The *StaDyn* Core Type System

Technical Report

Francisco Ortin

Computer Science Department,
University of Oviedo,
Calvo Sotelo s/n,
33007, Oviedo, Spain,
ortin@lsi.uniovi.es

Technical Report

August 24, 2009

Abstract

Although static typing provides early type error detection, better legibility and abstraction, and more opportunities for compiler optimizations, dynamic typing supplies a high degree of runtime adaptiveness, providing an agile and interactive development suitable for rapid prototyping. In this article, the minimal core type system of a programming language which main aim is to provide the benefits of both dynamic and static typing is presented. Keeping explicit typing, implicit type inference is introduced to combine dynamic and static typing in the same programming language. Union and intersection types are customized to express the dynamism of implicitly typed references. The main benefits of our work are early type error detection even in dynamic code, the integration of dynamic and static code in the same programming language, and a notable optimization of the generated code.

1 Introduction

Static typing involves an undeniable tool in software development, offering the programmer advantages such as early type error detection, a better documentation and abstraction, and more opportunities for compiler optimizations [34]. Nevertheless, dynamically typed languages provide a great flexibility at runtime, making them ideally suited for prototyping systems

with changing or unknown requirements, or that interact with other systems that change unpredictably (data and application integration) [23].

Taking the Web engineering area as an example, Ruby [40] has been successfully used together with the Ruby on Rails framework for creating database-backed web applications [41]. Nowadays, JavaScript [10] is being widely employed to create interactive web applications with AJAX (Asynchronous JavaScript And XML) [7], and PHP (PHP Hypertext Preprocessor) is one of the most popular languages to develop Web-based views. Python [36] is used for many different purposes, being the Zope application server (a framework for building content management systems, intranets and custom applications) [21] and the Django Web application framework [9] two well-known examples of Python frameworks.

Due to the recent success of dynamic languages, usual *static* languages –such as Java or C#– are gradually incorporating more dynamic features into their platforms. Taking Java as an example, the *Reflection API* became part of core Java with its release 1.1. This API offers introspection services to examine structures of object and classes at runtime, plus object creation and method invocation. The *Dynamic Proxy Class API* was added to Java 1.3. It allows defining a class at runtime that implements any interface, funneling all its method calls to an `InvocationHandler`. Java 1.5 introduced the `instrument` package that allows dynamic instrumentation of Java programs without modifying class interfaces. In Java 1.6, the new *Java Scripting API* permitted dynamic scripting programs to be executed from, and have access to, the Java platform [19]. Finally the *Java Specification Request* 292 [20], expected to be included in Java 1.7, incorporates the new `invokedynamic` opcode into the Java Virtual Machine (JVM) in order to support the implementation of dynamically typed object oriented languages. Since the computational model of dynamic languages requires extending the JVM semantics, Sun Microsystems launched the *Da Vinci* project in January 2008 [8]. This project is aimed at prototyping a number of enhancements to the JVM, so that it can run non-Java languages, especially dynamic ones, with a performance level comparable to that of Java itself.

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* [23]. One of the first approaches was *Soft Typing* [5], that applied static typing to a dynamically typed language such as Scheme [12]. However, soft typing does not control which parts in a program are statically checked, and static type information is not used to optimize the generated code either. The approach 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* [39], *Hybrid Typing* [13] and *Gradual Typing* [37] perform implicit conversion be-

tween dynamic and static code, employing subtyping relations in the case of quasi-static and hybrid typing and a consistency relation in gradual typing. The main difference between these approaches and the work presented in this paper is that we perform type-checking even when dynamic types are used, detecting type errors in dynamic code and, hence, improving its robustness. Gradual typing first identified unification-based constraint resolution as a suitable approach to integrate both dynamic and static typing [38]. However, with gradual typing a dynamic type is implicitly converted into static without any static type-checking, because type inference is not performed with dynamic references. Theoretical works on combining static with dynamic typing have been partially included in the implementation of programming languages such as Boo [26], Visual Basic .Net [43], Cobra [6], Dylan [11], Strongtalk [4] or the future C# 4.0 [42].

This paper presents the first step towards the formalization of the *StaDyn* programming language [29] specifying the type system of its minimal core. The main objective of *StaDyn* is to obtain the benefits of both dynamic and static typing in the same programming language. Since type-checking is also performed with dynamic references, it will be safer and more efficient, offering a direct interoperation between dynamic and static code. Depending on the specific features of (part of) an application, the programmer may select the dynamism that best suites her needs. For that purpose, we have defined a flow-sensitive type system [14] that performs type inference combining implicit and explicit typing. It joins syntax-directed and constraint-based type-checking [25], and models dynamism by means of the subtyping relation of union and intersection types [33].

The rest of this paper is structured as follows. Section 2 specifies the abstract syntax of the core language and a sample concrete program that will be used as an example throughout the paper. Section 3 defines the type system, and a brief depiction of the constraint resolution algorithm is described in Section 4. A summary of implementation issues are described in Section 5, and Section 6 presents the ending conclusions.

2 Syntax

This paper describes the minimal core type system of the *StaDyn* programming language [29]. *StaDyn* is an object-oriented programming language that extends C# 3.0 supporting both dynamic and static typing. Although current implementation of *StaDyn* offers classes, interfaces, information hiding, method overloading, inclusion polymorphism and dynamic binding, this paper does not include all these features. This work is focused on formalizing the major contribution of our work that is how to include dynamic and static typing in the same programming language. The core features that are specified in this first description of the type system are functions, objects

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

Figure 1: Abstract Syntax of the core language.

(not methods), arrays, assignments, and integer and boolean expressions. Type variables are also included to offer implicit type reconstruction [22] by means of extending the usage of the `var` reserved word added in C# 3.0. In *StaDyn*, `var` references can be set as static (by default) or dynamic, modifying how type-checking is performed. The semantic of the language does not depend on the dynamism.

First part of Figure 1 shows the abstract syntax of the core language (second and third parts are types and constraints respectively). 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 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 be only placed at 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 3.7). The $\oplus$ operator represents arithmetical operations, $\otimes$ logical ones and # symbolizes relational operators.

Figure 2 shows an example program of our core language. Those lines commented with `Error` are expressions statically detected as erroneous by the type system. Elements of the environment and constraints generated

are shown in the right part of the figure. We will use this figure throughout the paper to explain the behavior of the type system.

3 Type System

Types used to describe the core type system of *StaDyn* 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 only used by the type system without the knowledge of the programmer. The objective of avoiding the direct use of internal types is offering the programmer the best possible simplicity without losing the power of our proposition (we have also kept the C# concrete syntax). Constraint types can appear in the constraints of our type system; they are also transparent to the programmer. Object types are specified describing a collection of their fields between curly braces, not including methods as part of them (functions are used instead). Although the `var` keyword is the concrete syntax of type variables (included in C# 3.0 to allow avoid type specification of initialized local variables), the parser assigns them unique sequential numbers (X_i metavariables range over type variables). Dynamic or static (by default) dynamism can be set to type variables, intersection, and union types. Member types are the collection of fields that an object may hold. Since objects in C# do not offer structural subtyping (C# implements class-based inheritance subtyping), member types are the way we have used to specify structural subtyping of objects (see Section 3.5). Notice that intersection, union and member types are transparent to the programmer, appropriately inferred when `var` references are used.

3.1 Judgments

Type inference is specified with the general judgment $\Gamma; \Omega \vdash E : T \mid C; \Gamma'$. This judgment means that under constraints C, environment Γ, and context Ω, expression E has type T, producing the output environment Γ'. Environments (Γ) bind variables (identifies) to types in the scope represented by Γ, and they also bind type variables to types (if type variables have been inferred). For example, in the scope of the `testListOfTwo` function in Figure 2, Γ holds the assumptions $\Gamma($`integer`$)$:`int` and $\Gamma(X_{23})$:{*data*:`int`, *next*:{*data*:`bool`, *next*:`int`}}, being X_{23} the type of `listOfTwo`.

A context (Ω) stores the information of the function being analyzed, in order to type-check its statements. $\Omega._{locals}$ saves the identifier set of local variables in the current function, and $\Omega._{params}$ stores the parameter list; $\Omega._{rt}$ holds the declared return type of the function and $\Omega._{tifp}$ collects types inferred from function parameters (its usage will be detailed in Section 3.3).

We define two kinds of constraints (last part of Figure 1). Subtyping constraints ($CT \leq CT$) require the type on the left to be a subtype of (promote to) the type on the right [24]. Assignment constraints ($CT \leftarrow CT$) not only

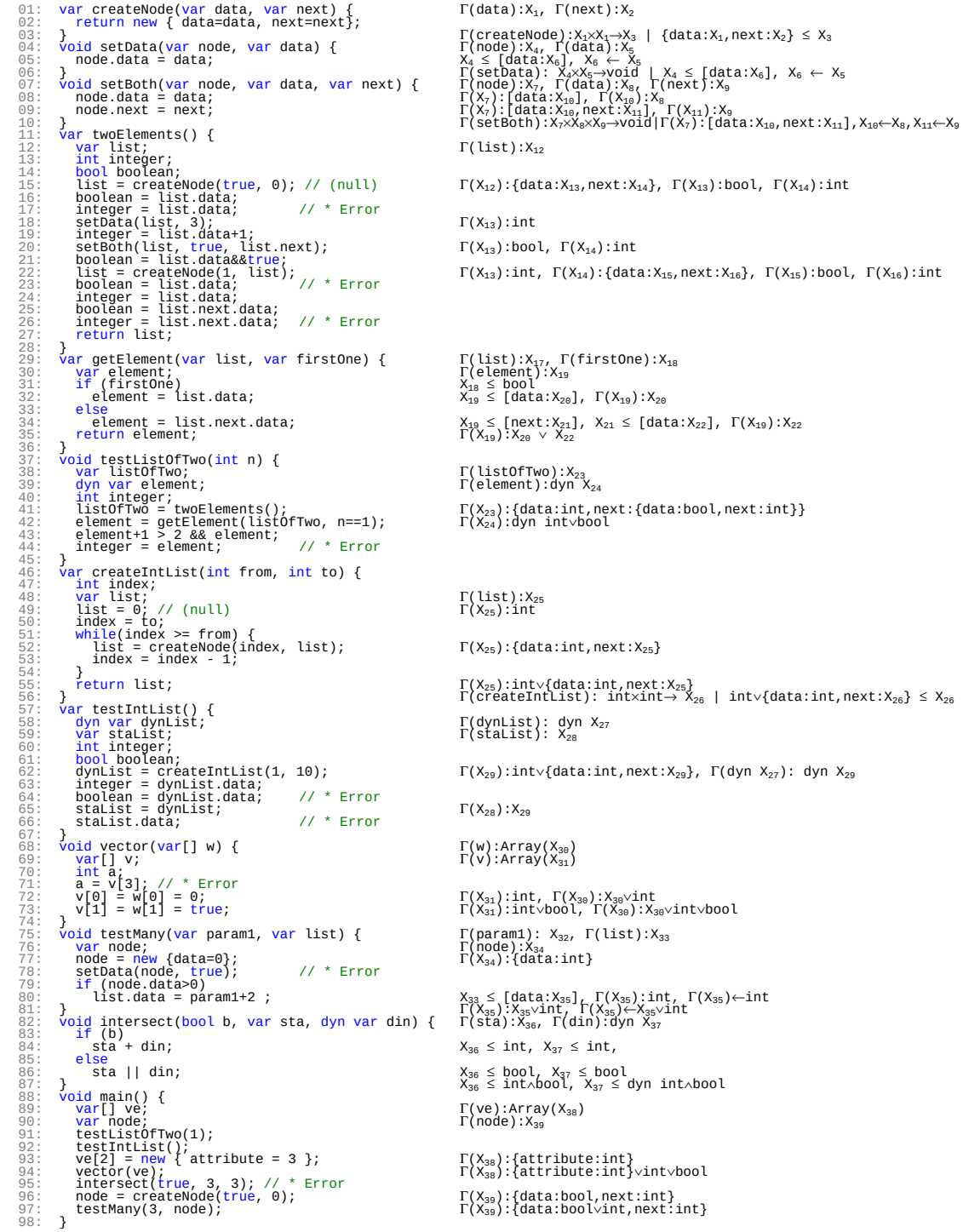

Figure 2: Example concrete program.

check that an assignment could be performed, but they are also used to infer types, binding a type variable to a particular one. Therefore, assignment constraints may modify type variable bindings in type environments when a function is called. In line 31 of Figure 2, a subtyping constraint is generated for variable `firstOne`; it should be a subtype of `bool` ($X_{18} \leq$ `bool`) when the function `getElement` is called. Line 5 is an example of assignment constraint generation. When the `setData` function is invoked, the `data` type of the argument (X_6) will be assigned the type of value (X_5).

Type inference judgments of the form $\Gamma; \Omega \vdash E : T \mid C; \Gamma$' have an input environment Γ, and an output environment Γ'. 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 [14], because variables and type variables may change their types depending on the control flow [15]. An example of this behavior is observed in the `twoElements` function in Figure 2. The assignment in line 15, binds the type variable of the `list`'s `data` (X_{13}) to `bool`, making the statement in line 16 to be accepted by the type system, and line 17 to be erroneous. Line 18 changes the type of the field (`int`) and, hence, line 19 compiles without any error. This example shows how the same variable can hold different types in the same scope, depending on the execution flow. This process has also been applied to control structures (Section 3.7).

3.2 Functions

Inference rules of functions, declarations and statements 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 into the output environment. It is bound to its type ($T_1 \times \ldots \times T_n \rightarrow T \mid C$) making it possible to type-check subsequent calls to it. As Figure 3 shows, function types include the constraint set that must be satisfied by the arguments at any invocation. It is worth noting that function declaration generates no constraint at all, and it only adds the type of the function to the output environment. This means that constraints, types of variables, and type variables are local to the function scope where they were inferred. The rest of the rules in Figure 3 generate no constraint at all, and output environments become the input of the following terms, obtaining flow-sensitive type checking.

(T-Func)

$$\frac{\begin{array}{c} id \notin dom(\Gamma) \qquad \Gamma; \Omega \vdash D : \diamond \mid \emptyset; \Gamma' \\ id_i \notin dom(\Gamma')^{i \in 1..n} \qquad \Gamma', id_1 : T_1 \ldots id_n : T_n; \Omega \vdash S : \diamond \mid C; \Gamma'' \\ \Gamma''' = \Gamma, id : T_1 \times \ldots \times T_n \to T \mid C \end{array}}{\Gamma; \Omega \vdash T\ id(T_1\ id_1 \ldots T_n\ id_n)\ D\ S : \diamond \mid \emptyset; \Gamma'''}$$

T-Decl

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

T-Decls

$$\frac{\Gamma; \Omega \vdash D_1 : \diamond \mid \emptyset; \Gamma' \qquad \Gamma'; \Omega \vdash D_2 : \diamond \mid \emptyset; \Gamma''}{\Gamma; \Omega \vdash D_1\ D_2 : \diamond \mid \emptyset; \Gamma''}$$

T-Funcs

$$\frac{\Gamma; \Omega \vdash F_1 : \diamond \mid \emptyset; \Gamma' \qquad \Gamma'; \Omega \vdash F_2 : \diamond \mid \emptyset; \Gamma''}{\Gamma; \Omega \vdash F_1\ F_2 : \diamond \mid \emptyset; \Gamma''}$$

T-Prog

$$\frac{\Gamma; \Omega \vdash F : \diamond \mid \emptyset; \Gamma' \qquad \Gamma'; \Omega \vdash S_1 : \diamond \mid \emptyset; \Gamma'' \qquad \Gamma''; \Omega \vdash S_2 : \diamond \mid \emptyset; \Gamma'''}{\vdash P : \diamond \mid \emptyset}$$

Figure 3: Program, declarations and functions.

$$\frac{\begin{array}{c}\Gamma; \Omega_{.params} = id_1 \ldots id_p, \Omega_{.locals} = id_{p+1} \ldots id_{p+l}, \\ \Omega_{.tr} = T, \Omega_{.tifp} = T_1 \ldots T_p \vdash S : \diamond\end{array}}{\Gamma; \Omega \vdash T\ id(T_1\ id_1 \ldots T_p\ id_p)\ T_{p+1}\ id_{p+1} \ldots T_{p+l}\ id_{p+l}\ S : \diamond} \quad (\Omega\text{-FUNC})$$

$$\frac{\Gamma; \Omega_1 \vdash E : T_1 \qquad \Gamma \vdash T_1 \leq [id : T_2] \qquad T \in \Omega_{1 \cdot tifp} \qquad \Omega_{2 \cdot tifp} = T_2}{\Gamma; \Omega_1 \cup \Omega_2 \vdash E.id : T_2} \quad (\Omega_{.tifp}\text{-FIELD})$$

$$\frac{\Gamma; \Omega_1 \vdash E_1 : T_1 \qquad \Gamma \vdash T_1 \leq Array(T_2) \qquad T \in \Omega_{1 \cdot tifp} \qquad \Omega_{2 \cdot tifp} = T_2}{\Gamma; \Omega_1 \cup \Omega_2 \vdash E_1[E_2] : T_2} \quad (\Omega_{.tifp}\text{-ARRAY})$$

Figure 4: Inference rules for Ω.

3.3 Context

It is necessary to store information regarding to a function in order to subsequently perform type checking of terms in the function scope. This information is saved in function contexts (Ω) by means of the rules shown in Figure 4. At function declaration (Ω-FUNC), local variables are stored in $\Omega_{.locals}$, parameters in $\Omega_{.params}$, and $\Omega_{.rt}$ saves the return type specified in function declaration.

Types inferred from type parameters are also stored in $\Omega_{.tifp}$ (it will be described later why this information is necessary to perform type-checking of assignments, field accessing and array indexing). First, Ω-FUNC adds parameter types to $\Omega_{.tifp}$; $\Omega_{.tifp}$-FIELD inserts field types when an object type is in $\Omega_{.tifp}$; $\Omega_{.tifp}$-ARRAY does the same with arrays. Notice that not only type variables are inserted in $\Omega_{.tifp}$, because objects and arrays may indirectly hold type variables in their fields or elements, respectively.

3.4 Expressions

This subsection describes type-checking of variables, object field access, vector indexing, and arithmetic, relational and logical expressions. Although assignments and function calls are also expressions, they will be described in Sections 3.6 and 3.9 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 free type variables in an environment. The predicate `lhsAssign` is used to know if an AST node is a direct left child of an assignment expression. Rules in Figure 5 do not generate any constraint and

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

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

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

Figure 5: Variables.

do not add any type information to output environments either.

T-VAR types a variable previously declared, when its type is not a type variable. When the type of an identifier is a type variable and it is bound to another type, T-BVAR types the identifier to the type bound to the type variable. This happens, for instance, in line 79 of Figure 2, where the type variable of `node` (X_{34}) is bound to {`data:int`} in line 77.

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. 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 bound to a type in the subsequent assignment). For example, the utilization of the `v` variable in line 71 of Figure 2 is not allowed because it is a free type variable, it is not a parameter, and it is not the left part of an assignment (T-AVAR). However, the `param1` variable can be used in line 80 because, under the same circumstances, it is a parameter.

Inference rules of arithmetic, relational and logical expressions (Figure 6) are deeply based on subtyping (following section). Operands of arithmetic and relational expressions must be subtypes of `int`; logical expressions should promote to `bool`. Output environments are used as input environments of subsequent expressions, being the last output environment the one returned by the whole expression. The constraint set generated by each expression is the union of all the constraints produced by each of the four premise judgments.

When accessing object fields (T-FIELD), the object should promote to the member type $[id : T_2]$. A member type is an internal type that denotes the set of fields an object should hold. Therefore, an object promotes to a

(T-Arith)

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq \texttt{int} \mid C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \qquad \Gamma''' \vdash T_2 \leq \texttt{int} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \oplus E_2 : \texttt{int} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

(T-Log)

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq \texttt{bool} \mid C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \qquad \Gamma''' \vdash T_2 \leq \texttt{bool} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \otimes E_2 : \texttt{bool} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

(T-Rel)

$$\frac{\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq \texttt{int} \mid C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma''' \qquad \Gamma''' \vdash T_2 \leq \texttt{int} \mid C_4;\Gamma''''}{\Gamma;\Omega \vdash E_1 \# E_2 : \texttt{bool} \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma''''}$$

(T-Field)

$$\frac{\Gamma;\Omega \vdash E : T_1 \mid C_1;\Gamma' \qquad \Gamma' \vdash T_1 \leq [id : T_2] \mid C_2;\Gamma'' \qquad T_2 \in ftv(\Gamma'') \wedge T_2 \notin \Omega._{tifp} \Rightarrow lhsAssign(E.id)}{\Gamma;\Omega \vdash E.id : T_2 \mid C_1 \cup C_2;\Gamma''}$$

(T-Array)

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

Figure 6: Expressions.

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

Figure 7: Type variable binding substitution.

member type following the same rules as structural subtyping for objects described in [2]. Moreover, if the object field is a free type variable not inferred from the parameters (not in $\Omega_{.tifp}$), it must be a direct left child of an assignment expression. Line 5 in Figure 2 is an example of a correct term. Although the `node`'s `data` is a free type variable (X_6) and it is not the left-hand-side of an assignment, its type is in $\Omega_{.tifp}$.

T-Array requires the first expression to be a subtype of array, and the index to be an integer. As with objects, if the calculated type is a free type variable, it should be the left-hand-side of an assignment. This predicate generates a compilation error in line 71 of Figure 2. The type of the elements of the `v` array is the free type variable X_{31}, not inferred from the parameters (it is a local variable), producing a compilation error because no value has been assigned to it.

3.5 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 type variables that might be bound to particular types. In effect, the T-TVBS rule in Figure 7 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-FTV rule in Figure 8 that, in addition, generates a subtyping constraint. The subtyping condition will be true if the constraint is fulfilled. The S-FTV inference rule includes the $X{:}T$ binding in Γ' because subtyping relation of the core language type system presented in this article is actually an equivalence relation, except for objects. This is the cause why we included the subtyping rule S-MemberI for member types. Future extensions of the type system will not add the $X{:}T$ binding to Γ'.

Subtyping relation is reflexive and transitive (Figure 8). When both type variables are not bound to any type, a subtyping constraint is produced (S-FTVs). Objects (S-Object) and arrays (S-Array) type constructors are covariant. However, an object promotes to another one when both have the same number of fields and equal field labels, maintaining the subtyping relation for each field type (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 there is covariant subtyping of their types (S-OMember).

It is also worth noting that member types have structural subtyping (S-MemberE) and that promotion to new members is offered by adding these new members to the constraint set and output environment (S-MemberI). This second rule is necessary because in T-Field (Figure 6) we add a single-field member type to the output environment (and to the constraint set). This means that object will only type-check object operations, but it does not mean that field is its sole member. The `setBoth` function in Figure 2 is an example of this mechanism. In line 8, `data` is assigned to `node.data`, binding the member type $[data{:}X_{10}]$ to the type of `node` (X_7). However, it is also possible to assign a new `next` field to the same object in line 9, because the S-MemberI now binds the member type $[data{:}X_{10}, next{:}X_{11}]$ to `node`.

Since *StaDyn* does not support higher-order functions yet (*delegates* in C# terminology), we do not specify subtyping of the function type constructor.

3.6 Assignments

The four inference rules in Figure 9 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 for 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. This involves a flow-sensitive type inference for object field type variables (e.g., the `twoElements` function in Figure 2). 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 18) changes the type of argument's `data`, making the statement in line 19 to be 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. A union type $T_1 \vee T_2$ denotes the ordinary union of the set of values belonging to T_1 and the set of values belonging to T_2 [33]. Therefore, the type of `v` in line 74 of Figure 2 is an

(S-Reflex)
$$\Gamma \vdash T \leq T \mid \emptyset; \Gamma$$

(S-Trans)
$$\frac{\Gamma \vdash T_1 \leq T_2 \mid C_1; \Gamma' \qquad \Gamma' \vdash T_2 \leq T_3 \mid C_2; \Gamma''}{\Gamma \vdash T_1 \leq T_3 \mid C_1 \cup C_2; \Gamma''}$$

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

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

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

(S-Object)
$$\frac{\Gamma \vdash T_1 \leq T_{n+1} \mid C_1; \Gamma_1 \ldots \ \Gamma_{n-1} \vdash T_n \leq T_{2n} \mid C_n; \Gamma_n}{\begin{array}{c}\Gamma \vdash \{id_1 : T_1 \ldots id_n : T_n\} \leq \{id_1 : T_{n+1} \ldots id_n : T_{2n}\} \mid \\ C_1 \cup \ldots \cup C_n; \Gamma_n\end{array}}$$

(S-OMember)
$$\frac{\forall\, i \in [n+1, n+m], \exists\, j \in [1, n], \qquad id_i = id_j \wedge \Gamma_{i-n} \vdash T_i \leq T_j \mid C_i; \Gamma_{i-n+1}}{\begin{array}{c}\Gamma_1 \vdash \{id_1 : T_1 \ldots id_n : T_n\} \leq [id_{n+1} : T_{n+1} \ldots id_{n+m} : T_{n+m}] \mid \\ C_1 \cup \ldots \cup C_i; \Gamma_{m+1}\end{array}}$$

(S-MemberI)
$$\frac{\Gamma(X) : [id_1 : T_1 \ldots id_n : T_n] \qquad \Gamma' = \Gamma, X : [id_1 : T_1 \ldots id_{n+1} : T_{n+1}] \qquad C = X \leq [id_1 : T_1 \ldots id_{n+1} : T_{n+1}]}{\Gamma \vdash X \leq [id_{n+1} : T_{n+1}] \mid C; \Gamma'}$$

(S-MemberE)
$$\frac{\Gamma(X) : [id_1 : T_1 \ldots id_n : T_n]}{\Gamma \vdash [id_i : T_i]^{i \in 1..n} \leq X \mid \emptyset; \Gamma}$$

Figure 8: Subtyping.

(T-Assign)

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \\ \neg tv(T_1) \qquad \Gamma';\Omega \vdash E_2 : T_2 \mid C_2;\Gamma'' \qquad \Gamma'' \vdash \texttt{sta}\ T_2 \leq T_1 \mid C_3;\Gamma'''\end{array}}{\Gamma;\Omega \vdash E_1 = E_2 : T_1 \mid C_1 \cup C_2 \cup C_3;\Gamma'''}$$

(T-BTVAssign)

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \\ \neg tv(T_1) \qquad \Gamma';\Omega \vdash E_2 : T_2 \mid C_2;\Gamma'' \qquad \Gamma'' \vdash \texttt{sta}\ T_2 \leq T_1 \mid C_3;\Gamma'''\end{array}}{\Gamma;\Omega \vdash E_1 = E_2 : T_1 \mid C_1 \cup C_2 \cup C_3;\Gamma'''}$$

(T-TVAssign)

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1 : X \mid C_1;\Gamma' \\ X \in ftv(\Gamma') \qquad \Gamma';\Omega \vdash E_2 : T \mid C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : T\end{array}}{\Gamma;\Omega \vdash E_1 = E_2 : T \mid C_1 \cup C_2;\Gamma'''}$$

(T-FAssign)

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1 : T_1 \mid C_1;\Gamma' \\ \Gamma' \vdash T_1 \leq [id : X] \mid C_2 \qquad tv(X) \qquad \Gamma';\Omega \vdash E_2 : T_2 \mid C_3;\Gamma'' \\ \Gamma''' = \Gamma'', X : T_2 \qquad \texttt{if}\ X \in \Omega_{.tip}, \texttt{then}\ C_4 = X \leftarrow T_2, \texttt{else}\ C_4 = \emptyset\end{array}}{\Gamma;\Omega \vdash E_1.id = E_2 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup C_4;\Gamma'''}$$

(T-AAssign)

$$\frac{\begin{array}{c}\Gamma;\Omega \vdash E_1[E_2] : X \mid C_1;\Gamma' \\ tv(X) \qquad \Gamma';\Omega \vdash E_3 : T \mid C_2;\Gamma'' \qquad \Gamma''' = \Gamma'', X : \Gamma''(X) \vee T \\ \texttt{if}\ X \in \Omega_{.tip}, \texttt{then}\ C_3 = X \leftarrow X \vee \Gamma''(X) \vee T, \texttt{else}\ C_3 = \emptyset\end{array}}{\Gamma;\Omega \vdash E_1[E_2] = E_3 : T \mid C_1 \cup C_2 \cup C_3;\Gamma'''}$$

Figure 9: Assignments.

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 74 in Figure 2). Unlike the type of `v`, the type of `w` (X_{30}) is included in the union type ($X_{30} \vee$ `int` $\vee$ `bool`), denoting that the type of the actual parameter in the invocation will be included in the union type. Therefore, the type of `w` when the function `vector` is called in line 94 is {`attribute:int`} $\vee$ `int` $\vee$ `bool` ({`attribute:int`} is the type of the `ve` argument).

Unlike subtyping constraints, the set of assignment constraints should keep insertion order, since the last assignment dictates type inference of

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

$$\text{(T-If)}\quad \frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \mid C_1;\Gamma' \qquad \Gamma' \vdash T \leq \texttt{int} \mid C_2;\Gamma'' \\ \Gamma'';\Omega \vdash S_1 : \diamond \mid C_3;\Gamma''' \qquad \Gamma'';\Omega \vdash S_2 : \diamond \mid C_4;\Gamma''''\end{array}}{\Gamma;\Omega \vdash \texttt{if}\ E\ S_1\ S_2 : \diamond \mid C_1 \cup C_2 \cup join(C_3, C_4);\Gamma''' \cup \Gamma''''}$$

$$\text{(T-While)}\quad \frac{\begin{array}{c}\Gamma;\Omega \vdash E : T \mid C_1;\Gamma' \\ \Gamma' \vdash T \leq \texttt{int} \mid C_2;\Gamma'' \qquad \Gamma'';\Omega \vdash S : \diamond \mid C_3;\Gamma'''\end{array}}{\Gamma;\Omega \vdash \texttt{while}\ E\ S : \diamond \mid C_1 \cup C_2 \cup join(C_3, \emptyset);\Gamma'' \cup \Gamma'''}$$

Figure 10: Statements.

arguments after function call. Notice than existing constraints could not be replaced by new ones because subtyping should be still checked. An example of this necessity is the `setData` function invocation in line 78 (Figure 2) that generates a compilation error because the assignment constraint $X_6 \leftarrow X_5$ cannot be performed (`bool` is not a subtype of `int`).

3.7 Statements

The core language includes the `return`, `if` and `while` statements, as well as expressions. Figure 10 shows inference rules for these statements. An expression is a valid statement only if a type can be set to the expression (T-Exp). The output environment and constraint list generated are those provided by the expression. 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 should be 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. For instance, the type of the `element` variable ($X_{20} \vee X_{22}$) –line 35 in Figure 2– is created from its types in line 32 (X_{20}) and 34 (X_{22}). The same happens with constraints: the joined constraint for the variable `sta` in line 87 ($X_{36} \leq$ `int`$\wedge$`bool`) is obtained from the constraint $X_{36} \leq$ `int` in line 84 and $X_{36} \leq$ `bool` generated in line 86.

The union of environments shown in Figure 10 is based on the `join` function described in Figure 11: variable bindings must be the same in both

$$\frac{\begin{array}{c}(\Gamma\text{-}\cup)\\ \forall id \in dom(\Gamma'), id \in dom(\Gamma'') \wedge \Gamma'(id) = \Gamma''(id)\\ \forall id \in dom(\Gamma''), id \in dom(\Gamma') \wedge \Gamma''(id) = \Gamma'(id)\\ tvb(\Gamma) = join(tvb(\Gamma'), tvb(\Gamma''))\end{array}}{\Gamma' \cup \Gamma'' = \Gamma}$$

Figure 11: $\cup$ applied to type environments.

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

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

Figure 12: The join algorithm.

environments, and the resulting type variable binding set (returned by the `tvb` function) is the `join` of type variable binding sets of each flow path.

Figure 12 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 with the axioms in Figure 13. The algorithm takes elements of both sets, adding new union and intersection types [33] to the return set. It first processes the elements of the first set, and then those included in the second set but not in the first one ($\div$).

As Figure 13 shows, comparisons between constraints are based on the type on the constraint left-hand-side. This is because constraints are always generated with a free type variable in its left-hand-side (otherwise, a constraint would have not been generated). Definitions of the `compare` and

(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)

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

$$union(Dyn\ X \leq T_1, Dyn\ X \leq T_2) = X \leq Dyn\ T_1 \wedge T_2$$

$$union(X : T_1, X : T_2) = X : T_1 \vee T_2$$

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

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

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

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

Figure 13: Comparison and union operations.

`union` operations in Figure 13 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 Section 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. An intersection type $T_1 \wedge T_2$ denotes all the values belonging to both T_1 and T_2 [33]. Therefore, 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 intersection type). $T_1 \wedge T_2$ is the greatest lower bound of T_1 and T_2, while $T_1 \vee T_2$ is the least upper bound [3]. The meaning of dynamic intersection and union types will be described in the following section.

The `union` function is also defined for constraints or type bindings generated in only one of the execution paths (last four axioms in Figure 13). 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 execution path –differences between dynamic and static types will be discussed in Section 3.8. 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. As an example, the `data` field of the `list` variable (X_{35}) has the type $X_{35} \vee$ `int` in line 81 of Figure 2. As it will be discussed in the following section, this means that its type can be either `int` or the free type variable X_{35}. The special case of assignment constraints, that are also generated in T-AAssign, is taken into account in the constraint resolution algorithm presented in Section 4.

3.8 Dynamism

3.8.1 Union Types

The type system presented in this paper considers the dynamism the programmer may set to type variables (`var` references). This dynamism is taken into consideration in the type-checking of union and intersection types. We have started from static typing rules defined by Benjamin Pierce [33] (S-SUnionL and S-UnionR in Figure 14, and S-SInterR and S-InterL in Figure 14), and then added new dynamic typing rules (D-SUnionL in Figure 14, and D-SInterR in Figure 15).

A union type holds all the possible types a reference may have, turning into their least upper bound [3]. If the type variable bound to a union type has been declared as static, the set of operations (e.g., addition, field access, assignment, invocation or indexing) that can be applied to that union type are those accepted by every type in the union type. However, if the reference is dynamic, type-checking is more optimistic. 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. 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 63 and 66 of Figure 2. The type of both `staList` and `dynList` is `int`$\vee\{$*data*:`int`, *next*:$X_{29}\}$, but `staList` is static whereas `dynList` is dynamic. This difference makes the field access operation in line 66 not to compile (the dot operator cannot be applied to an `int`), while it is correct in line 63 (the $\{$*data*:`int`, *next*:$X_{29}\}$ type accepts the addition operation).

The behavior described in the previous paragraph is achieved with the subtyping rules in Figure 14. Since our system makes extensive use of the subtyping relation to type-check terms, we have centralized the dynamism of our type system in the subtyping relation of union and intersection types. For static subtyping of union types, every type should promote to the other

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

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

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

Figure 14: Subtyping of union types.

type in the subtyping predicate (S-SUnionL). The generated type variable bindings and constraint set are the union of those produced with each subtyping, following the definition of environments union shown in Figure 11. S-DUnionL specifies dynamic subtyping, requiring at least one promotion; constraint and type variable binding created are the union of those successful promotions ($\cup\Gamma_i$ represents the union of the Γ_i).

3.8.2 Intersection Types

Since an intersection type collects the types a reference has, it denotes the greatest lower bound of those types [3]. 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). The dynamic behavior is 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). This behavior is directly connected with how subtyping constraints were joined (Figure 13): two subtyping constraints with the same type variable in their left-hand-side are joined to a unique constraint that requires this type variable to promote to the intersection type collection both right-hide-sides. The example function `intersect` in Figure 2 has two parameters: one static (X_{36}) and the other dynamic (X_{37}). The conditional statement generates two constraints: $X_{36} \leq$ `sta int` $\wedge$ `bool` and $X_{37} \leq$ `dyn int` $\wedge$ `bool`. This difference in the dynamism is shown when the function `intersect` is invoked in line 95: the first parameter generates a type error, but the second does not.

3.8.3 Dynamism Inference

Although Figures 14 and 15 describe subtyping of (dynamic and static) union and intersection types, the programmer only specifies the dynamism of free type variables (Figure 1). Starting from the dynamism of `var` references, inference rules in Figure 16 (together with J-Union axioms in Figure 13) infer the dynamism of union and intersection types. D-Var makes dynamic

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

(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 15: Subtyping of intersection types.

(D-Var)
$$\frac{\Gamma(id) : \texttt{dyn}\ X \qquad \Gamma(X) : T \qquad \Gamma' = \Gamma, \texttt{dyn}\ X : \texttt{dyn}\ T}{\Gamma \vdash id : \texttt{dyn}\ T \mid \emptyset; \Gamma'}$$

(D-TVUnion)
$$\frac{\Gamma(\texttt{dyn}\ X) = T_1 \vee \ldots \vee T_n}{\Gamma(\texttt{dyn}\ X) = \texttt{dyn}\ T_1 \vee \ldots \vee T_n}$$

(D-Array)
$$\frac{\begin{array}{c}\Gamma; \Omega \vdash E_1 : \mathit{Array}(\texttt{dyn}\ X); \Gamma' \\ \Gamma'; \Omega \vdash E_1[E_2] = E_3 : T_1; \Gamma'' \qquad \Gamma''; \Omega \vdash E_1 : T_2\end{array}}{\Gamma''; \Omega \vdash E_1 : \mathit{Array}(\texttt{dyn}\ T_2)}$$

Figure 16: Dynamism calculation.

the type of variables declared as dynamic. Since a union type is always bound to a type variable, D-TVUnion sets the dynamism of a type variable to the type it is bound to. The D-Array rule makes the dynamism of type variables in arrays prevail over assignments; i.e., if after an assignment the dynamic type variable in an array is bound to a new type, this new type will also be dynamic.

3.8.4 Converting Implicit to Explicit Types

The programmer could convert a dynamic implicitly typed union type to an explicit (particular) type (in assignments or function calls). When the union type is static, the subtyping rules described in Section 3.8.1 require types in the union type to promote to the explicit type. However, if the

(T-Inv)

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

(T-FTVInv)

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

Figure 17: Function invocation.

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 conversion is forced in both assignments (rule T-Assign in Figure 9) and function invocations (rules T-Inv and T-FTVInv y Figure 17). In line 44, Figure 2, a `dyn int∨bool` type is assigned to an integer, generating a compilation error even though `element` is dynamic, because the `integer` variable is explicitly typed (and, hence, static).

3.9 Function Invocation

Figure 17 shows the two inference rules of function invocation. The difference lies on the existence of type variable arguments (T-Inv for no 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 call. On each invocation, types of the arguments are computed and checked to be subtypes of parameter types.

If no argument is a free type variable, constraints resolution is performed (Section 4). 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 $\Gamma_{2n+2}(T)$, being Γ_{2n+2} a solution for C. Under these circumstances, the C constraint set is solved and, hence, it is not included in the constraint set 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 function invocation

(T-FTVInv).

4 Constraint Resolution

As we have mentioned, constraint resolution is performed when a function call with no type variable argument is type-checked, combining syntax-directed type-checking with constraint resolution.

Theorem 1. *The type system generates no unfulfilled constraint for the main function*

$$\vdash P : \diamond \mid \emptyset; \Gamma$$

Proof. Every constraint is generated by the appearance of a free type variable in an expression. Syntactically (Figure 1), the language syntax does not allow parameterization of the main function. Therefore, local variables are the unique free type variables that might be used in the main function. The four rules in Figure 5 type-check variables: T-Var requires the identifier not to be a type variable; T-BVar to be bound type variable; T-PVar to be a parameter; and T-AVar to be assigned a value and, hence, to be bound, without generating any constraint. For free type variable array elements and object members, both T-Array and T-Field (Figure 6) force free type variables not in $\Omega_{\cdot tifp}$ to be the left-hand side of an assignment, without generating any constraint. □

Proposition 1. *Properties of constraint resolution* $(\Gamma_s \Vdash C, \Gamma)$*:*

1. $\forall\, T_1 \leq T_2 \in C, \Gamma_s \vdash T_1 \leq T_2 \mid \emptyset; \Gamma_s$
2. $\forall\, X \leftarrow T \in C, \Gamma_s \vdash T \leq X \mid \emptyset; \Gamma_s$

The `solve` algorithm we have implemented for constraint resolution is shown in Figures 18 and 19. It is an adaptation of the algorithm defined by Aiken and Wimmers [3] that performs inclusion constraint resolution using union and intersection types.

Definition 1. *Under the typing environment* Γ*, a set of constraints* C *is satisfied by the typing environment* Γ_s*, written* $\Gamma_s \Vdash C, \Gamma$*, iff for the evaluation of* `solve`(C, Γ) *we obtain* $(\Gamma_s,$ `true`*).*

The `solve` algorithm (Figure 12) first checks the subtyping constraints and then the assignment ones. The subtyping rules are used to check the subtyping constraints, using Γ_s as the output environment. The algorithm employs the `union`(T_1, T_2) operation (Figure 19) to merge the equivalence classes of T_1 and T_2, that are initialized at the beginning of the `solve` algorithm. The `union` operation is not defined for union or intersection types

```
solve(C, Γ) ≡ (Γ_out, success) in
  Γ_s ← Γ
  foreach T ∈ C
    T.class ← {T}
  foreach T1 ≤ T2 ∈ C
    foreach T ∈ T2.class
      if not(Γ_s ⊢ T1 ≤ T | ∅; Γ_s)
        return (Γ_s, false)        /* Typing Error */
    union(T1, T2)
  foreach X ← T ∈ C
    T ← replaceRecursive(Γ_s, X, T)
    T.dynamism ← X.dynamism
    if not(Γ_s ⊢ T ≤ X | ∅; Γ_s)
      return (Γ_s, false)          /* Typing Error */
    foreach X1 ∈ X.class
      Γ_s ← Γ_s, X1 : T
  return(Γ_s, true)                /* Succeed */
```

Figure 18: Constraint resolution algorithm.

because they should be merged together with free type variables (first alternative of the `union` operation). For the assignment constraints, it is first checked that the right-hand-side is a subtype of the left-hand-side. Afterwards, the right-hand-side type is bound in Γ_s to all the types in the equivalence class of the left-hand-side.

In Figure 13 we saw how an assignment constraint $X \leftarrow X \vee T$ was generated when only one of the execution paths produced an assignment constraint for X ($X \leftarrow T$). The `replaceRecursive`(Γ,X,T) function solves this recursion. It takes the assignment right-hand-side (T) and replaces all the occurrences of T in X with $\Gamma(X)$. An example of this operation is the constraint resolution of the `testMany` function invocation (line 97 of Figure 2). The constraint $X_{35} \leftarrow X_{35} \vee$ `int` is satisfied when `testMany` is invoked, binding X_{35} (unified to the `data` field of the `node` argument) to `bool`$\vee$`int` and, hence, giving the `node` argument the type {*data*:`bool`$\vee$`int`,*next*:`int`}. Notice that the `replaceRecursive` function is not used in subtyping constraint resolution because no recursive subtyping constraint is generated in our type system.

$union(\Gamma, T_1, T_2) \equiv$
  $order(T_1, T_2)$ $\quad / * Avoid\ symmetric\ comparisons * /$
  `if` $tv(T_1)$ `and` $tv(T_2)$
    $T_1.class \leftarrow T_1.class \cup T_2.class$
    $T_2.class \leftarrow T_1.class$
  `else if` $T_1 \in dom(\Gamma)$
    $union(\Gamma, \Gamma(T_1), T_2)$
  `else if` $T_1 = Array(T_1')$ `and` $T_2 = Array(T_2')$
    $union(\Gamma, T_1', T_2')$
  `else if` $T_1 = \{id_1 : T_1', \ldots, id_n : T_n'\}$ `and`
        $T_2 = \{id_1 : T_1'', \ldots, id_n : T_n''\}$
    `foreach` $i \in [1, n]$
      $union(\Gamma, T_i', T_i'')$
  `else if` $T_1 = [id_1 : T_1', \ldots, id_n : T_n']$ `and`
      $T_2 = [id_1 : T_1'', \ldots, id_n : T_n'', \ldots, id_{n+m} : T_{n+m}'']$
    `foreach` $i \in [1, n]$
      $union(\Gamma, T_i', T_i'')$
  `else if` $T_1 = [id_1 : T_1', \ldots, id_n : T_n']$ `and`
      $T_2 = \{id_1 : T_1'', \ldots, id_n : T_n'', \ldots, id_{n+m} : T_{n+m}''\}$
    `foreach` $i \in [1, n]$
      $union(\Gamma, T_i', T_i'')$
  `else` $Error$ $\quad / * Typing\ Error * /$

Figure 19: The union operation.

5 Implementation

The core type system presented in this paper is based on the requirement that, before checking the type of a function call, the function body should be previously type-checked (Figure 17). Once the function has been analyzed, its body is not examined any more. Therefore, the type-checking algorithm starts analyzing the main function, typing each function body before checking its invocation. Although in the implementation of *StaDyn* we support recursive invocation, this feature has not been formalized yet.

Inference rules of our type system are non-deterministic. Some transformations of those rules (and merging between them) were performed to implement a deterministic type-checking algorithm, taking into consideration the process described in the paragraph above. To make deterministic the assignment rules, two premises were added to T-TVAssign. The AST node that represents the left-hand-side of the assignment could not be either an object field access or an array indexing expression. Inference rules that set the elements of Ω were mixed up with function invocation rules (T-Inv and T-FTVInv), array indexing (T-Array), object field access (T-Field), and assignment rules (Figure 9). Regarding to dynamism rules in Figure 16, D-Var was mixed with T-BVar, D-TVUnion with Γ-$\cup$, and T-Array with T-AAssign. The type variable binding substitution rule in Figure 7 was included in all the subtyping rules. The subtyping relation was implemented by means of the *Composite* design pattern [16], following the guidelines described in [27].

The type-checking algorithm has been developed in C# as part of the *StaDyn* programming language [29]. Although we are only generating CLR 2.0 binaries at present, we also plan to generate code for the DLR [17] and *Reflective Rotor* [35, 30] in future versions. Lexical and syntax analysis was developed using the AntLR tool [31], and syntax-directed type-checking follows the *Visitor* design pattern [16]. Binaries and source code of the whole implementation of *StaDyn* are freely available at:

`http://www.reflection.uniovi.es/stadyn`

The core language type system, its reduced syntax, and the example shown in Figure 2 are also included as part of the *StaDyn* source code distribution.

In [28] we describe a preliminary assessment of performance showing that current implementation generates code 43 and 51 times faster than Visual Basic 10 and C# 4.0 respectively, when implicit typing is used. If explicit typing is used instead, our compiler produces applications 4.6% slower than C# 4.0 and 14% faster than Visual Basic 10 [28].

6 Conclusions

This paper describes the first step towards the specification of the *StaDyn* type system, a programming language aimed at obtaining the benefits of both dynamic and static typing in the same programming language. The major contribution of this work is the definition of a type system that integrates syntax-directed and constraint-based analysis to perform type-checking, detecting type errors even in dynamic scenarios, and offering implicit coercions between static and dynamic types. Main advantages of our approach are early type error detection (even in dynamic code), dynamic and static code integration, and a valuable tool for code optimization. Explicit typing is combined with an implicit type reconstruction system. Differences between static and dynamic typing are centralized in the behavior of union and intersection types, employing the same type system in scenarios where efficiency and safety are required (static) and situations where adaptiveness or rapid-prototyping are appropriate (dynamic).

Future work will be focused on formalizing the operational semantics of the language core in order to prove its type safety. Afterwards, all those features of *StaDyn* not included in this paper will be defined: classes, interfaces, inheritance, inclusion polymorphism, dynamic binding, recursion, method overloading, and the inclusion of a wider set of built-in types.

7 Acknowledgments

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

References

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

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

[3] A. Aiken, and E.L. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1993.

[4] G. Bracha, and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications* (1993), 215–230.

[5] R. Cartwright, and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, 1991.

[6] The Cobra Programming Language. `http://cobra-language.com`

[7] D. Crane, E. Pascarello, and D. James. *Ajax in Action*. Manning Publications, 2005.

[8] *The Da Vinci Machine, a multi-language renaissance for the Java Virtual Machine architecture*. Sun Micryosystems OpenJDK. `http://openjdk.java.net/projects/mlvm`

[9] Django, the Web framework for perfectionists with deadlines. `http://www.djangoproject.com`

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

[11] N. Feinberg, S.E. Keene, R.O. Mathews, and P.T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman, 1997.

[12] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, 1996.

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

[14] J.S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Programming Language Design and Implementation*, 2002.

[15] M. Furr, A. Jong-Hoon, J.S. Foster, and M. Hicks. Static Type Inference for Ruby. In *ACM Symposium on Applied Computing*, 2009.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[17] J. Hugunin. Just Glue It! Ruby and the DLR in Silverlight. In *MIX Conference*, 2007.

[18] Hunt, A., and D. Thomas, "The Pragmatic Programmer", Addison-Wesley, 2000.

[19] JSR 223, *Scripting for the Java Platform*. `http://www.jcp.org/en/jsr/detail?id=223`

[20] JSR 292, *Supporting Dynamically Typed Languages on the Java Platform*. `http://www.jcp.org/en/jsr/detail?id=292`

[21] Latteier, A., M. Pelletier, C. McDonough, and P. Sabaini. *The Zope Book*. 2008. `http://www.zope.org/Documentation/Books/ZopeBook/`

[22] N. McCracken, The typechecking of programs with implicit type structure. In *Semantics of Data Types*, Lecture Notes in Computer Science 173, 1984.

[23] E. Meijer, and P. Drayton, P. 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)

[24] J.C., Mitchell. Type inference with simple subtypes. *Journal of Functional Programming* 1, 3 (1991).

[25] M. Odersky, M. Sulzmann, and M.Wehr. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999).

[26] R.B. de Oliveira. *The Boo programming language*. `http://boo.codehaus.org`

[27] F. Ortin, D. Zapico, and J.M. Cueva. Design Patterns for Teaching Type Checking in a Compiler Construction Course. *IEEE Transactions on Education* 50, 3 (2007).

[28] F. Ortin, and Perez-Schofield, J.B.G. Supporting both Static and Dynamic Typing.In *Programming and Languages Conference (PROLE)*, 2008.

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

[30] F. Ortin, Redondo, J.M., Perez-Schofield, J.B.G. Efficient Virtual Machine Support of Runtime Structural Reflection. *Science of Computer Programming, to be published.*

[31] Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[32] M.S. Paterson, and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences* 16 (1978).

[33] 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.

[34] B.C. Pierce. *Types and Programming Languages*. The MIT Press. February, 2002.

[35] J.M. Redondo, F. Ortin, and J.M. Cueva. Optimizing Reflective Primitives of Dynamic Languages. *International Journal of Software Engineering and Knowledge Engineering* 18, 6 (2008).

[36] G. van Rossum, L. Fred, and J.R. Drake. *The Python Language Reference Manual*. Network Theory, 2003.

[37] J. Siek, and W. Taha. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programing*, Lecture Notes In Computer Science 4609, 2007.

[38] J. Siek, and M. Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the Symposium on Dynamic Languages*, 2008.

[39] S. Thatte. Quasi-static typing. In *Proceedings of the ACM Symposium on Principles of programming languages*, 1990.

[40] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby, 2nd Edition*. Addison-Wesley Professional, 2004.

[41] D. Thomas, D.H. Hansson, A. Schwarz, T. Fuchs, L. Breed, and M. Clark. *Agile Web Development with Rails. A Pragmatic Guide*. Pragmatic Bookshelf, 2005.

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

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