The *StaDyn* core Language

Technical Report

Francisco Ortin

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

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

1. Introduction

Static typing is an unquestionably important tool for software development, offering the programmer advantages such as early type error detection, better documentation and abstraction, and more opportunities for compiler optimizations. Nevertheless, dynamically typed languages provide great flexibility at runtime, making them ideally suited for prototyping systems with changing or unknown requirements, or which interact with other systems that change unpredictably (data and application integration) [17].

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* [17]. One of the first approaches was *Soft Typing* [8], that applied static typing to a dynamically typed language such as Scheme. However, soft typing does not control which parts in a program are statically checked, neither is static type information used to optimize the generated code. The approach proposed in [1] adds a `Dynamic` type to lambda calculus, including two conversion operations (`dynamic` and `typecase`), generating a verbose code deeply dependent on its dynamism. The works of *Quasi-Static Typing* [26], *Hybrid Typing* [11] and *Gradual Typing* [25] perform implicit conversions between dynamic and

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

static code, employing subtyping relations in the case of quasi-static and hybrid typing, and a *consistency* relation in gradual typing. The main difference between these approaches and the work presented in this paper is that we perform type-checking even when dynamic types are used, detecting some type errors in dynamic code and, hence, improving its robustness.

Theoretical works on combining static with dynamic typing have been partially included in the implementation of programming languages such as Boo, Visual Basic (VB) .Net, Cobra, Dylan, Strongtalk, and the recently released C# 4.0. Some programming languages have taken the approach of adding a new dynamic type as proposed in [1] (`dynamic` in C# and Cobra, and `duck` in Boo), whereas others represent dynamic types by removing type annotations in variable declarations (VB and Dylan) [25]. Strongtalk follows a completely different approach based on the concept of *pluggable* type systems [7]. In these languages, dynamic types are implicitly coerced to static ones following the approach defined in [26] and [25], opposite to the explicit use of a conversion instruction like the `typecase` statement proposed by [1]. Since these implicit coercions may fail at runtime, a dynamic type-check is inserted in the generated code as described in [11].

This report describes the *StaDyn*[19] programming language, which supports both dynamic and static typing. It is described a minimal core of *StaDyn*, specifying its abstract syntax (2), type system (3) and erasure semantics by translating it into to C# (4).

2. Syntax

The first part of Figure 1 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. 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. 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 3.6).

Program	P	::=	$F^*\ D^*\ S^+$
Function	F	::=	$ST\ id\ \texttt{(}\ (ST\ id)^*\ \texttt{)}\ D^*\ S^*\ R^?$
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	::=	$id \mid id\ \texttt{(}\ E^*\ \texttt{)} \mid E \oplus E \mid E \otimes E \mid E \# E \mid E = E \mid E.id \mid$ $E[E] \mid \texttt{new}\ \{(id = (E \mid TV))^*\} \mid \texttt{new}\ ST\texttt{[}E\texttt{]}(\texttt{[]})^* \mid$ $\texttt{true} \mid \texttt{false} \mid \mathit{IntConstant}$
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	::=	$ST \mid \texttt{[}\ (id{:}T)^*\ \texttt{]} \mid ST \times \ldots \times ST \rightarrow ST \mid C^* \mid \{(id{:}T)^*\} \mid$ $Dyn^?\ T \vee \ldots \vee T \mid Dyn^?\ T \wedge \ldots \wedge T \mid \texttt{Array(}T\texttt{)}$
Constraints	C	::=	$T \leq T \mid TV \leftarrow T$

Figure 1: Abstract Syntax of the *StaDyn* minimal core.

The $\oplus$ operator represents arithmetic operations, $\otimes$ logical ones and $\#$ symbolizes relational operators. Objects are created following the syntax of the C# 3.0 feature of anonymous types [18]: between curly braces, there is listed a sequence of field identifiers followed by the assignment operator and an expression representing their initial values. The `new` expression for arrays creates one-dimensional arrays. Multidimensional arrays should be built in loops repeating the construction of one-dimensional arrays.

3. Type System

Types used to describe the *StaDyn* minimal core type system are shown in the second part of Figure 1. Syntax types are those that may be directly written by the programmer, whereas internal types 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. 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 *Sta-*

Dyn core type system. *StaDyn* (the whole programming language) does support method overriding by extending the method described in [16]: 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 [18]), the parser assigns them unique sequential numbers (X_i meta-variables 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 ($\texttt{[}(id\texttt{:}T)^*\texttt{]}$) represent the collection of fields an object may hold. Member types were introduced in constraints to define structural width coercion of object types to member types (S-OMember rule in Figure 10), because objects in *StaDyn* do not define width subtyping (S-Object in Figure 10). This subtyping relation is used in the constraint resolution algorithm when function calls are type-checked (T-Inv in Figure 16).

Type inference is specified with the general judgment $\Gamma; \Omega \vdash E : T \mid C; \Gamma'$, 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 [12], because type variables may change their types depending on the control flow [13].

Figure 2 shows another example program of our core language[1]. Elements of the environment and constraints generated are shown in the right part of the figure. X_i meta-variables range over type variables: whenever a new (`dyn`) `var` reference is analyzed by the parser, it creates a new type variable assigning it a unique sequential number. For example, in the scope

[1]We have added type inference information on the right side of some lines of code to later explain type inference.

```
01: var createNode(var data, var next) {                          Γ(data):X1, Γ(next):X2
02:   return new { data=data, next=next};
03: }                                  Γ(createNode):X1×X2→X3 | {data:X1,next:X2} ≤ X3
04: void setData(var node, var data) {                            Γ(node):X4, Γ(data):X5
05:   node.data = data;                                          X4 ≤ [data:X6], X6 ← X5
06: }                                Γ(setData): X4×X5→void | X4 ≤ [data:X6], X6 ← X5
07: void clearList(var list, bool clear) {                                   Γ(list):X7
08:   if (clear)
09:     list.next = 0;                                        X7 ≤ [next:X8], Γ(X8)←int
10: }            Γ(X8):X8∨int, Γ(clearList):X7xbool→void | X7 ≤ [next:X8], Γ(X8)←X8∨int
11: void main() {
12:   var list1;                                                             Γ(list1):X9
13:   var list2;                                                            Γ(list2):X10
14:   int increment;                                                    Γ(increment):int
15:   bool boolean;                                                     Γ(boolean):bool
16:   list1 = createNode(true, 0);       Γ(X9):{data:X11,next:X12},Γ(X11):bool, Γ(X12):int
17:   boolean = list1.data;
18:   list2 = createNode(boolean, list1);          Γ(X10):{data:X13,next:X9},Γ(X13):bool
19:   setData(list1, 3);                                                      Γ(X11):int
20:   increment = list2.next.data + 1;
21:   boolean = list1.data;       // * Error
22:   clearList(list2, false);                                 Γ(X10):{data:X13,next:X9∨int}
23: }
```

Figure 2: Example concrete program.

of the `main` function in Figure 2, Γ holds the assumptions Γ(`increment`):`int`, Γ(`list1`):X_9, and, in line 16, $\Gamma(X_9)$:{*data*:X_{11}, *next*:X_{12}}, $\Gamma(X_{11})$:`bool` and $\Gamma(X_{12})$:`int`. Since $\Gamma(X_{11})$:`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 3.6).

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.

We also define two kinds of constraints (the last part of Figure 1). Subtyping constraints ($IT \leq IT$) require the type on the left to be a subtype of (promote to) the type on the right. Assignment constraints ($TV \leftarrow IT$) 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,

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

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

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

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

Figure 3: Program, declarations and functions.

when function invocation expressions are checked. In line 5 of Figure 2, a subtyping constraint is generated for the `node` variable; it should be an object with a `data` field, i.e., subtype of a member type: $X_4 \leq [data{:}X_6]$. 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: $X_6 \leftarrow X_5$. When the `setData` function is invoked, the `data` type of the `node` argument X_6 will be assigned the type of the `data` parameter X_5. This is the reason why X_{11} is then bound to `int` in line 19.

3.1. Functions

We use $\diamond$ to denote well-formedness. Inference rules in Figure 3 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 $T_1 \times \dots \times T_n \to T \mid C$, 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 2 (line 6) has the two constraints $X_4 \leq [data{:}X_6]$ and $X_6 \leftarrow X_5$. The rest of the rules in Figure 3 generate no constraint at all, and output environments become the input of the following terms, obtaining a flow-sensitive type checking.

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

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

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

Figure 4: Inference rules for Ω.

3.2. Context

It is necessary to store information regarding a function in order to subsequently perform type checking of the terms in the function scope. This information is saved in the function context (Ω) by means of T-FUNC (Figure 3) and the rules shown in Figure 4. 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 3) adds parameter types to $\Omega_{.\texttt{tifp}}$; in Figure 4, $\Omega_{.\texttt{TIFP}}$-FIELD inserts field types in $\Omega_{.\texttt{tifp}}$ whenever an object type is in $\Omega_{.\texttt{tifp}}$; $\Omega_{.\texttt{TIFP}}$-ARRAY and $\Omega_{.\texttt{TIFP}}$-INV do the same with arrays and function calls, respectively. Notice that not only type variables are inserted in $\Omega_{.\texttt{tifp}}$, because objects may hold type variables in their fields indirectly, as may arrays, in their elements.

3.3. 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 3.5 and 3.7, respectively.

Figure 5 shows inference rules that type-check variables. The `tv` predicate tests whether a type is a type variable or not, and the `ftv` function returns the set of unbound type variables in an environment. T-VAR types a variable previously declared, when its type is not a type variable. When the type of an identifier is a type variable and it is bound to another type, T-BVAR types the identifier to the type bound to the type variable. This happens,

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

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

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

Figure 5: Variables.

for instance, in line 17 of Figure 2, where the type variable of `list1` (X_9) was previously bound to the object type $\{data{:}X_{11}, next{:}X_{12}\}$ 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 `v` parameter in lines 8 and 9 of Figure 6 is allowed because, despite its elements type being a free type variable, it is contained in $\Omega_{.\text{params}}$. `temp` can be used in line 9 because, although its type is not in $\Omega_{.\text{tifp}}$, it is in the left-hand side of an assignment.

Figure 7 shows the T-ARITH inference rule of arithmetic expressions (for the sake of brevity, relational and logical expressions are not shown). Operands of arithmetic and relational expressions must be subtypes of `int`; logical expressions should promote to `bool`. Output environments are used as the input environments of the subsequent expressions, the one returned by the whole expression being the last output 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. Lines 46 to 49 in Figure 6 are different examples of this inference rule. In a similar way, the T-NARRAY rule types the array construction expressions. The expression

[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.

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

Figure 6: Example coded in the minimal core of *StaDyn*.

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 expression in line 45 (Figure 6) is $Array(X)$, X 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 [2] (rule S-OMember in Figure 10). Moreover, if the object field is a free type variable not inferred from the parameters, i.e., not in $\Omega_{.\text{tifp}}$, it must be a direct left child of an assignment expression. Line 8 in Figure 6 is an example of a correct term. Although the `x` field of the `v[j]` object is a free type variable (X_4) and it is not the left-hand side of an assignment, its type is in $\Omega_{.\text{tifp}}$.

T-Array requires the first expression to be a subtype of an array, and

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

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

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

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

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

Figure 7: Basic expressions.

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 8. The type of the elements of the `v` array is the free type variable X_{22}, not inferred from the parameters (`v` is a local variable), producing a compilation error because no value has been assigned to it.

3.4. Subtyping

Judgments in subtyping rules ($\Gamma \vdash T_1 \leq T_2 \mid C;\Gamma'$) 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 9 types any expression to the type which is bound to the expression type variable. The output environment is used to bind a type variable to a type, when the type variable must be a subtype of a particular type. This is precisely the behavior of the S-FTVL and S-FTVR rules in Figure 10 that, in addition, generates a subtyping constraint. The subtyping condition will be true if the

```
01: void vector(var[] w) {                                  Γ(w):Array(X20)
02:   var[] v;                                              Γ(v):Array(X21)
03:   int a;
04:   v = new var[2];                                       Γ(X21):X22
05:   a = v[3]; // * Error
06:   v[0] = w[0] = 0;                          Γ(X22):int, Γ(X20):X20∨int
07:   v[1] = w[1] = true;              Γ(X22):int∨bool, Γ(X20):X20∨int∨bool
08: }
09: void main() {
10:   var[] ve;                                             Γ(ve):Array(X23)
11:   ve = new var[3];                                      Γ(X23):X24
12:   ve[2] = new { attribute = 3 };          Γ(X24):{attribute:int}
13:   vector(ve);                       Γ(X24):{attribute:int}∨int∨bool
14: }
```

Figure 8: Example use of arrays.

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

Figure 9: Type variable binding substitution.

constraint is fulfilled. When both type variables are not bound to any type, a subtyping constraint is produced (S-FTVs in Figure 10).

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

Member types were introduced for structural subtyping of objects. An object type is a subtype of a member type when the former has all the members of the latter, and their corresponding types are structurally equivalent (S-OMember). This rule is necessary in the fulfillment of subtyping constraints of function invocation (Section 3.7). As an example, the `setData` function in Figure 2 has the $X_4 \leq [data : X_6]$ constraint (line 6). When the function is called in line 19 passing the $\{data : X_{11}, next : X_{12}\}$ object as the first argument, the S-OMember subtyping rule confirms that the argument promotes to the parameter.

Subtyping rules for to union and intersection types are an enhancement

(S-Bool)
$$\Gamma \vdash \texttt{bool} \leq \texttt{bool} \mid \emptyset; \Gamma$$

(S-Int)
$$\Gamma \vdash \texttt{int} \leq \texttt{int} \mid \emptyset; \Gamma$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 10: Subtyping and type equivalence.

```
01:  var getElement(var list, var fstOrSnd) {
02:    var element;
03:    if (fstOrSnd)
04:      element = list.data;
05:    else
06:      element = list.next.data;
07:    return element;}
08:  }
09:  int increment(int value) {
10:    return value + 1;
11:  }
```

```
12:  void main() {
13:    int integer;
14:    var listOfTwo, sta;
15:    dyn var din;
16:    listOfTwo = createNode(1,createNode(true,0));
17:    din = sta = getElement(listOfTwo, true);
18:    integer = sta + 1;     // * Error
19:    integer = din + 1;
20:    increment(din);        // * Error
21:  }
```

Figure 11: Example use of dynamic and static references.

of the ones defined by other authors such as [23] and [10] (S-SUnionL, S-SUnionR, S-SInterR and S-SInterL), adding new dynamic typing rules (S-DUnionL and S-DInterR). 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)—$\cup\Gamma_i$ 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 11. The type of both `sta` and `din` variables is `int`$\vee$`bool`, but `sta` is static whereas `din` is dynamic. This difference prevents the arithmetic operation in line 18 from compiling (the plus operator cannot be applied to a `bool`), while it is correct in line 19 (addition is defined for integers).

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 the type is a subtype of at least one of the types in the intersection type (rule S-DInterR).

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

Since *StaDyn* does not yet support higher-order functions (*delegates* in

C# terminology), we do not specify subtyping of the function type constructor. Subtyping is not defined between member types either, because they only appear in constraints. Therefore, the current definition of the subtyping relation is neither reflexive nor transitive.

3.5. Assignments

The abstract syntax in Figure 1 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 12 describe assignment expressions. T-Assign types assignment expressions when the left-hand side expression type is not a type variable. This straightforward rule only requires the right-hand side to be a subtype of the left-hand side. In case the left-hand side is a type variable, 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 2 ($X_6 \leftarrow X_5$). Calling this function with an object as an argument (line 19) changes the type of the argument's `data` to `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 8 is an array of integers or booleans (`int`$\vee$`bool`) because it holds both. If the type variable has been inferred from the function parameters, a new assignment constraint will be generated including the own type variable in the right-hand side of the assignment. This is the case of the `w` variable in the `vector` function (line 8 in Figure 8). Unlike the type of `v`, the type of `w` (X_{20}) is included in the union type ($X_{20} \vee$ `int` $\vee$ `bool`), denoting that the type of the actual parameter in the invocation will be included in the union

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

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

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

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

Figure 12: Assignments.

type. Therefore, the type of the elements of `ve` when the function `vector` is called in line 13 is {`attribute`:`int`}∨`int`∨`bool`.

3.6. Statements

The minimal core includes the `return`, `if` and `while` statements (Figure 13). For the `return` statement, the expression type to be returned must be a subtype of the declared return type (T-Return).

Control-flow branches of `if` and `while` statements are taken into consideration to keep the flow-sensitiveness of our type system. The `join` of constraints and the union of type environments take into consideration this difficulty, taking the type information obtained on each execution path and combining both into a single constraint list and type environment. Each parameter represents the type information of an exclusive control flow. Since it might happen that the `while` body is not executed at runtime (it is not exclusive), the empty set is passed as its second argument (T-While). An example use of the `join` function is the type of the `result` variable ($X \vee$ `int`)—line 40 in Figure 6—that is created from its types in lines 37 (X) and 39 (`int`). The same happens with constraints: the joined constraint for the `v` parameter of the `nearestToOrigin3D` function ($X_1 \leq [dimensions{:}X_2] \wedge [x{:}X_3] \wedge [y{:}X_4] \wedge [z{:}X_5]$) is obtained from the constraints generated in lines 27 ($X_1 \leq [dimensions{:}X_2]$) and 28 ($X_1 \leq [x{:}X_3] \wedge [y{:}X_4] \wedge [z{:}X_5]$).

The union of environments used in Figure 13 is also based on the `join` function described in Figure 14: variable bindings must be the same in both

environments, and the resulting type variable binding set is the `join` of the type variable binding sets of each flow path.

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

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

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

Figure 13: Statements.

Figure 14 shows the algorithm used to implement the `join` function. Each set holds either constraints (subtyping and assignment) or type variable bindings ($X{:}T$ in environments). The algorithm has been defined employing the `compare` and `union` operations defined by the axioms in Figure 15. The algorithm takes elements of both sets, adding new union and intersection types [23] 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 ($\div$).

```
join(Set1, Set2) ≡ Set in
  Set ← ∅
  ∀ elem1 ∈ Set1
    if ∃ elem2 ∈ Set2, compare(elem1, elem2)
      Set ← Set ∪ union(elem1, elem2)
    else
      Set ← Set ∪ union(elem1)
  ∀ elem ∈ Set2 ÷ Set1
    Set ← Set ∪ union(elem)
```

```
Set1 ÷ Set2 ≡ Set in
  Set ← ∅
  ∀ elem1 ∈ Set1
    if ∄ elem2 ∈ Set2, compare(elem1, elem2)
      Set ← Set ∪ elem1
```

Figure 14: The join algorithm.

As Figure 15 shows, comparisons between constraints are based on the type in the constraint's left-hand side. This is because constraints are always generated with a free type variable in its left-hand side. Definitions of the

(J-Compare)
$compare(X_1 \leftarrow T_1, X_1 \leftarrow T_2)$ $compare(X_1 \leq T_1, X_1 \leq T_2)$ $compare(X_1 : T_1, X_1 : T_2)$

(J-Union)
$union(X \leftarrow T_1, X \leftarrow T_2) = X \leftarrow T_1 \vee T_2$ $union(\texttt{sta}\ X \leq T_1, \texttt{sta}\ X \leq T_2) = X \leq \texttt{sta}\ T_1 \wedge T_2$

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

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

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

Figure 15: Comparison and union operations.

`compare` and `union` operations in Figure 15 ensure that every constraint set will never have two different constraints with the same left-hand side type variable. The only statement that generates subtyping constraints with a particular type on the left-hand side is the `return` statement. However, this statement cannot appear in a control flow statement because of the AST transformation described in 2.

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

The `union` function is also defined for constraints or type variable bindings generated in only one of the optional execution paths (last four axioms in Figure 15). The union of a single static subtyping constraint is the own constraint, because static typing must check every possible flow of execution. However, if the type variable is dynamic, there is no resulting constraint because it has been produced in a single optional execution path and, since it is dynamic, the constraint fulfillment is not mandatory. In assignment constraints and type variable bindings, the type to be bound is included in the right-hand side of the assignment. This means that the type variable will be bound to a new union type including the type it was previously bound to, because a new type could be assigned to the existing one in an optional control flow. As an example, the `next` field of the `list` variable (X_8) has the type $X_8 \vee \texttt{int}$ in line 10 of Figure 2. This implies that the type of `list2` in line 22 is converted from $\{data{:}X_{13}, next{:}X_9\}$ (being $\Gamma(X_9){:}\{data{:}X_{11}, next{:}X_{12}\}$) to $\{data{:}X_{13}, next{:}X_9 \vee \texttt{int}\}$—the `next`

(T-Inv)

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

(T-FTVInv)

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

Figure 16: Function invocation.

field type is then $\{data{:}X_{11}, next{:}X_{12}\} \vee \texttt{int}$. The result is that the `next` field, whose type before the invocation was $\{data{:}X_{11}, next{:}X_{12}\}$, is changed to $\{data{:}X_{11}, next{:}X_{12}\} \vee \texttt{int}$ because one possible flow of execution in the `clearList` function may change its type to `int`.

3.7. Function Invocation

Figure 16 shows the two inference rules of function invocation. The difference is in the existence of free type variable arguments (T-Inv if there is no free type variable argument, and T-FTVInv otherwise). In both cases, the `shift` function takes a function type and returns an equivalent one, renaming the numbers of type variables to new fresh type variables. This process permits multiple invocations of the same function, creating new type variables for each function invocation. On each invocation, the types of the arguments are checked to be subtypes of the parameter types.

If no argument is a free type variable, constraints resolution is performed. The judgment $\Gamma_s \Vdash C, \Gamma$ means that under the 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 [3] that performs inclusion con-

straint resolution using union and intersection types. Its detailed description can be consulted in [20].

3.8. 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 3.4 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 12) and function invocations (rules T-Inv and T-FTVInv in Figure 16). As an example, the `din` variable (typed `dyn int`$\vee$`bool` in line 20 of Figure 11) is passed as an argument to the `increment` function that explicitly requires its `value` parameter to be `int`. Therefore, a compilation error is generated even though the argument is dynamic, because the `value` parameter is explicitly typed (and, hence, static).

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# [6], 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 [15]. 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 17 shows an example translation that will be used throughout this section. The *StaDyn* core source code is shown on the left, while the 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

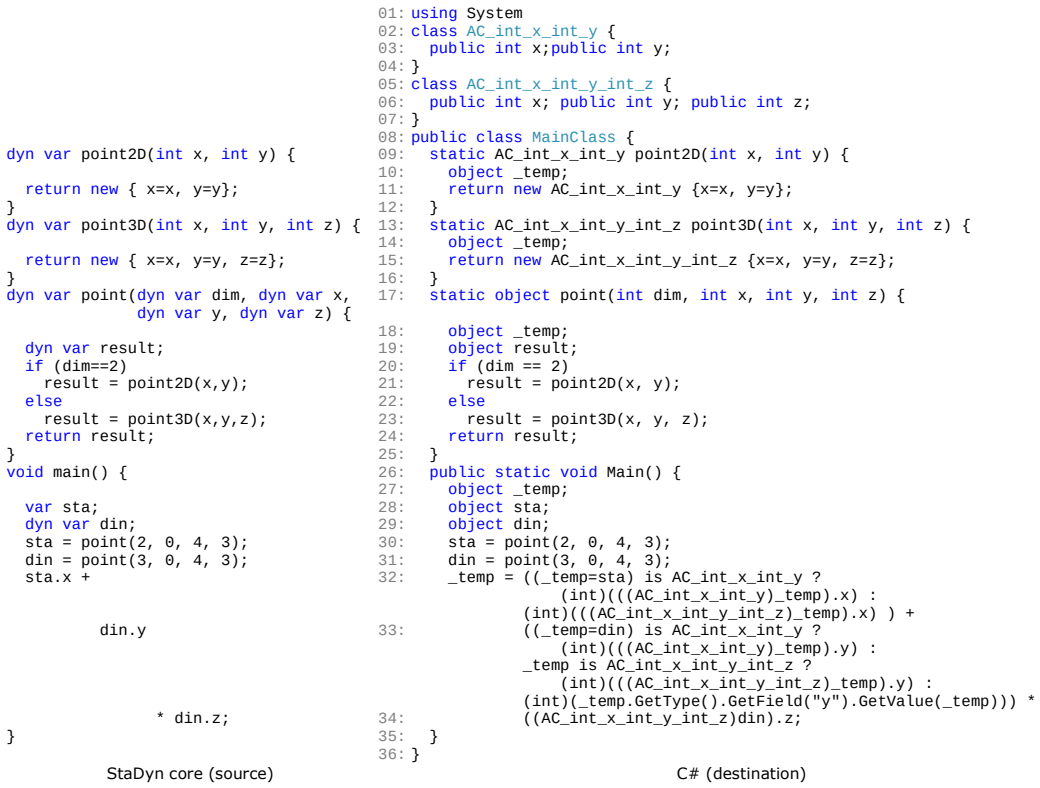

Figure 17: Example translation from *StaDyn* core to C#.

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

Figure 18 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).

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

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

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

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

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

where $id_1 \ldots id_n$ are lexicographically ordered, and

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

Figure 18: Type erasure definition.

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

4.2. Anonymous Classes

As shown in Figure 18, 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 ($[\![\ ]\!]_{\mathrm{AC}}$), 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.

$$\frac{\mathrm{AC\_}|T_1|_\Gamma\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_\Gamma\texttt{\_}id_n \notin classes}{[\![E = \texttt{new}\ \{id_1\texttt{=}E_1, \ldots, id_n\texttt{=}E_n\}]\!]_{\mathrm{AC}}(classes) \triangleq \\ \quad classes \leftarrow classes \cup \mathrm{AC\_}|T_1|_{\Gamma_E}\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_{\Gamma_E}\texttt{\_}id_n \\ \quad \texttt{class AC\_}|T_1|_{\Gamma_E}\texttt{\_}id_1\texttt{\_}\ldots\texttt{\_}|T_n|_{\Gamma_E}\texttt{\_}id_n\ \{}$$

```
        public |T_1|_{Γ_E} |id_1|_{Γ_E} ;
        ...
        public |T_n|_{Γ_E} |id_n|_{Γ_E} ;
    }
```
where $\Gamma_{E_1};\Omega_{E_1} \vdash E_1 : T_1 \ \ldots \ \Gamma_{E_n};\Omega_{E_n} \vdash E_n : T_n$,
$id_1 \ldots id_n$ 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

Figure 17 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).

Finally, an anonymous class declaration is generated when an object type is used and its class has not been previously declared.

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

$[\![T = \{id_1{:}T_1, \ldots, id_n{:}T_n\}]\!]_{\text{AC}}(classes) \triangleq$
$classes \leftarrow classes \cup \text{AC\_}|T_1|_{\Gamma_T}\text{\_}id_1\text{\_}\ldots\text{\_}|T_n|_{\Gamma_T}\text{\_}id_n$
```
class AC_|T_1|_{Γ_T}_id_1_..._|T_n|_{Γ_T}_id_n {
    public |T_1|_{Γ_T} |id_1|_{Γ_T} ;
    ...
    public |T_n|_{Γ_T} |id_n|_{Γ_T} ;
}
```
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

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 ($[\![\ ]\!]_{\text{AC}}$) (passing an empty set of classes) and the final generation of code ($[\![\ ]\!]_{\text{CG}}$).

$[\![P]\!]_{\text{program}} \triangleq$
```
import System ;
```
$[\![P]\!]_{\text{AC}}(\emptyset)$
$[\![P]\!]_{\text{CG}}$

Code generated for a program ($[\![P]\!]_{\text{CG}}$) 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 17 omits the two `_setValue` methods.

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

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

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

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

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

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

```
}
}
```

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

Definition 1. *Given a statement node S, we define:*

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

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 17). This involves a faster execution because no cast will be generated.

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

Definition 2. *Given two environments* Γ_1 *and* Γ_2*, we define:*

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

Definition 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 17) 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 C# code generated. Therefore, the $[\![\ ]\!]_{\text{CG}}$ 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 1—has also been extended with the appropriate Γ_{local} parameter):

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

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 2) of the local environments they return.

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

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

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

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

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

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

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

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

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

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

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

$[\![E_1\,\texttt{[}E_2\texttt{]}]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\quad \Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee [\![E_2]\!]_{\text{types}}(\Gamma_{\text{global}})$

if $\Gamma_{E_1}; \Omega_{E_1} \vdash E_1 : X_1$
$add(X_1, Array(T), \Gamma_{\text{local}})$
if $\Gamma_{E_2}; \Omega_{E_2} \vdash E_2 : X_2$
$add(X_2, \text{int}, \Gamma_{\text{local}})$
return Γ_{local}
where $\Gamma_{E_1[E_2]}; \Omega_{E_1[E_2]} \vdash E_1[E_2] : T$

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

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

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

$[\![\texttt{true}]\!]_{\text{types}}(\Gamma_{\text{global}}) = [\![\texttt{false}]\!]_{\text{types}}(\Gamma_{\text{global}}) =$
$[\![IntConstant]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$ 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.

$[\![id\texttt{(}E_1 \ldots E_n\texttt{)}]\!]_{\text{types}}(\Gamma_{\text{global}}) \triangleq$
$\Gamma_{\text{local}} \leftarrow [\![E_1]\!]_{\text{types}}(\Gamma_{\text{global}}) \vee \ldots \vee [\![E_n]\!]_{\text{types}}(\Gamma_{\text{global}})$
$\forall\ i \in [1, n]$
if $\Gamma_{E_i}; \Omega_{E_i} \vdash E_i : X_i$
$add(X_i, T_i, \Gamma_{\text{local}})$
return Γ_{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.

$[\![D = T\ id]\!]_{\text{CG}}(\Gamma_{\text{local}}) \triangleq |T|_{\Gamma_{\text{local}}}\ id$;
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.

$$
\begin{array}{ll}
[\![F = T\ id(T_1\ id_1 \ldots T_n\ id_n)\ D_1 \ldots D_m\ S_1 \ldots S_l\ R]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq & \\
\quad \texttt{static}\ |T'|_{\Gamma_{\mathrm{local}}}\ id(|T'_1|_{\Gamma_{\mathrm{local}}}\ id_1, \ldots, |T'_n|_{\Gamma_{\mathrm{local}}}\ id_n)\ \{ & \\
\quad\quad \texttt{object\ \_temp;} & \\
\quad\quad [\![D_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \ldots\ [\![D_m]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) & \\
\quad\quad \mathit{statement}(S_1, \Gamma_{\mathrm{local}}) \ldots\ \mathit{statement}(S_l, \Gamma_{\mathrm{local}}) & \\
\quad\quad [\![R]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) & \\
\quad \} & \\
\text{where} \quad \Gamma_{\mathrm{local}}; \Omega_F \vdash id_1 : T'_1, \ldots, \Gamma_{\mathrm{local}}; \Omega_F \vdash id_n : T'_n, \text{ and} & \\
\quad \Gamma_{\mathrm{local}}; \Omega_F \vdash id : T_{p_1} \times \ldots \times T_{p_n} \to T' &
\end{array}
$$

4.5. Basic Expressions

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

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

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

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

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

$$
\mathit{cast}(T_1, T_2, E, \Gamma) \equiv \begin{cases} \texttt{((}T_2\texttt{)(}[\![E]\!]_{\mathrm{CG}}(\Gamma)\texttt{))} & \text{if } (T_1 \neq T_2 \text{ and } T_2 \neq \texttt{object}) \text{ or} \\ & \quad (T_2 = \mathit{Array} \text{ and } T_1 \neq T([\,])^+) \\ \texttt{(}([\![E]\!]_{\mathrm{CG}}(\Gamma))\texttt{)} & \text{otherwise} \end{cases}
$$

The *cast* function generates code for the E expression including a cast when necessary. In case the types are the same, or the destination is `object`,

or an array type is casted 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:

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

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 code generated does not perform extra type checking at runtime because it is already done by the CLR.

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

At function invocation, each argument is converted to the corresponding parameter type. These parameter types are taken from the environment parameter (Γ_{local}). Therefore, the arguments may be casted to the actual C# types of the declared function. For instance, although the type erasure of the four parameters of the `point` function (Figure 17) is `object`, all of them were declared as integers. Therefore, arguments of any `point` function call should be casted to `int`, when necessary. The return type erasure follows the same process.

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

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

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

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

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

$$
\begin{array}{l}
[\![E = \texttt{new } T\texttt{[}E_1\texttt{] [] } \ldots \texttt{ []}]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq \\
\qquad \texttt{new } |T|_{\Gamma_E}\texttt{ [}[\![E_1]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}})\texttt{] [] } \ldots \texttt{ []} \\
\qquad \text{return } |T|_{\Gamma_E}\texttt{[ ][ ]} \ldots \texttt{[ ]}
\end{array}
$$

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.

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

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

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

4.7. Field Access

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

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

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

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

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

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

$[\![E_1\,.\,id_i = E_2]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$ `_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 code generated 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 VB [21]. 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.

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

$[\![E\,.\,id]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) \triangleq$

$\forall\, i \in [1,n], T_i \notin ftv(\Gamma_E)$

if *it is not the last iteration* or $\exists\, j \in [1,n], T_j \in ftv(\Gamma_E)$

$$\begin{cases} \texttt{\_temp=}[\![E]\!]_{\mathrm{CG}}(\Gamma_{\mathrm{local}}) & \text{if } \textit{it is the first iteration} \\ \texttt{: \_temp} & \text{otherwise} \end{cases} \quad \texttt{is}\ |T_i|_{\Gamma_E}\ \texttt{?}$$

$\texttt{(}|T|_{\Gamma_{E.id}}\texttt{)((}|T_i|_{\Gamma_E}\texttt{)\_temp).}id$

if $\exists\, i \in [1,n], T_i \in ftv(\Gamma_E)$

$\texttt{:(}|T|_{\Gamma_{E.id}}\texttt{)(\_temp.GetType().GetField("}id\texttt{").GetValue(\_temp))}$

return $|T|_{\Gamma_{E.id}}$

An example of the previous code generation template can be seen in line 32 of Figure 17. 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 code

[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.

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

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

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

Line 33 in Figure 17 is an example of accessing the `y` field of a dynamic union type. The ternary operator is the same as the previous field access (`sta.x`), but reflection is used in the last condition. We use reflection because the dynamic `din` reference may point to an object that does not implement the `y` field (it is a dynamic union type). Finally, line 34 generates faster code generating a direct cast because only one possible type in the union type ($\{x$:`int`$, y$:`int`$, z$:`int`$\}$) offers the `z` field.

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

4.8. Array Indexing

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

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

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

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.

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

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

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.

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

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

References

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

[2] M. Abadi, L. Cardelli, A Theory of Objects, Springer, Berlin, 1996.

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

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

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

[6] E. Börger, N.G. Fruja, V. Gervasi, R. F. Stärk, A High-Level Modular Definition of the Semantics of C#, Theoretical Computer Science 336(2-3) (2005) 235-284.

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

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

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

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

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

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

[13] M. Furr, A. Jong-Hoon, J.S. Foster, M. Hicks, Static Type Inference for Ruby, in: ACM Symposium on Applied Computing, 2009, pp. 1859–1866.

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

[15] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java - A Minimal Core Calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23(3) (2001) 396–450.

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

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

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

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

[20] F. Ortin, The StaDyn core Type System, Technical Report, Computer Science Department, University of Oviedo, August 2009, http://www.reflection.uniovi.es/stadyn/publications/stadyn.core.type.system.pdf

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

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

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

[24] J.C. Reynolds, Design of the programming language FORSYTHE, ALGOL-like Languages, Volume 1, 1997.

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

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

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

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

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