Union and Intersection Types to Support both Dynamic and Static Typing

Francisco Ortin, Miguel García

Computer Science Department, C/Calvo Sotelo s/n, 33007, Oviedo, Spain

Abstract

Although static typing provides undeniable benefits for the development of applications, dynamically typed languages have become increasingly popular for specific scenarios. Since each approach offers different benefits, the *StaDyn* programming language has been designed to support both dynamic and static typing. This paper describes the minimal core of the *StaDyn* programming language. Its type system performs type reconstruction over both dynamic and static implicitly typed references. A new interpretation of union and intersection types allows statically gathering the type information of dynamic references, which improves runtime performance and robustness. The evaluation of the generated code has shown how our approach offers an important runtime performance benefit.

Key words: programming languages, program specification, compilers, formal languages, type systems

1. Introduction

Static typing is an unquestionably important tool for software development, offering the programmer advantages such as early type error detection, better documentation and abstraction, and more opportunities for compiler optimizations. Nevertheless, dynamically typed languages provide great flexibility at runtime, making them ideally suited for prototyping systems with changing or unknown requirements, or which interact with other systems that change unpredictably (data and application integration) [13].

Email address: ortin@lsi.uniovi.es (Francisco Ortin)
URL: http://www.di.uniovi.es/~ortin (Francisco Ortin)

Since both dynamic and static typing offer important benefits, there have been approaches aimed at obtaining the advantages of both, following the philosophy of *static typing where possible, dynamic typing when needed* [13]. One of the first approaches was *Soft Typing* [6], that applied static typing to a dynamically typed language such as Scheme. However, soft typing does not control which parts in a program are statically checked, neither is static type information used to optimize the generated code. The approach proposed in [1] adds a `Dynamic` type to lambda calculus, including two conversion operations (`dynamic` and `typecase`), generating a verbose code deeply dependent on its dynamism. The works of *Quasi-Static Typing* [21], *Hybrid Typing* [9] and *Gradual Typing* [20] perform implicit conversions between dynamic and static code, employing subtyping relations in the case of quasi-static and hybrid typing, and a *consistency* relation in gradual typing. The main difference between these approaches and the work presented in this paper is that we perform type-checking even when dynamic types are used, detecting some type errors in dynamic code and, hence, improving its robustness.

Theoretical works on combining static with dynamic typing have been partially included in the implementation of programming languages such as Boo, Visual Basic (VB) .Net, Cobra, Dylan, Strongtalk, and the recently released C# 4.0. Some programming languages have taken the approach of adding a new dynamic type as proposed in [1] (`dynamic` in C# and Cobra, and `duck` in Boo), whereas others represent dynamic types by removing type annotations in variable declarations (VB and Dylan) [20]. Strongtalk follows a completely different approach based on the concept of *pluggable* type systems [5]. In these languages, dynamic types are implicitly coerced to static ones following the approach defined in [21] and [20], opposite to the explicit use of a conversion instruction like the `typecase` statement proposed by [1]. Since these implicit coercions may fail at runtime, a dynamic type-check is inserted in the generated code as described in [9].

The key contributions of this paper are:

- A new programming language (*StaDyn* [15]), which supports both dynamic and static typing. This paper describes a minimal core of *StaDyn*, focusing on the main inference rules of its type system and a runtime performance evaluation of the code generated. Its abstract syntax, type system and erasure semantics by translating it into to C# are detailed in [17].

- A type system that defines a novel subtyping relation for dynamic union

and intersection types (Section 3) in order to obtain static type information of dynamic references. The type system is flow-sensitive [10] and interprets static information along the control flow path, merging the type information of the incoming branches with union and intersection types [19]. This information is used to improve both the efficiency and the robustness of programs written in this language.

- A runtime performance assessment (Section 4) of a type erasure translation to C# that makes use of the static type information gathered by the compiler. We compare runtime performance of the *StaDyn* core with that of the C# 4.0 and VB 10 programming languages. For this evaluation, we have used two dynamically typed benchmarks, a hybrid statically and dynamically typed program, and a synthetic micro-benchmark that measures the relationship between execution time and type information inferred by the compiler.

2. A Hybrid Static and Dynamic Typing Example

The benefits offered by dynamically-typed programming languages have caused the addition of dynamic typing to some statically typed languages. A clear example of this trend is the newly added `dynamic` type to the C# 4.0 programming language [23]. This new type instructs the compiler to postpone every static type checking operation until runtime. With this new characteristic, it is possible to develop more flexible code, even in the presence of the advanced C# static type system. It is also possible to directly access dynamically typed programs written in IronPython, IronRuby and the JavaScript code used in Silverlight, exploiting *Dynamic Language Runtime* (DLR) services [11].

Figure 1 shows an example use of the `dynamic` type included in C# 4.0. The first benefit is *duck typing*. Duck typing [22] is a property offered by most dynamically typed 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. Therefore, it is possible to create a function that receives any figure object (`Circle`, `Square`, `Sphere` or `Cone`) and access to its `x` field, without creating a common superclass (or interface) for all the figures; the only requirement is that the object must expose a public `x` field. In line 33 (Figure 1) the `x` field of the elements in the `v` array are accessed regardless of

their type (they are declared as `dynamic`). This means that the parameter of the `sortXCoordinate` method must be an array of any object that implements an `x` field. These objects do not need to belong to a specific hierarchy defining the shared `x` message, and they do not have to be instances of the same type either (duck typing). An example of this flexibility is shown in Figure 1, where the `shapes` array passed to the `sortXCoordinate` (line 27) holds instances of four unrelated classes.

```
01: using System;
02: class Circle {
03:   public int x, y;
04:   public int radius, dimensions = 2;
05: }
06: class Square {
07:   public int x, y;
08:   public int side, dimensions = 2;
09: }
10: class Sphere {
11:   public int x, y, z;
12:   public int radius, dimensions = 3;
13: }
14: class Cone {
15:   public int x, y, z;
16:   public int radius, height,
                dimensions = 3;
17: }
18: class Shapes {
19:   static void Main() {
20:     dynamic[] shapes = new dynamic[] {
21:       new Circle{ x=2, y=-2, radius=5 },
22:       new Square{ x=-1, y=1, side=3 },
23:       new Sphere{ x=5, y=-6, z=2,
                      radius=7 },
24:       new Cone { x=0, y=1, z=1,
                     radius=2, height=5 }
25:     };
26:     //shapes[0] = 3; // No comp. error
27:     dynamic[]sorted = sortXCoordinate(
                                    shapes);
28:     var shape=nearestToOrigin3D(shapes);
29:   }
30:   static dynamic[] sortXCoordinate(
                                dynamic[] v) {
31:     for (int i = 0; i < v.Length - 1; i++)
32:       for (int j = v.Length - 1; j > i; j--)
33:         if (v[j-1].x > v[j].x) {
34:           var temp = v[j-1];
35:           v[j-1] = v[j];
36:           v[j] = temp;
37:         }
38:     return v;
39:   }
40:   static dynamic nearestToOrigin3D(
                                dynamic[] v) {
41:     //int center=v[0].center;//No comp.error
42:     double minDistance = Double.MaxValue;
43:     int indexOfMin = -1;
44:     for (int i = 0; i < v.Length; i++)
45:       if (v[i].dimensions == 3) {
46:         double distance =
47:                Math.Sqrt(Math.Pow(v[i].x,2)+
48:                Math.Pow(v[i].y, 2) +
49:                Math.Pow(v[i].z, 2));
50:         if (distance < minDistance) {
51:           minDistance = distance;
52:           indexOfMin = i;
53:         }
54:       }
55:     if (indexOfMin != -1)
56:       return v[indexOfMin];
57:     return null;
58:   }
59: }
```

Figure 1: Sample C# 4.0 code that makes use of dynamic typing.

Dynamic typing is also used in the `nearestToOrigin3D` method. In this case, the parameter should be a type of any object that implements a `dimensions` field comparable with an integer. Moreover, those objects whose `dimensions` field value is 3 must implement the `x`, `y` and `z` fields, and they must be subtypes of `double` (they are passed as parameters to the `Pow` method). The returned object is the one that fulfills these conditions, being that nearest to the origin of coordinates. This example shows how the type system considers dynamic conditions.

```
01: var[] sortXCoordinate(var[] v,
                          int size) {
02:   int i, j;
03:   var temp;
04:   i = 0;
05:   while (i<size-1) {
06:     j = size-1;
07:     while (j>i) {
08:       if (v[j-1].x > v[j].x) {
09:         temp = v[j-1];
10:         v[j-1] = v[j];
11:         v[j] = temp;
12:       }
13:       j = j-1;
14:     }
15:     i = i+1;
16:   }
17:   return v;
18:}
19: var nearestToOrigin3D(
          /*dyn*/ var[] v, int size){
20:   int i, minDistance, indexOfMin,
        distance;
21:   var result;
22:   //i = v[0].center; // Comp error
23:   minDistance = 2147483647;
24:   indexOfMin = 0-1;
25:   i = 0;
26:   while (i<size) {
27:     if (v[i].dimensions == 3){
28:       distance = v[i].x*v[i].x +
                   v[i].y*v[i].y +
                   v[i].z*v[i].z;
29:       if (distance<minDistance) {
30:         minDistance = distance;
31:         indexOfMin = i;
32:       }
33:     }
34:     i = i+1;
35:   }
36:   if (indexOfMin != 0-1)
37:     result = v[indexOfMin];
38:   else
39:     result = 0;
40:   return result;
41: }
42: void main() {
43:   var[] shapes, sorted;
44:   var shape;
45:   shapes = new var[4];
46:   shapes[0] = new {x=2, y=0-2,
47:        radius=5, dimensions=2};
48:   shapes[1] = new {x=0-1, y=1,
          side=3, dimensions=2};
49:   shapes[2] = new {x=5, y=0-6, z=2,
          radius=7, dimensions=3};
50:   shapes[3] = new { x=0, y=1, z=1,
          radius=2, height=5,
          dimensions=3 };
51:   //shapes[0] = 3; // Comp. Error
52:   sorted = sortXCoordinate(shapes, 4);
53:   shape = nearestToOrigin3D(shapes, 4);
54: }
```

Figure 2: Example coded in the minimal core of *StaDyn*.

3. The *StaDyn* core Language

StaDyn is an object-oriented programming language based on C# 3.0 that supports both dynamic and static typing. Although the current implementation of *StaDyn* offers most of the features of C# [15], its minimal core is focused on formalizing how to include dynamic and static typing in the same programming language. For that purpose, only its minimal core features are specified here: 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 [14]. In the *StaDyn* core, `var` references can be set as static (by default) or dynamic, modifying how type-checking is performed.

A formal specification of the *StaDyn* core programming language is presented in [17]: its abstract syntax; the hybrid static and dynamic type system;

and, based on the semantics of C#, the erasure semantics of the minimal core of *StaDyn*, depicting the translation templates used to generate .NET code optimized by means of the static type information gathered by the compiler.

Figure 2 shows the *StaDyn* core translation of the C# program in Figure 1. In the *StaDyn* core, the dynamism of `var` references is explicitly stated with the `dyn` reserved word. The major benefit of using the *StaDyn* core is that static type checking is performed even over dynamic references. For instance, the `sortXCoordinate` function statically checks that each object in the `v` array provides a public `x` field. Unlike C#, the *StaDyn* core prompts a compilation error in line 52 (function invocation in Figure 2), if code in line 51 is commented out. The error indicates that one of the elements in the array (the integer) does not provide the `x` message. In contrast, C# 4.0 compiles the code and the error is produced at runtime (line 26 in Figure 1).

3.1. Union Types

The *StaDyn* core gathers type information at compile time in order to perform static type checking over dynamic references. One of the elements we have used for this purpose is union types [19]. 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 [4], representing the least upper bound of T_1 and T_2 [2]. 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: S-SUnionL in Figure 3—subtyping rules are specified with the general judgment $\Gamma \vdash T_1 \leq T_2 \mid C; \Gamma'$, meaning that under constraints C, and environment Γ, the type T_1 is a subtype of T_2, producing the output environment Γ'.

Union types were already included in object-oriented languages, in type systems where they were explicitly declared [12] or inferred from implicitly typed references [3]. We have taken the subtyping rules defined by other authors such as [19] and [8] (S-SUnionL and S-SUnionR), adding the new dynamic typing rule S-DUnionL. If a union type is 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). But 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)—$\cup\Gamma_i$ and $\cup C_i$ represent the union of all the Γ_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. With this new interpretation of union types, only static union types $\texttt{sta}\ T_1 \vee T_2$ are the least upper bound of T_1 and T_2, not the dynamic ones, $\texttt{dyn}\ T_1 \vee T_2$, because $\texttt{dyn}\ T_1 \vee T_2 \not\leq T_1$ and $\texttt{dyn}\ T_1 \vee T_2 \not\leq T_2$.

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

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

Figure 3: Subtyping relation for union types.

In our example, the type inferred for the `shapes` array in line 52 (Figure 2) is *Array*(`Circle`∨`Square`∨`Sphere`∨`Cone`)[1]. In the invocation of the `sortXCoordinate` function (line 52), it is statically checked that the argument is a subtype of an array of objects each of which provides an `x` field. Since this condition is statically fulfilled, the program is compiled by the *StaDyn* core compiler without errors (and the static type information is used to optimize its execution). However, if we uncomment line 51, an error message will be shown.

3.2. Intersection Types

The `nearestToOrigin3D` function imposes more constraints on the `v` parameter. Objects in the array must provide the `dimensions`, `x`, `y` and `z` fields. We represent these constraints by means of intersection types [19]. $T_1 \wedge T_2$ denotes all the values belonging to both T_1 and T_2 [4], representing the greatest lower bound of T_1 and T_2 [2]. 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). Just as with union types, we have added a new subtyping rule for dynamic types to be more lenient, accepting the promotion when a type promotes to at least one of the types in the dynamic intersection type (rule S-DInterR). This prevents the dynamic intersection type $\texttt{dyn}\ T_1 \wedge T_2$ from being the greatest lower bound of T_1 and T_2.

[1] Although the *StaDyn* core does not name the object types, we use the class identifiers for the sake of legibility.

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

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

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

Figure 4: Subtyping relation for intersection types.

$$\text{(S-OMember)}\quad \frac{\forall\, i \in [1,m],\ \exists\, j \in [1,n],\ id'_i = id_j,\ \Gamma_{i-1} \vdash T'_i \equiv T_j \mid C_i; \Gamma_i}{\Gamma_0 \vdash \{id_1 : T_1 \ldots id_n : T_n\} \leq [id'_1 : T'_1 \ldots id'_m : T'_m] \mid C_1 \cup \ldots \cup C_m; \Gamma_m}$$

Figure 5: Subtyping relation between object and member types.

In the *StaDyn* core type system, X_i meta-variables range over type variables. The parser assigns them a unique sequential number when a (`dyn`) `var` type (its concrete syntax) is parsed. Object types are specified describing a collection of their fields between curly braces, not including methods as part of them. Therefore, the circle object created in line 46 of Figure 2 has the object type $\{$`x`:`int`, `y`:`int`, `radius`:`int`, `dimensions`:`int`$\}$. Member types (`[`$(id{:}T)^*$`]`) represent the collection of fields an object may hold. We have introduced member types in constraints to define structural width coercion of object types to member types, because objects in *StaDyn* do not define width subtyping. Therefore, as shown in the S-OMember subtyping rule in Figure 5, an object promotes to a member type when the object has all the fields in the member types, and their types are equivalent among themselves.

In the example code in Figure 2, the constraints of the `nearestToOrigin3D` function require its `v` argument to be a subtype of $Array(X_1)$, being $X_1 \leq$ [`dimensions`:X_2] $\wedge$ [`x`:X_3] $\wedge$ [`y`:X_4] $\wedge$ [`z`:X_5] (an object with all these four fields). Therefore, the invocation in line 53 produces a compilation error because the type of `shapes` is $Array($`Circle` $\vee$ `Square` $\vee$ `Sphere` $\vee$ `Cone`$)$. Since the four types in the union type should promote to the intersection type (S-SUnionL), and neither `Circle` nor `Square` (S-SInterR) promote to [`z`:X_5] (S-OMember), the program is rejected by the compiler.

StaDyn offers a more lenient type system without renouncing static type checking. The `v` parameter of the `nearestToOrigin3D` function can be de-

clared as dynamic (uncommenting the `dyn` type qualification in line 19, Figure 2). 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). Then, the program would generate no error because the four types offer a public `dimensions` field—the output environment ($\cup\Gamma_i$) and the constraint set ($\cup C_i$) generated by applying the S-DInterR rule are only the ones generated by the coercion to [`dimensions`:X_2]. This relaxation of the subtyping relation when references are declared as dynamic is also applied to union types (S-DUnionL in Figure 3): 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 time, even when the programmer uses dynamic references. As an example, if line 22 in Figure 2 is commented out, an error is shown even though `v` has been declared as dynamic (the `center` message is not accepted by any of the four possible types); whereas C# compiles the code, producing the type error at runtime (line 41 in Figure 1). This example shows the objective of *StaDyn*: to offer both the flexibility of dynamic typing and the robustness and efficiency of static typing.

4. Runtime Performance

We have evaluated the runtime performance of the *StaDyn* core presented in this paper. An assessment of the whole *StaDyn* implementation can be consulted in [18].

4.1. Methodology

We have compared the *StaDyn* core runtime performance with probably the two most widely used programming languages over the .Net platform, compiled with their maximum optimization options:

1. **C# 4.0**. The C# programming language version 4.0 combines static and dynamic typing [23]. When dynamic code is used, the recently released *Dynamic Language Runtime* (DLR) is used to optimize the execution of dynamic code [7]. The DLR is now part of the .Net framework 4.0.
   The translation of programs from the *StaDyn* core to C# 4.0 has been accomplished by coding functions as `static` methods, translating every (`dyn`) `var` reference into a `dynamic` one, and assigning expressions

(excluding function invocation and assignment) to temporary object references.

2. **Visual Basic 10**. The VB 10 programming language also supports both dynamic and static typing [24]. A dynamic reference is declared with the `Dim` reserved word, without setting a type. With this syntax, the compiler does not infer any type information statically, performing type checking at runtime. The main difference between VB 10 and C# 4.0 is that the former uses the *Common Language Runtime* (CLR), whereas the latter employs the DLR. Translation from the *StaDyn* core to VB has been done the same way as to C# 4.0, but using the VB syntax.
3. ***StaDyn* core**. Programs coded in the *StaDyn* core programming language presented in this paper.

We have not included other dynamic programming languages such as Python or Ruby to avoid the introduction of a bias in the translation of source code (translation from C# to VB is almost direct). Both C# and VB 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.

We have divided the programs we have used to make the comparison into three different groups:

1. **Micro-benchmark**. We have coded a synthetic micro-benchmark to evaluate the influence of static type information gathered by the compiler, taking 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 type reference declaration, when the compiler is able to infer types. Although `dyn var` references are used, the *StaDyn* core compiler infers their possible types statically. Different types are inferred as a single union type. The number of possible types in the union type produces different runtime performance. In this micro-benchmark we have considered this, writing programs where 1, 5, 10 or 50 different possible types are statically inferred.

- Implicit dynamic type reference declaration, when the compiler does not infer any type.

For each scenario, we perform three different operations: accessing a field of an object, accessing an element of an array, and performing an arithmetical operation over two variables. These three operations are performed in a loop of 5 million iterations.

2. **Hybrid static and dynamic typing code**. To evaluate hybrid statically and dynamically typed code, we have extended the *StaDyn* core program in Figure 2, filling the `shapes` array with 1,000 random figures (`Circle`, `Square`, `Sphere` or `Cone`). The two `sortXCoordinate` and `nearestToOrigin3D` functions are called passing the `shapes` array as an argument.
3. **Existing benchmarks** for dynamically typed languages to obtain an estimate of possible benefits over dynamically typed languages. For this scenario we have taken two well-known benchmarks for the Python programming language: Pystone (a translation of the Dhrystone benchmark) and Pybench (a collection of tests that provides a standardized way to measure the performance of Python implementations). From the second one we have selected those tests that could be translated into the *StaDyn* core (arithmetic, calls, constructs, instances, lists, lookups, new instances and numbers). Python code was first translated into the *StaDyn* core; afterwards, the *StaDyn* core code was translated into both C# 4.0 and VB following the method described above.

Since the *StaDyn* core type system does not support method overriding, all the tests in the selected benchmarks make no use of dynamic binding in order to not bias the runtime performance measurements.

The code has been instrumented with hooks to evaluate runtime performance, recording the value of the processor's time stamp counter. We have measured the difference between the value between the beginning and the end of each benchmark to obtain the total execution time of each program.

All the programs have been executed over the .NET framework 4.0 on a lightly loaded E6750 2.67 GHz Core 2 Duo system with 2 GB of RAM running Windows 7 Professional, build 6.1.7600. Every test has been compiled without debugging information and with full optimization. To evaluate average percentages, ratios and orders of magnitude, we have used the geometric mean.

Benchmark		Test	*StaDyn* core	C#	VB
Micro-benchmark		Explicit Typing	15.63	15.63	15.63
		One possible type	15.63	2,906.25	28,953.13
		Five possible types	406.25	2,937.50	29,640.63
		Ten possible types	484.38	2,984.38	29,484.38
		Fifty possible type	921.88	3,156.25	29,937.50
		No type information	1,671.88	3,175.65	30,328.13
Hybrid		Shapes	843.75	2,031.25	8,265.63
Dynamically Typed	Pybench	Arithmetic	31.25	2,109.38	671.88
		Calls	203.13	2,765.63	2,796.88
		Constructs	31.25	3,343.75	3,250.00
		Instances	296.88	2,421.88	1,109.38
		Lists	812.50	20,765.63	76,109.38
		Lookups	93.75	2,453.13	52,062.50
		NewInstances	31.25	1,796.88	9,421.88
		Numbers	31.25	1,250.00	78.13
	Pystone		281.25	2,937.50	9,218.75

Table 1: Execution time expressed in milliseconds.

4.2. Assessment

Table 1 shows the results expressed in milliseconds. The first six rows show the results of the micro-benchmark; following this, the hybrid static and dynamic typing Shapes example. Finally, the dynamic typing benchmarks: Pybench (8 rows) and Pystone (last row).

Beginning with the micro-benchmark, the test with explicit type declaration reveals that the three implementations offer exactly the same runtime performance (the IL code generated is almost the same). The performance assessment when the exact single type of every `dyn var` reference is inferred shows the repercussion of our approach. Runtime performance of the *StaDyn* core is the same as using explicitly typed references (in fact, the code generated is precisely the same). In this special scenario, the *StaDyn* core shows a huge performance improvement. If the compiler infers the exact type of `dyn var` references, the *StaDyn* core is more than 1,252 VB and, in the same situation, 185 times faster than C# 4.0. This vast difference is caused by the lack of static type inferencing in both VB and C# 4.0. These two languages perform every type-checking operation over dynamic references at runtime, using reflection. The use of reflective operations in the .NET platform has an important performance cost [16]. The difference between C# and VB shows the performance benefit of using the DLR in this scenario.

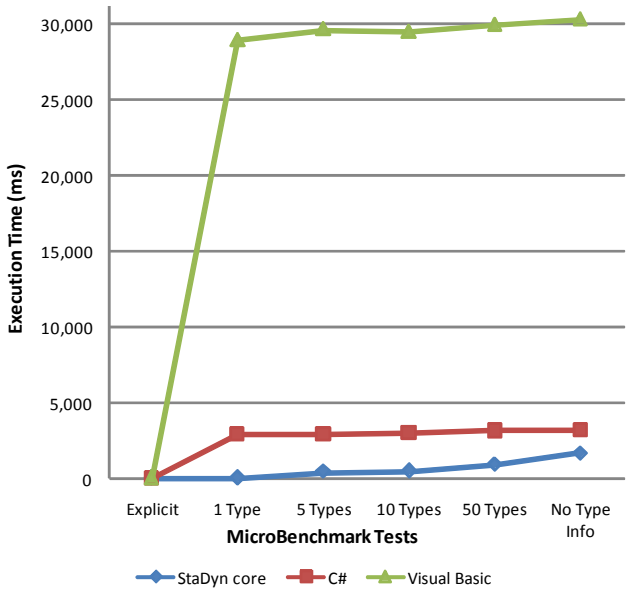

Figure 6: Execution time of the micro-benchmark.

Figure 6 shows the progression of execution time when the compiler infers 1, 5, 10 or 50 possible types. The last value is when no type information is gathered by the compiler. Although C# is 8.5 times faster than VB, both runtime performance trends are nearly constant: the standard deviation of VB is 1.7% and that of C# is 4.14%. This small variation is caused by the lack of static type information gathered for dynamic references. Therefore, the generated code does not seem to depend on the number of possible types.

The runtime performance of *StaDyn* core programs evolves in a different way. Execution time shows a linear increase in the number of types inferred by the compiler (the performance benefit drops when the number of possible types increases). As an example, the runtime performance benefit drops to 32 and 3.42 times better than VB and C# respectively, when the compiler infers 50 possible types for `dyn var` references. This difference between our approach and others is justified by the amount of type information gathered by the compiler. *StaDyn* continues collecting type information, even when references are set as dynamic, and this information is used to optimize the generated code. In contrast, both C# 4.0 and VB perform no static type inference once a reference is declared as dynamic.

When the compiler obtains no static type information, runtime perfor-

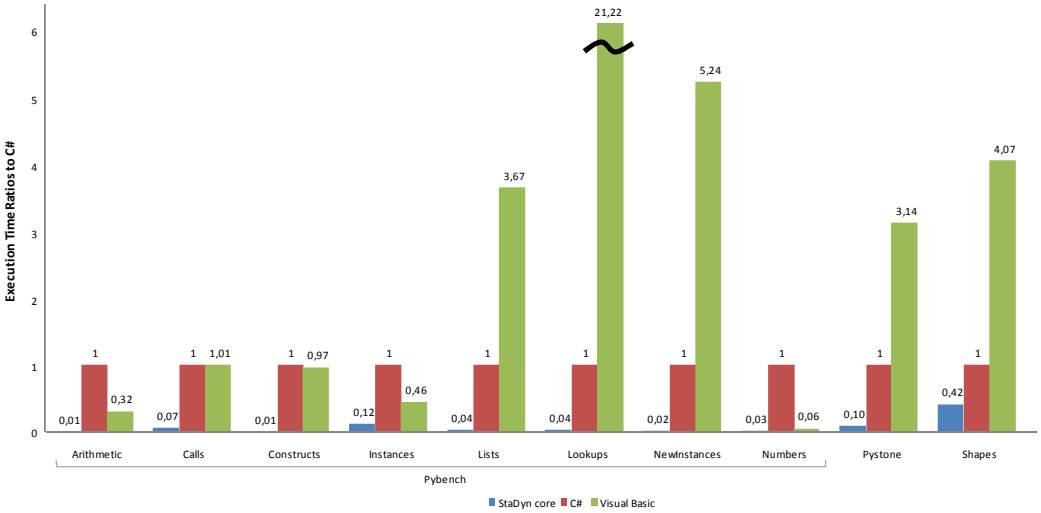

Figure 7: Execution time ratios to C#.

mance is the worst in the three programming languages. However, the *StaDyn* core requires 5.52% and 52.65% of the execution time that VB and C# respectively employ to run the same programs. In this scenario, the DLR implies a considerable performance improvement (C# vs. VB).

Figure 7 shows the ratios of execution time to C# for the hybrid (Shapes) and dynamic typing benchmarks (Pybench and Pystone). Running hybrid code, the performance benefit is 140.74% and 879.63% compared to C# and VB respectively. This benefit increases as the number of dynamic references in the code grows: average benefit (geometric mean) running the dynamic typing code is 3,106.29% (C#) and 3,381.3% (VB). Since the *StaDyn* core optimizations are obtained by means of collecting type information of dynamic references, the compiler has more opportunities to optimize the code when `dyn var` references are used. Therefore, our language offers the flexibility of dynamic typing, and a number of optimizations to come closer to the runtime performance of static typing.

The lowest performance benefit obtained by the *StaDyn* core running dynamic code is with the *numbers* test of Pybench (150% compared to VB). Since this test performs almost all the operations over constant numbers (few variables are used), our optimizations are hardly applied. Differences between C# and VB may be due to the appropriateness of using the DLR (C#) as opposed to the `Reflection` namespace (VB) for dynamically typed

code.

5. Conclusions

The *StaDyn* programming language combines static and dynamic typing, improving the runtime performance of dynamic typing and the flexibility of static typing. Its type system performs type inference of both static and dynamic implicit references in order to improve its runtime performance and statically type-check dynamic types. At the same time, the type information gathered by the compiler allows interoperation between both kinds of code, sharing the same type system. The key feature of the type system is a new interpretation of union and intersection types to consider the differences between type checking dynamic and static types.

The static type information has been used to optimize the .NET code generated. Runtime performance has been compared with C# 4.0 and VB 10. The *StaDyn* core type inference system has entailed significant performance benefits. Since the *StaDyn* optimizations are based on statically obtaining type information of dynamic references, the highest benefit is achieved when running dynamically typed benchmarks (on average, more than 30 times faster). However, our approach has also exhibited a notable improvement running hybrid code (from 141% to 880%). The worst scenario is when the type system does not infer any type information of dynamic references, obtaining at least a performance benefit of 89.9%.

Future work will be focused on formalizing the operational semantics of the language core aimed at proving its type safety when static references are used, and demonstrating specific properties of the type system—in particular those regarding dynamic code.

The C# implementation of the *StaDyn* minimal core, including its type system, the translation to C# (described in [17]), and all the examples presented in this paper, are freely available at `http://www.reflection.uniovi.es/stadyn/download/2010/ipletters`.

The current release of the whole *StaDyn* programming language implementation and its source code can be downloaded from `http://www.reflection.uniovi.es/stadyn`.

Acknowledgments

This work has been funded by Microsoft Research, under the project entitled *Extending dynamic features of the SSCLI*, awarded in the *Phoenix and*

SSCLI, Compilation and Managed Execution Request for Proposals, 2006. It has been also funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: project TIN2008-00276, entitled *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software.*

References

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

[2] A. Aiken, E.L. Wimmers, Type Inclusion Constraints and Type Inference, in: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, 1993, pp. 31–41.

[3] D. Ancona, G. Lagorio, Coinductive Type Systems for Object-Oriented Languages, in: Proceedings of the European Conference on Object-Oriented Programming, 2009, pp. 2–26.

[4] F. Barbanera, M. Dezani-Ciancaglini, U. de'Liguoro, Intersection and Union Types: Syntax and Semantics, Information and Computation 119(2) (1995) 202–230.

[5] G. Bracha, Pluggable Type Systems, in: OOPSLA Workshop on Revival of Dynamic Languages, 2004.

[6] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation, 1991, pp. 278–292.

[7] B. Chiles, A. Turner, Dynamic Language Runtime, `http://dlr.codeplex.com/Project/Download/FileDownload.aspx?DownloadId=97300`

[8] F.M. Damm, Subtyping with Union Types, Intersection Types and Recursive Types, in: Theoretical Aspects of Computer Software, 1994, pp. 687–706.

[9] C. Flanagan, S.N. Freund, A. Tomb, Hybrid types, invariants, and refinements for imperative objects, in: International Workshop on Foundations and Developments of Object-Oriented Languages, 2006.

[10] J.S. Foster, T. Terauchi, A. Aiken, Flow-Sensitive Type Qualifiers, in: Programming Language Design and Implementation, 2002, pp. 1–12.

[11] J. Hugunin, Just Glue It! Ruby and the DLR in Silverlight, in: MIX Conference, 2007.

[12] A. Igarashi, H. Nagira, Union Types for Object-Oriented Programming, Journal of Object Technology 6(2) (2007) 5–30.

[13] E. Meijer, P. Drayton, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages, in: Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, 2004.

[14] Microsoft corporation. C# 3.0 Language Specification. `http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc`

[15] F. Ortin, The StaDyn Programming Language, `http://www.reflection.uniovi.es/stadyn`

[16] F. Ortin, J.M. Redondo, J.B.G. Perez-Schofield, Efficient Virtual Machine Support of Runtime Structural Reflection, Science of Computer Programming 70(10) (2009) 836–860.

[17] F. Ortin. The StaDyn core Language, Technical Report, Computer Science Department, University of Oviedo, December 2010, `http://www.reflection.uniovi.es/stadyn/publications/2010/stadyn.core.pdf`

[18] F. Ortin, D. Zapico, J.B.G. Perez-Schofield, M. Garcia, Including both Static and Dynamic Typing in the same Programming Language, IET Software 4(4) 2010 (268–282).

[19] B.C. Pierce, Programming with intersection types, union types, and polymorphism, Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[20] J. Siek, W. Taha, Gradual Typing for Objects, in: Proceedings of the 21st European Conference on Object-Oriented Programing, Lecture Notes In Computer Science 4609, 2007, pp. 2–27.

[21] S. Thatte, Quasi-static typing, in: Proceedings of the ACM Symposium on Principles of programming languages, 1990, pp. 367–381.

[22] D. Thomas, C. Fowler, A. Hunt, Programming Ruby, second ed., Pragmatic Bookshelf, 2004.

[23] M. Torgersen, New features in C# 4.0, Microsoft Corporation, 2009.

[24] P. Vick, The Microsoft Visual Basic Language Specification, Microsoft Corporation, 2007.