SoftwareX 20 (2022) 101211

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

The *StaDyn* programming language

Francisco Ortin [a,b,*], Miguel Garcia [a], Baltasar Garcia Perez-Schofield [c], Jose Quiroga [a]

[a] University of Oviedo, Computer Science Department, Federico Garcia Lorca 18, 33007, Oviedo, Spain
[b] Munster Technological University, Department of Computer Science, Rossa Avenue, Bishopstown, Cork, Ireland
[c] University of Vigo, Computer Science Department, As Lagoas s/n, 32004 Ourense, Spain

ARTICLE INFO

Article history:
Received 20 May 2022
Received in revised form 3 August 2022
Accepted 18 September 2022

Keywords:
Hybrid static and dynamic typing
Compiler optimizations
.NET platform
Programming language
StaDyn

ABSTRACT

Hybrid static and dynamic typing languages are aimed at combining the benefits of both kinds of languages: the early type error detection and compile-time optimizations of static typing, together with the runtime adaptability of dynamically typed languages. The *StaDyn* programming language is a hybrid typing language, whose main contribution is the utilization of the type information gathered by the compiler to improve compile-time error detection and runtime performance. *StaDyn* has been evaluated as the hybrid typing language for the .NET platform with the highest runtime performance and the lowest memory consumption. Although most optimizations are performed statically by the compiler, compilation time is yet lower than the existing hybrid languages implemented on the .NET platform.

Code metadata

Current code version	2.1.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00121
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	C# 5.0
Compilation requirements, operating environments & dependencies	.NET Framework 4.8
If available Link to developer documentation/manual	https://reflection.uniovi.es/stadyn/
Support email for questions	ortin@uniovi.es

1. Motivation and significance

Dynamically typed languages commonly support runtime-adaptable features such as reflection, metaprogramming, dynamic code generation, duck typing, and dynamic reconfiguration and distribution. The great runtime flexibility provided by dynamic languages has made them suitable for different scenarios such as runtime adaptable systems, rapid prototyping, dynamic aspect-oriented programming, and data processing and integration systems [1]. Taking the web development scenario as an example, Ruby [2] is used for the rapid development of database-backed web applications with the Ruby on Rails framework [3]. This framework has confirmed the simplicity of implementing the DRY (Do not Repeat Yourself) [4] and the Convention over Configuration [3] principles in a dynamically typed language. JavaScript [5] is being widely employed to create both front- and back-end modules of web applications [6]. Python [7] is used for many different purposes including web development with numerous frameworks (e.g., Django, CherryPy, Pyramid and Flask).

The great flexibility of dynamic languages is, however, counteracted by limitations derived from the lack of static type checking. The type information gathered by statically typed languages is commonly used to perform different optimizations and to provide the early detection of type errors [1]. Statically typed languages offer the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime [8]. Moreover, avoiding the runtime type inspection and type checking performed by dynamically typed languages commonly involve a runtime performance improvement [9,10].

* Corresponding author at: University of Oviedo, Computer Science Department, Federico Garcia Lorca 18, 33007, Oviedo, Spain.
E-mail addresses: ortin@uniovi.es (Francisco Ortin), garciarmiguel@uniovi.es (Miguel Garcia), jbgarcia@uvigo.es (Baltasar Garcia Perez-Schofield), quirogajose@uniovi.es (Jose Quiroga).

https://doi.org/10.1016/j.softx.2022.101211

Since both dynamic and static typing offer important benefits, there have been approaches aimed at obtaining the advantages of both approaches, following the philosophy of *static typing where possible, dynamic typing when needed* [1]. Out of all the theoretical research works, gradual typing has probably been the one with the highest impact [11]—soft typing [12], quasi-static typing [13] and hybrid typing [14] are other well-known works. Gradual typing defines a *consistency* relation that formally defines the interaction between static and dynamic types [15]. Ever since its conception, gradual typing has been an active research topic, which we detail in Section 5.

The research work in gradual and hybrid dynamic and static typing has influenced the design and implementation of some commercial programming languages. Different hybrid typing languages were created, such as Visual Basic, Objective-C, Dylan, Boo, Fantom and Cobra. Likewise, some dynamically typed languages, such as Groovy and PHP, became gradually typed after including static type annotations [16,17]. Moreover, the statically typed C# language included the `dynamic` type [18] to indicate the compiler to postpone type checks until runtime. Kotlin, when compiled to JavaScript, follows a similar approach with its `dynamic` type [19].

The *StaDyn* programming language presented in this article is a full-fledged object-oriented programming language that provides hybrid static and dynamic typing for the .Net platform. The main contribution of *StaDyn* compared with the existing hybrid languages is that it gathers type information from dynamically typed code and uses it for two main purposes: detecting at compile time type errors of dynamically typed code; and providing better runtime performance by reducing dynamic type-checking operations.

The rest of this article is structured as follows. An illustrative example is presented in Section 2. Section 3 describes the *StaDyn* programming language. An evaluation of the efficiency of *StaDyn* and some use case scenarios are presented in Section 4. Section 5 depicts the related work and conclusions are drawn in Section 6.

2. Illustrative example

Fig. 1 shows an illustrative example of *StaDyn* code with static and dynamic typing. The `DistanceToOrigin` method in Fig. 1 receives a `Circumference`, so the `DistanceToOrigin(cir)` invocation in line 26 is accepted by the compiler. On the contrary, the compiler rejects the next `DistanceToOrigin(rec)` invocation because `rec` is a `Rectangle`, not a `Circumference`. That invocation is rejected even though `rec` implements the two messages passed to the `figure` parameter (`GetX` and `GetY`). If the `figure` parameter had been dynamically typed, it would have produced neither a compiler nor a runtime error, because `rec` provides valid implementations of both `GetX` and `GetY`—that is the so-called duck typing feature of dynamic languages.

`Distance` provides an example of the dynamic typing approach, since its two parameters are declared as `dynamic`. In the invocation in line 28, a rectangle and a circumference are passed to `Distance`. That function call produces no runtime error because both objects provide `GetX` and `GetY`. That is not the case with the following `Distance(cir, tri)` invocation, since triangles do not provide `GetX` and `GetY`. In C#, the invocation is compiled and it throws an error at runtime, just like most dynamic languages do. However, the *StaDyn* compiler manages to show an error in line 29. That is because *StaDyn* performs type checking of dynamically typed code and knows that `tri` does not provide the implementation of `GetX` and `GetY`. Moreover, the type information gathered by the compiler is also used to optimize the code generated (see Section 4.1). This compile-time type inference and type checking of dynamically typed code is the main difference between *StaDyn* and the rest of the dynamically typed languages.

3. Software description

StaDyn is a hybrid static and dynamic typing object-oriented language for the .Net framework. It was created as an extension of C# 3 in 2007 by the Computational Reflection research group of the University of Oviedo, as a research project partially funded by Microsoft Research. *StaDyn* enhances the behavior of the C# implicitly typed local references (i.e., the `var` keyword) and the `dynamic` type that was later included in C# 4.

In *StaDyn*, the `var` keyword can be used to declare implicitly typed references. This means that `var` is a valid type for declaring fields, parameters, return values and local variables—in C#, `var` can only be used to define initialized local variables. In the example code in Fig. 1, all the `dynamic` references used (line 19) can be replaced with `var`. The main difference between `var` and `dynamic` is that the compiler is more lenient with `dynamic` references. With `var`, the compiler performs classical static typing, making sure that every possible execution flow is safe. However, with `dynamic` it allows constructs with at least one valid execution flow[1] (details are given in Section 3.4).

The *StaDyn* compiler is implemented in C# [20]. For lexical and syntactic analysis, we use the ANTLR LL(*) parser generator. The compiler generates assembly code in the intermediate language (IL) for the .Net platform. Finally, the IL assembler is used to generate the .Net binaries.

3.1. Variables holding different types at runtime

In dynamic languages, it is very common to use one unique variable to hold different types in the same scope. For example, the code on the left-hand side of Fig. 2 uses the `value` variable to hold the same value with different types. First, `value` stores the string representation of a real number (line 3) passed to the program from the command line prompt. In that case, the `Length` message can be passed to `value`, because `Length` is a property provided by `String`. Then, line 5 converts the `String` into a `double` number, so it can be passed to `Sqrt` (line 6). The `Length` property is no longer provided by `value`, so the *StaDyn* compiler produces a compile-time error in line 7 (unlike C#, which shows the error at runtime).

Common statically typed platforms such as Java and .Net do not allow variables to have different types in the same scope. So, if the `Object` type is used as the assembly translation of `dynamic` and `var`, the required casts or the use of reflection will slow down the execution of the application [21]. To avoid this runtime performance penalty, the *StaDyn* compiler transforms the program (its abstract syntax tree, AST) into another one where each dynamically typed reference is (statically) assigned at most once. That is, programs are transformed into Static Single Assignment (SSA) form [22].

The SSA transformation is shown in Fig. 2. The AST of the program on the left is translated into the AST of the program on the right [23]—the same occurs with `var` references. In this way, the type-checking phase of the *StaDyn* compiler infers a unique type for the `value` $_0$ (`string`) and `value` $_1$ (`double`) variables. Besides the robustness of detecting the error in line 7, the generated code provides significantly higher runtime performance by avoiding unnecessary casts and reflection (Section 4.1).

[1] The first version of *StaDyn* only supported our extension of the `var` type, not `dynamic`. Afterwards, C# 4.0 was launched and included the new `dynamic` type. We then created *StaDyn* 2.0 to include a new interpretation of `dynamic`. The exact differences between `var` and `dynamic` are detailed in Section 3.4

```
01:  public class Circumference {
02:    private double x, y, radius;
03:    public double GetX() { return this.x; }
04:    public double GetY() { return this.y; }
05:    public double GetRadius() { return this.radius; }
06:  }

07:  public class Rectangle {
08:    private double x, y, width, height;
09:    public double GetX() { return this.x; }
10:    public double GetY() { return this.y; }
11:  }

12:  class Triangle {
13:    private double x1, y1, x2, y2, x3, y3;
14:  }

15:  class Program {
16:    static double DistanceToOrigin(Circumference figure) {
17:      return Math.Sqrt(Math.Pow(figure.GetX(), 2) +
                          Math.Pow(figure.GetY(), 2));
18:    }
19:    static dynamic Distance(dynamic figure1, dynamic figure2) {
20:      return Math.Sqrt(Math.Pow(figure1.GetX()-figure2.GetX(), 2) +
                          Math.Pow(figure1.GetY()-figure2.GetY(), 2));
21:    }
22:    static void Main() {
23:      Circumference cir = new Circumference(0, 0, 10);
24:      Rectangle rec = new Rectangle(1.2, 3.7, 1.0, 2.0);
25:      Triangle tri = new Triangle(1, 2, -1, 3, 7, 2);
26:      DistanceToOrigin(cir);
27:      DistanceToOrigin(rec);
28:      Distance(cir, rec);
29:      Distance(cir, tri);
30:    }
31:  }
```

Fig. 1. Example *StaDyn* program with hybrid static and dynamic typing (class constructors are obviated).

```
01: public class Program {
02:   static void Main(String[] args) {
03:     dynamic value = args[0];
04:     Console.WriteLine("Characters: {0}.",
                          value.Length);
05:     value = Convert.ToDouble(value);
06:     Console.WriteLine("Square root: {0}.",
                          Math.Sqrt(value));
07:     int length = value.Length;
08:   }
09: }
```

```
01:  public class Program {
02:    static void Main(String[] args) {
03:      dynamic value₀ = args[0];
04:      Console.WriteLine("Characters: {0}.",
                           value₀.Length);
05:      value₁ = Convert.ToDouble(value₀);
06:      Console.WriteLine("Square root: {0}.",
                           Math.Sqrt(value₁));
07:      int length = value₁.Length;
08:    }
09:  }
```

Fig. 2. Left-hand side: *StaDyn* source code where value holds two different types. Right-hand side: the transformed program produced by the SSA algorithm.

```
01:  class Program {
02:    static dynamic Factory(int figureType, double x, double y) {
03:      if (figureType == CIRCUMFERENCE)
04:        return new Circumference(x, y);
05:      if (figureType == RECTANGLE)
06:        return new Rectangle(x, y);
07:      throw new ArgumentException("Unknown figure type");
08:    }
09:    static void Main(String[] args) {
10:      var figure = Factory(Convert.ToInt32(args[0]), 10, 20);
11:      figure.GetX();
12:      figure.GetRadius();
13:      figure.GetArea();
14:    }
15:  }
```

Fig. 3. Flow-sensitive types in *StaDyn*.

3.2. Flow-sensitive types

As in dynamic languages, *StaDyn* allows variables whose type depends on the execution flow. In the source code of Fig. 3, the type of `figure` (line 10) depends on the dynamic value of `args[0]`. It may be `Circumference` or `Rectangle`.

StaDyn models flow-sensitive types through union types [24]. The type of the `figure` variable is the `Circumference`∨ `Rectangle` union type. A union type $T_1 \vee T_2$ denotes the ordinary union of the set of values belonging to T_1 and T_2 [25], representing the least upper bound of T_1 and T_2 [26]. Therefore, the set of operations (e.g., addition, field access, assignment, invocation and indexing) that can be applied to a union type are those accepted by every type in the union type. For this reason, it is safe to call the `GetX` method of `figure` in line 11 of Fig. 3 (both `Circumference` and `Rectangle` provide `GetX`). This is what is called duck typing in the dynamic language community [27]. An important benefit of *StaDyn* is that duck typing is statically typed [28].

The behavior of `var` and `dynamic` references are not the same when it comes to type-check union types. For `var`, the type system checks, for a given operation, that such operation is supported by *all* the types in the union type. In our example, `figure.GetX()` is safe because both `Circumference` and `Rectangle` provide that method. However, the `GetRadius` message cannot be passed to `figure` (line 12) because it is not implemented by `Rectangle`. This is the classical interpretation of union types [29].

StaDyn provides a new interpretation of union types for `dynamic` Refs. [30]. In this case, the compiler is more lenient to follow the flavor of dynamic languages, but static typing is still performed. An operation is allowed when *at least one* of the types in the union type supports that operation. For example, if `figure` is declared as `dynamic` in Fig. 3 (line 10), the `figure.GetRadius()` statement in line 12 will be accepted by the compiler, because `GetRadius` is implemented by `Circumference`. On the other hand, `figure.GetArea()` is detected as a compiler error, even if `figure` is `dynamic`, because neither `Circumference` nor `Rectangle` supports that message. A formal description can be consulted in [31].

For both `dynamic` and `var` union types, *StaDyn* adds runtime type checks in the generated code [31]. As the number of types in the union type grows, those type checks consume more execution time [32]. Although the method specialization technique implemented (Section 3.5) significantly reduces this cost, union types holding a massive number of types are better avoided by defining a common supertype. Otherwise, the *StaDyn* compiler uses the type cache of the Dynamic Language Runtime (DLR) (Section 3.6) when a union type holds 126 types or more—from this value on, the DLR provides better performance.

3.3. Dynamically typed parameters

The type of both `dynamic` and `var` variables are inferred with a unification algorithm [33]. The *StaDyn* compiler models the type of `dynamic` and `var` variables as type variables that can be unified (i.e., instantiated or substituted) with any other type [34]. This unification is performed on method invocation, object construction and assignments.

Fig. 4 shows a *StaDyn* program excerpt with gray type annotations on the right. Type variables are represented as X_i, where i is a unique identifier. The type of `cir` (line 15) is the type variable X_6 that, initially, is not instantiated (a substitution has not been found for it). The assignment in line 15 unifies the type of `cir` (X_6) to `Circumference`. The SSA algorithm assures that one dynamically typed local variable is assigned once and hence has a unique type.

`var` and `dynamic` parameters cannot be unified with a single type. Since they represent the type of all the possible arguments to be passed, they cannot be inferred in the same way as local variables. This is the case of the `figure` parameter in line 2. In the first invocation to `Distance` (line 17), the parameter type variable (X_1) is unified to `Circumference` but, in the next invocation (line 18), X_1 will be `Rectangle`. Thus, the type of the parameter (X_1) varies depending on the invocation—the same happens for methods returning `var` or `dynamic`.

To support the type inference of `var` and `dynamic` parameters and return values, *StaDyn* includes constraints in its type system [35]. For example, the type of `Distance` is a method receiving X_1 and returning X_2, with the following constraints: X_1 must provide the `GetX()` and `GetY()` messages ($X_1 \leq$ {`GetX`(): X_3}, $X_1 \leq$ {`GetY`(): X_4}), since they are called in the method body; likewise, the two values returned by `GetX` and `GetY` (X_3 and X_4) must be subtypes of `double`, because they are passed to `Math.Pow`; and, finally, the type returned by `Distance` is `double`, since that is the type returned by `Math.Sqrt`.

```
01: class Program {
02:   static dynamic Distance(dynamic figure) {   Distance: X1→X2 | X1≤{GetX():X3},
                                                      X1≤{GetY():X4}, X3≤double,
                                                      X4≤double, X2←double
03:     return Math.Sqrt(Math.Pow(figure.GetX(), 2) + Math.Pow(figure.GetY(), 2));
04:   }
05:
06:   static dynamic Factory(int figureType,       Factory: int,double,double→X5  |
                        double x, double y) {          X5←Circumference∨Rectangle
07:     if (figureType == CIRCUMFERENCE)
08:       return new Circumference(x, y);
09:     if (figureType == RECTANGLE)
10:       return new Rectangle(x, y);
11:     throw new ArgumentException("Unknown figure type");
12:   }
13:
14:   static void Main(String[] args) {
15:     dynamic cir = new Circumference(0,0,10);            cir: X6, X6←Circumference
16:     dynamic rec = new Rectangle(0,0,10,20);             rec: X7, X7←Rectangle
17:     dynamic distCir = Distance(cir);                    distCir: X8, X8←double
18:     dynamic distRec = Distance(rec);                    distRec: X9, X9←double
19:     dynamic figure = Factory(Convert.ToInt32(args[0]),  figure: X10,
                                 10, 20);                   X10←Circumference∨Rectangle
20:     dynamic distFig = Distance(figure);                 distFig: X11, X11←double
21:     Distance(new Triangle(0,0,0,0,0,0));
22:   }
23: }
```

Fig. 4. Dynamically typed parameters.

At every invocation, the type variables of the parameters are unified with the type of the arguments, and constraints are checked. If the constraints are fulfilled, the returning type is then inferred. Otherwise, a compiler error is shown. The three invocations in lines 17, 18 and 20 fulfill the constraints of `Distance` and return `double`. However, the invocation in line 21 does not meet the constraints because `Triangle` does not implement `GetX()` and `GetY()`—a compiler error is shown.

3.4. Static and dynamic typing

One of the development scenarios of gradual typing is rapid prototyping. In such a scenario, dynamically typed variables are used to implement prototypes. If the language also supports static typing, dynamically typed variables can be gradually converted into statically typed ones to make the application more robust and efficient.

StaDyn supports the gradual modification of dynamically typed code with different compiler options. As mentioned, the language provides `var` as a new type. By default, `var` is statically typed and shows a compiler error when an operation is not supported by all the types in a flow-sensitive union type (Section 3.2). However, if the `everythingDynamic` option is passed to the compiler, all the `var` references are treated as `dynamic`, making the type system to be more lenient. That option would no longer be used when we want to convert a prototype into a safe program [28].

StaDyn also provides a Visual Studio plug-in that facilitates the conversion of rapid prototypes into safe applications [36]. Among other functionalities, it performs the automatic replacement of `var` and `dynamic` references with the types inferred by the compiler, when they are inferred to one single static type (flow-sensitive union types are not replaced) [37]. It supports this option for a single variable, one file and the whole application.

3.5. Method specialization

The *StaDyn* type inference system combines constraints, union types, unification and SSA transformations to detect compile-time errors of dynamically typed code. Moreover, the type information gathered by the compiler is used to improve the runtime performance of the generated code. One of the most significant optimizations performed by *StaDyn* is method specialization [28]. The type information inferred for the arguments in a method invocation is used to specialize the code of the method being called. `var` and `dynamic` parameters are replaced with the inferred types of the arguments, and a new implementation for the invoked method is generated. In that new version of the method, no dynamic typing is performed, so runtime performance is significantly increased (Section 4.1).

Fig. 5 shows an example of the method specialization technique (the left-hand side shows the original code, while the right-hand side depicts its specialized version). For the first invocation to `Distance` (line 7), a new `Distance_1` method is created, replacing the `dynamic` type of the parameter (line 2) with the type of the argument (`Circumference`); the same specialization is performed for the return type. Therefore, when the specialized invocation calls `Distance_1`, runtime performance will be increased because dynamic typing is avoided.

The same specialization takes place if the argument is a flow-sensitive union type (e.g., `figure` argument in line 9). In this case, a new `Distance_3` method is created. The only runtime

```
01: class Program {
02:   static dynamic Distance(dynamic figure) {
03:     return Math.Sqrt(Math.Pow(figure.GetX(),2) +
              Math.Pow(figure.GetY(), 2));
04:   }

05:   static void Main(String[] args) {
06:     dynamic cir = new Circumference(0,0,10);
07:     dynamic distCir = Distance(cir);
08:     dynamic figure = Factory(Convert.ToInt32(
                                  args[0]), 10, 20);
09:     dynamic distFigure = Distance(figure);
10:   }
```

```
class Program {
  static double Distance_1(Circumference figure) {
    return Math.Sqrt(Math.Pow(figure.GetX(), 2) +
           Math.Pow(figure.GetY(), 2));
  }
  static double Distance_2(Rectangle figure) {
    return Math.Sqrt(Math.Pow(figure.GetX(), 2) +
           Math.Pow(figure.GetY(), 2));
  }
  static double Distance_3(Object figure) {
    return figure is Circumference ?
             Distance_1((Circumference)figure) :
             Distance_2((Rectangle)figure);
  }
  static void Main(String[] args) {
    dynamic cir = new Circumference(0,0,10);
    dynamic distCir = Distance_1(cir);
    dynamic figure = Factory(Convert.ToInt32(
                                  args[0]), 10, 20);
    dynamic distFigure = Distance_3(figure);
  }
```

Fig. 5. Original *StaDyn* code (left-hand side) and its specialized version (right-hand side).

type check performed in `Distance_3` is knowing whether the parameter is `Circumference` or `Rectangle`. No other type should be checked because the type of the argument is inferred to `Circumference`∨`Rectangle`. `Distance_3` calls either `Distance_1` or `Distance_2`, where no dynamic type checking is performed.

3.6. Runtime type cache optimization

Although method specialization is performed for each invocation, the original `Distance` method with the `dynamic` parameter and return type is maintained in the generated code. This is because the compiled assembly can be used as a .NET component from another application and, in that case, it would not be specialized.

For that particular case scenario, the *StaDyn* compiler provides another optimization. For those unspecialized methods with dynamically typed parameters, a runtime type cache is used [38]. In that case, the code generated by the compiler uses the services of the Dynamic Language Runtime (DLR). The DLR implements different cache levels for the typical operations of dynamically typed code [39]. The use of the runtime cache provides significant performance benefits when methods are repeatedly called with the same argument types (a common scenario in most applications) [38].

4. Impact

StaDyn is a suitable programming language for scenarios where both dynamic and static typing are appropriate to build a .NET component or application. The main contribution of *StaDyn* is the static type checking performed by the compiler for the dynamically typed code. So far, we have described how that information is used to produce safer programs with fewer runtime errors (Section 3). We now evaluate how that type information is used to generate more efficient code (Section 4.1). Finally, we describe some scenarios where the *StaDyn* programming language has been used (Section 4.2).

4.1. Runtime performance and memory consumption

We compare the efficiency of *StaDyn* with the following hybrid typing languages for the .NET platform: C# 7.3, Visual Basic 15, Boo 0.9.7, Fantom 1.0.77 and Cobra 0.9.6. We also include the IronPython 2.7.7 implementation of Python for the .NET platform due to its good performance. All the source code has no type annotations (everything is `dynamic`). For C#, we also measure the code with all the type annotations to see how close the implemented optimizations are to fully type-annotated C#.

There exist some other dynamic languages that implement state-of-the-art optimizations to improve the runtime performance of dynamically typed code. Although they do not generate code for the .NET platform, we include them in our analysis to compare their optimizations with the ones provided by *StaDyn*. For Python, we measure PyPy 2.7, evaluated as the fastest Python implementation [40], and the CPython 2.7.14 reference implementation. The JavaScript engines V8 8.7 and SpiderMonkey 24.4 (with and without IonMonkey) are also included in the evaluation because of their excellent performance [20]. We also include the JavaScript GraalVM implementations over the high-performance GraalVM polyglot virtual machine (both native image and JVM runtimes) [41].

We measure different applications and benchmarks. First, we take the Pybench and Pystone well-known dynamically typed benchmarks. Second, we include Section 2 (kernels) and Section 3 (large-scale applications) of the statically typed Java Grande benchmark. The Points hybrid static and dynamic benchmark is also measured [31]. We translate all the programs into the different languages to be evaluated, and all the type annotations are replaced with dynamically typed references.

We follow the two methodologies proposed by Georges et al. [42]: (1) *startup* performance is how quickly a system can run a relatively short-running application; (2) *steady-state* performance concerns long-running applications, where runtime optimizations have been executed. For startup, each program is executed 30 times, computing the average result and the student's t 95% confidence interval of the total execution time. As for startup, steady-state programs are executed 30 times. However,

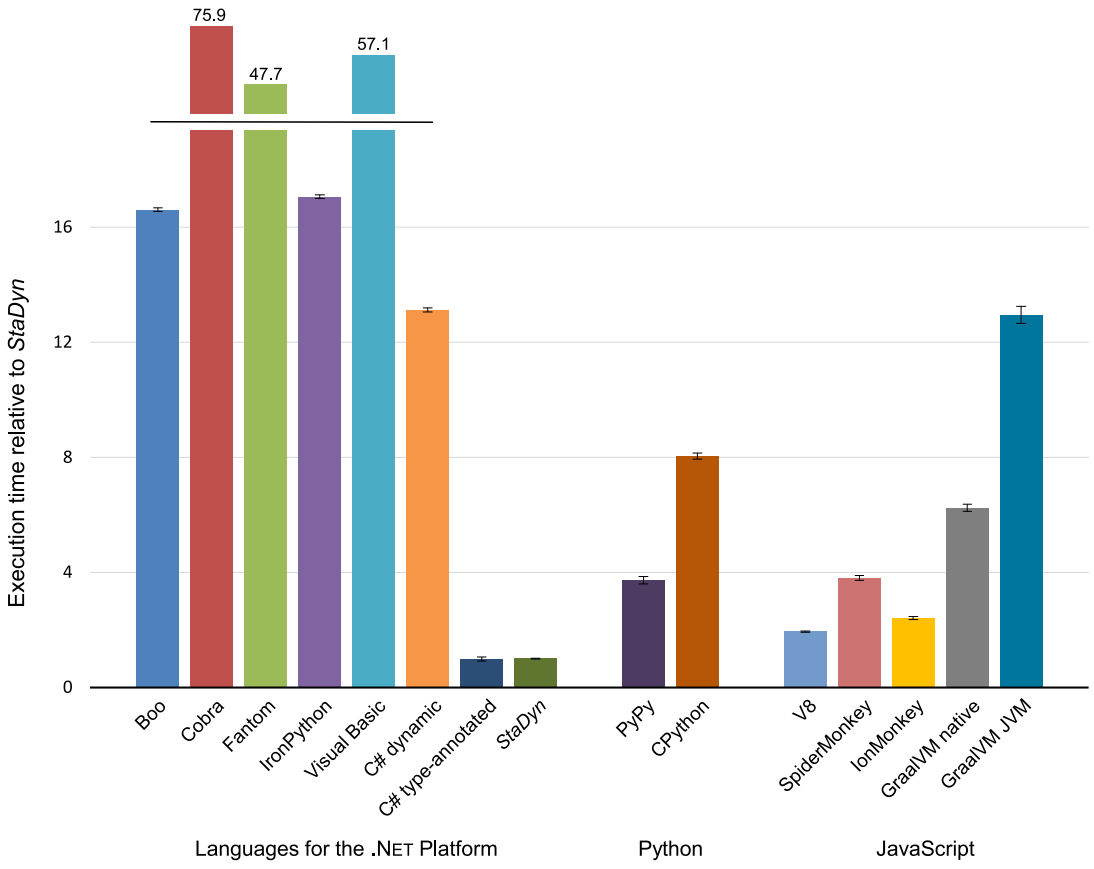

Fig. 6. Average startup execution times relative to *StaDyn* (whiskers represent 95% confidence intervals).

each program execution in steady-state runs the benchmark in a loop, and it returns the average value of the last 10 iterations when the coefficient of variation of those 10 iterations is lower than 2% (*i.e.*, a steady state is reached)—a more detailed description can be consulted in [42].

Memory consumption is evaluated with the startup methodology. For each program execution, we measure the maximum size of working set memory employed by each process ever since it started (*i.e.*, `PeakWorkingSet`). The working set of a process is the set of memory pages currently visible to a process in physical RAM memory, which is available to be used without triggering a page fault.

All the programs are measured on a 2.5 GHz Intel Core i7 system with 8 GB of RAM, running an updated 64-bit version of Windows 10 and .NET Framework 4.8.

Fig. 6 shows the startup execution times, where *StaDyn* outperforms the rest of the languages measured but C# with full type annotations. The optimizations implemented by *StaDyn* make code with all `dynamic` variables to be 1.8% slower than fully type-annotated C#, but there are no significant differences since both confidence intervals overlap [42]. When comparing the same source code – C# and *StaDyn* with all `dynamic` variables – runtime performance is 12.1 times higher than C#. This value grows up to 74.9 factors when compared to Cobra, the .NET language with the lowest runtime performance.

When compared to PyPy, a Python implementation optimized with a tracing JIT compiler [43], *StaDyn* provides a 272.8% runtime performance benefit for startup. The JavaScript engine evaluated with the best startup performance is V8, which implements runtime adaptive optimizations [44]. In the applications measured, V8 requires 93.4% more execution time than *StaDyn*.

For the steady-state methodology (Fig. 7), the dynamic optimizations implemented by the tracing JIT compilers of IonMonkey and GraalVM for JVM show important benefits compared to the startup approach. IonMonkey shows the best JavaScript performance with 127.7% more execution time than *StaDyn*. *StaDyn* keeps showing the second best steady-state performance, consuming 5.1% more execution time than C# with all the type annotations. All these data show that our optimizations make `dynamic` code in *StaDyn* to be very close to type-annotated code, and provide significantly higher runtime performance than the existing approaches to optimize dynamically typed code.

Fig. 8 shows the average runtime memory consumed by the different languages. *StaDyn* is the programming language that requires fewer memory resources at runtime. This is because the method specialization optimization implemented by *StaDyn* is performed statically by the compiler.[2] On the contrary, highly optimized implementations, such as PyPy, IonMonkey, V8 and GraalVM, perform all the optimizations dynamically, consuming additional memory resources. C# with `dynamic` and IronPython use the runtime cache of the DLR, and Fantom implements its own cache, causing significantly higher memory consumption.

As mentioned, most of the optimizations provided by *StaDyn* take place statically, at compile time. That provides lower execution times and memory consumption than the existing approaches, but requires additional compilation time. Thus, we compare the compilation time of the selected programs with different compilers. The .NET Foundation supports two C# and

[2] Fully type-annotated C# consumes 2.4% more memory than *StaDyn* because of the DLR used in the Points benchmark. *StaDyn* does not make use of the DLR, due to the method specialization technique implemented [28].

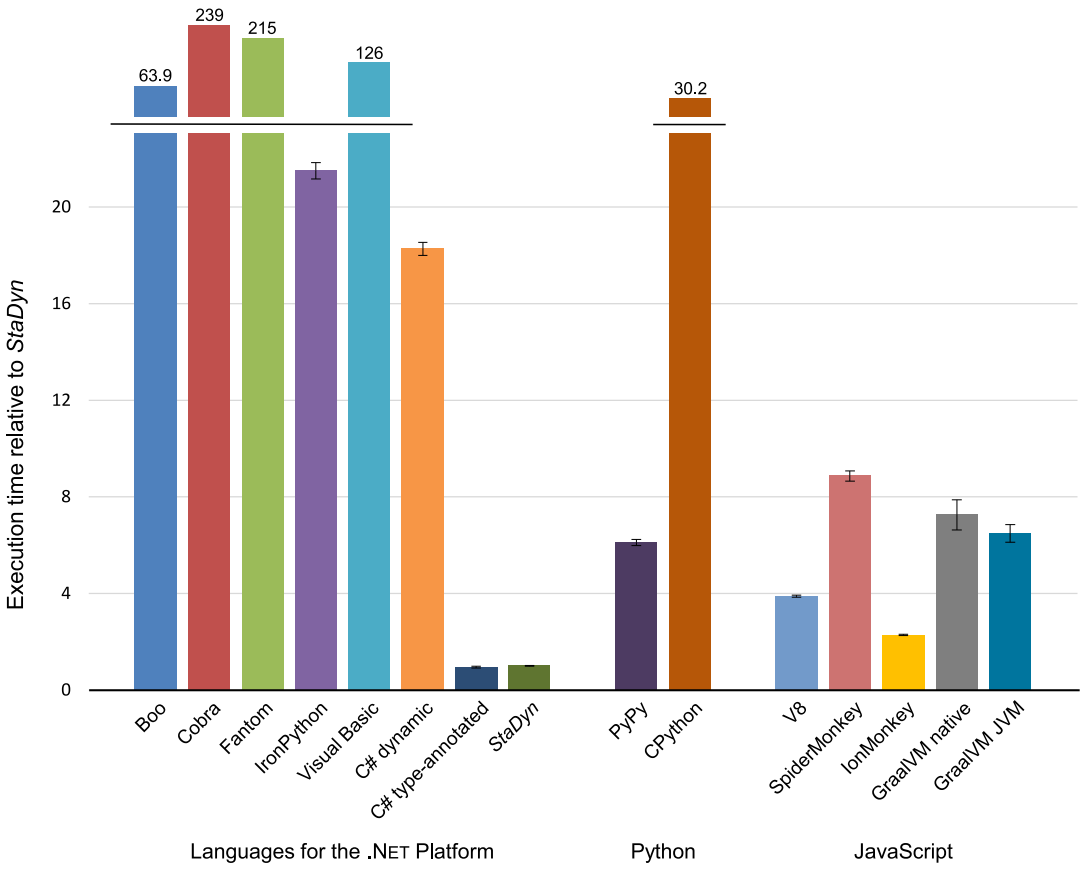

Fig. 7. Average steady-state execution times relative to *StaDyn* (whiskers represent 95% confidence intervals).

Visual Basic compilers implemented on the .NET platform (known as Roslyn compilers), similar to *StaDyn*. The average compilation times of these C# and Visual Basic compilers are, respectively, 92.9% and 101% higher than *StaDyn*. Compared to the Microsoft native implementations (i.e., binary compilers), C# and Visual Basic use 15.9% and 24.6% of the compilation time consumed by *StaDyn*. *StaDyn* compiles the selected programs 4.5%, 12.4% and 63.4% faster than, respectively, Fantom, Boo and Cobra, the rest of the hybrid typing languages for the .NET platform.

4.2. Usages of StaDyn

The *StaDyn* programming language was used in the development of the OneRate credit risk analysis system [45]. OneRate was built as a highly adaptable framework that supports the common features of credit risk analysis systems, while allowing their customization to the particular requirements of each customer. These two objectives are achieved with the combination of static and dynamic typing.

StaDyn was also used in the development of part of DIMAG, a framework for the declarative implementation of native mobile applications [46]. The code generation module of DIMAG was implemented in *StaDyn*, due to its adequacy to add new target devices using duck typing. The same approach was used to develop the Lizard native view generation system [47].

We used *StaDyn* in the implementation of statically typed multimethods for the .NET platform [9]. A multimethod dispatcher uses `dynamic` as the type of its polymorphic arguments. Those arguments are passed to statically typed overloaded methods implementing each multimethod [48]. The type system of *StaDyn* allows detecting errors statically, while obtaining high runtime performance. *StaDyn* was also used to implement part of the dynamic weaving module of the DSAW aspect weaver platform [49].

Besides software development, *StaDyn* has also been utilized in different academic scenarios. Our language is used to teach the differences between dynamic and static typing in a Programming Paradigms and Technologies course [50]. Likewise, its source code is employed to teach the different parts of a compiler in a Programming Language Design and Implementation module [51]. Finally, we use *StaDyn* to teach its different optimizations in a Programming Languages and Platforms research course of a software engineering master's degree.

5. Related work

There have been many works aimed at obtaining the advantages of static and dynamic typing in the very same programming language. Soft typing applies static typing to the Scheme dynamically typed language [12]. In soft typing, the static type checker inserts runtime type checks in dynamically typed operations that may be erroneous. Abadi et al. add a `Dynamic` type to lambda calculus, including two conversion operations: `dynamic` to construct values of `Dynamic` type and `typecase` for inspecting them, producing verbose code deeply dependent on its dynamism [52].

The works of quasi-static typing [13] and hybrid typing [14] perform implicit conversions between dynamic and static code via subtyping relations. Gradual typing is based on the consistency relation, first defined in the $\lambda^{?}_{\rightarrow}$ functional calculus [15]. Unlike subtyping, the consistent subtyping relation is not transitive. Consistency was later combined with subtyping ($\lesssim$) and included in object-oriented abstractions [11]. Gradual typing has also been integrated with ownership types [53], refinement types [54], session types [55] and type inference [56].

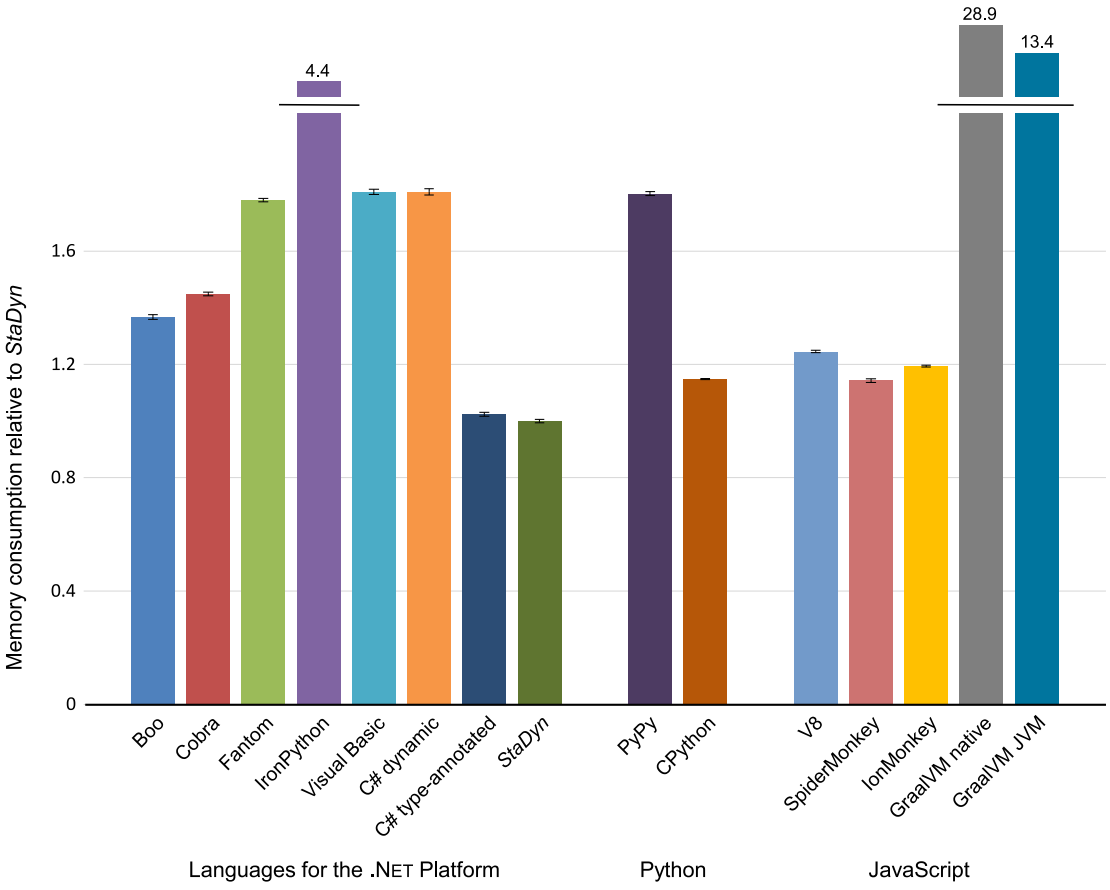

Fig. 8. Average runtime memory consumption relative to *StaDyn* (whiskers represent 95% confidence intervals).

Garcia et al. propose a new foundation for gradual typing called Abstract Gradual Typing (AGT) [57]. AGT derives consistent predicates and gradual typing judgements by using abstract interpretation to give gradual types a semantics in terms of pre-existing static types. Gradual types are interpreted as sets of possible static types. AGT yields a formal account of consistency that subsumes and generalizes the traditional notions of consistency. AGT includes a systematic approach to developing dynamic semantics for gradual programs as proof reductions over source-language typing derivations.

Inspired by the AGT work on using abstract interpretation to understand gradual typing [57], Toro and Tanter define gradual union types. Gradual union types combine the traditional static approach of untagged ($T_1 + T_2$) and tagged ($T_1 \vee T_2$) unions, viewing a union type as a kind of gradual type [58]. Thus, $T_1 \bigoplus T_2$ is a union gradual type that represents both T_1 and T_2, meaning that the use of a value type $T_1 \bigoplus T_2$ is accepted if the operations make sense for either T_1 or T_2. If such operations are valid for one of the types (T_1 or T_2), a runtime check is introduced that may cause a dynamic cast error. No runtime check is necessary if the operation is valid for both T_1 and T_2. If the operation is valid for neither T_1 nor T_2, the program is rejected statically. Since gradual union types do not allow full dynamic type checking, the unknown type ? is also included in the meta-theory of gradual union types.

Castagna and Lanvin enrich gradual type systems with union and intersection types, making the transition between dynamic and static typing smoother and finer grained [59]. Union and intersection types can be used by programmers to instruct the system to perform fewer runtime checks. They use the name of set-theoretic types because, interpreting types as sets of values, union and intersection types are interpreted as the corresponding set-theoretic operations. Likewise, the subtyping relation is defined as set containment. Castagna and Lanvin define the static and dynamic semantics of a language with gradual typing and set-theoretic type connectives, and prove its soundness.

Muehlboeck and Tate present a calculus, called MonNom, that allows mixing untyped structural code with typed nominal code in gradually typed object-oriented languages [60]. Their system allows programs to transition between untyped structural and typed nominal approaches, while still ensuring gradual guarantee [61] and soundness. MonNom was implemented with an ahead-of-time compiler using LLVM, extending their calculus with a standard library, generics and new primitives. The evaluation showed that the worst-case overhead of the implementation stays under 25%.

Sound gradually typed languages insert runtime checks to provide type soundness for the overall program. Therefore, the programmer can rely on the language implementation to provide meaningful error messages at runtime. However, these runtime type checks imply a significant performance overhead [62]. Rastogi et al. measured the runtime performance of the sound gradual type system provided by Safe TypeScript, reporting that the average performance cost for dynamic code with no type annotations was 22 factors [63].

Due to the runtime performance cost of sound gradually typed languages, there have been many works focused on their optimization. Siek et al. propose monotonic references to avoid the runtime overhead of dynamic typing in statically typed regions [64]. The work of Herman et al. aims at reducing the type checking operations by combining adjacent type coercions, providing potential optimizations in space and time [65].

Rastogi et al. design a sound type inference algorithm to improve the performance of gradually typed programs, without introducing any new runtime failures [66]. In this way, static types can often be inferred, thereby removing unnecessary runtime checks. The algorithm performs an asymmetric treatment of types that flow in and out to an unknown type, and an escape analysis is performed to decide which types are safe to infer. Their type inference algorithm was included in an implementation of ActionScript, showing an average runtime performance improvement of 60% [66].

Reticulated Python is a gradually typed variant of the Python programming language [67]. The transient strategy for gradual typing in Reticulated Python inserts lightweight constant-time checks (type tag inspections) rather than using proxies. However, such transient gradual typing still shows a runtime performance cost linear to the number of type annotations, presenting a worst-case overhead of 6 factors compared to CPython [68]. Reticulated Python was later optimized by removing unnecessary checks with a type inference algorithm that uses subtyping and check constraints generated by the transient checks [69]. The linear cost disappeared, showing a 6% average overhead compared to CPython and 1% when run on PyPy.

Pycket implements a tracing JIT compiler for the gradually typed Racket language [70]. Pycket is implemented in the RPython meta-tracing framework, originally created for PyPy. RPython automatically generates tracing JIT compilers from interpreters [71]. Among other optimizations, Pycket performs runtime type specialization of different data structures. The tracing JIT compiler provided by RPython has allowed Pycket to eliminate more than 90% of the gradual typing overhead introduced by Typed Racket [72].

6. Conclusions

The *StaDyn* programming language combines the benefits of static and dynamic typing in the very same language. Compared to the existing hybrid typing languages, its main contribution is that it statically gathers type information of dynamically typed code. Such type information is used to provide early detection of type errors of dynamically typed code and a significant runtime performance improvement. Dynamic memory consumption is also reduced due to its compile-time method-specialization optimization. Although most optimizations take place at compile time, compilation time is lower than the hybrid typing languages implemented on the .Net platform. *StaDyn* has been successfully used to implement several software applications and to teach different topics in university courses.

A virtual machine with all the benchmarks and the software used to measure the different language implementations evaluated in this article are available for download at

https://www.reflection.uniovi.es/stadyn/download/2022/softwarex

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Francisco Ortin reports financial support was provided by Ministry of Science Technology and Innovations. Francisco Ortin reports financial support was provided by Government of Principality of Asturias.

Data availability

The data used in this article is produced by the execution of the benchmarks described in the paper. The source code of all the benchmarks is available for download.

Acknowledgments

The *StaDyn* programming language was partially funded by Microsoft Research, United States with the project entitled *Extending Dynamic Features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution Request for Proposals*. It was also funded by the Spanish Department of Science, Innovation, and Universities (project RTI2018-099235-B-I00) and the University of Oviedo, Spain (GR-2011-0040). We thank Cristina González, Daniel Zapico, Francisco Moreno and Anton Morant for their contribution to the implementation of the *StaDyn* programming language.

References

[1] Meijer E, Drayton P. 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: ACM; 2004, p. 1–6.
[2] Thomas D, Fowler C, Hunt A. Programming ruby. second ed.. Addison-Wesley; 2004.
[3] Thomas D, Hansson DH, Schwarz A, Fuchs T, Breed L, Clark M. Agile web development with rails. In: A pragmatic guide. Pragmatic Bookshelf; 2005.
[4] Hunt A, Thomas D. The pragmatic programmer: from Journeyman to master. Addison-Wesley Longman Publishing Co., Inc. 1999.
[5] ECMA-262. ECMAScript 2021 language specification. European Computer Manufacturers Association; 2021.
[6] Bush E, van der Linden M. Full-stack javascript development: develop, test and deploy with MongoDB, express, angular and node on AWS. Red Hat Press; 2016.
[7] van Rossum G, Fred L, Drake J. The python language reference manual. Network Theory; 2003.
[8] Pierce BC. Types and programming languages. Cambridge, Massachusetts: The MIT Press; 2002.
[9] Ortin F, Garcia M, Redondo JM, Quiroga J. Combining static and dynamic typing to achieve multiple dispatch. Inf Int Interdiscip J 2013;16(12):8731–50.
[10] Ortin F, Conde P, Lanvin DF, Izquierdo R. Runtime performance of `invokedynamic`: an evaluation with a Java library. IEEE Softw 2014; 31(4):82–90.
[11] Siek JG, Taha W. Gradual typing for objects. In: Proceedings of the 21st European conference on object-oriented programming. Berlin, Germany: Springer-Verlag; 2007, p. 2–27.
[12] Cartwright R, Fagan M. Soft typing. In: Proceedings of the conference on programming language design and implementation. Toronto, Canada: ACM; 1991, p. 278–92.
[13] Thatte S. Quasi-static typing. In: Proceedings of the 17th symposium on principles of programming languages. San Francisco, California, United States: ACM; 1990, p. 367–81.
[14] Flanagan C, Freund SN, Tomb A. Hybrid types, invariants, and refinements for imperative objects. In: Proceedings of the international workshop on foundations and developments of object-oriented languages. San Antonio, Texas: ACM; 2006, p. 1–11.
[15] Siek JG, Taha W. Gradual typing for functional languages. In: Scheme and functional programming workshop. 2006, p. 1–12.
[16] Strachan J. Groovy 2.0 release notes. 2022, http://groovy-lang.org/releasenotes/groovy-2.0.html.
[17] Zandstra M. PHP objects, patterns, and practice. fifth ed.. A Press; 2016, p. 488.
[18] Bierman G, Meijer E, Torgersen M. Adding dynamic types to C#. In: Proceedings of the 24th European conference on object-oriented programming. ECOOP, Maribor, Slovenia: Springer-Verlag; 2010, p. 76–100.
[19] JetBrains. Kotlin dynamic type. 2022, https://kotlinlang.org/docs/dynamic-type.html.
[20] Garcia M, Ortin F, Quiroga J. Design and implementation of an efficient hybrid dynamic and static typing language. Softw - Pract Exp 2015;46(2):199–226.
[21] Ortin F, Redondo JM, Garcia Perez-Schofield JB. Efficient virtual machine support of runtime structural reflection. Sci Comput Program 2009;70(10):836–60.
[22] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 1991;13(4):451–90.
[23] Quiroga J, Ortin F. SSA transformations to facilitate type inference in dynamically typed code. Comput J 2017;60(9):1300–15.
[24] Pierce BC. Programming with intersection types and bounded polymorphism. Tech. Rep. CMU-CS-91-106, Pittsburgh, PA, USA: School of Computer Science; 1992.

[25] Barbanera F, Dezani-Ciancaglini M, De'Liguoro U. Intersection and union types: syntax and semantics. Inform and Comput 1995;119:202–30.
[26] Aiken A, Wimmers EL. Type inclusion constraints and type inference. In: Proceedings of the conference on functional programming languages and computer architecture. Copenhagen, Denmark: ACM; 1993, p. 31–41.
[27] Chugh R, Rondon PM, Jhala R. Nested refinements: a logic for duck typing. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: ACM; 2012, p. 231–44.
[28] Ortin F, Garcia M, McSweeney S. Rule-based program specialization to optimize gradually typed code. Knowl-Based Syst 2019;179.
[29] Hüinette F, Hedbor P. The pike programming language. 2022, http://pike.lysator.liu.se.
[30] Ortin F, Garcia M. Union and intersection types to support both dynamic and static typing. Inform Process Lett 2011;111(6):278–86.
[31] Ortin F. Type inference to optimize a hybrid statically and dynamically typed language. Comput J 2011;54(11):1901–24.
[32] Ortin F, Zapico D, Perez-Schofield JBG, Garcia M. Including both static and dynamic typing in the same programming language. IET Softw 2010;4(4).
[33] Robinson JA. Computational logic: the unification computation. Mach Intell 1971;6:63–72.
[34] Milner R. A theory of type polymorphism in programming. J Comput System Sci 1978;17:348–75.
[35] Odersky M, Sulzmann M, Wehr M. Type inference with constrained types. Theory Pract Object Syst 1999;5:35–55.
[36] Ortin F, Moreno F, Morant A. Static type information to improve the IDE features of hybrid dynamically and statically typed languages. J Vis Lang Comput 2014;25(4).
[37] Ortin F, Morant A. IDE support to facilitate the transition from rapid prototyping to robust software production. In: Proceedings - international conference on software engineering. 2011, p. 40–3.
[38] Quiroga J, Ortin F, Llewellyn-Jones D, Garcia M. Optimizing runtime performance of hybrid dynamically and statically typed languages for the .Net platform. J Syst Softw 2016;113.
[39] Chiles B, Turner A. Dynamic language runtime. 2022, https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview.
[40] Redondo JM, Ortin F. A comprehensive evaluation of widespread python implementations. IEEE Softw 2015;34(4):76–84.
[41] Šipek M, Mihaljević B, Radovan A. Exploring aspects of polyglot high-performance virtual machine graalvm. In: 2019 42nd International convention on information and communication technology, electronics and microelectronics. IEEE; 2019, p. 1671–6.
[42] Georges A, Buytaert D, Eeckhout L. Statistically rigorous java performance evaluation. In: Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems and applications. OOPSLA, New York, NY, USA: ACM; 2007, p. 57–76.
[43] Rigo A, Fijalkowski M, Bolz CF, Cuni A, Peterson B, Gaynor A. PyPy, a fast, compliant alternative implementation of the Python language. 2022, http://pypy.org.
[44] Google Inc. V8, the Google's high performance, open source, JavaScript engine. 2022, https://developers.google.com/v8.
[45] Redondo JM, Ortin F. A saas framework for credit risk analysis services. IEEE Lat Am Trans 2017;15(3).
[46] Miravet P, Marin I, Ortin F, Rodriguez J. Framework for the declarative implementation of native mobile applications. IET Softw 2014;8(1):19–32.
[47] Marin I, Ortin F, Pedrosa G, Rodriguez J. Generating native user interfaces for multiple devices by means of model transformation. Front Inform Technol Electron Eng 2015;16(12).
[48] Ortin F, Quiroga J, Redondo JM, Garcia M. Attaining multiple dispatch in widespread object-oriented languages. Dyna 2014;182(186):242–50.
[49] Felix J, Ortin F. Efficient aspect weaver for the .NET platform. IEEE Lat Am Trans 2015;13(5).
[50] Ortin F, Redondo J, Quiroga J. Design and evaluation of an alternative programming paradigms course. Telemat Inform 2017;34(6).
[51] Ortin F, Quiroga J, Rodriguez-Prieto O, Garcia M. An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. PLoS One 2022;17(3:1932-6203).
[52] Abadi M, Cardelli L, Pierce BC, Rémy D. Dynamic typing in polymorphic languages. J Funct Programming 1995;5(1):111–30.
[53] Sergey I, Clarke D. Gradual ownership types. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics). 7211 LNCS, 2012, p. 579–99.
[54] Jafery KA, Dunfield J. Sums of uncertainty: refinements go gradual. In: Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages. POPL, (1):2017, p. 804–17.
[55] Igarashi A, Thiemann P, Vasconcelos V, Wadler P. Gradual session types. In: Proceedings of the ACM international conference on functional programming. Vol. 1. Oxford, United Kingdom; 2017, p. 1–38.
[56] Garcia R, Cimini M. Principal type schemes for gradual programs. In: Proceedings of the 42nd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: ACM; 2015, p. 303–15.
[57] Garcia R, Clark AM, Tanter É. Abstracting gradual typing. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: Association for Computing Machinery; 2016, p. 429–42.
[58] Toro M, Tanter É. A gradual interpretation of union types. In: Ranzato F, editor. Static analysis. Cham: Springer International Publishing; 2017, p. 382–404.
[59] Castagna G, Lanvin V. Gradual typing with union and intersection types. In: Proceedings of the ACM on programming languages. Vol. 1. (ICFP). New York, NY, USA: Association for Computing Machinery; 2017.
[60] Muehlboeck F, Tate R. Transitioning from structural to nominal code with efficient gradual typing. In: Proceedings of the ACM on programming languages. Vol. 5. (OOPSLA). New York, NY, USA: Association for Computing Machinery; 2021.
[61] Siek JG, Vitousek MM, Cimini M, Boyland JT. Refined criteria for gradual typing. In: Ball T, Bodik R, Krishnamurthi S, Lerner BS, Morrisett G, editors. 1st Summit on advances in programming languages. Leibniz international proceedings in informatics (LIPIcs), vol. 32, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2015, p. 274–93.
[62] Takikawa A, Feltey D, Greenman B, New MS, Vitek J, Felleisen M. Is sound gradual typing dead? In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL 2016, New York, NY, USA: ACM; 2016, p. 456–68.
[63] Rastogi A, Swamy N, Fournet C, Bierman G, Vekris P. Safe & efficient gradual typing for type script. In: Conference record of the annual ACM symposium on principles of programming languages. 2015-Janua. 2015, p. 167–80.
[64] Siek JG, Vitousek MM, Cimini M, Tobin-Hochstadt S, Garcia R. Monotonic references for efficient gradual typing. In: Vitek J, editor. Programming languages and systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2015, p. 432–56.
[65] Herman D, Tomb A, Flanagan C. Space-efficient gradual typing. In: Higher-order and symbolic computation. Vol. 23. (2):2010, p. 167–89.
[66] Rastogi A, Chaudhuri A, Hosmer B. The ins and outs of gradual type inference. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. New York, NY, USA: Association for Computing Machinery; 2012, p. 481–94.
[67] Vitousek MM, Kent AM, Siek JG, Baker J. Design and evaluation of gradual typing for python. In: Proceedings of the 10th ACM symposium on dynamic languages. New York, NY, USA: Association for Computing Machinery; 2014, p. 45–56.
[68] Vitousek MM, Swords C, Siek JG. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In: Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages. New York, NY, USA: Association for Computing Machinery; 2017, p. 762–74.
[69] Vitousek MM, Siek JG, Chaudhuri A. Optimizing and evaluating transient gradual typing. In: Proceedings of the 15th ACM SIGPLAN international symposium on dynamic languages. DLS 2019, New York, NY, USA: Association for Computing Machinery; 2019, p. 28–41.
[70] Bauman S, Bolz CF, Hirschfeld R, Kirilichev V, Pape T, Siek JG, et al. Pycket: A tracing JIT for a functional language. In: Proceedings of the 20th ACM SIGPLAN international conference on functional programming. ICFP 2015, New York, NY, USA: Association for Computing Machinery; 2015, p. 22–34.
[71] Ancona D, Ancona M, Cuni A, Matsakis ND. RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the 2007 symposium on dynamic languages. 2007, p. 53–64.
[72] Bauman S, Friedrich Bolz-Tereick C, Siek J, Tobin S. Sound gradual typing: Only mostly dead. (OOPSLA):2017, p. 1–24.