed parameters. In this situation, the same as the other languages, *StaDyn* uses reflection to discover the types of these parameters at runtime. This is the reason why the performance difference for these two tests is not as important as that for the *JG.HeapSort* test.

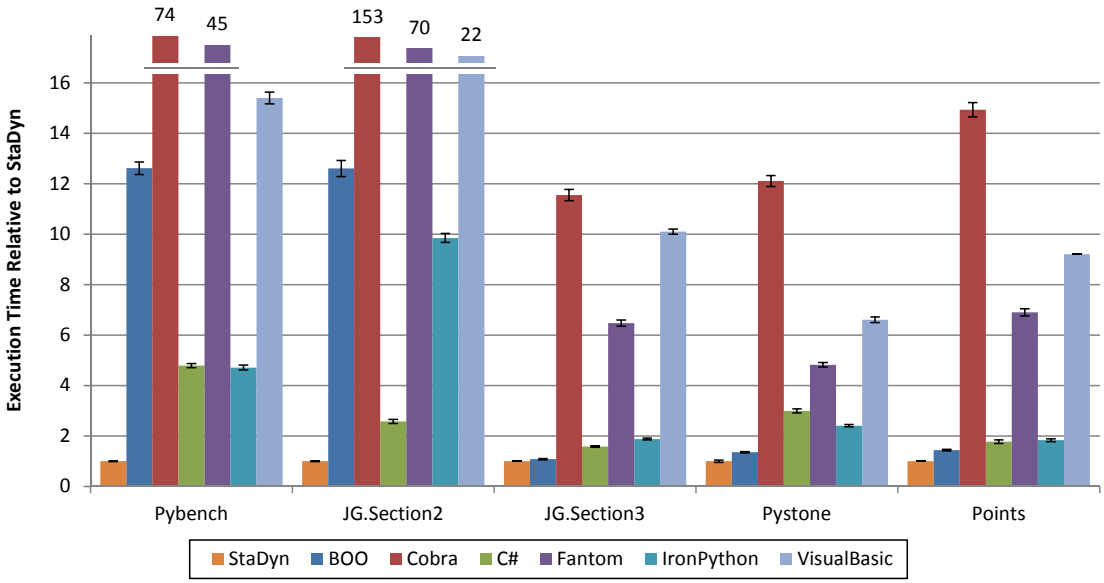



Figure 6.5: Execution time of the dynamically typed benchmarks.

Regarding the *JG.RayTracer* application (Java Grande, section 3), the performance differences are significantly lower than in the other measured applications of Java Grande. *StaDyn* is 2.6 times faster than the average measurements of the rest of languages (Boo, C#, IronPython, Fantom, VB and Cobra require 8%, 58%, 87%, 547%, 909% y 1,055% more execution time than *StaDyn*, respectively). Throughout the code, there are multiple method invocations of dynamically typed parameters. As *StaDyn* does not use type information inferred to optimize the code, the use of reflection causes a lower performance benefit. However, *StaDyn* remains the fastest implementation, due to the benefit of gathering type information of other expressions in the code. For example, *StaDyn* infers the type of all the dynamically typed variables which are declared in the scope of a method or a loop, and different types of fields. The results obtained in the evaluation of this real application confirm those obtained in the micro-benchmark, where *StaDyn* remained the fastest implementation when no type information was inferred (§ 6.2).

In order to complete the evaluation of the dynamic typing benchmarks, the popular *Pystone* benchmark has also been evaluated. This benchmark, in contrast to *Pybench*, does not measure specific language features, but the execution of a whole computation intense program. Figure 6.5 shows the results of all the dynamically typed benchmarks evaluated in this section, including *Pystone* (for Java Grande section 2, the geometric mean is shown). The *Pystone* results are quite similar to those of the *JG.Raytracer*. *StaDyn* is the fastest language, and its performance benefit compared to the average value of the rest of languages is 290% (Table B.3 shows the full list of times). *Pystone* uses a lot of dynamically typed parameters, the same as *JG.Raytracer*.

Finally, the evaluation of a modified version of the *Points* example shown in Figure 4.1 is included. This hybrid static and dynamic typing code has been modified in this section to make it fully dynamically typed. Figure 6.5 shows that the obtained results are quite similar to the results of the *JG.RayTracer* and

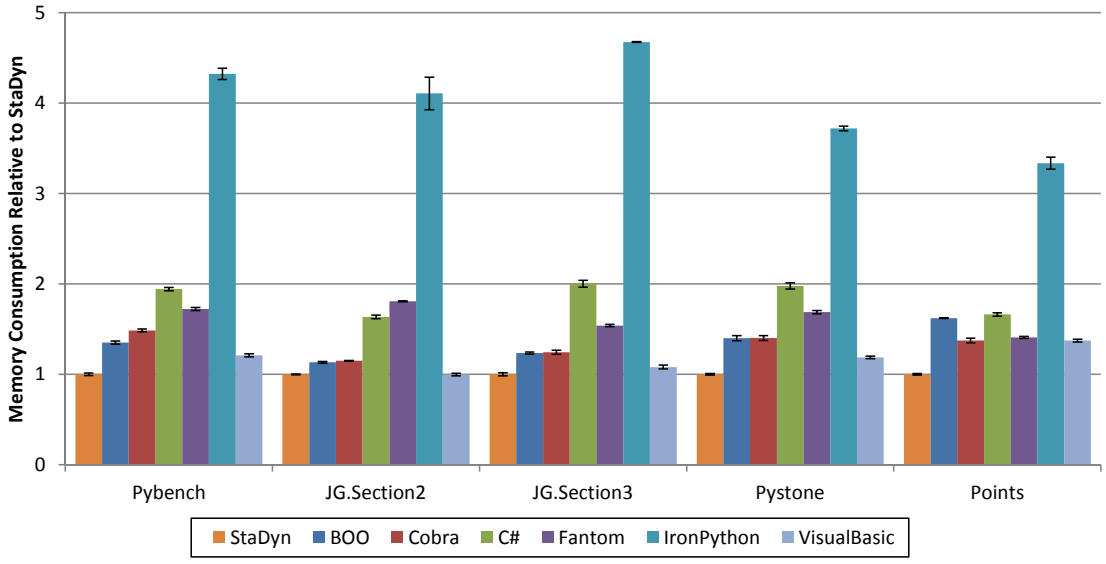



Figure 6.6: Memory consumption of the dynamically typed benchmarks.

the *Pystone* benchmarks. *StaDyn* performance is 305% faster than the average performance of the rest of languages (Table B.4).

Figure 6.5 summarizes all the results of the dynamic typing benchmarks. In the tests where the *StaDyn* compiler infers less type information (Java Grande section 3, *Pystone* and *Points*), its average runtime performance benefit is around 290%. This situation occurs when the source code has many calls to methods with the parameters declared as dynamic. For this reason, the immediate future work will be focused on adding program specialization for this kind of functions (see 7.1), using the type information gathered by *StaDyn*. We think this optimization will involve a notable runtime performance improvement, since the use of reflection causes an important performance penalty [97]. When *StaDyn* uses the inferred static type to optimize the generated code (*Pybench* and Java Grande section 2 benchmarks), the runtime performance of *StaDyn* is significantly higher than the other languages, being of 1.2 orders of magnitude on average. This influence of the type information gathered by *StaDyn* on its runtime performance benefits confirms the results obtained in the micro-benchmark (§ 6.2).

### 6.3.1 Memory Consumption

Figure 6.6 shows the average memory consumption relative to *StaDyn*, running the dynamically typed benchmarks. *StaDyn* presents the lowest memory consumption for every benchmark. The same as for the micro-benchmark, VB is the closet language, requiring 16% more memory than *StaDyn*, on average. The languages that use the DLR as a part of their runtime consume more memory resources: IronPython and C# consume, on average, 83.6% and 300% more memory than *StaDyn*. Therefore, the call site cache implemented by the DLR [13] provides runtime performance improvement, but also involves higher memory consumption.

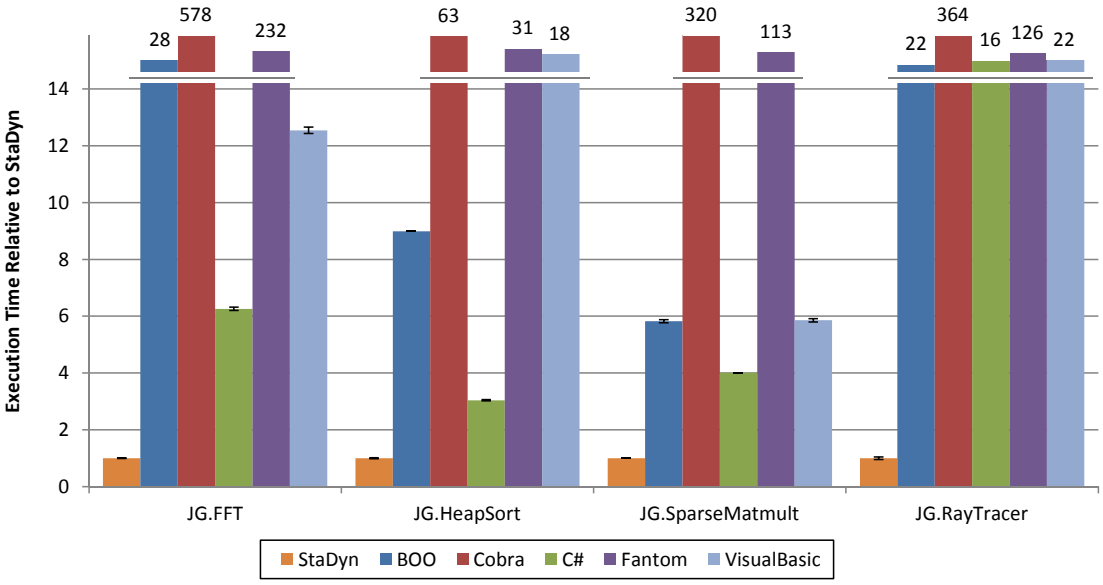



Figure 6.7: Execution time of the Java Grande benchmarks using hybrid typing.

There are two programs where *StaDyn* is not the language that consumes less memory resources. For the *JG.FFT* and *JG.HeapSort* tests, *StaDyn* requires 5.8% and 13.8% more memory than VB (Table C.2). These two algorithms use many dynamically typed local variables in multiple nested scopes, causing a significant memory consumption increase of *StaDyn* because of the SSA algorithm used ( § 3.2.1).

After analyzing the results of both memory consumption and runtime performance of dynamically typed applications, we have seen how the optimizations implemented by the *StaDyn* compiler do not involve more memory consumption –conversely to the languages that use the DLR.

## 6.4 Hybrid Dynamic and Static Typing Code

In this section, all the hybrid static and dynamic typing[1] programming languages that generate .NET code are compared. The same benchmarks as in the previous section (except *Pybench*) have been evaluated. To evaluate the performance of hybrid typing, their source code have been modified. We have explicitly declared the types of class fields and parameters of methods when possible, and we have used dynamic typing for all the local variables. Since the tests of the *Pybench* benchmark have neither class fields nor method parameters, it was not included in this evaluation.

Figure 6.7 shows the average performance runtime relative to *StaDyn* for the Java Grande applications, when hybrid typing is used (Table B.6). *StaDyn* is the fastest language in all the programs. On average, it is 4, 8, 16, 53 and 127 times faster than C#, Boo, VB, Fantom and Cobra, respectively.

[1]For the sake of brevity, we use from now on *hybrid typing* is used to refer to *hybrid static and dynamic typing*.

In the execution of the *JG.FFT* program, the runtime performance of *StaDyn* is considerably better than those of the other languages, being 5 times faster than the second one (C#) and 2.76 orders of magnitude higher than the one with the worse performance (Cobra). This is because in this application, *StaDyn* infers the type of all the dynamically typed references, and therefore it generates statically typed code. However, the rest of languages use reflection to discover the types of some variables, implying a significant performance penalty.

Comparing the new hybrid version of *JG.FFT* with the fully dynamic (Figure 6.4), the performance benefit between *StaDyn* and the other languages grows significantly. Running the dynamically typed version, *StaDyn* is 10 times faster than the average execution times of the rest of languages. With hybrid typing this difference grows up to 50 times. The hybrid typing code explicitly states the type all the parameters. This causes *StaDyn* to significantly improve its runtime performance, running the hybrid version of the *JG.FFT* application 9 times faster than the dynamically typed one. Although the rest of languages have also shown an improvement, it has been lower. They are about 2 times faster with respect to the dynamically typed version, except VB that is 7 times faster.

In the *JG.HeapSort* test also obtains the best runtime performance because of static type inference. However, comparing this program with the fully dynamic version, the performance differences between *StaDyn* and the other languages have decreased. Using hybrid typing, *StaDyn* is 15 times faster than average measurements of the other languages, but this value is 48 for dynamically typed code (Figure 6.4). In hybrid typing, the class fields are explicitly typed. Since *JG.HeapSort* makes extensive use of an array declared as a field, Cobra, Fantom, VB, C# and Boo running the hybrid version 24, 18, 9, 8 and 4 times faster than the dynamic one, respectively. However, the performance benefit of *StaDyn* using hybrid typing is only 300%, because the type of the array field was already inferred in the fully dynamic version.

In the *JG.SparseMatmult* program, C#, VB and Boo present the best execution times relative to *StaDyn*, being less than 5 factors (Cobra and Fantom are around 2 orders of magnitude slower than *StaDyn*). This algorithm makes wide use of an array local variable, declared as dynamic. While the *StaDyn* compiler infers its type, the other languages do not gather that type information. The performance difference among the rest of languages depends on how each one performs accesses to dynamically typed arrays. Boo and VB have specific operations to perform this operation (SetSlice and LateIndexSet, respectively), C# uses the DLR, and Cobra and Fantom use reflection, which is the option with the worst performance.

Boo and VB have decreased their relative execution time to *StaDyn*, comparing this version of *JG.SparseMatmult* with the dynamic typing one; nevertheless, it is the opposite for C#, Cobra and Fantom. The performance benefit of *StaDyn* using hybrid versus dynamic typing is 1,100%, while that value is around 500% for C#, Cobra and Fantom. However, Boo and VB shows 4,500% faster execution of the hybrid version than the dynamic typing one (Figure 6.4). The difference between the improvement obtained by *StaDyn* and the other languages is ex-

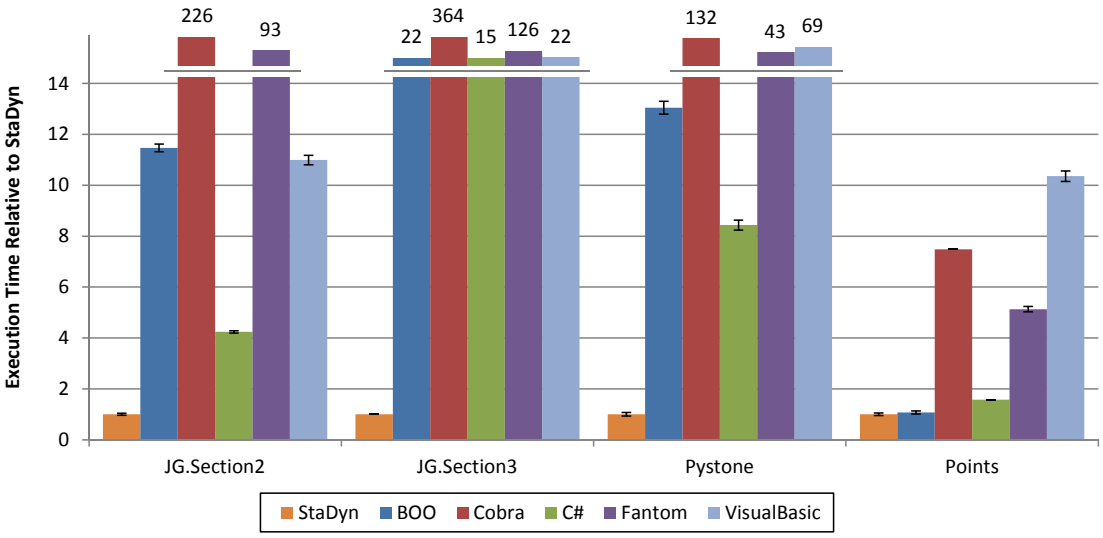



Figure 6.8: Execution time of the hybrid static and dynamic typing benchmarks.

plained by a combination of the two previously described situations (the *JG.FFT* and the *JG.HeapSort* applications). On one hand, as with the *JG.FFT*, *StaDyn* obtains a performance benefit by using explicitly typed parameters. But on the other hand, as with the *JG.HeapSort*, explicitly typed attributes does not implies a performance benefit to *StaDyn*, because it already infers the type of dynamic class fields. For the other languages, the explicit declaration of parameters and class fields implies a performance improvement, which is more remarkable for Boo and VB rather than for C#, Fantom and Cobra.

The final evaluation presented in Figure 6.7 is the *JG.RayTracer* application. In this test, *StaDyn* presents the major performance difference compared to the rest of languages, being 15, 21, 21, 126 and 364 times faster than C#, Boo, VB, Fantom and Cobra, respectively. This huge difference is because *StaDyn*, unlike the other measured languages, manages to infer the type of all the `var` references in the program.

Comparing the hybrid version of *JG.RayTracer* with the fully dynamic one, *StaDyn* shows the highest performance difference with the rest of languages. *StaDyn* runs the dynamic *JG.RayTracer* 4.2 times faster than the average execution times of the other languages (Figure 6.4); using hybrid typing this average increases to 48 factors (in *JG.FFT* it grows from 10 to 50, in *JG.HeapSort* it decreases from 48 to 15, and in *JG.SparseMatmult* remains around 21). This difference is because the dynamic typing version of this program presents the worst scenario for *StaDyn* (many dynamically typed parameters); and hybrid typing is the best scenario, since the types of all the `var` references are inferred. Therefore, the runtime performance of *StaDyn* with hybrid typing is 128 times faster than with dynamic typing (versus 9 times in *JG.FFT*, 3 times in *JG.HeapSort* and 11 times in *JG.SparseMatmult*).

Figure 6.8 summarizes all the results of the measured hybrid typing benchmarks. The result of the *Pystone* benchmark is similar to the second section of the Java Grande test suite. *StaDyn* presents the best runtime performance,

being 33 times faster than the average measurements of the rest of languages (Table B.7). The comparison with its dynamic typing version is similar to the *JG.RayTracer* application: *StaDyn* performance is increased in 20 times, and the rest of languages in about two factors.

The evaluation of the *Points* application is also included in Figure 6.8. Although *StaDyn* is the fastest language, in this application the performance difference compared to the rest of languages is smaller than in the previous scenarios (Table B.8). The execution time required by Boo is 6.9% more than *StaDyn*, and C# uses 57.5% more CPU time than our language. The average execution time of all the languages (excluding *StaDyn*) is 3.67 times higher than *StaDyn*. This application uses few dynamic variables (Figure 4.1) and therefore, gives little room for the optimizations implemented by *StaDyn* (the rest of variables are explicitly typed). Besides, most dynamic references are method parameters that *StaDyn* does not optimize. Although *StaDyn* is able to infer the type of the attributes, the type of dynamic parameters is obtained at runtime, as the rest of languages. The hybrid typing version of *StaDyn* is 15% faster than the fully dynamic one; the average value for the rest of languages is 50%. This lower significance in the explicit declaration of types in *StaDyn* is because it infers much type information in the dynamically typed version.

In this subsection we have seen how, when the method parameters are explicitly typed, all languages obtain a performance improvement. *StaDyn* is the language with the most remarkable improvement. On average, *StaDyn* runs the hybrid typing programs 10 times faster than the dynamically typed ones. Comparing the execution time of *StaDyn* with the average execution time of the rest of languages, our language is 10 times faster running dynamically typed code, and 22 times faster executing hybrid typing applications. Hybrid programs explicitly state the type of function parameters. The next optimization we plan to use is to employ the inferred types of arguments to implement function specialization optimizations § 7.1. Since this is similar to explicit type declaration, the hybrid typing code assessment gives us an estimate of the performance benefit that could be obtained with function specialization for functions.

### 6.4.1 Memory Consumption

In terms of memory usage, Figure 6.9 shows that *StaDyn* is the language with the lowest memory consumption for every benchmark. VB is the second one, consuming 21.2% more average memory resources than *StaDyn*. Boo, Cobra, Fantom and C# require 43%, 48%, 93% and 93% more memory than *StaDyn* (§ C.3), respectively. It should be noted that, although C# reduces the usage of the DLR when using hybrid typing, its memory consumption relative to *StaDyn* remains significant (93%).

Comparing these results with the ones obtained in the evaluation of dynamic typing (§ 6.3.1), all languages decrease its memory consumption with hybrid typing is used. The one with the highest memory decrease is *StaDyn* (20.9%) followed by VB (15.3%), C# (14.8%), Boo (13%), Fantom (7.4%) and Cobra

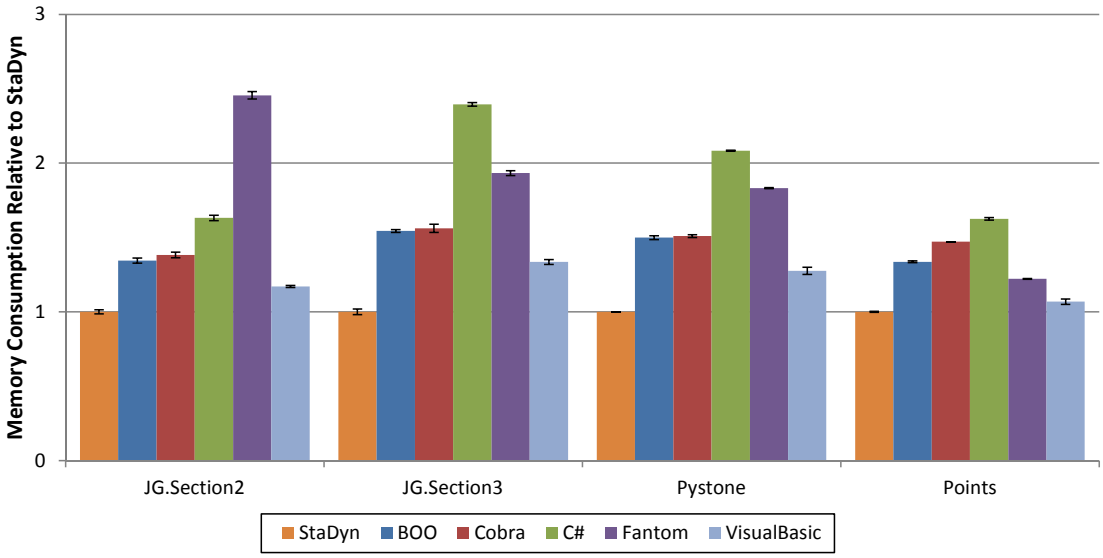



Figure 6.9: Memory consumption of the hybrid static and dynamic typing benchmarks.

(5.2%).

As with the evaluation of dynamically typed programs, the results presented in this section confirm that the performance difference between *StaDyn* and the rest of languages is not obtained by means of consuming more memory resources.

## 6.5 Explicitly Typed Code

In this section the same benchmarks with 100% explicitly typed code are evaluated. The main objective of this evaluation is to compare the optimizations of each compiler when no dynamic type is used. All the benchmarks in this section have been modified explicitly declaring the types of every variable, and hence dynamic typing is not used at all. As the *Points* application cannot be implemented without dynamic types [78], it has not been included in this evaluation.

Figure 6.10 shows the average execution time relative to *StaDyn* for the Java Grande and *Pystone* benchmarks. C# is the fastest language for all the programs, being *StaDyn* the second one. C# generates code that runs, on average, 2.5% faster than the one produced by *StaDyn*. This difference shows how the commercial C# compiler performs more optimizations than *StaDyn* when the application is fully statically typed. *StaDyn* is 24%, 48%, 243% and 334% faster than Cobra, VB, Boo and Fantom, respectively. These results are similar to the ones obtained in the micro-benchmark, when the types of all the variables were explicitly declared (§ 6.2).

In the Java Grande applications the runtime performance of Boo and Fantom is appreciably worse than the other languages (Table B.9); on average, they require 414% and 735% more execution time than *StaDyn* (23.3% and 59.7% in the case of Cobra and VB). The worse performance of Boo and Fantom is caused by the dynamically typed operations used in their runtime, even when the code explicitly

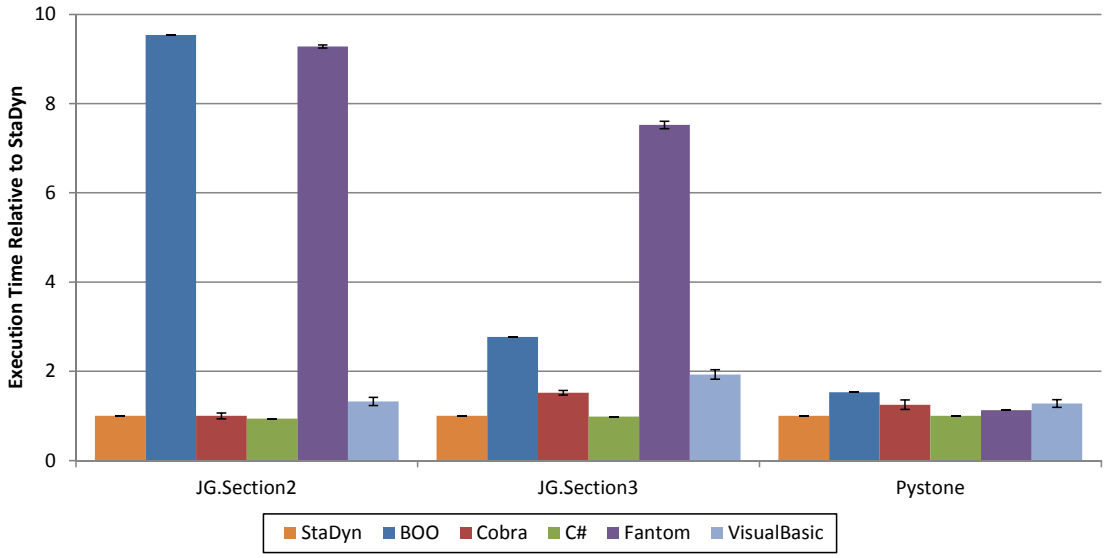



Figure 6.10: Execution time of the explicitly typed benchmarks.

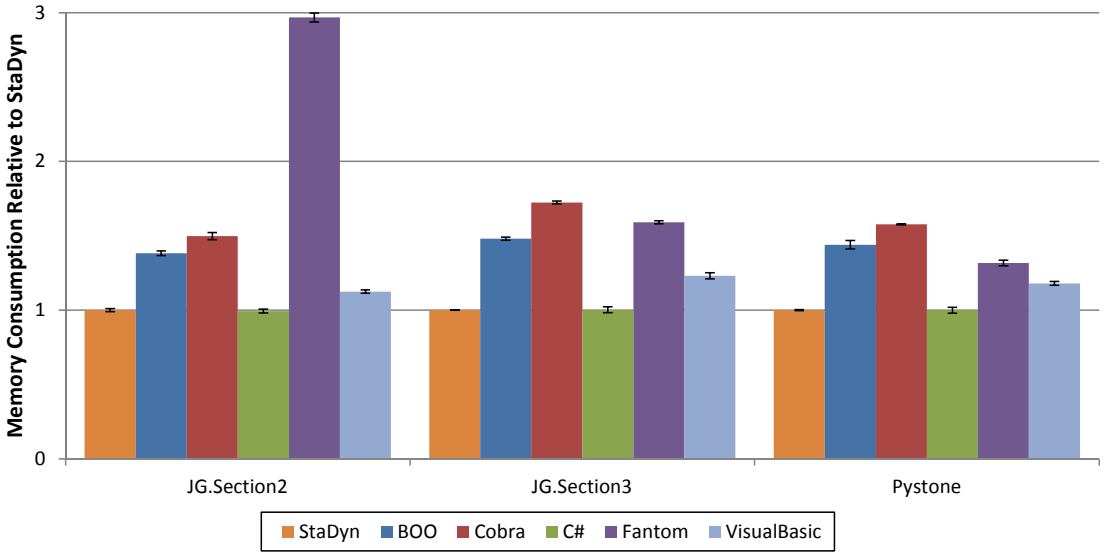



Figure 6.11: Memory consumption of the explicitly typed benchmarks.

states the static types. The other languages use the statically typed IL opcodes, causing significant performance differences.

The *Pystone* benchmark shows a different scenario. This application barely accesses to arrays or any other operation that uses the dynamic typing services of the language runtime. Consequently, the runtime performance of all languages is very close (Table B.10). Fantom, Cobra, VB and Boo require, respectively, 13%, 25%, 27% and 53% more execution time than *StaDyn* to run the *Pystone* benchmark. For this kind of code (explicitly declaring the types of all the variables), runtime performance differences among languages are considerably reduced.

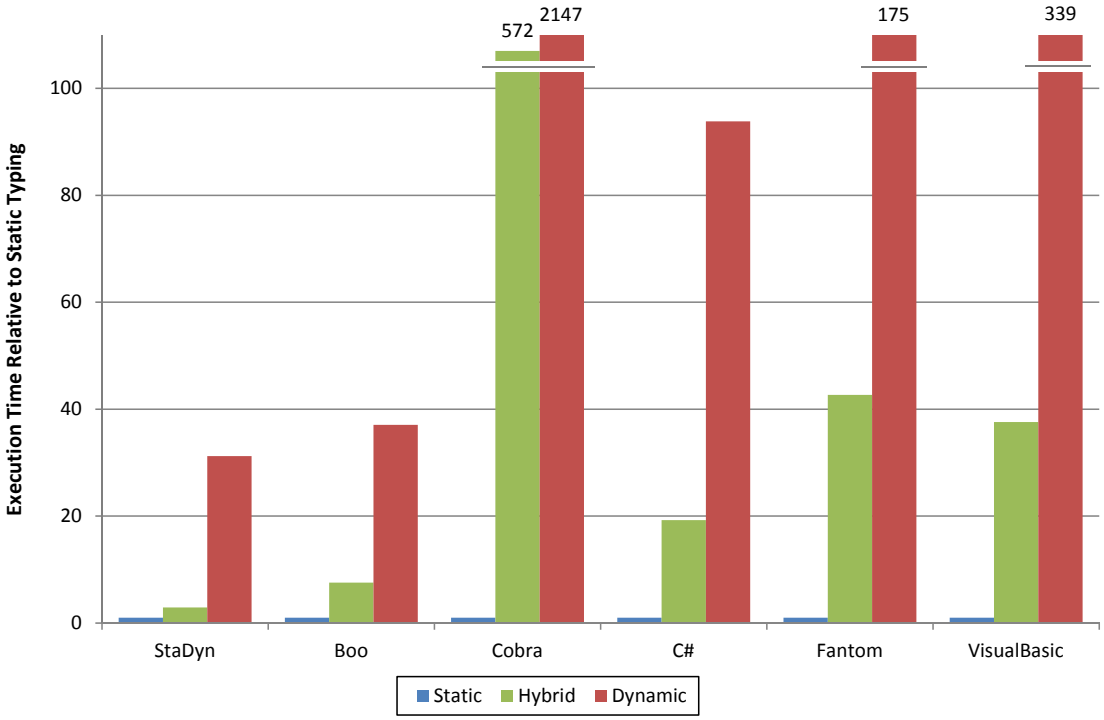



Figure 6.12: Influence of dynamic typing on runtime performance.

### 6.5.1 Memory Consumption

Figure 6.11 shows memory consumption with explicit type declaration. *StaDyn* and C# are the languages that consume less memory (the difference is lower than the error interval). VB, Boo, Cobra and Fantom require 17%, 43%, 59% and 95% more memory than *StaDyn* (§ C.4). This higher consumption is caused by the additional runtime implemented by these languages –the DLR is not used when all the references are statically typed.

When using hybrid typing, all the languages but C# consume around 10% more memory than using explicit type declaration. C# requires 121% more memory, due to the cost of using the DLR. *StaDyn* consumes 9.6% more memory in hybrid typing due to the cost of the SSA algorithm implemented (§ 3.2.1).

## 6.6 Influence of Dynamic Typing on Runtime Performance

Figure 6.12 shows the dependency of dynamic, hybrid and static typing on the runtime performance of each programming language. The values shown in Figure 6.12 are relative to the execution time of each programming language, when all the variables are declared explicitly with their types. Therefore, the figure shows the cost of using dynamic (and hybrid) typing in each programming language. This evaluation includes all the presented benchmarks but the micro-benchmark, not included because, since it is a synthetic application, it could introduce a bias

in the results.

*StaDyn* is the language that introduces the lowest performance penalty of using both hybrid and dynamic typing. Comparing static and hybrid typing, the cost is of 2 factors. Boo, C#, VB, Fantom and Cobra require 7, 18, 36, 41 and 571 more execution time running hybrid typing code than statically typed one. The difference with *StaDyn* is because these languages do not infer type information of dynamic references at compile time.

Finally, the statically typed programs in *StaDyn* run 1.4 orders of magnitude faster than dynamically typed applications. This value is of 1.5, 1.9, 2.2, 2.5 y 3.3 orders of magnitude for Boo, C#, Fantom, VB and Cobra. *StaDyn* shows the lowest difference because of the type inference implemented by the compiler. Boo is the second one, but the reason is different: the Boo runtime uses dynamic typing even when all the variables are explicitly declared. Namely, Boo has low runtime performance when variables are explicitly typed, thereby involving a low difference with hybrid and dynamic typing.

# Chapter 7

# Conclusions

Gathering type information of dynamic (and hybrid) typing code at compile time can be used to improve early type error detection and runtime performance of the code generated, without increasing memory consumption at runtime. In order to empirically show this, we have implemented the *StaDyn* programming language, which combines static and dynamic typing in the very same programming language. The major contribution of *StaDyn* is that static type inference and type checking is performed by the compiler even over dynamic references, offering a high level of flexibility and a better robustness and efficiency, closer to static typing.

Most dynamic languages allow references to have different types in the same scope. However, statically typed languages force variables to have the same type within a scope. *StaDyn* offers references to have different types in the same scope *statically*. We have developed a version of the SSA algorithm, which guarantees that every reference is assigned exactly once by means of creating new temporary references. As a result, code can be easily generated to the .NET platform, which requires a single type for each variable.

*StaDyn* extends the behavior of the implicitly typed local references of the C# 3.0 programming 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# 3.0 only provides its use in the declaration of initialized local references. For this purpose, we have implemented the Hindley-Milner unification algorithm to infer the *concrete* type of each `var` reference.

The implemented type system is flow-sensitive because it takes into account the flow context of each `var` reference. It gathers concrete type information knowing all the possible types a `var` reference may hold. Depending on flow context, different concrete types could be inferred (by the unification algorithm) for the same variable. The key technique we have used to represent this concrete-type flow-sensitive type system is union types. When a branch in the code is detected, a union type with all the possible concreted types is inferred. The result is a type-based *alias analysis* algorithm to know all the concrete types that a local

`var` reference may be pointing to.

Implicitly typed parameters cannot be unified to a single concrete type, because they represent the type of any possible argument. Since they cannot be inferred the same way as local references, we have enhanced the *StaDyn* type system to be constraint-based. Method types have been extended with an ordered set of constraints specifying the restrictions to be fulfilled by the parameters. An abstract interpretation mechanism is implemented to perform type checking following the method call graph, generating the appropriate set of constraints. All the constraints defined for a `var` parameter are defined as an *intersection type* the argument must promote to (it must be a subtype of all the types in the intersection type). At present, the type information of `var` parameters gathered by the compiler is only used for early type error detection –method specialization has not been implemented yet.

*StaDyn* also provides the use of `var` types in class fields (attributes). Implicitly typed attributes extend the constraint-based type system in the sense that concrete types `var` fields can be modified on each method invocation expression. To model this behavior, we have added a new kind of assignment constraint to the type system. Each time a value is assigned to a `var` attribute, an *assignment* constraint is added to the method being analyzed. This constraint postpones the unification of the concrete type of the attribute to be performed later, when an actual object is used in the invocation. This mechanism allows static type checking of `var` fields, improving both compile-time type error detection and runtime performance optimizations. Besides, it extends the type-based alias analysis to object and class attributes.

The use of `var` as a new type to be used in local variables, parameters and attributes provides an added value to the *StaDyn* language. Since the compiler infers all the possible concrete types for an implicitly typed variable, the type system provides *static* duck typing. An operation can be passed to a `var` reference when all the possible types it may be pointing to accept that operation.

*StaDyn* performs static type checking of both dynamic and static `var` references, improving the interoperation between these two kinds of code. For this purpose we define a new interpretation of union and intersection types. A union type is promoted to another type when all the types in the union type are subtypes of the given type (least upper bound). This coercion is more lenient when the union type is dynamic: only one type in the union type must be a subtype of the given type. A type is converted to a static intersection type when that type is a subtype of all the types in the intersection type (greatest lower bound). This condition is relaxed to "'at least one type in the intersection type"' when the reference is dynamic.

The dynamism of `var` references is not explicitly stated in the source code. It is specified in an external XML file, that is transparently managed by the programming IDE. Therefore, dynamically typed code can be converted into statically typed one without modifying the source code, and vice versa. Consequently, *StaDyn* facilitates the transition of rapid prototyping into robust and efficient software production. By separating the *dynamism* concern, it is possible to cus-

tomize the trade-off between runtime flexibility of dynamic typing and runtime performance and robustness of static typing, minimizing the changes in the application source code.

We have evaluated runtime performance and memory consumption of *StaDyn*, and compared it with the existing hybrid and dynamic programming languages for the .NET Framework. A set of benchmarks with dynamic, hybrid and static typing code were executed. Running fully dynamically typed code, the average runtime performance of *StaDyn* is at least 150% (up to 2.1 orders of magnitude) faster. In the case of hybrid statically and dynamically typed code, *StaDyn* is form 5 to 126 times faster than the rest of languages. *StaDyn* has outperformed the rest of languages for each single program of these two kinds of benchmarks.

To measure the static optimizations performed by the compiler, we have also executed some benchmarks where all the variables are explicitly typed (no variable is declared as dynamic). In this scenario, *StaDyn* has been the second fastest language, requiring, on average, 2.5% more execution time that C#. The rest of languages have required from 24% to 334% more average execution time than *StaDyn*. The difference is mainly caused by the fact that *StaDyn* does not require any runtime.

*StaDyn* is the language that introduces the lowest performance penalty of using hybrid static and dynamic typing with respect to explicit type declaration (200%). Similarly, the difference between hybrid and dynamic typing has also been the lowest one (10 factors). Since our programming language gathers type information of dynamic references, the execution time of running the hybrid and dynamic benchmarks is closer to static typing than the rest of languages –that do not perform any type inference.

In terms of memory consumption, *StaDyn* has showed the lowest memory resource requirements. The rest of languages require at least 16% and 21% more memory for running dynamic and hybrid typing code, respectively. When all the types are explicitly declared, memory consumption of *StaDyn* and C# are the same (the rest of languages consume no less than 17% more memory). However, when one single dynamic reference is used, the DLR runtime used by C# consumes 74% more memory than *StaDyn*. In the assessment performed, the SSA algorithm implemented to assign one single type for each reference in the generated code has not penalized the memory consumption of *StaDyn*.

The evaluation shows that *StaDyn* provides the best runtime performance when dynamic references are used, without requiring additional memory resources at runtime. The type information gathered by the compiler is used to generate optimized code, avoiding the use of runtime caches to reduce the cost of using of reflection.

## 7.1 Future Work

The CLR does not provide the runtime adaptive (HotSpot) optimizations implemented by Java, where server applications are dynamically recompiled and reoptimized. As we have seen in the evaluation section, *StaDyn* generally outperforms the rest of languages. In the case of `var` parameters reflection is used, slowing down the execution of applications. If the number of invocations is really high (e.g., server applications), the implementation of a call-site cache, similar to the one provided by the DLR [13], would produce better performance results. Therefore, we plan to offset the lack of HotSpot optimizations with the explicit identification of server applications, for which we will implement a cache. When the –server option is passed to the *StaDyn* compiler, the DLR cache will be used instead of reflection for `var` parameters.

As mentioned, the use of `var` parameters in *StaDyn* is translated to reflective code. However, *StaDyn* does infer type information about the arguments in method invocation. Therefore, the immediate future work will be focused on specializing methods with the type information of their arguments. The same as a C++ compiler does, different versions of the same method will be generated for a particular set of argument types, speeding up the execution of *StaDyn* applications [98].

The next step in the development of *StaDyn* is adding structural intercession (i.e., the capability of dynamically adapting the structure of objects and types) to the language. Relying in the concrete type system and the type-based alias analysis provided, the structural type representation of objects will be adapted at compile-time. The idea is to represent object adaptation similarly to the calculus defined by Cardelli and Mitchell for operations on records [99]. Using a similar approach to the assignment constraints defined for the assignment of `var` parameters, structural intercession could be included in the constraint-based type system.

The structural intercession services will be supported by the `ExpandoObjects`, using the DLR as the compiler back-end. At the same time, we want to use the ЯRotor platform we developed as a previous research work [88]. This project modifies a shared-source implementation of the CLI to incorporate structural intercession as part of the JIT-compiled primitives. For these operations, ЯRotor has shown a notable runtime performance improvement [97].

Once *StaDyn* offers structural intercession, we plan to use it in the development of dynamic aspect-oriented applications. We have previously created a platform that provides these services over.Net [100]. Our intention is to employ StaDyn for implementing the same application, and compare the runtime performance results obtained [101].

Finally, we have previously done research on related fields, where we have used widespread dynamic languages such as Python [102]. We would like to apply the last version of our programming language to these same scenarios in the future. Similarly, the development of applications that compute runtime changing data turned out to be really difficult with statically typed languages [103]. Alternative

implementations combining static and dynamic typing could be a topic of future research.

The current release of the *StaDyn* programming language, its source code, and all the benchmarks and examples presented in this PhD thesis are freely available at http://www.reflection.uniovi.es/stadyn.

# Appendix A

# Syntax of the *StaDyn* Programming Language

## A.1 Syntax Specification

| | | |
|---|---|---|
| *qualifiedIdentifier* | ::= | IDENTIFIER ( DOT *qualifiedIdentifier* )? |
| *type* | ::= | ( ( *predefinedTypeName* \| *qualifiedIdentifier* ) \| VOID ) *rankSpecifiers* |
| *argumentList* | ::= | *argument* ( COMMA *argument* )* |
| *argument* | ::= | *expression* |
| *constantExpression* | ::= | *expression* |
| *booleanExpression* | ::= | *expression* |
| *expressionList* | ::= | *expression* ( COMMA *expression* )* |
| *expression* | ::= | *assignmentExpression* |
| *assignmentExpression* | ::= | *conditionalExpression* ( ( ASSIGN \| PLUS_ASSIGN \| MINUS_ASSIGN \| STAR_ASSIGN \| DIV_ASSIGN \| MOD_ASSIGN \| BIN_AND_ASSIGN \| BIN_OR_ASSIGN \| BIN_XOR_ASSIGN \| SHIFTL_ASSIGN \| SHIFTR_ASSIGN ) *assignmentExpression* )? |
| *conditionalExpression* | ::= | *conditionalOrExpression* ( QUESTION *assignmentExpression* COLON *conditionalExpression* )? |
| *conditionalOrExpression* | ::= | *conditionalAndExpression* ( LOG_OR *conditionalAndExpression* )* |
| *conditionalAndExpression* | ::= | *inclusiveOrExpression* ( LOG_AND *inclusiveOrExpression* )* |
| *inclusiveOrExpression* | ::= | *exclusiveOrExpression* ( BIN_OR *exclusiveOrExpression* )* |
| *exclusiveOrExpression* | ::= | *andExpression* ( BIN_XOR *andExpression* )* |
| *andExpression* | ::= | *equalityExpression* ( BIN_AND *equalityExpression* )* |
| *equalityExpression* | ::= | *relationalExpression* ( ( EQUAL \| NOT_EQUAL ) *relationalExpression* )* |
| *relationalExpression* | ::= | *shiftExpression* ( ( ( LTHAN \| GTHAN \| LTE \| GTE ) *additiveExpression* )* \| IS *type* ) |
| *shiftExpression* | ::= | *additiveExpression* ( ( SHIFTL \| SHIFTR ) *additiveExpression* )* |
| *additiveExpression* | ::= | *multiplicativeExpression* ( ( PLUS \| MINUS ) *multiplicativeExpression* )* |
| *multiplicativeExpression* | ::= | *unaryExpression* ( ( STAR \| DIV \| MOD ) *unaryExpression* )* |
| *unaryExpression* | ::= | OPEN_PAREN *type* CLOSE_PAREN *unaryExpression* \| INC *unaryExpression* \| DEC *unaryExpression* \| PLUS *unaryExpression* \| MINUS *unaryExpression* \| LOG_NOT *unaryExpression* \| BIN_NOT *unaryExpression* \| *primaryExpression* |
| *basicPrimaryExpression* | ::= | *literal* \| IDENTIFIER \| OPEN_PAREN *assignmentExpression* CLOSE_PAREN \| THIS \| BASE ( DOT IDENTIFIER \| OPEN_BRACK *expression* CLOSE_BRACK ) \| *newExpression* |
| *primaryExpression* | ::= | *basicPrimaryExpression* ( OPEN_PAREN *argumentList*? CLOSE_PAREN \| OPEN_BRACK *expression* CLOSE_BRACK \| DOT IDENTIFIER \| INC \| DEC )* |

*newExpression* ::= NEW *type* ( OPEN_PAREN *argumentList*? CLOSE_PAREN |
*arrayInitializer* | OPEN_BRACK *expressionList*
CLOSE_BRACK *rankSpecifiers arrayInitializer*? )

*literal* ::= TRUE |
FALSE |
INT_LITERAL |
DOUBLE_LITERAL |
CHAR_LITERAL |
STRING_LITERAL |
NULL

*predefinedType* ::= BOOL |
CHAR |
DOUBLE |
INT |
OBJECT |
STRING |
VAR

*predefinedTypeName* ::= BOOL |
CHAR |
DOUBLE |
INT |
OBJECT |
STRING |
VAR

*statement* ::= *declarationStatement* |
*declarationStatement* |
*embeddedStatement*

*embeddedStatement* ::= *block* |
SEMI |
*expressionStatement* |
*selectionStatement* |
*iterationStatement* |
*jumpStatement* |
*tryStatement*

*body* ::= *block* |
SEMI

*block* ::= OPEN_CURLY *statement*∗ CLOSE_CURLY

*statementList* ::= *statement*+

*declarationStatement* ::= *localVariableDeclaration* SEMI |
*localConstantDeclaration* SEMI

*localVariableDeclaration* ::= *type localVariableDeclarators*

*localVariableDeclarators* ::= *localVariableDeclarator* ( COMMA *localVariableDeclarator* )*

*localVariableDeclarator* ::= IDENTIFIER ( ASSIGN *localVariableInitializer* )?

*localVariableInitializer* ::= *expression* |
*arrayInitializer*

*localConstantDeclaration* ::= CONST *type localConstantDeclarators*

*localConstantDeclarators* ::= *localConstantDeclarator* ( COMMA *localConstantDeclarator* )*

*localConstantDeclarator* ::= IDENTIFIER ASSIGN *constantExpression*

*constantDeclarators* ::= *constantDeclarator* ( COMMA *constantDeclarator* )*

*constantDeclarator* ::= IDENTIFIER ASSIGN *constantExpression*

*expressionStatement* ::= *statementExpression* SEMI

*statementExpression* ::= *assignmentExpression*

*selectionStatement* ::= *ifStatement* |
*switchStatement*

*ifStatement* ::= IF OPEN_PAREN *booleanExpression* CLOSE_PAREN
*embeddedStatement elseStatement*?

*elseStatement* ::= ELSE *embeddedStatement*

*switchStatement* ::= SWITCH OPEN_PAREN *expression* CLOSE_PAREN *switchBlock*

*switchBlock* ::= OPEN_CURLY *switchSections*? CLOSE_CURLY

*switchSections* ::= *switchSection*+

*switchSection* ::= *switchLabels statementList*

*switchLabels* ::= *switchLabel*+

*switchLabel* ::= CASE *constantExpression* COLON |
DEFAULT COLON

*iterationStatement* ::= *whileStatement* |
*doStatement* |
*forStatement* |
*foreachStatement*

*whileStatement* ::= WHILE OPEN_PAREN *booleanExpression* CLOSE_PAREN
*embeddedStatement*

*doStatement* ::= DO *embeddedStatement* WHILE OPEN_PAREN *booleanExpression*
CLOSE_PAREN SEMI

*forStatement* ::= FOR OPEN_PAREN *forInitializer* SEMI
*forCondition* SEMI *forIterator* CLOSE_PAREN *embeddedStatement*

*forInitializer* ::= ( *localVariableDeclaration* | *localVariableDeclaration* | *statementExpressionList* )?
*forCondition* ::= *booleanExpression*?
*forIterator* ::= *statementExpressionList*?
*statementExpressionList* ::= *statementExpression* ( COMMA *statementExpression* )*
*foreachStatement* ::= FOREACH OPEN_PAREN *type* IDENTIFIER IN *expression* CLOSE_PAREN *embeddedStatement*
*jumpStatement* ::= *breakStatement* | *continueStatement* | *returnStatement* | *throwStatement*
*breakStatement* ::= BREAK SEMI
*continueStatement* ::= CONTINUE SEMI
*returnStatement* ::= RETURN *expression*? SEMI
*throwStatement* ::= THROW *expression*? SEMI
*tryStatement* ::= TRY *block catchClause*∗ *finallyClause*?
*catchClause* ::= CATCH OPEN_PAREN *qualifiedIdentifier IDENTIFIER*? CLOSE_PAREN *block*
*finallyClause* ::= FINALLY *block*
*compilationUnit* ::= *usingDirectives namespaceMemberDeclarations* EOF
*usingDirectives* ::= *usingDirective*∗
*usingDirective* ::= USING *qualifiedIdentifier* SEMI
*namespaceMemberDeclarations* ::= *namespaceMemberDeclaration*∗
*namespaceMemberDeclaration* ::= *namespaceDeclaration* | *modifiers typeDeclaration*
*typeDeclaration* ::= *classDeclaration* | *interfaceDeclaration*
*namespaceDeclaration* ::= NAMESPACE *qualifiedIdentifier namespaceBody SEMI*?
*namespaceBody* ::= OPEN_CURLY ( *modifiers typeDeclaration* )* CLOSE_CURLY
*modifiers* ::= *modifier*∗
*modifier* ::= ABSTRACT | NEW | OVERRIDE | PUBLIC | PROTECTED | INTERNAL | PRIVATE | STATIC | VIRTUAL
*classDeclaration* ::= CLASS IDENTIFIER *classBase classBody SEMI*?
*classBase* ::= ( COLON *type* ( COMMA *type* )* )?
*classBody* ::= OPEN_CURLY *classMemberDeclarations* CLOSE_CURLY
*classMemberDeclarations* ::= *classMemberDeclaration*∗
*classMemberDeclaration* ::= *modifiers typeMemberDeclaration*
*typeMemberDeclaration* ::= CONST *type constantDeclarators* SEMI | IDENTIFIER OPEN_PAREN *formalParameterList*? CLOSE_PAREN *constructorInitializer*? *constructorBody* | *voidAsType* IDENTIFIER OPEN_PAREN *formalParameterList*? CLOSE_PAREN *methodBody* | *type* ( *variableDeclarators* SEMI | IDENTIFIER ( OPEN_CURLY *accessorDeclarations* CLOSE_CURLY | OPEN_PAREN *formalParameterList*? CLOSE_PAREN *methodBody* ) )
*variableDeclarators* ::= *variableDeclarator* ( COMMA *variableDeclarator* )*
*variableDeclarator* ::= IDENTIFIER ( ASSIGN *variableInitializer* )?
*variableInitializer* ::= *expression* | *arrayInitializer*
*returnType* ::= *voidAsType* | *type*
*methodBody* ::= *body*
*formalParameterList* ::= *fixedParameters*
*fixedParameters* ::= *fixedParameter* ( COMMA *fixedParameter* )*
*fixedParameter* ::= *type* IDENTIFIER
*accessorDeclarations* ::= *getAccessorDeclaration setAccessorDeclaration*? | *setAccessorDeclaration getAccessorDeclaration*?
*getAccessorDeclaration* ::= *'get' accessorBody*
*setAccessorDeclaration* ::= *'set' accessorBody*
*accessorBody* ::= *body*
*constructorInitializer* ::= COLON ( BASE OPEN_PAREN *argumentList*? CLOSE_PAREN | THIS OPEN_PAREN *argumentList*? CLOSE_PAREN )
*constructorBody* ::= *body*
*nonArrayType* ::= *type*
*rankSpecifiers* ::= *rankSpecifier*∗
*rankSpecifier* ::= OPEN_BRACK *COMMA*∗ CLOSE_BRACK
*arrayInitializer* ::= OPEN_CURLY ( CLOSE_CURLY | *variableInitializerList COMMA*? CLOSE_CURLY )

| | | |
|---|---|---|
| *variableInitializerList* | ::= | *variableInitializer* ( COMMA *variableInitializer* )* |
| *interfaceDeclaration* | ::= | INTERFACE IDENTIFIER *interfaceBase interfaceBody SEMI*? |
| *interfaceBase* | ::= | ( COLON *type* ( COMMA *type* )* )? |
| *interfaceBody* | ::= | OPEN_CURLY *interfaceMemberDeclarations* CLOSE_CURLY |
| *interfaceMemberDeclarations* | ::= | *interfaceMemberDeclaration*∗ |
| *interfaceMemberDeclaration* | ::= | *NEW*? ( *voidAsType* IDENTIFIER OPEN_PAREN *formalParameterList*? CLOSE_PAREN SEMI \| *type* IDENTIFIER ( OPEN_PAREN *formalParameterList*? CLOSE_PAREN SEMI \| OPEN_CURLY *interfaceAccessors* CLOSE_CURLY ) ) |
| *interfaceAccessors* | ::= | *getAccessorDeclaration setAccessorDeclaration*? \| *setAccessorDeclaration getAccessorDeclaration*? |
| *voidAsType* | ::= | VOID |

## A.2 Lexical Specification

```
KEYWORDS          ::=  ABSTRACT | BASE | BOOL | BREAK | CASE | CATCH | CHAR | CLASS |
                       CONST | CONTINUE | DEFAULT | DO | DOUBLE | ELSE | FALSE |
                       FINALLY | FOR | FOREACH | IF | IN | INT | INTERFACE | INTERNAL |
                       IS | NAMESPACE | NEW | NULL | OBJECT | OVERRIDE | PRIVATE |
                       PROTECTED | PUBLIC | RETURN | STATIC | STRING | SWITCH | THIS |
                       THROW | TRUE | TRY | USING | VAR | VIRTUAL | VOID | WHILE
IDENTIFIER        ::=  '@'? ( '_' | LETTER_CHARACTER )
                       (
                        LETTER_CHARACTER | DECIMAL_DIGIT
                       )*
INT_LITERAL       ::=  '0' ( 'x' | 'X' ) HEX_DIGIT + | DECIMAL_DIGIT +
DOUBLE_LITERAL    ::=  DOT
                       (
                        DECIMAL_DIGIT +
                        (
                         ( 'e' | 'E' ) ( '+' | '-' )? DECIMAL_DIGIT +
                        )?
                        ( 'd' | 'D' )?
                       )? |
                       DECIMAL_DIGIT +
                       (
                        DOT DECIMAL_DIGIT +
                        (
                         ( 'e' | 'E' ) ( '+' | '-' )? DECIMAL_DIGIT +
                        )? ( 'd' | 'D' )? |
                        ( 'e' | 'E' ) ( '+' | '-' )? DECIMAL_DIGIT + ( 'd' | 'D' )? |
                        ( 'd' | 'D' )
                       )?

  CHAR_LITERAL        ::=  ' ' ' ( LETTER_CHARACTER | ESCAPED_LITERAL ) ' ' '
  STRING_LITERAL      ::=  ' " ' ( LETTER_CHARACTER | ESCAPED_LITERAL ) * ' " '
  ESCAPED_LITERAL     ::=  '\\'
                           ( ' ' ' | ' " ' | '\\' | '0' | 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' |
                             'x' HEX_DIGIT ( HEX_DIGIT ( HEX_DIGIT HEX_DIGIT ? )? )?
                           )
  DECIMAL_DIGIT       ::=  [0-9]
  HEX_DIGIT           ::=  [0-9] | [A-F] | [a-f]
```

# Appendix B

# Runtime Performance Tables

## B.1 Micro-benchmark

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Explicit | 2.39 | ±6.0% | 5.82 | ±0.0% | 2.43 | ±0.1% | 2.14 | ±0.1% | 9.41 | ±0.1% | 2.90 | ±0.1% |
| One | 2.40 | ±0.0% | 461 | ±0.1% | 1,496 | ±0.0% | 224 | ±0.0% | 932 | ±0.0% | 5,442 | ±0.1% |
| Five | 24.00 | ±9.2% | 431 | ±0.0% | 1,513 | ±0.0% | 293 | ±0.1% | 971 | ±0.1% | 5,839 | ±0.0% |
| Ten | 45.67 | ±9.1% | 473 | ±0.0% | 1,679 | ±0.1% | 360 | ±0.0% | 1,023 | ±0.1% | 6,083 | ±0.0% |
| Fifty | 60.00 | ±0.0% | 698 | ±0.1% | 1,974 | ±0.1% | 942 | ±0.1% | 1,063 | ±0.1% | 6,190 | ±0.0% |
| Hundred | 82.67 | ±9.0% | 824 | ±0.1% | 2,130 | ±0.1% | 1,470 | ±0.1% | 1,116 | ±0.1% | 6,497 | ±0.1% |
| **Total** | **3,524** | **±6.0%** | **8,492** | **±0.0%** | **14,328** | **±0.1%** | **9,504** | **±0.1%** | **8,514** | **±0.1%** | **37,221** | **±0.1%** |

Table B.1: Execution times ($\mu$seconds) of the micro-benchmark.

## B.2 Dynamically Typed Code

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FFT | 2,007 | ±1.9% | 16,601 | ±3.4% | 175,963 | ±1.8% | 2,897 | ±5.8% | 90,730 | ±1.0% | 5,678 | ±1.4% | 20,583 | ±2.8% |
| HeapSort | 3,475 | ±2.0% | 40,193 | ±2.0% | 1,427,218 | ±1.9% | 24,611 | ±1.5% | 544,441 | ±1.2% | 114,774 | ±1.9% | 169,128 | ±1.3% |
| SparseMatmult | 1,700 | ±0.0% | 35,551 | ±2.2% | 173,147 | ±1.9% | 2,844 | ±2.0% | 85,753 | ±1.0% | 17,374 | ±2.0% | 41,506 | ±1.1% |
| RayTracer | 6,240 | ±0.0% | 6,744 | ±2.0% | 72,102 | ±1.9% | 9,882 | ±1.9% | 40,404 | ±1.9% | 11,726 | ±1.9% | 63,020 | ±1.0% |
| **Total** | **13,421** | **±1.0%** | **99,089** | **±2.4%** | **1,848,430** | **±1.9%** | **40,234** | **±2.8%** | **761,328** | **±1.3%** | **149,553** | **±1.8%** | **294,236** | **±1.6%** |

Table B.2: Execution times (ms) of the *Java Grande* benchmarks with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 1,977 | ±4.8% | 2,683 | ±2.0% | 23,940 | ±1.8% | 5,914 | ±2.8% | 9,540 | ±1.9% | 4,764 | ±2.0% | 13,060 | ±1.7% |

Table B.3: Execution times (ms) of the *Pystone* benchmark with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Points | 171 | ±0.0% | 247 | ±2.0% | 2,554 | ±1.9% | 303 | ±4.0% | 1,180 | ±2.0% | 314 | ±2.9% | 1,575 | ±0.0% |

Table B.4: Execution times (ms) of the *Points* application with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Aritmethic.SimpleFloatArithmetic | 31 | ±0.0% | 2,021 | ±2.0% | 33,633 | ±1.8% | 1,063 | ±1.8% | 34,191 | ±1.7% | 437 | ±0.0% | 707 | ±3.8% |
| Arithmetic.SimpleIntegerArithmetic | 15 | ±0.0% | 2,180 | ±2.0% | 149,866 | ±1.9% | 1,034 | ±2.0% | 31,987 | ±2.0% | 338 | ±2.0% | 801 | ±1.9% |
| Arithmetic.SimpleIntFloatArithmetic | 15 | ±0.0% | 2,200 | ±2.0% | 148,232 | ±1.6% | 1,030 | ±1.9% | 34,424 | ±1.9% | 330 | ±2.2% | 795 | ±1.9% |
| Calls.FunctionCalls | 15 | ±0.0% | 171 | ±0.0% | 159 | ±6.1% | 705 | ±1.9% | 640 | ±0.0% | 309 | ±5.6% | 187 | ±0.0% |
| Calls.MethodCalls | 157 | ±1.9% | 1,154 | ±0.0% | 8,502 | ±1.7% | 631 | ±1.9% | 4,914 | ±0.0% | 2,215 | ±0.0% | 26,634 | ±1.1% |
| Calls.Recursion | 121 | ±3.6% | 1,950 | ±0.0% | 64,401 | ±1.9% | 561 | ±2.2% | 12,694 | ±1.9% | 414 | ±1.9% | 624 | ±0.0% |
| Constructs.ForLoops | 698 | ±2.1% | 5,335 | ±0.0% | 122,810 | ±1.7% | 4,695 | ±1.3% | 120,846 | ±1.8% | 908 | ±1.7% | 10,384 | ±1.4% |
| Constructs.IfThenElse | 46 | ±0.0% | 1,963 | ±2.3% | 717 | ±0.0% | 373 | ±3.4% | 1,557 | ±1.9% | 250 | ±0.0% | 421 | ±0.0% |
| Constructs.NestedForLoops | 507 | ±3.3% | 3,691 | ±1.9% | 82,789 | ±1.4% | 4,087 | ±1.5% | 172,866 | ±1.5% | 834 | ±2.9% | 59,820 | ±0.8% |
| Dicts.DictCreation | 452 | ±0.0% | 907 | ±2.0% | 23,852 | ±1.2% | 1,418 | ±1.9% | 5,954 | ±0.6% | 1,231 | ±9.4% | 6,006 | ±1.5% |
| Dicts.DictWithFloatKeys | 738 | ±4.3% | 5,670 | ±2.8% | 36,418 | ±1.9% | 1,490 | ±1.6% | 53,336 | ±1.6% | 1,747 | ±1.8% | 10,522 | ±1.4% |
| Dicts.DictWithIntegerKeys | 489 | ±2.5% | 7,413 | ±2.7% | 41,832 | ±1.3% | 1,709 | ±1.8% | 28,330 | ±1.0% | 1,314 | ±1.6% | 13,973 | ±1.9% |
| Dicts.DictWithStringKeys | 856 | ±1.8% | 6,700 | ±2.0% | 39,257 | ±1.1% | 1,794 | ±0.0% | 29,164 | ±1.6% | 1,342 | ±0.0% | 13,973 | ±1.8% |
| Dicts.SimpleDictManipulation | 515 | ±2.7% | 8,216 | ±1.9% | 69,739 | ±1.6% | 655 | ±0.0% | 32,194 | ±2.0% | 1,689 | ±1.3% | 47,632 | ±1.2% |
| Exceptions.TryExcept | 78 | ±0.0% | 296 | ±0.0% | 2,021 | ±2.0% | 67 | ±8.4% | 671 | ±0.0% | 647 | ±1.9% | 280 | ±2.7% |
| Exceptions.TryRaiseExcept | 1,591 | ±1.5% | 2,050 | ±1.5% | 1,884 | ±1.5% | 1,856 | ±0.0% | 2,179 | ±1.7% | 3,900 | ±0.0% | 1,669 | ±0.0% |
| Instances.CreateInstances | 358 | ±0.0% | 686 | ±3.0% | 11,103 | ±2.0% | 1,204 | ±3.2% | 2,955 | ±1.8% | 4,891 | ±2.0% | 819 | ±2.0% |
| Lists.ListSlicing | 89 | ±5.6% | 1,482 | ±0.0% | 2,367 | ±1.7% | 256 | ±2.3% | 698 | ±2.0% | 3,034 | ±1.8% | 2,932 | ±0.0% |
| Lists.SimpleListManipulation | 889 | ±0.0% | 5,271 | ±1.9% | 38,520 | ±1.7% | 3,151 | ±0.0% | 53,907 | ±2.0% | 883 | ±2.0% | 14,363 | ±1.8% |
| Lookups.NormalClassAttribute | 109 | ±0.0% | 98 | ±9.4% | 377 | ±1.9% | 265 | ±0.0% | 1,659 | ±1.9% | 2,274 | ±1.9% | 210 | ±2.8% |
| Lookups.NormalInstanceAttribute | 132 | ±4.5% | 3,801 | ±1.0% | 14,149 | ±0.0% | 1,063 | ±1.5% | 21,473 | ±0.7% | 2,145 | ±2.0% | 63,601 | ±1.6% |
| NewInstances.CreateNewInstances | 558 | ±6.6% | 727 | ±1.8% | 11,320 | ±2.0% | 1,180 | ±1.9% | 2,782 | ±1.3% | 12,888 | ±1.9% | 909 | ±3.9% |
| Numbers.CompareFloats | 93 | ±0.0% | 2,139 | ±1.9% | 77,412 | ±1.9% | 1,388 | ±1.8% | 1,196 | ±2.0% | 406 | ±0.0% | 943 | ±2.4% |
| Numbers.CompareFloatsIntegers | 78 | ±0.0% | 1,631 | ±1.9% | 61,547 | ±2.0% | 1,174 | ±1.8% | 48,068 | ±2.0% | 339 | ±2.2% | 727 | ±3.0% |
| Numbers.CompareIntegers | 296 | ±0.0% | 3,469 | ±1.7% | 108,850 | ±0.9% | 1,774 | ±1.2% | 2,803 | ±1.3% | 380 | ±2.0% | 1,318 | ±1.9% |
| Strings.CompareStrings | 4,056 | ±0.0% | 6,583 | ±3.9% | 102,018 | ±1.8% | 5,627 | ±1.8% | 28,699 | ±1.8% | 5,340 | ±1.8% | 159,975 | ±1.3% |
| Strings.ConcatStrings | 2,297 | ±1.9% | 1,985,232 | ±1.7% | 48,691 | ±1.5% | 3,010 | ±1.8% | 31,351 | ±1.0% | 2,814 | ±1.8% | 3,322 | ±0.0% |
| Strings.CreateStringsWithConcat | 1,045 | ±0.0% | 1,948,555 | ±1.7% | 45,353 | ±1.6% | 1,330 | ±1.6% | 27,866 | ±2.0% | 1,323 | ±2.0% | 1,591 | ±0.0% |
| Strings.StringMappings | 1,263 | ±0.0% | 2,320 | ±1.8% | 5,358 | ±1.4% | 1,497 | ±0.0% | 11,005 | ±1.7% | 812 | ±1.9% | 21,461 | ±1.5% |
| Strings.StringSlicing | 499 | ±0.0% | 5,080 | ±2.0% | 40,436 | ±0.0% | 1,528 | ±0.0% | 39,482 | ±2.0% | 902 | ±2.0% | 110,003 | ±1.7% |
| Strings.StringPredicates | 40 | ±0.0% | 481 | ±4.1% | 1,548 | ±1.9% | 807 | ±1.9% | 14,969 | ±2.0% | 967 | ±0.0% | 2,948 | ±1.7% |
| **Total** | **18,126** | **±1.4%** | **4,019,470** | **±2.0%** | **1,395,162** | **±1.6%** | **48,422** | **±1.7%** | **854,859** | **±1.5%** | **57,303** | **±1.9%** | **579,550** | **±1.5%** |

Table B.5: Execution times (ms) of the *Pybench* benchmark.

## B.3 Hybrid Dynamic and Static Typing Code

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FFT | 205 | ±8.8% | 5,899 | ±2.0% | 118,652 | ±1.9% | 1,281 | ±1.9% | 47,783 | ±1.7% | 2,569 | ±1.8% |
| HeapSort | 904 | ±3.0% | 8,127 | ±0.0% | 56,994 | ±1.2% | 2,750 | ±1.4% | 28,595 | ±0.8% | 16,338 | ±1.2% |
| SparseMatmult | 140 | ±0.0% | 816 | ±1.9% | 44,912 | ±0.0% | 561 | ±0.0% | 16,006 | ±1.0% | 820 | ±2.0% |
| RayTracer | 48 | ±0.0% | 1,099 | ±2.0% | 17,655 | ±1.0% | 793 | ±2.0% | 6,143 | ±1.5% | 1,093 | ±1.9% |
| **Total** | **1,297** | **±2.9%** | **15,940** | **±1.5%** | **238,212** | **±1.0%** | **5,386** | **±1.3%** | **98,527** | **±1.3%** | **20,820** | **±1.7%** |

Table B.6: Execution times (ms) of the *Java Grande* benchmarks with hybrid typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 87 | ±6.7% | 1,137 | ±1.9% | 11,576 | ±1.9% | 735 | ±2.3% | 3,795 | ±1.7% | 6,063 | ±1.8% |

Table B.7: Execution times (ms) of the *Pystone* benchmark with hybrid typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Points | 148 | ±5.2% | 158 | ±5.6% | 1,107 | ±0.0% | 233 | ±0.0% | 759 | ±2.0% | 1,531 | ±2.0% |

Table B.8: Execution times (ms) of the *Points* application with hybrid typing.

## B.4 Explicitly Typed Benchmarks

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FFT | 31 | ±0.0% | 296 | ±0.0% | 33 | ±8.0% | 31 | ±0.0% | 31 | ±0.0% | 67 | ±8.4% |
| HeapSort | 374 | ±0.0% | 3,588 | ±0.0% | 336 | ±3.3% | 312 | ±0.0% | 16,526 | ±1.1% | 378 | ±5.6% |
| SparseMatmult | 46 | ±0.0% | 436 | ±0.0% | 48 | ±8.4% | 46 | ±0.0% | 827 | ±0.0% | 48 | ±6.8% |
| RayTracer | 17 | ±0.0% | 46 | ±2.8% | 25 | ±1.6% | 16 | ±0.0% | 125 | ±0.0% | 32 | ±5.5% |
| **Total** | **468** | **±0.0%** | **4,366** | **±0.7%** | **443** | **±5.3%** | **405** | **±0.0%** | **17,508** | **±0.3%** | **526** | **±6.6%** |

Table B.9: Execution times (ms) of the *Java Grande* benchmarks with explicit typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 92 | ±7.8% | 75 | ±0.0% | 60 | ±0.0% | 68 | ±6.9% | 60 | ±0.0% | 77 | ±8.7% |

Table B.10: Execution times (ms) of the *Pystone* benchmark with explicit typing.

# Appendix C

# Memory Consumption Tables

## C.1 Micro-benchmark

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Explicit | 10,256 | ±1% | 15,503 | ±1% | 18,472 | ±1% | 10,374 | ±2% | 21,590 | ±0% | 10,863 | ±0% |
| One | 10,278 | ±2% | 17,805 | ±0% | 18,391 | ±1% | 17,923 | ±1% | 21,731 | ±1% | 14,590 | ±1% |
| Five | 10,454 | ±2% | 17,841 | ±2% | 18,454 | ±2% | 18,241 | ±1% | 21,748 | ±1% | 14,795 | ±2% |
| Ten | 10,482 | ±2% | 17,893 | ±0% | 18,439 | ±1% | 18,440 | ±2% | 21,731 | ±1% | 14,811 | ±2% |
| Fifty | 14,236 | ±2% | 18,229 | ±1% | 18,630 | ±1% | 18,706 | ±1% | 21,919 | ±1% | 14,946 | ±1% |
| Hundred | 22,026 | ±1% | 18,394 | ±2% | 18,945 | ±1% | 19,494 | ±2% | 22,158 | ±1% | 14,903 | ±2% |
| No Inference | 18,817 | ±0% | 18,140 | ±1% | 18,799 | ±0% | 18,306 | ±2% | 22,166 | ±0% | 14,862 | ±1% |
| **Total** | **96,549** | **±1%** | **123,806** | **±1%** | **130,130** | **±1%** | **121,485** | **±1%** | **153,043** | **±1%** | **99,769** | **±1%** |

Table C.1: Memory consumption (Kbytes) of the micro-benchmark.

## C.2 Dynamically Typed Code

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JG.SparseMatmultBench | 17,289 | ±0.0% | 22,970 | ±1.0% | 22,995 | ±0.7% | 29,215 | ±1.9% | 45,957 | ±0.2% | 137,680 | ±9.7% | 20,705 | ±1.3% |
| JG.HeapSortBench | 20,634 | ±0.5% | 22,187 | ±0.6% | 22,143 | ±0.0% | 28,395 | ±1.9% | 38,179 | ±0.8% | 53,610 | ±1.8% | 19,487 | ±1.5% |
| JG.FFTBench | 19,520 | ±0.8% | 19,847 | ±0.9% | 20,748 | ±0.2% | 36,641 | ±0.1% | 23,388 | ±0.0% | 65,293 | ±1.7% | 17,142 | ±1.4% |
| JG.RayTracerBench | 14,944 | ±1.7% | 18,459 | ±1.0% | 18,599 | ±1.8% | 29,913 | ±1.9% | 23,009 | ±0.8% | 69,849 | ±0.1% | 16,153 | ±1.9% |
| **Total** | **72,387** | **±0.8%** | **83,463** | **±0.9%** | **84,485** | **±0.7%** | **124,164** | **±1.4%** | **130,533** | **±0.4%** | **326,431** | **±3.3%** | **73,487** | **±1.5%** |

Table C.2: Memory consumption (Kbytes) of the *Java Grande* benchmarks with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 13,294 | ±0.8% | 18,603 | ±2.0% | 18,629 | ±2.0% | 26,283 | ±1.7% | 22,427 | ±1.1% | 49,445 | ±0.7% | 15,772 | ±1.2% |

Table C.3: Memory consumption (Kbytes) of the *Pystone* benchmarks with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Points | 16,011 | ±0.9% | 25,942 | ±0.1% | 22,002 | ±1.8% | 26,608 | ±1.1% | 22,559 | ±0.8% | 53,387 | ±2.0% | 21,996 | ±1.1% |

Table C.4: Memory consumption (Kbytes) of the *Points* example with dynamic typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | IronPython | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arithmetic.SimpleFloatArithmetic | 12,757 | ±0.9% | 16,671 | ±1.3% | 18,398 | ±1.9% | 27,249 | ±0.7% | 22,275 | ±0.6% | 54,295 | ±2.0% | 14,584 | ±1.9% |
| Arithmetic.SimpleIntegerArithmetic | 12,678 | ±0.8% | 16,850 | ±0.7% | 18,545 | ±1.9% | 27,199 | ±1.8% | 22,225 | ±1.7% | 51,474 | ±1.8% | 14,552 | ±1.7% |
| Arithmetic.SimpleIntFloatArithmetic | 12,733 | ±1.9% | 16,818 | ±1.3% | 18,485 | ±1.2% | 27,224 | ±0.1% | 21,797 | ±1.2% | 51,346 | ±1.8% | 14,499 | ±1.9% |
| Calls.FunctionCalls | 11,403 | ±1.9% | 16,634 | ±0.4% | 20,129 | ±0.5% | 23,721 | ±0.6% | 21,780 | ±0.2% | 51,697 | ±0.9% | 13,187 | ±2.0% |
| Calls.MethodCalls | 12,310 | ±1.9% | 17,800 | ±1.7% | 19,829 | ±1.1% | 25,082 | ±0.1% | 22,256 | ±0.2% | 55,411 | ±1.5% | 21,978 | ±0.5% |
| Calls.Recursion | 12,003 | ±1.5% | 16,935 | ±2.0% | 18,635 | ±0.2% | 19,315 | ±1.0% | 21,727 | ±0.8% | 53,219 | ±1.4% | 14,369 | ±1.4% |
| Constructs.ForLoops | 13,062 | ±2.0% | 16,930 | ±1.2% | 21,634 | ±1.5% | 29,184 | ±1.3% | 22,088 | ±1.5% | 52,079 | ±1.8% | 15,470 | ±1.6% |
| Constructs.IfThenElse | 12,790 | ±0.3% | 16,821 | ±1.8% | 18,578 | ±1.4% | 19,243 | ±0.4% | 21,854 | ±0.2% | 58,287 | ±1.6% | 14,576 | ±1.2% |
| Constructs.NestedForLoops | 12,453 | ±2.0% | 16,876 | ±1.5% | 18,585 | ±1.7% | 21,923 | ±1.1% | 22,078 | ±1.2% | 45,743 | ±2.0% | 15,488 | ±1.0% |
| Dicts.DictCreation | 12,609 | ±2.0% | 16,364 | ±1.3% | 21,785 | ±1.2% | 26,792 | ±0.8% | 22,002 | ±0.5% | 52,218 | ±0.5% | 15,079 | ±0.2% |
| Dicts.DictWithFloatKeys | 12,317 | ±1.2% | 17,558 | ±1.5% | 19,819 | ±0.7% | 24,973 | ±1.1% | 22,096 | ±1.2% | 54,886 | ±2.0% | 14,646 | ±1.6% |
| Dicts.DictWithIntegerKeys | 12,444 | ±1.2% | 17,566 | ±0.2% | 19,147 | ±1.9% | 24,750 | ±0.7% | 22,217 | ±2.0% | 50,685 | ±1.2% | 14,668 | ±1.2% |
| Dicts.DictWithStringKeys | 12,800 | ±2.0% | 17,516 | ±1.7% | 18,602 | ±0.2% | 24,792 | ±1.8% | 22,188 | ±2.0% | 50,737 | ±0.6% | 15,213 | ±1.9% |
| Dicts.SimpleDictManipulation | 12,586 | ±2.1% | 17,831 | ±1.6% | 20,728 | ±1.0% | 23,322 | ±1.2% | 22,199 | ±1.8% | 53,715 | ±1.2% | 15,869 | ±1.7% |
| Exceptions.TryExcept | 20,214 | ±0.7% | 18,421 | ±1.8% | 18,651 | ±0.2% | 18,822 | ±1.6% | 22,110 | ±1.4% | 106,167 | ±1.9% | 24,343 | ±1.5% |
| Exceptions.TryRaiseExcept | 12,467 | ±2.0% | 17,189 | ±1.9% | 18,614 | ±0.2% | 19,030 | ±1.6% | 21,262 | ±1.0% | 50,803 | ±0.4% | 14,846 | ±1.2% |
| Instances.CreateInstances | 12,174 | ±1.8% | 17,087 | ±1.2% | 18,522 | ±2.0% | 19,317 | ±0.8% | 21,965 | ±0.2% | 76,612 | ±0.7% | 14,580 | ±1.7% |
| Lists.ListSlicing | 12,350 | ±1.5% | 19,626 | ±0.6% | 18,633 | ±1.2% | 23,388 | ±1.1% | 22,157 | ±0.8% | 45,962 | ±1.7% | 16,050 | ±1.1% |
| Lists.SimpleListManipulation | 16,971 | ±2.3% | 17,758 | ±1.8% | 20,120 | ±1.9% | 36,774 | ±1.2% | 22,267 | ±1.0% | 52,253 | ±1.5% | 16,200 | ±1.2% |
| Lookups.NormalClassAttribute | 12,767 | ±0.8% | 15,538 | ±1.7% | 18,573 | ±1.5% | 21,830 | ±0.8% | 22,075 | ±1.2% | 53,040 | ±1.7% | 14,508 | ±0.5% |
| Lookups.NormalInstanceAttribute | 13,906 | ±1.6% | 17,594 | ±1.9% | 18,395 | ±0.4% | 28,930 | ±0.0% | 22,110 | ±1.0% | 52,837 | ±1.9% | 15,337 | ±0.9% |
| NewInstances.CreateNewInstances | 12,313 | ±2.0% | 17,076 | ±1.9% | 18,492 | ±0.7% | 19,478 | ±0.8% | 22,166 | ±0.8% | 78,254 | ±1.0% | 14,617 | ±1.3% |
| Numbers.CompareFloats | 12,487 | ±0.3% | 16,785 | ±0.0% | 18,548 | ±1.7% | 32,969 | ±1.0% | 22,190 | ±1.8% | 54,714 | ±1.6% | 14,896 | ±1.7% |
| Numbers.CompareFloatsIntegers | 12,289 | ±1.9% | 16,791 | ±1.9% | 18,627 | ±0.6% | 32,909 | ±0.7% | 22,276 | ±0.2% | 54,651 | ±1.1% | 14,819 | ±1.7% |
| Numbers.CompareIntegers | 12,184 | ±0.3% | 16,841 | ±0.7% | 18,510 | ±1.2% | 32,870 | ±0.2% | 22,036 | ±1.0% | 54,468 | ±1.8% | 14,875 | ±1.6% |
| Strings.CompareStrings | 12,421 | ±1.9% | 18,164 | ±0.7% | 19,127 | ±1.2% | 28,680 | ±1.3% | 22,133 | ±1.8% | 56,662 | ±1.8% | 17,177 | ±1.9% |
| Strings.ConcatStrings | 12,509 | ±1.9% | 17,701 | ±1.9% | 18,630 | ±1.2% | 24,353 | ±0.9% | 22,136 | ±1.0% | 55,738 | ±1.9% | 15,370 | ±1.6% |
| Strings.CreateStringsWithConcat | 12,482 | ±0.7% | 17,828 | ±0.2% | 18,495 | ±0.6% | 23,247 | ±1.4% | 22,124 | ±1.5% | 55,989 | ±2.0% | 14,672 | ±1.8% |
| Strings.StringMappings | 12,391 | ±1.8% | 18,238 | ±1.5% | 18,597 | ±1.3% | 24,549 | ±0.7% | 22,163 | ±0.7% | 50,838 | ±0.5% | 15,368 | ±1.1% |
| Strings.StringSlicing | 12,430 | ±1.8% | 18,147 | ±1.1% | 18,979 | ±0.6% | 27,263 | ±1.3% | 22,202 | ±0.8% | 54,457 | ±1.9% | 17,465 | ±1.4% |
| Strings.StringPredicates | 12,420 | ±0.9% | 17,852 | ±0.8% | 18,594 | ±1.3% | 24,451 | ±0.2% | 22,263 | ±1.0% | 53,346 | ±1.4% | 15,460 | ±0.7% |
| **Total** | **399,722** | **±1.5%** | **537,805** | **±1.3%** | **591,006** | **±1.1%** | **783,629** | **±0.9%** | **684,419** | **±1.0%** | **1,742,581** | **±1.4%** | **484,762** | **±1.4%** |

Table C.5: Memory consumption (Kbytes) of the *Pybench* benchmark.

## C.3 Hybrid Dynamic and Static Typing Code

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JG.SparseMatmultBench | 16,941 | ±1.9% | 21,902 | ±1.7% | 22,762 | ±1.5% | 25,313 | ±0.6% | 44,741 | ±0.8% | 19,575 | ±1.3% |
| JG.HeapSortBench | 16,208 | ±0.2% | 22,046 | ±1.9% | 22,010 | ±1.8% | 24,154 | ±0.8% | 43,651 | ±0.3% | 19,407 | ±0.4% |
| JG.FFTBench | 13,701 | ±2.0% | 18,973 | ±0.2% | 19,855 | ±0.8% | 26,842 | ±1.9% | 28,682 | ±2.0% | 16,065 | ±0.0% |
| JG.RayTracerBench | 11,732 | ±1.9% | 18,222 | ±0.6% | 18,334 | ±1.8% | 28,291 | ±0.5% | 22,798 | ±0.8% | 15,712 | ±1.3% |
| **Total** | **58,583** | **±1.5%** | **81,143** | **±1.1%** | **82,961** | **±1.5%** | **104,600** | **±1.0%** | **139,873** | **±1.0%** | **70,759** | **±0.7%** |

Table C.6: Memory consumption (Kbytes) of the *Java Grande* benchmarks with hybrid typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 12,222 | ±0.0% | 18,240 | ±0.8% | 18,381 | ±0.7% | 25,440 | ±0.1% | 22,364 | ±0.2% | 15,438 | ±1.9% |

Table C.7: Memory consumption (Kbytes) of the *Pystone* benchmark with hybrid typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Points | 12,493 | ±0.3% | 16,679 | ±0.4% | 18,420 | ±0.0% | 20,271 | ±0.5% | 15,262 | ±0.2% | 13,244 | ±1.8% |

Table C.8: Memory consumption (Kbytes) of the *Points* application with hybrid typing.

## C.4 Explicitly Typed Benchmarks

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JG.SparseMatmultBench | 15,112 | ±1.7% | 20,544 | ±1.5% | 21,940 | ±1.5% | 15,177 | ±1.7% | 44,042 | ±0.1% | 15,933 | ±1.2% |
| JG.HeapSortBench | 14,452 | ±0.8% | 19,649 | ±1.6% | 21,243 | ±1.8% | 14,302 | ±1.5% | 57,184 | ±1.4% | 15,077 | ±0.7% |
| JG.FFTBench | 11,882 | ±0.6% | 16,984 | ±0.2% | 18,721 | ±1.6% | 11,736 | ±0.8% | 26,926 | ±1.6% | 15,356 | ±1.0% |
| JG.RayTracerBench | 10,707 | ±0.0% | 15,841 | ±0.7% | 18,452 | ±0.5% | 10,726 | ±2.0% | 17,037 | ±0.6% | 13,169 | ±1.7% |
| **Total** | **52,154** | **±0.8%** | **73,018** | **±1.0%** | **80,356** | **±1.4%** | **51,941** | **±1.5%** | **145,190** | **±0.9%** | **59,535** | **±1.1%** |

Table C.9: Memory consumption (Kbytes) of the *Java Grande* benchmarks with explicit typing.

| Test | *StaDyn* | | Boo | | Cobra | | C# | | Fantom | | VB | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pystone | 11,684 | ±0.3% | 16,820 | ±2.0% | 18,418 | ±0.2% | 11,677 | ±1.9% | 15,376 | ±1.4% | 13,782 | ±1.1% |

Table C.10: Memory consumption (Kbytes) of the *Pystone* benchmark with explicit typing.

# Appendix D

# Publications

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

– Articles published in journal in the *Journal Citation Reports*:

1. Union and intersection types to support both dynamic and static typing. Francisco Ortin, Miguel Garcia. *Information Processing Letters*, Volume 111, Issue 6, 2011.

2. Applying dynamic separation of aspects to distributed systems security: a case study. Miguel Garcia, David Llewellyn-Jones, Francisco Ortin, Madjid Merabti. *IET Software*, Volume 6, Issue 3, 2012.

3. Including both static and dynamic typing in the same programming language. Francisco Ortin, Daniel Zapico, J. Baltasar G. Perez-Schofield, Miguel Garcia. *IET Software*, Volume 4, Issue 4, 2010.

4. On the suitability of dynamic languages for hot-reprogramming a robotics framework: a Python case study. Francisco Ortin, Sheila Mendez, Vicente Garcia-Diaz and Miguel Garcia. *Software Practice and Experience*, DOI: 10.1002/spe.2162. To be published; accepted on October, 2012.

5. Towards a practical solution for data grounding in a semantic web services environment. Miguel Garcia, Jose M. Alvarez, Diego Berrueta, Luis Polo, Jose E. Labra, Patricia Ordoñez. *Journal of Universal Computer Science*, Volume 18, Issue 11, 2012.

– Articles published other journals:

1. A Programming language that combines the benefits of static and dynamic Typing. Francisco Ortin, Miguel Garcia. *Communications in Computer and Information Science*, Volume 170. ISSN: 1865-0929. Springer Verlag. 2013.

2. Achieving multiple dispatch in hybrid statically and dynamically typed languages. Francisco Ortin, Miguel Garcia, Jose M. Redondo, Jose

Quiroga. *Advances in Intelligent Systems and Computing*, Volume 206, ISSN: 2194-5357. Springer Verlag. 2013.

3. Modularizing different responsibilities into separate parallel hierarchies. Francisco Ortin, Miguel Garcia. *Communications in Computer and Information Science*, Volume 275. ISSN: 1865-0929. Springer Verlag. 2013.

– Articles presented in conferences:

1. A programming language to facilitate the transition from rapid prototyping to efficient software production. Francisco Ortin, Daniel Zapico, Miguel Garcia. *5$^{th}$ International Conference on Software and Data Technologies* (ICSOFT), Athens (Greece). July 2010.

2. A performance cost evaluation of aspect weaving. Miguel Garcia, Francisco Ortin, David Llewellyn-Jones, Madjid Merabti. *ACM 36$^{th}$ Australian Computer Science Conference* (ACSC), Adelaide, (Australia). January 2013.

3. Supporting Dynamic and Static Typing by means of Union and Intersection Types. Francisco Ortin, Miguel Garcia. *IEEE International Conference on Progress in Informatics and Computing* (PIC), Shanghai (China). December 2010.

4. A type safe design to allow the separation of different responsibilities into parallel hierarchies. Francisco Ortin, Miguel Garcia. *6$^{th}$ International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE), Bejing (China). June 2011.

5. Computational reflection in order to support context-awareness in a robotics framework. Sheila Mendez, Francisco Ortin, Miguel Garcia, Vicente Garcia-Diaz. *23$^{rd}$ International Conference on Software Engineering & Knowledge Engineering* (SEKE), Miami, Florida (USA). July 2011.

6. Separating different responsibilities into parallel hierarchies. Francisco Ortin, Miguel Garcia. *ACM 4$^{th}$ International C* Conference on Computer Science and Software Engineering* (C3S2E), Montreal (Canada). May 2011

# References


[1] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*. Addison-Wesley Professional, Raleigh, North Carolina, 2nd edition, 2004. 1, 15

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

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

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

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

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

[7] Amos Latteier, Michel Pelletier, Chris McDonough, and Peter Sabaini. The Zope book. http://www.zope.org/Documentation/Books/ZopeBook/, 2008. 1

[8] Wim Vanderperren and Davy Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Dynamic Aspects Workshop*, pages 120–134, 2004. 2

[9] Markus Dahm. Byte code engineering. In *In Java-Information's Tage*, pages 267–277. Springer-Verlag, 1999. 2

[10] Shigeru Chiba. Load-time structural reflection in Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 313–336, 2000. 2

[11] Mike Grogan. JSR 223. Scripting for the Java platform. http://www.jcp.org/en/jsr/detail?id=223, 2012. 2

[12] Sun Microsystems. JSR 292, supporting dynamically typed languages on the java platform. http://www.jcp.org/en/jsr/detail?id=292. 2

[13] Bill Chiles and Alex Turner. Dynamic Language Runtime. http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512. 2, 6, 57, 61, 67, 80

[14] Jim Hugunin. Just glue it! Ruby and the DLR in Silverlight. In *The MIX Conference*, Las Vegas, Nevada, 30 April - 7 May 2007. 2, 6, 22

[15] Benjamin Crawford Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002. 2, 11

[16] 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. 2, 5

[17] Martín Abadi, Luca Cardelli, Benjamin Crawford Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991. 2, 7

[18] Martín Abadi, Luca Cardelli, Benjamin Crawford Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. 2

[19] Francisco Ortin, Daniel Zapico, Jose Baltasar Garcia Perez-Schofield, and Miguel Garcia. Including both static and dynamic typing in the same programming language. *IET Software*, 4(4):268–282, 2010. 3

[20] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. *SIGPLAN Notices*, 28(10):215–230, October 1993. 5

[21] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. 5

[22] Gilad Bracha. Pluggable Type Systems. In *Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, October 2004. ACM. 5, 17

[23] 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. 5

[24] Rodrigo B. De Oliveira. The Boo programming language. http://boo.codehaus.org. 6, 58

[25] Paul Vick. *The Microsoft Visual Basic Language Specification*. Microsoft Corporation, Redmond, Washington, 2007. 6, 57

[26] Mads Torgersen. *New features in C# 4.0*. Microsoft Corporation, Redmond, Washington, 2009. 6

[27] Francisco Ortin, Miguel Garcia, Jose Manuel 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. 6

[28] 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. 6, 8

[29] Stephen Kochan. *Programming in Objective-C 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 6

[30] TIOBE Software. The TIOBE programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. 6

[31] 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. 7, 58

[32] Brian Frank and Andy Frank. Fantom, the language formerly known as Fan. `http://fantom.org`, 2012. 7, 58

[33] Robert Cartwright and Miken Fagan. Soft Typing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Canada, 26-28 June 1991. ACM. 7

[34] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 7, 14

[35] 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. 7

[36] 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. 7

[37] 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. 7

[38] 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. 8

[39] 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. 8

[40] 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. 8

[41] Giovanni Lagorio and Elena Zucca. Just: Safe unknown types in java-like languages. *Journal of Object Technology*, 6(2):69–98, 2007. 8

[42] Giovanni Lagorio and Elena Zucca. Introducing safe unknown types in Java-like languages. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1429–1434, Dijon, France, 23-27 April 2006. ACM. 8

[43] Michael Furr, Jong-hoon David An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the ACM symposium on Applied Computing (SAC)*, pages 1859–1866, Honolulu, Hawaii, 9-12 March 2009. ACM. 8, 27

[44] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 428–452, Glasgow, UK, 9-11 June 2005. Springer. 8

[45] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. 9

[46] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. 9

[47] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM. 9

[48] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 587–606, New York, NY, USA, 2012. ACM. 9

[49] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997. 11

[50] Microsoft Corporation. The C# Programming Language. http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc. 13, 23, 26, 27

[51] 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. 14, 18

[52] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. 14

[53] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, , et al. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992. 14

[54] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. 14, 18

[55] 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. 15

[56] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995. 15, 22

[57] 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. 15

[58] 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. 15

[59] 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. 16

[60] 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. 16

[61] Benjamin Crawford Pierce. Programming with intersection types and bounded polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA, 1992. 16, 24, 33, 37

[62] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures*, 35:48–62, April 2009. 17

[63] 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. 18

[64] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997. 19

[65] 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. 19, 20, 40

[66] 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. 21

[67] 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. 21

[68] 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. 21

[69] Andrew W. Appel. *Modern compiler implementation in ML: basic techniques*. Cambridge University Press, New York, NY, USA, 1997. 21

[70] 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. 22

[71] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. 22

[72] 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. 22

[73] 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. 22

[74] Francisco Ortin and Miguel Garcia. Modularizing Different Responsibilities into Separate Parallel Hierarchies. *Communications in Computer and Information Science*, 275:16–31, January 2013. 22

[75] 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. 22

[76] Jose Manuel Redondo and Francisco Ortin. Optimizing reflective primitives of dynamic languages. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):759–783, 2008. 22, 53

[77] Francisco Ortin and Miguel Garcia. Union and intersection types to support both dynamic and static typing. *Information Processing Letters*, 111(6):278–286, 2011. 23, 33

[78] Francisco Ortin. Type inference to optimize a hybrid statically and dynamically typed language. *Computer Journal*, 54(11):1901–1924, November 2011. 23, 25, 72

[79] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119:202–230, June 1995. 24

[80] 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. 24, 40

[81] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the Symposium on Applied Computing (SAC)*, SAC '06, pages 1435–1441, Dijon, France, 23-27 April 2006. ACM. 24, 26

[82] Giovanni Lagorio and Davide Ancona. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 5653, pages 2–26, Genova, Italy, 6-10 July 2009. Springer-Verlag. 24

[83] Atsushi Igarashi, Benjamin Crawford Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23:396–450, May 2001. 26, 41

[84] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, 17-19 June 2002. ACM. 27

[85] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, Secaucus, NJ, USA, 1996. 32

[86] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*, pages 687–706, Sendai, Japan, 19-22 April 1994. Springer-Verlag. 33

[87] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A High-Level Modular Definition of the Semantics of C#. *Theoretical Computer Science*, 336:235–284, May 2005. 41

[88] Francisco Ortin, Jose Manuel Redondo, and Jose Baltasar Garcia Perez-Schofield. Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming*, 70(10):836–860, 2009. 53, 61, 64, 80

[89] IronPython. The IronPython programming language. http://ironpython.net/. 57

[90] Python Software Foundation. Pybench benchmark project trunk page. http://svn.python.org/projects/python/trunk/Tools/pybench/, 2012. 58

[91] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984. 58

[92] Krintz, Chandra. A collection of phoenix-compatible C# benchmarks. http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks, 2012. 58

[93] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007. 59

[94] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. 59

[95] MicrosoftTechnet. Windows server techcenter: Windows performance monitor. http://technet.microsoft.com/en-us/library/cc749249.aspx, 2012. 59

[96] Microsoft. Windows management instrumentation. http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx, 2012. 59

[97] Jose Manuel Redondo and Francisco Ortin. Efficient support of dynamic inheritance for class- and prototype-based languages. *Journal of Systems and Software*, 86(2):278–301, 2013. 67, 80

[98] ANSI/ISO C++ Committee. Programming languages–c++ iso/iec 14882: 1998 (e). *American National Standards Institute*, 1998. 80

[99] Luca Cardelli and John C. Mitchell. Operations on records. *Mathematical structures in computer science*, 1(01):3–48, 1991. 80

[100] Miguel Garcia, David Llewellyn-Jones, Francisco Ortin, and Madjid Merabti. Applying dynamic separation of aspects to distributed systems security: a case study. *IET Software*, 6(3):231–248, 2012. 80

[101] Miguel Garcia, Francisco Ortin, David Llewellyn-Jones, and Madjid Merabti. A performance cost evaluation of aspect weaving. In *Australian Computer Science Conference*, ACSC, jan 2013. 80

[102] Francisco Ortin, Sheila Mendez, Vicente Garcia-Diaz, and Miguel Garcia. On the suitability of dynamic languages for hot-reprogramming a robotics framework: a python case study. *Software Practice and Experience*, To be published; accepted on October, 2012. 80

[103] Miguel Garcia, Jose Maria Alvarez, Diego Berrueta, Luis Polo, Jose Emilio Labra, and Patricia Ordoñez. Towards a practical solution for data grounding in a semantic web services environment. *Journal of Universal Computer Science*, 18(11):1576–1597, jun 2012. 80