Type Inference to Optimize a Hybrid Statically and Dynamically Typed Language

FRANCISCO ORTIN

Computational Reflection Research Group, Department of Computer Science, Calvo Sotelo s/n, 33007, Oviedo, Spain
Email: ortin@lsi.uniovi.es

Dynamically typed languages are becoming increasingly popular for different software development scenarios such as Web engineering, rapid prototyping, or the construction of applications that require runtime adaptiveness. In contrast, statically typed languages have undeniable advantages such as early type error detection and more opportunities for compiler optimizations. Since both approaches offer different benefits, hybrid statically and dynamically typed programming languages have emerged, and some statically typed languages have also incorporated dynamic typing capabilities. In this paper we present the minimal core of *StaDyn*, a hybrid typing language that performs static type inference of both statically and dynamically typed references. The type information gathered by the compiler is used to generate efficient .NET code, obtaining a significant runtime performance improvement compared to C# 4.0 and Visual Basic 10.

Keywords: Hybrid Static and Dynamic Typing; Union Types; Intersection Types; Runtime Performance; Type Systems

Received 15 February 2011

1. INTRODUCTION

Dynamically typed programming languages have recently turned out to be really suitable for specific scenarios such as Web development, application frameworks, game scripting, interactive programming, rapid prototyping, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software. Common features of dynamic languages are metaprogramming, reflection, mobility, and dynamic reconfiguration and distribution.

Taking the Web engineering area as an example, Ruby [1] has been successfully used together with the *Ruby on Rails* framework for creating database-backed web applications [2]. This framework has confirmed the simplicity of implementing the DRY (*Don't Repeat Yourself*) [3] and the *Convention over Configuration* [2] principles with this kind of languages. Nowadays, JavaScript [4] is being widely employed to create interactive Web applications with AJAX (*Asynchronous JavaScript And XML*) [5], while PHP (*PHP Hypertext Preprocessor*) is one of the most popular languages to develop Web-based views. Python [6] is used for many different purposes, being the *Zope* application server [7] (a framework for building content management systems, intranets and custom applications) and the *Django* Web application framework [8] two well-known examples. Due to its small size, portability and ease of integration, *Lua* [9] has gained great popularity for extending games [10]. Finally, a wide range of dynamic aspect-oriented tools has been built over dynamic languages [11, 12, 1, 13], offering a higher runtime adaptiveness than the common static ones.

The benefits offered by dynamically typed programming languages have caused the recent 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 [14]. 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 the *Dynamic Language Runtime* (DLR) services [15].

Java also seems to follow this trend. The

Java Specification Request (JSR) 292 [16], expected to be included in Java 1.7, incorporates the new `invokedynamic` opcode to the Java Virtual Machine (JVM) in order to support the implementation of dynamically typed object-oriented languages. Since the computational model of dynamic languages requires extending the JVM semantics, Sun Microsystems launched the *Da Vinci Machine* project in January 2008 [17]. This project is aimed at prototyping a number of enhancements to the JVM, so that it can run non-Java languages, especially dynamic ones, with a performance level comparable to that of Java itself.

The great flexibility of dynamic languages is, however, counteracted by limitations derived by the lack of static type checking. This deficiency implies two major drawbacks: no early detection of type errors, and commonly a considerable runtime performance penalty. Static typing offers the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime –when the programmer's efforts might be aimed at some other task, or even after the program has been deployed [18]. Moreover, the runtime type inspection and type checking performed by dynamic type systems commonly involve a significant performance penalty.

Since both approaches offer important benefits, there have been former works on providing both typing approaches in the same language (see Section 6). Meijer and Drayton maintained that instead of providing programmers with a black or white choice between static or dynamic typing, it could be useful to strive for softer type systems [19]. Static typing allows earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance. Dynamic typing languages provide a solution to a kind of computational incompleteness inherent to statically-typed languages, offering, for example, storage of persistent data, inter-process communication, dynamic program behavior customization, or generative programming [20]. Hence, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [21]. That is, *static typing where possible, dynamic typing when needed* [19].

Our work breaks the programmers' black or white choice between static or dynamic typing. We have designed a programming language, called *StaDyn* [22], that supports both static and dynamic typing in the very same programming language—an informal description of the language can be consulted in [23]. Dynamic typing offers higher flexibility, whereas static typing implies better robustness and performance. The major contribution of our programming language, compared to the existing hybrid typing languages, is that the compiler keeps performing type checking even over dynamic references. The type information gathered at compile time is used for both improving the runtime performance and the early type error detection of the programming language.

In this paper we reduce the *StaDyn* programming language to its minimal core, making it easy to describe its type system and its erasure semantics. The key contributions of this paper are:

- A type system to infer static type information of dynamic references (Section 3.2). The type system is flow-sensitive [24] and interprets static information along the control flow path, merging the type information of the incoming branches with union and intersection types [25]. This information is used to improve both the efficiency and the robustness of programs written in this language.
- Its erasure semantics specification by translating the *StaDyn* core to C# (Section 4). The translation scheme uses the static type information gathered by the compiler to generate efficient .Net code.
- A runtime performance assessment to measure the runtime performance of our proposal (Section 5.2). We compare runtime performance of the *StaDyn* core with that of the C# 4.0 and Visual Basic 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.

The rest of this paper is structured as follows. In the next section, we provide an informal overview of the *StaDyn* core to motivate our work. Section 3 formally describes the abstract syntax (Section 3.1) and the type system (Section 3.2) of the *StaDyn* core. Section 4 presents its erasure semantics by translating it into C#, and a runtime performance assessment is detailed in Section 5. Related work is commented in Section 6 and, finally, Section 7 presents the conclusions and future work.

2. AN INFORMAL OVERVIEW OF THE *STADYN* CORE

Figure 1 shows an example use of the `dynamic` type recently included in C# 4.0. The first benefit is *duck typing*. Duck typing [1] 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. In line 33 (Figure 1) the `x` field of the `data` attribute in the `l` list is accessed regardless of its type (`list` and `l` have been declared as `dynamic`). This means that the first parameter of the `positiveX` method could be any linked list whose `data` objects implement an `x` field. These objects do not need to

```
01: using System;
02: class Points {
03:   static dynamic createNode(dynamic data, dynamic next){
04:     return new { data = data, next = next };
05:   }
06:   static dynamic createPoint(int dimensions, int x,
                                  int y, int z) {
07:     dynamic point;
08:     if (dimensions == 2)
09:       point = new { x=x, y=y, dimensions=dimensions };
10:     else
11:       point = new { x=x, y=y, z=z, dimensions=3 };
12:     return point;
13:   }
14:   static dynamic createPoints(int number) {
15:     int i;
16:     dynamic list, point;
17:     i = 0;
18:     list = null;
19:     while (i < number) {
20:       point = createPoint(i%2 + 2, number/2 - i, i, i);
21:       list = createNode(point, list);
22:       i = i + 1;
23:     }
24:     return list;
25:   }
26:   static dynamic positiveX(dynamic list, int n) {
27:     int i;
28:     dynamic l, result;
29:     i = 0;
30:     result = null;
31:     l = list;
32:     while (i < n) {
33:       if (l.data.x >= 0)
34:         result = createNode(l.data, result);
35:       l = l.next;
36:       i = i + 1;
37:       }
38:     return result;
39:   }
40:   static double distance3D(dynamic point) {
41:     double value;
42:     value = Double.MaxValue;
43:     // point.center++; // No compiler error
44:     if (point.dimensions == 3)
45:       value = Math.Sqrt(point.x*point.x +
                    point.y*point.y + point.z*point.z);
46:     return value;
47:   }
48: static dynamic closestToOrigin3D(dynamic list, int n) {
49:   int i;
50:   double minDistance;
51:   dynamic l, point3D = null;
52:   minDistance = Double.MaxValue;
53:   l = list;
54:   i = 0;
55:   while (i < n) {
56:     if (distance3D(l.data) < minDistance) {
57:       minDistance = distance3D(l.data);
58:       point3D = l.data;
59:     }
60:     l = l.next;
61:     i = i + 1;
62:   }
63:   return point3D;
64:   }
65:   static void Main() {
66:     int numberOfPoints;
67:     dynamic list, positive, point;
68:     numberOfPoints = 10;
69:     list = createPoints(numberOfPoints);
70:     // list.data = 10; // No compiler error
71:     positive = positiveX(list, numberOfPoints);
72:     point = closestToOrigin3D(list, numberOfPoints);
73:   }
74: }
```

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

belong to a specific hierarchy defining the shared `x` message, and they do not have to be instances of the same type either. An example of this flexibility is shown in Figure 1, where the `list` reference passed to the `positiveX` method (line 71) holds a linked list with objects of two different types (two and three dimensional points). The method returns another list containing those objects whose `x` field value is positive, regardless of their type.

Dynamic typing is also used in the `closestToOrigin3D` method. In this case, the first parameter should be a linked list whose `data` is 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 `Math.Sqrt` method). The returned object is the one that fulfills these conditions, being that nearest to the origin of coordinates. This example shows how the C# 4.0 type system can consider dynamic conditions.

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# 3.0 [22], 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 [26]. 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 later in this paper: its abstract syntax is specified in Section 3.1; Section 3.2 details the hybrid (static and dynamic) type system; and, based on the semantics of C#, Section 4 describes 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 version 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 *StaDyn* is that static type checking is performed even over dynamic references. For instance, the `positiveX` function statically checks that each `data` object in `list` provides a public `x` field. Unlike C#, the *StaDyn* core prompts a compilation error in line 70 (function invocation in Figure 2), if code in line 69 is commented out. The error indicates that one of the elements in

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

FIGURE 2. Example coded in the minimal core of *StaDyn*.

the list (the integer) does not provide the `x` message. In contrast, C# 4.0 compiles the code and the error is produced at runtime (line 70 in Figure 1).

Our compiler gathers type information at compile time in order to perform static type checking over dynamic references. One of the elements we have used for this purpose is union types [25]. 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 [27], representing the least upper bound of T_1 and T_2 [28]. A union type holds all the possible types a reference may have. The set of operations (e.g., addition, field access, assignment, invocation or indexing) that can be applied to a union type are those accepted by every type in the union type (inference rules of static union types are S-SUnionL and S-SUnionR in Figure 11). Union types were already included in object-oriented languages, in type systems where they were explicitly declared [29] or inferred from implicitly typed references [30].

In our example, the type inferred for `list` in line 68 (Figure 2) is a list of $\{x{:}\texttt{int}, y{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\} \vee \{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\}$, meaning two or three dimensional points. In the invocation of the `positiveX` function (line 70), it is statically checked that the argument is a list of objects that provide an `x` field. Since this condition is statically fulfilled, the program is compiled without errors (and the static type information is used to optimize its execution). However, if we uncomment line 69, an error message will be shown.

The `closestToOrigin3D` function imposes more constraints to the `list` parameter. Objects in `list` must provide the `dimensions`, `x`, `y` and `z` fields because of the invocation to `distance3D`. We represent these constraints by means of intersection types [25]. $T_1 \wedge T_2$ denotes all the values belonging to both T_1 and T_2 [27], representing the greatest lower bound of T_1 and T_2 [28]. 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 11).

In our example, the argument `list` of the `closestToOrigin3D` function must be a list of X type, being $X \leq [\texttt{dimensions}{:}X_1] \wedge [\texttt{x}{:}X_2] \wedge [\texttt{y}{:}X_3] \wedge [\texttt{z}{:}X_4]$ (an object with all these four fields). However, the invocation in line 71 produces a compilation error because `list` holds a union type of both two and three dimensional points, and the former do not provide the `z` field. Our approach is to make the type system more lenient, without renouncing static type checking. The `point` parameter of the `distance3D` function can be declared as dynamic (uncommenting the `dyn` type qualification in line 38 of Figure 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 in Figure 11). Then, the program would generate no error

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

FIGURE 3. Abstract Syntax of the *StaDyn* minimal core.

because both types of points offer a public `dimensions` field. This relaxation of the subtyping relation when references are declared as dynamic is also applied to union types (S-DUnionL): the promotion should be fulfilled by at least one of the types in the union type.

It is worth noting that type checking is still performed at compile type even when the programmer uses dynamic references. As an example, if line 41 in Figure 2 is uncommented, an error is shown even though `point` has been declared as dynamic (the `center` message is not accepted by either of the two possible points); whereas C# compiles the code, producing the type error at runtime (line 43 in Figure 1). This example informally shows the objective of *StaDyn*: to offer both the flexibility of dynamic typing and the robustness and efficiency of static typing.

3. THE *STADYN* CORE LANGUAGE

After an informal overview of the aim of the *StaDyn* core programming language, we describe its syntax and type system. Next section describes its erasure semantics by translating it into to C#.

3.1. Syntax

The first part of Figure 3 shows the abstract syntax of the minimal core (the second and third parts are, respectively, types and constraints). EBNF is used, where $^+$ means repetition of at least one element, * matches zero or more occurrences, $^?$ means optionally matching the previous element, and | represents alternative.

A program (P) is composed of a sequence of function declarations (F^*) followed by the local variable declarations (D^*) and statements (S^+) of the main function. Although the programmer may use the `return` statement the same way as in C#, it could only be placed as the last statement of the abstract syntax. This transformation is performed by the parser to facilitate type inference in conditional and iterative control structures (Section 3.2.6).

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

The $\oplus$ operator represents arithmetic operations, $\otimes$ logical ones and # symbolizes relational operators. Objects are created following the syntax of the C# 3.0 feature of anonymous types [26]: between curly braces, there is listed a sequence of field identifiers followed by the assignment operator and an expression representing their initial values (see lines 9 and 11 in Figure 1). The `new` expression for arrays creates one-dimensional arrays. Multidimensional arrays should be built in loops repeating the construction of one-dimensional arrays.

3.2. Type System

Types used to describe the *StaDyn* minimal core type system are shown in the second part of Figure 3. Syntax types (ST) are those that may be directly written by the programmer, whereas internal types (T) are internally used by the type system without the knowledge of the programmer. The point of avoiding the direct use of internal types is to offer the programmer the greatest possible simplicity without losing the expressive power of the type system.

Object types are specified describing a collection of their fields between curly braces, not including methods[1]. Methods can be represented by functions where `this` is the first parameter. Although this representation does not support method overriding, it allows us to significantly reduce the *StaDyn* core type system. *StaDyn* (the whole programming language) does support method overriding by extending the

[1]A class-based core like the one proposed in [31] would be more appropriate to formalize methods and overriding.

behavior described in [29]: when a message is passed to a dynamic union type, it is checked that at least for one possible signature, the actual argument types are subtypes of the corresponding formal parameters; the type of the method invocation expression is the union of the return types declared by those methods that satisfy the previous condition.

Although the **var** keyword is part of the concrete syntax of type variables (included in C# 3.0 to allow avoiding type specification of initialized local variables [26]), the parser assigns them unique sequential numbers (X_i metavariables range over type variables). Type variables can be declared as dynamic (**dyn**) or, by default, static (**sta**). Only intersection and union types can also be qualified as dynamic or static.

Member types ($\texttt{[}(id{:}IT)^*\texttt{]}$) represent the collection of fields an object may hold. Member types have been introduced in constraints to define structural width coercion of object types to member types (S-OMember rule in Figure 11), because objects in *StaDyn* do not define width subtyping (S-Object in Figure 11). This subtyping relation is used in the constraint resolution algorithm when function calls are type-checked (T-Inv in Figure 17).

Type inference is specified with the general judgment $\Gamma;\Omega \vdash E : T \parallel C;\Gamma'$, meaning that under constraints C, environment Γ, and context Ω, expression E has type T, producing the output environment Γ'. Environments (Γ) bind variables (identifiers) to types in the scope represented by Γ, and they also bind type variables to types (if type variables have been inferred). Γ holds the environment before the scope of E, and Γ' stores the environment after typing E. Γ' might differ from Γ, containing inferred types of local variables and new types bound to type variables inferred in E. Output environments have already been used to define flow-sensitive type systems [24], because type variables may change their types depending on the control flow [32].

A context (Ω) stores the information of the function being analyzed, in order to type-check its statements. $\Omega._{\text{params}}$ saves the parameter list of the current function, $\Omega._{\text{rt}}$ holds its declared return type, and $\Omega._{\text{tifp}}$ collects the types inferred from function parameters (see Section 3.2.2).

Figure 4 shows another example program of our core language. Elements of the environment and generated constraints are shown in the right part of the figure. For example, in the scope of the **main** function in Figure 4, Γ holds the assumptions $\Gamma(\texttt{increment}){:}\texttt{int}$, $\Gamma(\texttt{list1}){:}X_{18}$, and, in line 16, $\Gamma(X_{18}){:}\{data{:}X_{20}, next{:}X_{21}\}$, $\Gamma(X_{20}){:}\texttt{bool}$ and $\Gamma(X_{21}){:}\texttt{int}$. Since $\Gamma(X_{20}){:}\texttt{bool}$ in line 16, the statement in line 17 is accepted by the type system. However in line 19 the type of the object **data** field is changed to **int** and, hence, line 20 compiles without any error, whereas line 21 is now erroneous. This example shows how a variable can hold different types in the same scope, depending on the execution flow. This is a common feature of dynamically typed languages, but *StaDyn* offers it in a statically typed way. This process has also been applied to control structures (Section 3.2.6).

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

FIGURE 4. Example concrete program.

We also define two kinds of constraints (the last part of Figure 3). Subtyping constraints ($T \leq T$) require the type on the left to be a subtype of (promote to) the type on the right. Assignment constraints ($TV \leftarrow T$) not only check that an assignment could be performed, but they are also used to infer types, binding a type variable to another type. Therefore, assignment constraints may modify type variable bindings in type environments, when function invocation expressions are checked. In line 5 of Figure 4, a subtyping constraint is generated for the **node** variable; it should be an object with a **data** field, i.e., a subtype of a member type: $X_{13} \leq [data{:}X_{15}]$. This constraint must be statically fulfilled wherever the function **setData** is called, e.g., line 19. Line 5 is also an example of an assignment constraint generation: $X_{15} \leftarrow X_{14}$. When the **setData** function is invoked, the **data** type of the **node** argument X_{15} will be assigned the type of the **data** parameter X_{14}. This is the reason why X_{20} is then bound to **int** in line 19.

3.2.1. Functions

We use $\diamond$ to denote well-formedness. Inference rules in Figure 5 not only check well-formedness, but they also generate output environments and constraints that are used for type-checking subsequent expressions. As an example, T-Func adds the identifier of the function being declared to the output environment. This identifier type is now $T_1 \times \ldots \times T_n \to T \parallel C$, denoting that it is possible to type-check subsequent calls to it. Function types include the constraint set (C) that must be satisfied by the arguments at each invocation. These constraints are those produced by the statements within the function. For instance, the type expression of the **setData** function in Figure 4 (line 6) has the two

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

(T-Decl)
$$\frac{\begin{array}{c} id \notin dom(\Gamma) \\ \Gamma' = \Gamma, id : T\end{array}}{\Gamma;\Omega \vdash T\ id : \diamond \parallel \emptyset; \Gamma'}$$

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

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

FIGURE 5. Program, declarations and functions.

constraints $X_{13} \leq [data{:}X_{15}]$ and $X_{15} \leftarrow X_{14}$. The rest of the rules in Figure 5 generate no constraint at all, and output environments become the input of the following terms, obtaining a flow-sensitive type checking.

3.2.2. Context

It is necessary to store information regarding a function in order to subsequently perform type checking of the terms in the function scope. This information is saved in the function context (Ω) by means of T-Func (Figure 5) and the rules shown in Figure 6. At function declaration (T-Func), local variables are stored in $\Omega._{\texttt{locals}}$, parameters in $\Omega._{\texttt{params}}$, and $\Omega._{\texttt{rt}}$ saves the return type specified in the function declaration. The types inferred from the type parameters are stored in $\Omega._{\texttt{tifp}}$ (it will be described later why this information is necessary to perform type-checking of assignments, field accessing and array indexing). First, T-Func (Figure 5) adds parameter types to $\Omega._{\texttt{tifp}}$; in Figure 6, $\Omega._{\text{TIFP}}$-Field inserts field types in $\Omega._{\texttt{tifp}}$ whenever an object type is in $\Omega._{\texttt{tifp}}$; $\Omega._{\text{TIFP}}$-Array and $\Omega._{\text{TIFP}}$-Inv do the same with arrays and function calls, respectively.

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

3.2.3. Basic Expressions

This subsection describes the type-checking of variables, object field access, vector indexing, arithmetic, relational and logical expressions. Although assignments and function calls are also expressions, they will be described in Sections 3.2.5 and 3.2.7, respectively.

Figure 7 shows inference rules that type-check variables. The `tv` predicate tests whether a type is a type variable or not, and the `ftv` function returns the set of unbound type variables in an environment. T-Var types a variable previously declared, when its

($\Omega._{\text{TIFP}}$-Field)
$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E : \{id_1 : T_1, \ldots, id_n : T_n\} \parallel C; \Gamma' \\ \{id_1 : T_1, \ldots, id_n : T_n\} \in \Omega._{\text{tifp}}\end{array}}{\text{include } T_1 \ldots T_n \text{ in } \Omega._{\text{tifp}}}$$

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

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

FIGURE 6. Inference rules for Ω.

(T-Var)
$$\frac{\begin{array}{c}\Gamma(id) : T \\ \neg tv(T)\end{array}}{\Gamma;\Omega \vdash id : T \parallel \emptyset; \Gamma}$$

(T-BVar)
$$\frac{\begin{array}{c}\Gamma(id) : X \\ \Gamma(X) : T\end{array}}{\Gamma;\Omega \vdash id : T \parallel \emptyset; \Gamma}$$

(T-PVar)
$$\frac{\begin{array}{c}\Gamma(id) : X \qquad X \in ftv(\Gamma) \\ id \in \Omega._{\text{params}} \\ \neg lhsAssign(id)\end{array}}{\Gamma;\Omega \vdash id : X \parallel \emptyset; \Gamma}$$

(T-AVar)
$$\frac{\begin{array}{c}\Gamma(id) : X \qquad X \in ftv(\Gamma) \\ id \notin \Omega._{\text{params}} \\ lhsAssign(id)\end{array}}{\Gamma;\Omega \vdash id : X \parallel \emptyset; \Gamma}$$

FIGURE 7. Variables.

type is not a type variable. When the type of an identifier is a type variable and it is bound to another type, T-BVar types the identifier to the type bound to the type variable. This happens, for instance, in line 17 of Figure 4, where the type variable of `list1` (X_{18}) was previously bound to the object type $\{data{:}X_{20}, next{:}X_{21}\}$ in line 16.

Both T-PVar and T-AVar type-check identifiers when their types are free type variables (not bound to any other type). In the first case, the variable can be used when it is a parameter (thus, it has a value) and it is not the left-hand side of an assignment[2]. On the other hand, T-AVar allows a free type variable that is not a parameter to be used as an expression as long as it is the left-hand side of an assignment (because the type variable will be then bound to a type in the subsequent expression). For example, the utilization of the `data` parameter in line 2 of Figure 4 is allowed because, despite its type is a free type variable (X_{10}), it is contained in $\Omega._{\text{params}}$. `list1` can be used in line 16 because, although its type (X_{18}) is not in $\Omega._{\text{tifp}}$, it is in the left-hand side of an assignment.

Figure 8 shows the T-Arith inference rule of arithmetic expressions (for the sake of brevity, relational and logical expressions are not shown). Operands

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

(T-Arith)

$$\frac{\begin{array}{cc}\Gamma;\Omega\vdash E_1 : T_1 \parallel C_1;\Gamma' & \Gamma'\vdash T_1 \leq \texttt{int} \parallel C_2;\Gamma'' \\ \Gamma'';\Omega\vdash E_2 : T_2 \parallel C_3;\Gamma''' & \Gamma'''\vdash T_2 \leq \texttt{int} \parallel C_4;\Gamma''''\end{array}}{\Gamma;\Omega\vdash E_1 \oplus E_2 : \texttt{int} \parallel C_1\cup C_2\cup C_3\cup C_4;\Gamma''''}$$

(T-NObject)

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

(T-NArray)

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

(T-Field)

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

(T-Array)

$$\frac{\begin{array}{c}\Gamma;\Omega\vdash E_1 : T_1 \parallel C_1;\Gamma' \\ X \text{ fresh} \qquad \Gamma'\vdash T_1 \leq Array(X) \parallel C_2;\Gamma'' \\ \Gamma'';\Omega\vdash E_2 : T_2 \parallel C_3;\Gamma''' \qquad \Gamma'''\vdash T_2 \leq \texttt{int} \parallel C_4;\Gamma'''' \\ X\in ftv(\Gamma'''')\wedge X\notin\Omega._{\mathrm{tifp}} \Rightarrow lhsAssign(E_1[E_2])\end{array}}{\Gamma;\Omega\vdash E_1[E_2] : X \parallel C_1\cup C_2\cup C_3\cup C_4;\Gamma''''}$$

FIGURE 8. Basic expressions.

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

FIGURE 9. Example use of arrays.

of arithmetic and relational expressions must be subtypes of `int`; logical expressions should promote to `bool`. Output environments are used as the input environments of the subsequent expressions, the one returned by the whole expression being the last environment. The constraint set generated by each expression is the union of all the constraints produced by each of the four premise judgments.

The `new` object expression (T-NObject) infers an object type comprising the field labels and types of the corresponding expressions. Line 2 in Figure 4 is an example of this inference rule, where the type of the expression is $\{data{:}X_{10}, next{:}X_{11}\}$. In a similar way, the T-NArray rule types the array construction expressions. The expression that specifies the array size must promote to integer. Only one-dimensional arrays can be constructed at a time, and the type returned is an array of the same dimensions as pairs of square brackets. The type of the `new` expression in line 11, Figure 9, is $Array(X_{34})$, X_{34} being a new fresh type variable.

When accessing object fields (T-Field), the object should promote to the member type $[id : X]$, X being a new fresh type variable. A member type is an internal type that denotes the set of fields an object should hold. Therefore, an object promotes to a member type following the same rules as structural subtyping for objects described in [33] (rule S-OMember in Figure 11). Moreover, if the object field is a free type variable not inferred from the parameters, i.e., not in $\Omega._{\mathrm{tifp}}$, it must be a direct left child of an assignment expression. Line 42 in Figure 2 is an example of a correct term. Although the `dimensions` field of the `point` object is a free type variable and it is not the left-hand side of an assignment, its type is in $\Omega._{\mathrm{tifp}}$.

T-Array requires the first expression to be a subtype of an array, and the index to be an integer. As with objects, if the calculated type is a free type variable, it should be the left-hand side of an assignment. This predicate generates a compilation error in line 5 of Figure 9. The type of the elements of the `v` array is the free type variable X_{32}, not inferred from the parameters (`v` is a local variable), producing a compilation error because no value has been assigned to it.

3.2.4. Subtyping

Judgments in subtyping rules ($\Gamma\vdash T_1 \leq T_2 \parallel C;\Gamma'$) require an input environment (Γ) and generate a set of constraints (C) and an output environment (Γ'). The input environment is used to know the type variables that might be bound to other types. In effect, the T-TVBS rule in Figure 10 types any expression to the type which is bound to the expression type variable.

The output environment is used to bind a type variable to a type when the type variable must be a subtype of a particular type. This is precisely the behavior of the S-FTVL and S-FTVR rules in Figure 11 that, in addition, generate a subtyping constraint. An example expression where the S-FTVL rule is applied is `node.data` in Figure 4, line 5. The T-Field rule requires the type of `node` (the X_{13} free type variable), to be a subtype of the member type [`data`:X_{15}]—X_{15} being a new fresh type variable. Then, the S-FTVL rule generates a new $X_{13} \leq$ [`data`:X_{15}] constraint and binds [`data`:X_{15}] to X_{13} in the output environment Γ'.

S-FTVR offers the same functionality when a concrete type must promote to a free type variable. This rule is used in `return` statements inside functions that return a type variable (e.g., line 2 in Figure 4). When both type variables are not bound to any type, only a subtyping constraint is produced (S-FTVs in Figure 11).

The arrays (S-Array) and objects (S-Object) type constructors are invariant. $Array(T_1)$ is a subtype of $Array(T_2)$ when T_1 and T_2 are equivalent. T_1 and T_2 are equivalent under the subtype relation, when $T_1 \leq T_2$ and $T_2 \leq T_1$ (E-Types). An object promotes

(T-TVBS)

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

FIGURE 10. Type variable binding substitution.

to another one when both have the same number of fields and equal field labels, and the corresponding types are equivalent (S-Object).

Member types were introduced for structural subtyping of objects. An object type is a subtype of a member type when the former has all the members of the latter, and their corresponding types are structurally equivalent (S-OMember). This rule is necessary in the fulfillment of subtyping constraints of function invocation (Section 3.2.7). As an example, the `setData` function in Figure 4 has the $X_{13} \leq [data : X_{15}]$ constraint (line 6). When the function is called in line 19 passing the $\{data : X_{20}, next : X_{21}\}$ object as the first argument, the S-OMember subtyping rule confirms that the argument promotes to the parameter.

Subtyping rules for union and intersection types are an enhancement of the ones defined by other authors such as [25] and [34] (S-SUnionL, S-SUnionR, S-SInterR and S-SInterL), adding new dynamic typing rules (S-DUnionL and S-DInterR)—[35] details this new interpretation of union and intersection types. If the type variable bound to a union type has been declared as static, the set of operations that can be applied to that union type are those accepted by every type in the union type (S-SUnionL). However, if the reference is dynamic, type-checking is more permissive. In that case, it is possible to perform an operation when it is accepted by at least one of the types in the union type (S-DUnionL)—in the conclusion of the rule, $\cup\Gamma_i$ and $\cup C_i$ represent the union of all the 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. This behavior can be seen in lines 18 and 19 of Figure 12. The type of both `sta` and `din` variables is `int`$\vee$`bool`, but `sta` is static whereas `din` is dynamic. This difference prevents the arithmetic operation in line 18 from compiling (the plus operator cannot be applied to a `bool`), while it is correct in line 19 (addition is defined for integers).

In parallel, a type promotes to a static intersection type only if it is a subtype of all the types collected by the intersection type (rule S-SInterR). Similarly, we have defined the dynamic behavior to be more lenient, accepting the promotion when a type promotes to at least one of the types in the intersection type (rule S-DInterR). An example is the function call in line 71 of Figure 2: `data` of the `list` argument must be a subtype of $[dimensions{:}X_1] \wedge [x{:}X_2] \wedge [y{:}X_3] \wedge [z{:}X_4]$; the program is compiled only when the `point` parameter in

(S-Bool)

$$\Gamma \vdash \texttt{bool} \leq \texttt{bool} \parallel \emptyset;\Gamma$$

(S-Int)

$$\Gamma \vdash \texttt{int} \leq \texttt{int} \parallel \emptyset;\Gamma$$

(S-FTVL)

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

(S-FTVR)

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

(S-FTVs)

$$\frac{X_1 \in ftv(\Gamma) \qquad X_2 \in ftv(\Gamma) \qquad C = X_1 \leq X_2}{\Gamma \vdash X_1 \leq X_2 \parallel C;\Gamma}$$

(E-Types)

$$\frac{\Gamma \vdash T_1 \leq T_2 \parallel C_1;\Gamma' \qquad \Gamma' \vdash T_2 \leq T_1 \parallel C_2;\Gamma''}{\Gamma \vdash T_1 \equiv T_2 \parallel C_1 \cup C_2;\Gamma''}$$

(S-Array)

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

(S-Object)

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

(S-OMember)

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

(S-SUnionL)

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

(S-SUnionR)

$$\Gamma \vdash\ T_i^{\ i \in 1..n} \leq \texttt{sta}\ T_1 \vee \ldots \vee T_n \parallel \emptyset;\Gamma$$

(S-DUnionL)

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

(S-SInterL)

$$\Gamma \vdash \texttt{sta}\ T_1 \wedge \ldots \wedge T_n \leq\ T_i^{\ i \in 1..n} \parallel \emptyset;\Gamma$$

(S-SInterR)

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

(S-DInterR)

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

FIGURE 11. Subtyping and type equivalence.

line 38 is dynamic (two and three dimensional points provide `dimensions`, `x`, and `y` fields), producing an error in case it is static (the `z` field is not implemented by two dimensional points).

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

```
var getElement(var list, var fstOrSnd) {
  var element;
  if (fstOrSnd)
    element = list.data;
  else
    element = list.next.data;
  return element;
}
int increment(int value) {
  return value + 1;
}
```

```
void main() {
  int integer;
  var listOfTwo, sta;
  dyn var din;
  listOfTwo = createNode(1,createNode(true,0));
  din = sta = getElement(listOfTwo, true);
  integer = sta + 1;     // Compiler error
  integer = din + 1;
  increment(din);        // Compiler error
}
```

FIGURE 12. Example use of dynamic and static references.

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

3.2.5. Assignments

The abstract syntax in Figure 3 allows any expression to be the left-hand side of an assignment. The type system rejects all those expressions that cannot be used in that context. Only identifiers, array indexing and field access expressions can be the left-hand side of an assignment. For the sake of brevity, we do not show those rules.

The four inference rules in Figure 13 describe assignment expressions. T-Assign types assignment expressions when the left-hand side expression type is not a type variable. This straightforward rule only requires the right-hand side to be a subtype of the left-hand side. In case the left-hand side is a type variable, it will from now on be bound to the type of the right-hand side expression (T-TVAssign).

T-FAssign types the assignment of an object field when it is a type variable. As with the T-Field rule, the object should be a subtype of a member type with the specific field label. The new field type will be the type of the right-hand side expression, regardless of its previous type. Finally, if the field type has been inferred from a parameter, a new assignment constraint will be generated. This constraint will cause changing the field type of the argument when the function is called. An example of this kind of assignment constraint generation is shown in the `setData` function in Figure 4 ($X_{15} \leftarrow X_{14}$). Calling this function with an object as the first argument (line 19) changes the type of the argument's `data` to the type of the second argument (`int`), making the statement in line 20 correct.

For an array type whose elements are type variables (T-AAssign), the new type of its elements will be a union type comprising its previous type and the type of the right-hand side. Therefore, the type of `v` in line 8 of Figure 9 is an array of integers or booleans (`int`$\vee$`bool`) because it holds both. If the type variable

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

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

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

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

FIGURE 13. Assignments.

has been inferred from the function parameters, a new assignment constraint will be generated including the own type variable in the right-hand side of the assignment. This is the case of the `w` variable in the `vector` function (line 8 in Figure 9). Unlike the type of `v`, the type of `w` (X_{30}) is included in the union type ($X_{30} \vee$ `int` $\vee$ `bool`), denoting that the type of the actual parameter in the invocation will be included in the union type. Therefore, the type of the elements of `ve` when the function `vector` is called in line 13 is {`attribute`:`int`}$\vee$`int`$\vee$`bool`.

3.2.6. Statements

The minimal core includes the `return`, `if` and `while` statements (Figure 14). For the `return` statement, the expression type to be returned must be a subtype of the declared return type (T-Return).

Control-flow branches of `if` and `while` statements are taken into consideration to keep the flow-sensitiveness of our type system. The `join` of constraints and the union of type environments take into consideration this difficulty, taking the type information obtained on each execution path and combining both into a single constraint list and type environment. Each parameter represents the type information of an exclusive control flow. Since it might happen that the `while` body is not executed at runtime (it is not exclusive), the empty set is passed as its second argument (T-While). An example use of the `join` function is the type of the `point` variable ($\{x{:}\texttt{int}, y{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\} \vee \{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\}$)—line 10 in Figure 2—that is created from its types in lines 7 ($\{x{:}\texttt{int}, y{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\}$) and 9 ($\{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}, \mathit{dimensions}{:}\texttt{int}\}$). The same happens with constraints: the joined constraint for the `point` parameter of the `distance3D` function ($X_1 \leq [\mathit{dimensions}{:}X_2] \wedge [x{:}X_3] \wedge [y{:}X_4] \wedge [z{:}X_5]$) is obtained from the constraints generated in lines 42 ($X_1 \leq [\mathit{dimensions}{:}X_2]$) and 43 ($X_1 \leq [x{:}X_3] \wedge [y{:}X_4] \wedge [z{:}X_5]$).

The union of environments used in Figure 14 is also based on the `join` function described in Figure 15: variable bindings must be the same in both environments, and the resulting type variable binding set is the `join` of the type variable binding sets of each flow path.

(T-Return)

$$\frac{\Gamma;\Omega \vdash E : T \parallel C_1;\Gamma' \qquad \Gamma' \vdash T \leq \Omega._{\mathrm{rt}} \parallel C_2;\Gamma''}{\Gamma;\Omega \vdash \texttt{return}\ E : \diamond \parallel C_1 \cup C_2;\Gamma''}$$

(T-If)

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

(T-While)

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

FIGURE 14. Statements.

Figure 15 shows the algorithm used to implement the `join` function. Each set holds either constraints (subtyping and assignment) or type variable bindings ($X{:}T$ in environments). The algorithm has been defined employing the `compare` and `union` operations defined by the axioms in Figure 16. The algorithm takes elements of both sets, adding new union and intersection types [25] to the return set. It first processes the elements in the first set, and then those included in the second set but not in the first one ($\div$).

As Figure 16 shows, comparisons between constraints are based on the type in the constraint's left-hand side. This is because constraints are always generated with a free type variable in its left-hand side. Definitions of the `compare` and `union` operations in Figure 16 ensure that every constraint set will never have two different constraints with the same left-hand side type variable. The only statement that generates subtyping constraints with a particular type on the left-hand side

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

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

FIGURE 15. The join algorithm.

(J-Compare)

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

$$compare(X_1{:}T_1, X_1{:}T_2)$$

(J-Union)

$$union(X{\leftarrow}T_1, X{\leftarrow}T_2) = X \leftarrow T_1{\vee}T_2$$

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

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

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

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

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

FIGURE 16. Comparison and union operations.

is the `return` statement. However, this statement cannot appear in a control flow statement because of the AST transformation described in Section 3.1.

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

The `union` function is also defined for constraints or type variable bindings generated in only one of the optional execution paths (last four axioms in Figure 16). The union of a single static subtyping constraint is the own constraint, because static typing must check every possible flow of execution. However, if the type variable is dynamic, there is no resulting constraint because it has been produced in a single optional execution path and, since it is dynamic, the constraint fulfillment is not mandatory. In assignment constraints and type variable bindings, the type to be bound is included in the right-hand side of the assignment. This means that the type variable will be bound to a new union type including the type it was previously bound to, because a new type could be assigned to the existing one in an optional control flow. As an example, the `next` field of the `list` variable (X_{17}) has the type $X_{17}{\vee}\texttt{int}$ in line

(T-Inv)

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

(T-FTVInv)

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

FIGURE 17. Function invocation.

10 of Figure 4. This implies that the type of `list2` in line 22 is converted from $\{data{:}X_{22}, next{:}X_{18}\}$ (being $\Gamma(X_{18}){:}\{data{:}X_{20}, next{:}X_{21}\}$) to $\{data{:}X_{22}, next{:}X_{18} \vee$ `int`$\}$. The result is that the `next` field is changed because one possible flow of execution in the `clearList` function may change its type to `int`.

3.2.7. Function Invocation

Figure 17 shows the two inference rules of function invocation. The difference is in the existence of free type variable arguments (T-Inv if there is no free type variable argument, and T-FTVInv otherwise). In both cases, the `shift` function takes a function type and returns an equivalent one, renaming the numbers of type variables to new fresh type variables. This process permits multiple invocations of the same function, creating new type variables for each function invocation. On each invocation, the types of the arguments are checked to be subtypes of the parameter types.

If no argument is a free type variable, constraints resolution is performed. The judgment $\Gamma_s \Vdash C, \Gamma$ means that under the Γ input environment, Γ_s is a *solution* for C; i.e., Γ_s holds all the substitutions to fulfill C under the Γ environment. In that case, the type of the function call is the substitution $\Gamma_{2n+2}(T)$, where Γ_{2n+2} is a solution for C. Under these circumstances, the C constraint set is solved and, hence, it is not included in the constraints generated by the function call. This shows how constraint resolution is part of the type inference process (it is not global, i.e., it is not performed after traversing the whole AST). If any argument is a free type variable, C is added to the constraint set produced by a function invocation expression (T-FTVInv).

The constraint resolution algorithm implemented is an adaptation of the algorithm defined by Aiken and Wimmers [28] that performs inclusion constraint resolution using union and intersection types. Its detailed description can be consulted in [36].

3.2.8. Converting Implicit to Explicit Types

Our language defines an automatic conversion of dynamic implicitly typed union types to explicit (particular) types (in assignments and function calls). When the union type is static, the subtyping rules described in Section 3.2.4 require types in the union type to promote to the explicit type. However, if the implicit type is dynamic, the conversion is too lenient because only one single promotion is necessary to allow the conversion. This is why a static promotion is forced in both assignments (rule T-Assign in Figure 13) and function invocations (rules T-Inv and T-FTVInv in Figure 17). As an example, the `din` variable (typed `dyn int`$\vee$`bool` in line 20 of Figure 12) is passed as an argument to the `increment` function that explicitly requires its `value` parameter to be `int`. Therefore, a compilation error is generated even though the argument is dynamic, because the `value` parameter is explicitly typed (and, hence, static).

4. ERASURE TRANSLATION

The objective of this section is twofold. First, to describe the translation templates used to generate code for the .Net platform employing the static type information gathered by the compiler. Second, based on the semantics of C# [37], to describe the erasure semantics of the minimal core of *StaDyn*.

The *StaDyn* core may be translated into C# following either of two implementation styles: first, by type-passing, augmenting the runtime system to carry information about type parameters; second, by erasure, removing all information about type parameters at runtime [31]. We have used the second approach, giving an erasure mapping from the *StaDyn* minimal core into C#. This style corresponds to the current implementation of *StaDyn*, which is compiled into the .Net platform by generating IL code (before the executable files), maintaining no information about type parameters at runtime—here we describe the translation to C# for simplicity. Figure 18 shows an example translation that will be used throughout this section. The *StaDyn* core source code is shown on the left, while the corresponding output C# program is displayed on the right.

The translation is performed traversing the AST. This traversal is performed after type checking, where the AST nodes were annotated with their types and a copy of the state of the type environment (Γ) and context (Ω) in the conclusion of each typing rule (written Γ_{node} and Ω_{node}).

4.1. Type Erasure

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

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

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

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

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

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


          din.y




                * din.z;
}
```

StaDyn core (source)

```
using System
class AC_int_x_int_y {
  public int x;public int y;
}
class AC_int_x_int_y_int_z {
  public int x; public int y; public int z;
}
public class MainClass {
  static AC_int_x_int_y point2D(int x, int y) {
    object _temp;
    return new AC_int_x_int_y {x=x, y=y};
  }
  static AC_int_x_int_y_int_z point3D(int x, int y, int z) {
    object _temp;
    return new AC_int_x_int_y_int_z {x=x, y=y, z=z};
  }
  static object point(int dim, int x, int y, int z) {

    object _temp;
    object result;
    if (dim == 2)
      result = point2D(x, y);
    else
      result = point3D(x, y, z);
    return result;
  }
  public static void Main() {
    object _temp;
    object sta;
    object din;
    sta = point(2, 0, 4, 3);
    din = point(3, 0, 4, 3);
    _temp = ((_temp=sta) is AC_int_x_int_y ?
                (int)(((AC_int_x_int_y)_temp).x) :
            (int)(((AC_int_x_int_y_int_z)_temp).x) ) +
            ((_temp=din) is AC_int_x_int_y ?
                (int)(((AC_int_x_int_y)_temp).y) :
            _temp is AC_int_x_int_y_int_z ?
                (int)(((AC_int_x_int_y_int_z)_temp).y) :
            (int)(_temp.GetType().GetField("y").GetValue(_temp))) *
            ((AC_int_x_int_y_int_z)din).z;
  }
}
```

C# (destination)

FIGURE 18. Example translation from *StaDyn* core to C#.

top type in C#, `object`, is the erased type that an expression should have.

$$|\texttt{int}|_\Gamma = \texttt{int} \qquad |\texttt{bool}|_\Gamma = \texttt{bool} \qquad |\texttt{void}|_\Gamma = \texttt{void}$$

$$|Array(T)|_\Gamma = |T|_\Gamma\,\texttt{[]}$$

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

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

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

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

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

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

FIGURE 19. Type erasure definition.

Figure 19 shows type erasures of the *StaDyn* minimal core. Function and intersection type erasures are not used in our translation rules, because our language does not support high-order functions, and intersection types only appear in constraints (no code is generated for them).

4.2. Anonymous Classes

As shown in Figure 19, an anonymous class ($\texttt{AC\_}|T_1|_\Gamma\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_\Gamma\texttt{\_}id_n$) is the type erasure of each different object structure. Since subtyping rules in our language require two objects to have the same structure (S-Object), we create a unique anonymous class for each object structure. To do so, the name of the anonymous class is the concatenation of each field name (lexicographically ordered) followed by its type erasure—arrays $T[\,]_1 \ldots [\,]_n$ are replaced with T_n because square brackets are not allowed in C# identifiers.

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

$$\frac{\text{AC\_}|T_1|_\Gamma\text{\_}id_1\text{\_}\ldots\text{\_}|T_n|_\Gamma\text{\_}id_n \notin classes}{}$$

```
⟦E = new {id1=E1, ..., idn=En}⟧AC(classes) ≜
        classes ← classes ∪ AC_|T1|ΓE_id1_..._|Tn|ΓE_idn
        class AC_|T1|ΓE_id1_..._|Tn|ΓE_idn {
            public |T1|ΓE |id1|ΓE ;
            ...
            public |Tn|ΓE |idn|ΓE ;
        }
    where ΓE1;ΩE1 ⊢ E1 : T1 ... ΓEn;ΩEn ⊢ En : Tn,
        id1 ... idn are lexicographically ordered, and
        in AC_|T1|Γ_id1_..._|Tn|Γ_idn,
        T[ ]1 ... [ ]n is replaced with T_n
```

Figure 18 shows how two anonymous classes (lines 2 to 7 of the C# code on the right) are created in the traversal of two `new` object nodes (lines 11 and 15 of the *StaDyn* core code on the left).

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

$$\frac{\text{AC\_}|T_1|_\Gamma\text{\_}id_1\text{\_}\ldots\text{\_}|T_n|_\Gamma\text{\_}id_n \notin classes}{}$$

```
⟦T = {id1:T1, ..., idn:Tn}⟧AC(classes) ≜
        classes ← classes ∪ AC_|T1|ΓT_id1_..._|Tn|ΓT_idn
        class AC_|T1|ΓT_id1_..._|Tn|ΓT_idn {
            public |T1|ΓT |id1|ΓT ;
            ...
            public |Tn|ΓT |idn|ΓT ;
        }
    where id1 ... idn are lexicographically ordered, and
        in AC_|T1|Γ_id1_..._|Tn|Γ_idn,
        T[ ]1 ... [ ]n is replaced with T_n
```

4.3. Translation of Programs

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

```
⟦P⟧program ≜  import System ;
              ⟦P⟧AC(∅)
              ⟦P⟧CG
```

Code generated for a program ($[\![P]\!]_{\mathrm{CG}}$) consists of a C# public class (`MainClass`) followed by two helper `_setValue` methods (explained in Sections 4.7 and 4.8). Each function is translated into a corresponding `static` C# method, and the main declarations and statements are placed inside the program's entry point (the C# `Main` method of the `MainClass`)—the example translation in Figure 18 omits the two `_setValue` methods.

```
⟦P = F1 ... Fn D1 ... Dm S1 ... Sl⟧CG ≜
    public class MainClass {
    private static object _setValue(object obj,
            string id, object value) {
      obj.GetType().GetField(id).SetValue(obj,value);
    return value;
    }
    private static object _setValue(Array array,
            object value, int index) {
      array.SetValue(value, index);
      return value;
    }
    ⟦F1⟧CG ... ⟦Fn⟧CG
    public static void Main() {
      object _temp;
      ⟦D1⟧CG ... ⟦Dm⟧CG
      statement(S1) ... statement(Sl)
    }
  }
```

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

Definition 4.1. *Given a statement node S, we define:*

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

4.4. Declarations

The .Net platform forces the declaration of each single variable with a unique type. We could simply declare variables and function parameters with their type erasures. However, this would generate many unnecessary casts. As an example, if a free type variable parameter is always used as an integer it is better to declare it as `int` rather than as `object`—its type erasure—(examples are the `x`, `y` and `z` parameters of the `point` function in Figure 18). This involves a faster execution because no cast will be generated.

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

Definition 4.2. *Given two environments* Γ_1 *and* Γ_2*, we define:*

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

Definition 4.3. *Given a type variable or identifier x, a type T, and an environment* Γ*, we define:*

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

To obtain all the possible types of a local variable, it is also necessary to know the actual C# types of the generated *global* functions. As an example, the `x`, `y` and `z` parameters in the `point` function (Figure 18) are only used in function invocations (lines 21 and 23). Since parameters of both `point2D` and `point3D` were declared as `int` in the C# destination code, the three `point` function parameters should also be declared as integers. Consequently, we define the *types* traversal not only returning the Γ of local variables, but also receiving the Γ that holds the type of every *global* function.

Once we obtain all the possible types of each local variable, we can pass them as a parameter to the translation process in order to optimize the generated C# code. Therefore, the $[\![\ ]\!]_{\mathrm{CG}}$ code generation function will from now on receive a Γ parameter. This parameter contains all the possible types of each local variable in the current scope, plus the C# types of the previously declared functions. We should then extend the code generation template for a program, adding the following code to the translation scheme shown above (the *statement* function—Definition 4.1—has also been extended with the appropriate Γ_{local} parameter):

```
$[\![P = F_1 \ldots F_n\ D_1 \ldots D_m\ S_1 \ldots S_l]\!]_{\mathrm{CG}} \triangleq$
    $\Gamma_{\mathrm{global}} \leftarrow \emptyset$
    $\Gamma_{\mathrm{local}_1} \leftarrow [\![F_1]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
    $[\![F_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}_1} \vee \Gamma_{\mathrm{global}})$
    $\ldots$
    $\Gamma_{\mathrm{local}_n} \leftarrow [\![F_n]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
    $[\![F_n]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}_n} \vee \Gamma_{\mathrm{global}})$
    $\Gamma_{\mathrm{local}_{\mathrm{main}}} \leftarrow [\![D_1 \ldots D_m\ S_1 \ldots S_l]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
    $[\![D_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}_{\mathrm{main}}} \vee \Gamma_{\mathrm{global}})$
    $\ldots$
    $[\![D_m]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}_{\mathrm{main}}} \vee \Gamma_{\mathrm{global}})$
    $\mathit{statement}(S_1, \Gamma_{\mathrm{local}_{\mathrm{main}}} \vee \Gamma_{\mathrm{global}})$
    $\ldots$
    $\mathit{statement}(S_l, \Gamma_{\mathrm{local}_{\mathrm{main}}} \vee \Gamma_{\mathrm{global}})$
```

We now define how types of local variables are obtained, i.e., the $[\![\ ]\!]_{\mathrm{types}}$ traversal. Local types in declarations and statements are the union (Definition 4.2) of the local environments they return.

```
$[\![D_1 \ldots D_m\ S_1 \ldots S_l\ R]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \triangleq$
    return $[\![D_1]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee \ldots \vee [\![D_m]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee$
        $[\![S_1]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee \ldots \vee [\![S_l]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee$
        $[\![R]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
```

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

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

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

```
$[\![\texttt{if}\ E\ S_1 \ldots S_n\ S_{n+1} \ldots S_{n+m}]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \triangleq$
    $\Gamma_{\mathrm{local}} \leftarrow [\![E]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
    if $\Gamma_{\mathrm{E}}; \Omega_{\mathrm{E}} \vdash E : X$
        $\mathit{add}(X, \mathrm{bool}, \Gamma_{\mathrm{local}})$
    return $\Gamma_{\mathrm{local}} \vee [\![S_1]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee \ldots$
        $\vee\ [\![S_n]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee [\![S_{n+1}]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee \ldots$
        $\vee\ [\![S_{n+m}]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$

$[\![\texttt{while}\ E\ S_1 \ldots S_n]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \triangleq$
    $\Gamma_{\mathrm{local}} \leftarrow [\![E]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$
    if $\Gamma_{\mathrm{E}}; \Omega_{\mathrm{E}} \vdash E : X$
        $\mathit{add}(X, \mathrm{bool}, \Gamma_{\mathrm{local}})$
    return $\Gamma_{\mathrm{local}} \vee [\![S_1]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) \vee ...$
        $\vee\ [\![S_n]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}})$

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

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

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

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

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

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

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

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

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

$[\![\texttt{true}]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) = [\![\texttt{false}]\!]_{\mathrm{types}}(\Gamma_{\mathrm{global}}) =$
```

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

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

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

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

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

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

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

4.5. Basic Expressions

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

```
$[\![\texttt{true}]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$   true
                                 return bool

$[\![\texttt{false}]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$  false
                                 return bool

$[\![IntLiteral]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$  $IntLiteral$
                                 return int

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

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

DEFINITION 4.4. *Given two type erasures T_1 and T_2, an expression node E, and an environment Γ, we define:*

```
$cast(T_1, T_2, E, \Gamma) \equiv$
    $((T_2)[\![E]\!]_{\text{CG}}(\Gamma))$   if $T_1 \neq T_2$ and $T_2 \neq$ object and
                          not($T_2{=}Array$ and $T_1{=}T([\,])^+$)
    $([\![E]\!]_{\text{CG}}(\Gamma))$        otherwise
```

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

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

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

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

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

```
$[\![E_1 \oplus E_2]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$
        $cast(T_1, \text{int}, E_1, \Gamma_{\text{local}})\ \texttt{op}_\oplus\ cast(T_2, \text{int}, E_2, \Gamma_{\text{local}})$
        return int
where   $T_1 = [\![E_1]\!]_{\text{CG}}(\Gamma_{\text{local}})$, $T_2 = [\![E_2]\!]_{\text{CG}}(\Gamma_{\text{local}})$,
        $\texttt{op}_+ = \texttt{+}$, $\texttt{op}_- = \texttt{-}$, $\texttt{op}_* = \texttt{*}$, and $\texttt{op}_/ = \texttt{/}$
```

At function invocation, each argument is converted to the corresponding parameter type. These parameter types are taken from the environment parameter (Γ_{local}). Therefore, the arguments may be cast to the actual C# types of the declared function. For instance, although the type erasure of the four parameters of the `point` function (Figure 18) is `object`, all of them were declared as integers. Therefore, arguments of any `point` function call should be cast to `int`, when necessary. The return type erasure follows the same process.

```
$[\![id(E_1 \ldots E_n)]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$
        $id(cast(T_1, |T_{p_1}|_{\Gamma_{\text{local}}}, E_1, \Gamma_{\text{local}}), \ldots,$
                $cast(T_n, |T_{p_n}|_{\Gamma_{\text{local}}}, E_n, \Gamma_{\text{local}}))$
        return $|T|_{\Gamma_{\text{local}}}$
where $T_1 = [\![E_1]\!]_{\text{CG}}(\Gamma_{\text{local}}), \ldots, T_n = [\![E_n]\!]_{\text{CG}}(\Gamma_{\text{local}})$, and
      $\Gamma_{\text{local}}(id) = T_{p_1} \times \ldots \times T_{p_n} \to T$
```

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

```
$[\![E_1$=$E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
        $[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$ = $cast(T_2, T_1, E_2, \Gamma_{\mathrm{local}})$
        return $T_1$
where $T_1 = [\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$, $T_2 = [\![E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$
```

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

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

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

4.6. Statements

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

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

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

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

4.7. Field Access

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

$$\frac{\Gamma_E; \Omega_E \vdash E : \{id_1{:}T_1, \ldots, id_n{:}T_n\}}{}$$

```
$[\![E\,.\,id_i]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
        $cast(T, |\{id_1{:}T_1, \ldots, id_n{:}T_n\}|_{\Gamma_E}, E, \Gamma_{\mathrm{local}})$ . $id_i$
        return $|T_i|_{\Gamma_E}$
where $T = [\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$
```

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

$$\frac{\Gamma_E; \Omega_E \vdash E : T \qquad T \in ftv(\Gamma_E) \text{ or } T = [..., id_i{:}T_i, ...]}{}$$

```
$[\![E\,.\,id_i]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
        (_temp=$[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$).GetType()
                    .GetField("$id_i$").GetValue(_temp)
        return object
```

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

$$\frac{\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : T \qquad T{\in}ftv(\Gamma_{E_1}) \text{ or } T{=}[..., id_i{:}T_i, ...]}{}$$

```
$[\![E_1\,.\,id_i$ = $E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$
    _setValue($[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})$,"$id_i$",$[\![E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{global}})$)
    return object
```

In the case of static union types, the generated code is optimized using the type information gathered statically. We use the ternary conditional operator to dynamically check the actual type from all the possible ones inferred by the compiler[3]. At runtime, this conditional code is significantly faster than reflection, which is the implementation of dynamic typing for both C# and Visual Basic [38, 39]. If the union type holds one (or more) free type variables, the last alternative in the conditional expression obtains the field value using reflection. Since this is the slowest alternative, we generate it as the last option in order to optimize runtime performance of the generated code.

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

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

An example of the previous code generation template can be seen in line 32 of Figure 18. The type of sta is $\{x{:}\texttt{int}, y{:}\texttt{int}\} \vee \{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}\}$. In the first iteration, the object expression (sta) is assigned to _temp and it is checked whether it is $\{x{:}\texttt{int}, y{:}\texttt{int}\}$. If so, a cast is performed and the x field is obtained. The second condition is similar, but asking for the $\{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}\}$ type. Since the union type does not hold any free type variable, reflection is not used in another last condition.

When the expression type is dynamic, it should be taken into consideration that there may be types in the union type that do not provide the expected field. The first optimization consists in generating code only for those types that accept the specific field access operation, using the ternary conditional operator. A performance benefit is obtained because the generated code only checks for those types that are applicable. The last alternative generated

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

is reflection. At runtime, if the field is still not found, a runtime exception will be thrown. This may happen when dynamic references are used, because it is not guaranteed that the field actually exists. Another final optimization is implemented when only one possible type fulfills the condition. In this case, a direct access to the field is generated (an `InvalidCastException` could be raised by the CLR).

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

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

Line 33 in Figure 18 is an example of accessing the `y` field of a dynamic union type. The ternary operator is the same as the previous field access (`sta.x`), but reflection is used in the last condition. We use reflection because the dynamic `din` reference may point to an object that does not implement the `y` field (it is a dynamic union type). Finally, line 34 generates faster code generating a direct cast because only one possible type in the union type ($\{x{:}\texttt{int}, y{:}\texttt{int}, z{:}\texttt{int}\}$) offers the `z` field.

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

4.8. Array Indexing

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

$$\frac{\Gamma_{E_1};\Omega_{E_1} \vdash E_1 : Array(T)}{}$$

```
$[\![E_1\,\texttt{[}E_2\texttt{]}]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq$
      $cast(T_1, T[\,], E_1, \Gamma_{\text{local}})$ [ $cast(T_2, \text{int}, E_2, \Gamma_{\text{local}})$ ]
      return $|T|_{\Gamma_{E_1[E_2]}}$
where $T_1 = [\![E_1]\!](\Gamma_{\text{local}})$, and $T_2 = [\![E_2]\!](\Gamma_{\text{local}})$
```

If the first expression is not an array, reflection is used (the `GetValue` method of the .Net's `Array` class). Notice that it cannot be a union of arrays because of the way we create union types (Section 4.5). In that case, the type would be an array of union types.

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

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

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

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

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

5. RUNTIME PERFORMANCE

We have evaluated the runtime performance of the *StaDyn* core presented in this paper, following the translation scheme described in Section 4.1. An assessment of the whole *StaDyn* implementation (that generates IL code instead of C#) can be consulted in [23].

5.1. Methodology

In order to assess the *StaDyn* core translation to C#, we have compared its 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 [14]. When dynamic code is used, the recently released *Dynamic Language Runtime* (DLR) is used to optimize the execution of dynamic code [40]. 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 [41]. 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 (whose abstract syntax is presented in Section 3.1). The source code is checked by the type system described in Section 3.2 and translated to C# 4.0 following the translation templates defined in Section 4.

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. A micro-benchmark to evaluate the influence of static type information gathered by the compiler. For this purpose, we have developed a synthetic micro-benchmark that takes the following scenarios into account:

   - Explicit static type declaration. No `var` references are used at all, explicitly stating the type of every variable.
   - Implicit dynamic 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 `list` with 10,000 random two and three dimensional points. The two `positiveX` and `closestToOrigin3D` functions are called passing the `list` reference as an argument.
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 beginning and the end of each benchmark to obtain the total execution time of each program.

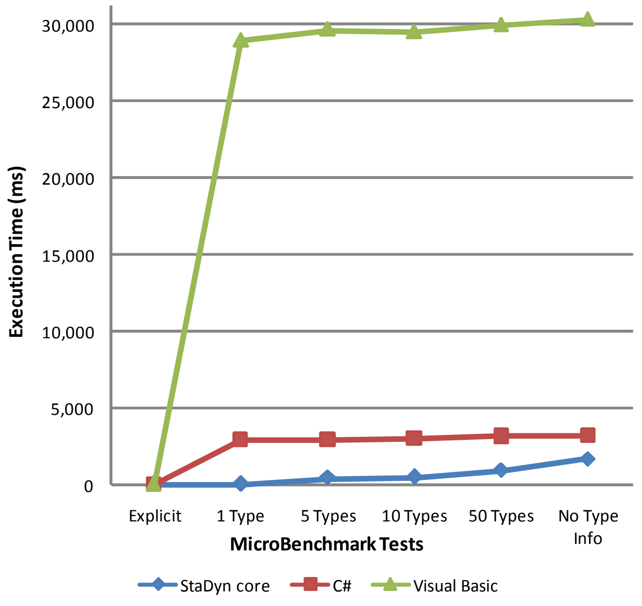

FIGURE 21. Execution time of the micro-benchmark.

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. 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.

5.2. 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 Points 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 generated IL code is almost the same). The performance assessment when the exact single type of `dyn var` references is inferred shows the repercussion of our approach. Runtime performance of *StaDyn* core is the same as using explicitly typed references (in fact, the generated code is precisely the same). In this special scenario, *StaDyn* shows a huge performance improvement. If the compiler infers the exact type of `dyn var` references, the *StaDyn* core is more than 1,252 times faster than VB and, in the same situation, 185 times faster than C# 4.0. 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 [38]. The difference between C# and VB shows the performance benefit of using the DLR in this scenario.

Figure 21 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

Benchmark		Test	*StaDyn* core	C#	Visual Basic
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		Points	309.62	1,859.27	4,921.87
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.

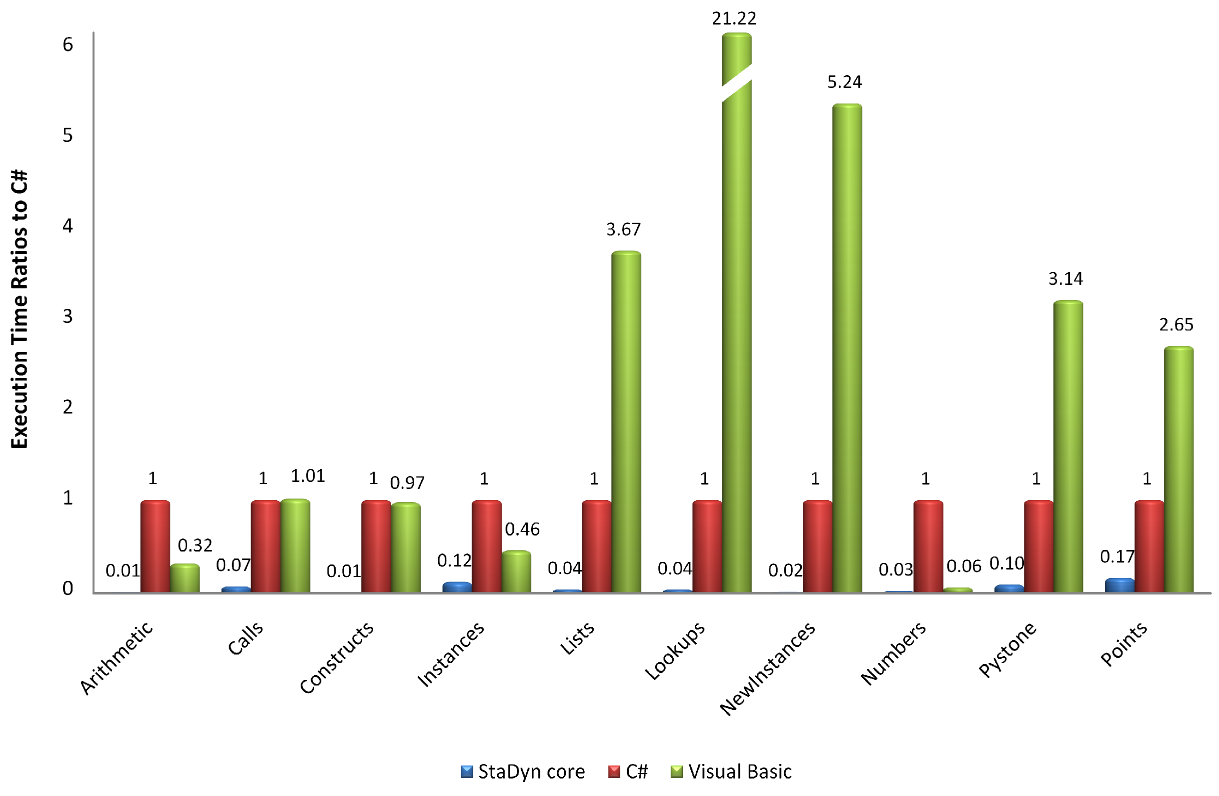

FIGURE 20. Execution time ratios to C#.

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 performance is the worst in the three programming languages. However, *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 20 shows the ratios of execution time to C# for the hybrid (Points) and dynamic typing benchmarks (Pybench and Pystone). Running hybrid code, the performance benefit is 500.5% and 1,489.65% compared to C# and VB respectively. This benefit increases as the number of dynamic references in the code grows: average benefit running the dynamic typing code is 3,106.29% (C#) and 3,381.3% (VB). Since *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 *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. One example is the *lookups* test that accesses to dynamic fields of an object, using another dynamic reference; under these circumstances, the VB implementation is extraordinary slower than C#.

6. RELATED WORK

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* [19].

One of the first approaches was *Soft Typing* [42], that applied static typing to a dynamically typed language such as Scheme. 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 [20] adds a `Dynamic` type to the lambda calculus, including two conversion operations (`dynamic` and `typecase`), generating a verbose code deeply dependent on its dynamism.

The works of *Quasi-Static Typing* [43], *Hybrid Typing* [44] and *Gradual Typing* [45] 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. Gradual typing already identified unification-based constraint resolution as a suitable approach to integrate both dynamic and static typing [46]. However, with gradual typing a dynamic type is always implicitly converted into static without any static type-checking, because type inference is not performed over dynamic references. 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.

The work developed by Wrigstad *et al.* allows the combination of dynamic and static typing in the *Thorn* programming language [47]. Thorn offers `like` types, an intermediate point between static and dynamic types [48]. Occurrences of `like` types variables are checked statically within their scope but, as they may be bound to dynamic values, their usage must be still checked at runtime. `like` types increase the robustness of the Thorn programming language, and programs developed using `like` types have been assessed to be about 3x and 6x faster than using dynamic types in the same programming language [48].

Although the *Just* programming language [49] does not combine dynamic and static typing, it added implicit type reconstruction to an explicitly typed language such as Java to obtain statically checked duck typing. The combination of syntax-directed and constraint-based type-checking allows the programmer to write generic code without defining class hierarchies [50]. This approach, however, does not consider methods that generate constraints (polymorphic methods) to invoke other polymorphic methods.

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 [51]. Some programming languages have taken the approach of adding a new dynamic type as proposed in [20] (`dynamic` in C# and Cobra, and `duck` in Boo), whereas others represent dynamic types by removing type annotations in variable declarations (VB and Dylan) [41]. Strongtalk follows a completely different approach based on the concept of *pluggable* type systems [52]. In these languages, dynamic types are implicitly coerced to static ones following the approach defined in [43] and [45], opposite to the explicit use of a conversion instruction like the `typecase` statement proposed by [20]. Since these implicit coercions may fail at runtime, a dynamic type-check is inserted in the generated code as described in [44].

There are also some works aimed at performing static type inference of dynamically typed languages to discover type errors before program execution. *Diamondback Ruby* (DRuby) is a tool that blends Ruby's dynamic type system with a static typing discipline [32]. When possible, DRuby infers static types to discover type errors in Ruby programs. In many cases, the DRuby programmer must annotate programs with types in order to obtain compile-time type errors. Since DRuby trusts annotations to be correct, improperly annotated code may cause run-time type errors, and these errors may be misleading. Anderson, Giannini and Drossopoulou formalized a subset of JavaScript (JS_0), defining a type inference algorithm that is sound with respect to a type system [53]. Therefore, programmers can benefit from the safety offered by the type system, without the need to write explicitly types in their programs. Different features of the JavaScript programming language such as dynamic removal of members or dynamic code evaluation are not supported. Neither of these works (DRuby and JS_0) use the statically inferred type information to optimize the generated code.

7. CONCLUSIONS

The *StaDyn* programming language combines static and dynamic typing in the very same programming language offering early type error detection, improved runtime performance, and direct interoperation between dynamically and statically typed code. The major contribution of *StaDyn* is that static type inference and type checking is performed by the compiler even over dynamic references, offering a high level of flexibility, and a better robustness and efficiency closer to static typing.

In order to formally describe the *StaDyn* programming language, we have reduced it to its minimal core. The key features of its type system are a new interpretation of union and intersection types, the combination of syntax-directed and constraint-based type-checking, type inference of implicitly-typed dynamic and static references, and flow-sensitive type-checking.

The type information gathered by the compiler is used to generate optimized C# code. When running dynamic languages benchmarks, the average runtime performance improvement has been 23 and 27 orders of magnitude compared to C# and VB respectively. When running hybrid static and dynamic typing code, the runtime performance benefit drops to 5 and 15 orders of magnitude. The lowest benefit is obtained, 89.95% (C#) and 1,714% (VB), when the compiler does not manage to infer any type information of dynamic references. Finally, the code generation scheme does not seem to involve any performance penalty, obtaining the same results when types are explicitly stated.

The C# implementation of the *StaDyn* minimal core, including a parser for its concrete syntax, its type system, the translation to C# described in Section 4, and all the examples and benchmarks used in this paper, are freely available at `http://www.reflection.uniovi.es/stadyn/download/2011/computerjournal`.

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

ACKNOWLEDGEMENTS

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] Thomas, D., Fowler, C., and Hunt, A. (2004) *Programming Ruby*, 2nd edition. Addison-Wesley Professional, Raleigh, North Carolina.

[2] Thomas, D., Hansson, D., Schwarz, A., Fuchs, T., Breed, L., and Clark, M. (2005) *Agile Web Development with Rails. A Pragmatic Guide.* Pragmatic Bookshelf, Raleigh, North Carolina.

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

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

[5] Crane, D., Pascarello, E., and James, D. (2005) *AJAX in Action.* Manning Publications, Greenwich, Connecticut.

[6] van Rossum, G., Fred, L., and Drake, J. (2003) *The Python Language Reference Manual.* Network Theory, United Kingdom.

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

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

[9] Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. (1996) Lua – an extensible extension language. *Software Practice & Experience*, **26**, 635–652.

[10] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007) The evolution of lua. *Proceedings of the conference on History of Programming Languages (HOPL)*, San Diego, California, 9-10 June, pp. 1–26. ACM.

[11] Hermann, J. The Pythius Web Site. `http://pythius.sourceforge.net`.

[12] Böllert, K. (1999) On weaving aspects. *Proceedings of the Workshop on Object-Oriented Technology*, Lisbon, Portugal, 14-18 June, pp. 301–302. Springer-Verlag.

[13] Ortin, F. and Cueva, J. M. (2004) Dynamic adaptation of application aspects. *Journal of Systems and Software*, **71**, 229–243.

[14] Torgersen, M. (2009) *New features in C# 4.0.* Microsoft Corporation, Redmond, Washington.

[15] Hugunin, J. (2007) Just glue it! Ruby and the DLR in Silverlight. *The MIX Conference*, Las Vegas, Nevada, 30 April - 7 May.

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

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

[18] Pierce, B. C. (2002) *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts.

[19] Meijer, E. and Drayton, P. (2004) Static typing where possible dynamic typing when needed: The end of the cold war between programming languages. *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages*, Vancouver, Canada, 24-28 October. ACM.

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

[21] Abadi, M., Cardelli, L., Pierce, B. C., and Rémy, D. (1995) Dynamic typing in polymorphic languages. *Journal of Functional Programming*, **5**, 111–130.

[22] Ortin, F. The StaDyn programming language. `http://www.reflection.uniovi.es/stadyn`.

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

[24] Foster, J., Terauchi, T., and Aiken, A. (2002) Flow-sensitive type qualifiers. *Proceedings of the*

Programming Language Design and Implementation (PLDI), Berlin, Germany, 17-19 June, pp. 1–12. ACM.

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

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

[27] Barbanera, F., Dezani-Ciancaglini, M., and De'Liguoro, U. (1995) Intersection and union types: syntax and semantics. *Information and Computation*, **119**, 202–230.

[28] Aiken, A. and Wimmers, E. L. (1993) Type inclusion constraints and type inference. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 9-11 June, pp. 31–41. ACM Press.

[29] Igarashi, A. and Nagira, H. (2006) Union types for object-oriented programming. *Proceedings of the Symposium on Applied Computing (SAC)*, Dijon, France, 23-27 April SAC '06, pp. 1435–1441. ACM.

[30] Lagorio, G. and Ancona, D. (2009) Coinductive type systems for object-oriented languages. In Drossopoulou, S. (ed.), *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Genova, Italy, 6-10 July, pp. 2–26. Springer-Verlag.

[31] Igarashi, A., Pierce, B. C., and Wadler, P. (2001) Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, **23**, 396–450.

[32] Furr, M., An, J.-h. D., Foster, J. S., and Hicks, M. (2009) Static type inference for Ruby. *Proceedings of the ACM symposium on Applied Computing (SAC)*, Honolulu, Hawaii, 9-12 March, pp. 1859–1866. ACM.

[33] Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*. Springer, Secaucus, NJ, USA.

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

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

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

[37] Börger, E., Fruja, N. G., Gervasi, V., and Stärk, R. F. (2005) A High-Level Modular Definition of the Semantics of C#. *Theoretical Computer Science*, **336**, 235–284.

[38] Ortin, F., Redondo, J. M., and Perez-Schofield, J. B. G. (2009) Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming*, **70**, 836–860.

[39] Redondo, J. M. and Ortin, F. (2008) Optimizing reflective primitives of dynamic languages. *International Journal of Software Engineering and Knowledge Engineering*, **18**, 759–783.

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

[41] Vick, P. (2007) *The Microsoft Visual Basic Language Specification*. Microsoft Corporation, Redmond, Washington.

[42] Cartwright, R. and Fagan, M. (1991) Soft Typing. *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, 26-28 June, pp. 278–292. ACM.

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

[44] Flanagan, C., Freund, S., and Tomb, A. (2006) Hybrid types, invariants, and refinements for imperative objects. *Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)*, San Antonio, Texas, 23 January. ACM.

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

[46] Siek, J. G. and Vachharajani, M. (2008) Gradual typing with unification-based inference. *Proceedings of the Dynamic Languages Symposium*, Paphos, Cyprus, 25 July, pp. 7:1–7:12. ACM.

[47] Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strnisa, R., Vitek, J., and Wrigstad, T. (2009) Thorn—robust, concurrent, extensible scripting on the JVM. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Orlando, Florida, 25-29 October, pp. 117–136. ACM.

[48] Wrigstad, T., Nardelli, F. Z., Lebresne, S., Östlund, J., and Vitek, J. (2010) Integrating typed and untyped code in a scripting language. *Proceedings of the 37th annual symposium on Principles of Programming Languages (POPL)*, Madrid, Spain, 17-23 January POPL '10, pp. 377–388. ACM.

[49] Lagorio, G. and Zucca, E. (2007) Just: Safe unknown types in java-like languages. *Journal of Object Technology*, **6**, 69–98.

[50] Lagorio, G. and Zucca, E. (2006) Introducing safe unknown types in java-like languages. *Proceedings of the Symposium on Applied Computing (SAC)*, Dijon, France, 23-27 April, pp. 1429–1434. ACM.

[51] Bierman, G., Meijer, E., and Torgersen, M. (2010) Adding dynamic types to c#. *Proceedings of the 24th European Conference on Object-Oriented Programming*, Maribor, Slovenia, 21-25 June ECOOP'10, pp. 76–100. Springer-Verlag.

[52] Bracha, G. (2004) Pluggable Type Systems. *Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, October. ACM.

[53] Anderson, C., Giannini, P., and Drossopoulou, S. (2005) Towards type inference for javascript.

Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Glasgow, UK, 9-11 June, pp. 428–452. Springer.

[54] Hunt, A. and Thomas, D. (2000) *Dylan programming: an object-oriented and dynamic language.* Addison Wesley Longman, Reading, Massachusetts.