NOTICE: This is the author's version of a work accepted for publication by Elsevier. Changes resulting from the publishing process, including peer review, editing, corrections, structural formatting and other quality control mechanisms, may not be reflected in this document. A definitive version was subsequently published in the Journal of Visual Languages & Computing, Volume 25, Issue 4, pp. 346-362, August 2014.

Static Type Information to Improve the IDE Features of Hybrid Dynamically and Statically Typed Languages

Francisco Ortin[a,*], Francisco Moreno[b], Anton Morant[c]

[a] *University of Oviedo, Computer Science Department, C/ Calvo Sotelo s/n, 33007, Oviedo, Spain*
[b] *Alisys Software S.L.U., C/ Menendez Valdes 40, 33201, Gijon, Spain*
[c] *University of Oxford, Wolfson College, Linton Road, OX26UD, Oxford, UK*

Abstract

The flexibility offered by dynamically typed programming languages has been appropriately used to develop specific scenarios where dynamic adaptability is an important issue. This has made some existing statically typed languages gradually incorporate more dynamic features to their implementations. As a result, there are some programming languages considered hybrid dynamically and statically typed. However, these languages do not perform static type inference on dynamically typed code, lacking those common features provided when statically typed code is used. This lack is also present in the corresponding IDEs that, when dynamically typed code is used, do not provide the services offered for static typing. We have customized an IDE for a hybrid language that statically infers type information of dynamically typed code. By using this type information, we show how the IDE can provide a set of appealing services that the existing approaches do not support, such as compile-time type error detection, code completion, transition from dynamically to statically typed code (and vice versa), and significant runtime performance optimizations. We have evaluated the programmer's performance improvement obtained with our IDE, and compared it with similar approaches.

Keywords: Hybrid dynamic and static typing, IDE support, type inference, code completion, separation of concerns, plug-in, Visual Studio

*Corresponding author
Email addresses: `ortin@lsi.uniovi.es` (Francisco Ortin), `francisco.moreno@alisys.net` (Francisco Moreno), `anton.morant@comlab.ox.ac.uk` (Anton Morant)
URL: `http://www.di.uniovi.es/~ortin` (Francisco Ortin)

1. Introduction

Dynamic languages have turned out to be suitable for specific scenarios such as rapid prototyping, Web development, interactive programming, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software.

Taking the Web engineering area as an example, Ruby [1] has been successfully used together with the Ruby on Rails framework for creating database-backed Web applications [2]. This framework has confirmed the simplicity of implementing the DRY (*Don't Repeat Yourself*) [3] and the *Convention over Configuration* [2] principles with this kind of languages. Nowadays, JavaScript [4] is being widely employed to create interactive Web applications with AJAX [5], while PHP is one of the most popular languages to develop Web-based views. Python [6] is used for many different purposes, being the Zope application server [7] and the Django Web application framework [8] two well-known examples.

Due to the recent success of dynamic languages, other statically typed ones such as Java and C# are gradually incorporating more dynamic features into their platforms. Taking C# as an example, the .NET platform was initially released with introspective and low-level dynamic code generation services. Version 2.0 included dynamic methods and the `CodeDom` namespace to generate the structure of high-level source code documents. The Dynamic Language Runtime (DLR) adds to the .NET platform a set of services to facilitate the implementation of dynamic languages. A new `dynamic` type has been included in C# 4.0 to support dynamically typed code. When a reference is declared as `dynamic`, the compiler performs no static type checking, postponing all the type verifications until runtime [9]. With this new characteristic, C# 4.0 offers direct access to dynamically typed code in IronPython, IronRuby and the JavaScript code in Silverlight.

Java also seems to follow this trend. The last addition to support features commonly provided by dynamic languages has been the Java Specification Request (JSR) 292, partially included in Java 7. The JSR 292 incorporates the new `invokedynamic` opcode to the Java Virtual Machine so that it can run dynamic languages with a performance level comparable to that of Java itself [10].

The flexibility of dynamic languages is, however, counteracted by limitations derived from the lack of static type checking. This deficiency implies two major drawbacks: no early detection of type errors, and less opportunities for compiler optimizations. Static typing offers the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime –when the programmer's efforts might be aimed at some other task, or even after the program has been deployed. Moreover, since runtime adaptability of dynamic languages is mostly implemented with

dynamic type systems, runtime type inspection and checking commonly involve a significant performance penalty [11].

Since both static and dynamic typing approximations offer different benefits, there have been former works to provide both typing approaches in the same language (see Section 5). Meijer and Drayton [12] maintain that instead of providing programmers with a black or white choice between static or dynamic typing, it could be useful to strive for softer type systems. Static typing allows earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance. Dynamic typing languages provide a solution to a kind of computational incompleteness inherent to statically typed languages, offering, for example, storage of persistent data, inter-process communication, dynamic program behavior customization or generative programming [12]. Therefore, there are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [13]. That is, *static typing where possible, dynamic typing when needed* [12].

As proposed by Meijer and Drayton, we break the programmers' black or white choice between static or dynamic typing. We have developed a programming language called *StaDyn* that provides both type systems [14]. *StaDyn* is an extension of C# 3.0, which supports static and dynamic typing. *StaDyn* permits the straightforward development of adaptable software and rapid prototyping, without sacrificing application robustness and runtime performance. The programmer indicates whether high flexibility is required (dynamic typing) or stronger type checking (static) is preferred. It is also possible to combine both approaches, making parts of an application more flexible, whereas the rest of the program maintains its robustness and runtime performance.

The main contribution of this paper is a visual IDE that takes advantage of the specific features of hybrid statically and dynamically typed languages, showing how the type information gathered by the compiler can be used to provide new features plus others that are commonly offered for statically typed code only. The *StaDyn* IDE separates the *dynamism* concern [15] facilitating the transition from rapidly developed prototypes to final robust and efficient applications, detects many type errors of dynamically typed code at compile time, provides code completion for dynamically typed code, and performs significant code optimizations.

The rest of this paper is structured as follows. In Section 2, the *StaDyn* IDE is described, emphasizing the new features added to Visual Studio (VS) in order to support specific features of hybrid dynamically and statically typed languages. Section 3 describes the implementation technologies used to customize the IDE, and its integration with the *StaDyn* compiler. In Section 4, we evaluate the programmer's performance improvement using our IDE, comparing it with similar approaches. Section 5 discusses related work, and the conclusions and future work are presented in Section 6.

2. A Visual IDE for Hybrid Statically and Dynamically Typed Languages

The proposed IDE can be applied to any object-oriented hybrid statically and dynamically typed language, such as Visual Basic, Objective-C, Boo, C# 4.0, Groovy 2.0, Fantom and Cobra. We have selected the *StaDyn* programming language, a hybrid static and dynamic typing language developed for research purposes [14]. *StaDyn* is an extension of C# 3.0 [16] that enhances the behavior of its implicitly typed local references (i.e., its `var` keyword). In *StaDyn*, the type of references can be explicitly declared, while it is also possible to use the `var` keyword to declare implicitly typed references. *StaDyn* includes this keyword as a whole new type (it can be used to declare uninitialized local variables, fields, method parameters and return types), whereas C# only provides its use in the declaration of initialized local references. An informal description of the language can be consulted in [14], while its static and dynamic semantics is detailed in [17].

In a previous prototype, we developed a first version of an IDE for the first implementation of the *StaDyn* language [18]. This first prototype was implemented as a plug-in for VS 2008. The one presented in this paper provides new features for both VS 2010 and 2012, using the Managed Extensibility Framework (MEF) included in the .NET Framework 4.0 (Section 3). The new functionalities of the present IDE significantly increase the ones of the previous prototype, representing a new contribution (Section 5 details the differences).

2.1. Basic Features

Our customization of Visual Studio provides the typical features of most programming language IDEs. Some of these features, already supported by the previous prototype, are the creation of *StaDyn* projects, the common VS editing services for *StaDyn* files, the same color syntax highlighting of C#, and the typical *build*, *rebuild*, *clean* and *start* commands (debugging is not provided yet).

Besides, some other basic services have been included in this new version, such as brace, parenthesis and square bracket matching, code completion (called *IntelliSense* in VS) in the initialization of `var` references, IntelliSense for built-in types and overloaded methods, automatic code completion by pressing the space bar or the dot key, and a new classification of the elements proposed by IntelliSense with the following tabs: `All`, `Keywords`, `Types`, `MembersInScope`, and `ClassMembers`. Figure 1 shows a snapshot of our language service in VS 2012, where one *StaDyn* file in a project is being edited. The rest of figures show the IDE for VS 2010.

2.2. Type Hints of Implicitly Typed References

Dynamic (and hybrid) languages allow the declaration of references without specifying a type (or simply declaring it as `dynamic`). These references can hold values of any type. In

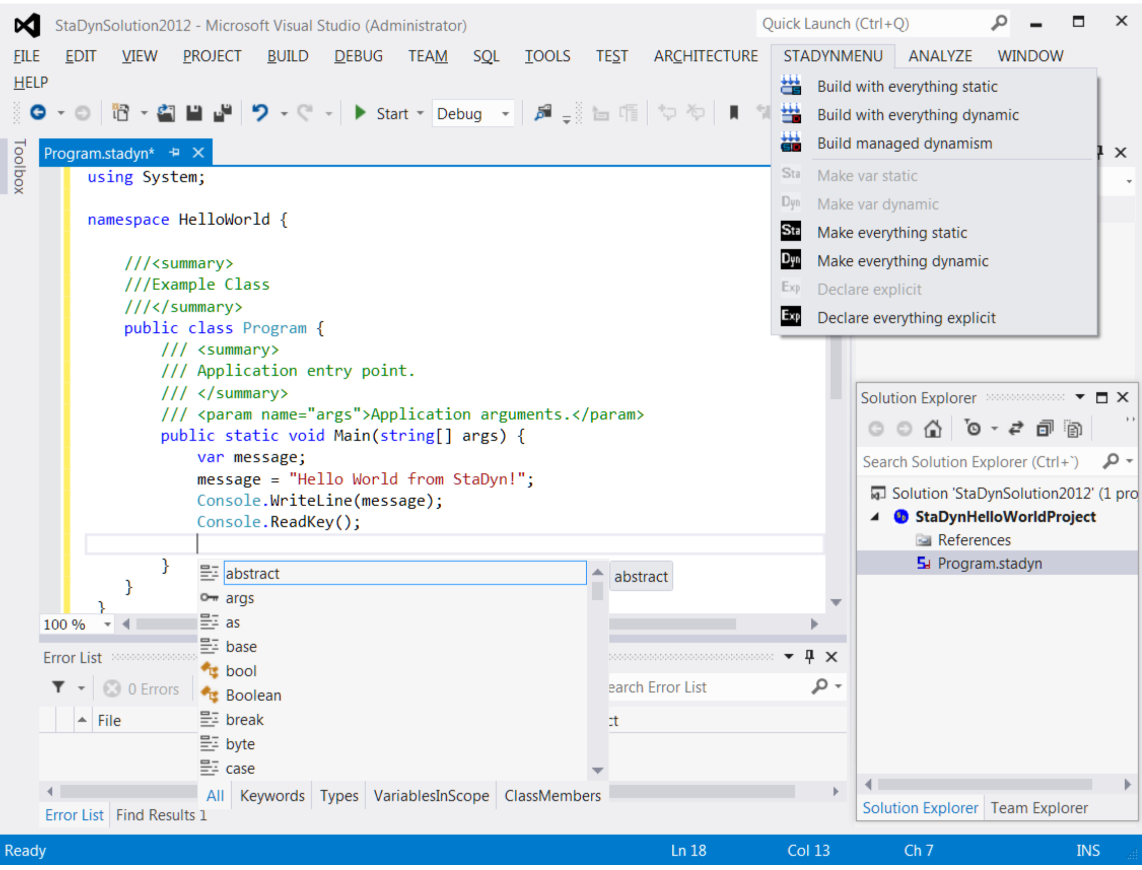

Figure 1: The *StaDyn* IDE for Visual Studio 2012.

fact, they can also hold values of different types without a common supertype but `Object`. Dynamic languages easily implement this feature with a dynamic type system. In contrast, statically typed languages force a variable declared with a type T to have the same type T within the scope in which it is bound to a value. Even languages with powerful static type inference (type reconstruction) such as ML [19] or Haskell [20] do not permit the assignment of different types to the same reference.

StaDyn offers this feature with *static* type checking, considering the concrete type of each reference. The *StaDyn* program shown in Figure 2 is an example of this capability. The `number` reference is first declared to hold any type (`var`). This variable declaration is an extension of the implicitly typed `var` references introduced in C# 3.0. In C#, `var` can only be used to declare a local variable if a default value is assigned to it at declaration. However, `var` variables in *StaDyn* can be declared without a default value, as shown in Figure 2.

The type of `number` is inferred by the compiler, and it is used by the IDE to provide different services such as code completion (Section 2.3), showing type errors at compile time (Section 2.7), or improving runtime performance (Section 2.9). At its declaration, `number` has no type (its type variable has not been unified), then a `String` is assigned to it, and it finally holds a `double` value.

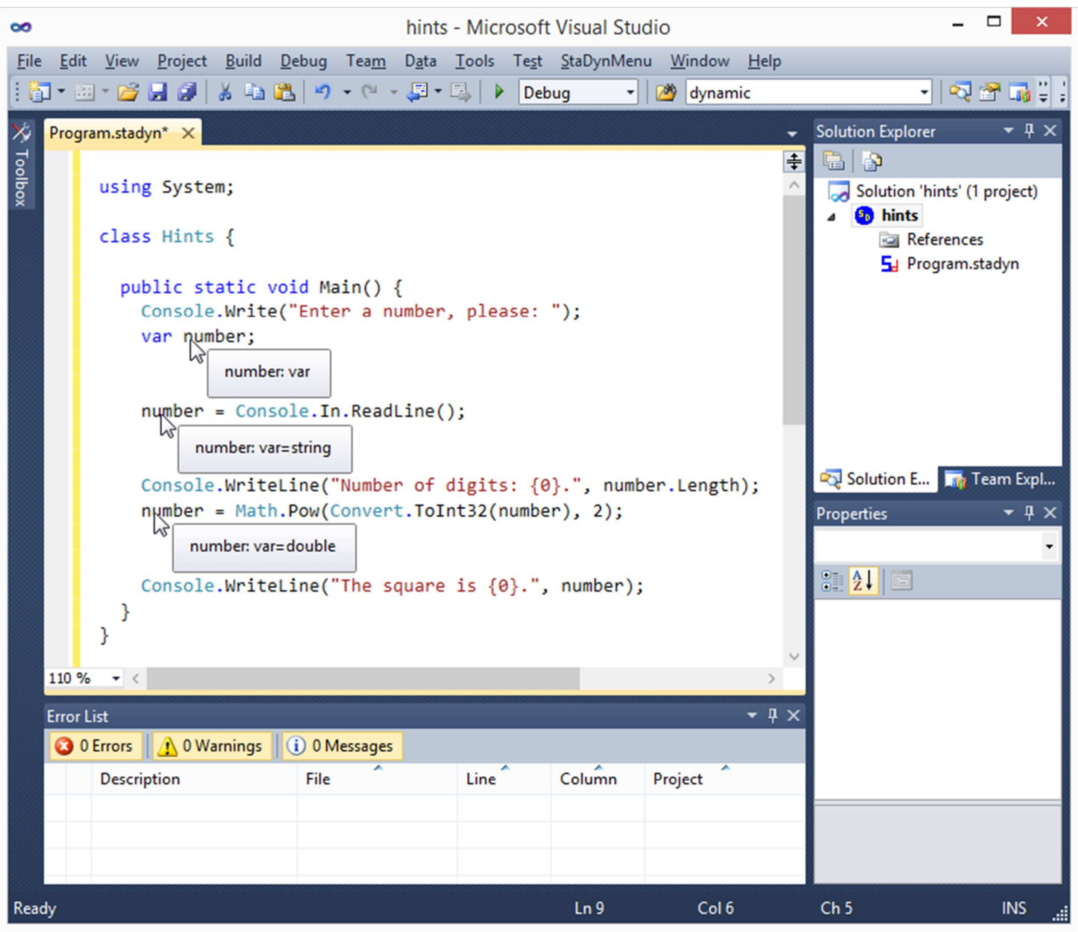

Figure 2: Type hints of local variables.

Occasionally, the programmer may be interested in knowing the inferred type of a `var` reference (e.g., to select the appropriate implementation of an overloaded method, when the reference is passed as a parameter). For these cases, the IDE provides this type information as hints. As shown in Figure 2, the inferred type (if any) is shown when the mouse moves over a `var` reference.

2.2.1. Storing Values of Different Types

Types of local variables are inferred with a modification [21] of the Hindley-Milner type inference algorithm [22]. The Hindley-Milner type system implements a unification algorithm to provide parametric polymorphism [23] but, unlike dynamic languages, it forces a reference to have the same static type in its scope. To overcome this drawback, we have developed a version of the single static assignment (SSA) algorithm [24]. This algorithm guarantees that every reference is assigned exactly once by creating new temporary references. Therefore, the `number` reference in Figure 2 is replaced with three different references. This is the reason why the IDE shows three different types for the `number` variable. The first fresh type variable is not unified, and the second and third ones are unified to `string` and `double`, respectively.

2.2.2. References as Pointers

References can be used to create dynamic data structures. Since *StaDyn* provides `var` as a new type to be used in object fields and method parameters, inferring the type of `var` references becomes a complex task. For example, the code in Figure 3 uses a polymorphic `Node` class to create a polymorphic `List`, which has a `head` field pointing to the first `Node`. In the `Main` method, a `Node` whose `data` is a `bool` value is created. Then, a `List` object that references to the original `node` is built. If we obtain the object inside the `Node` inside the `List`, we get a `bool` value (`data1`). Then, an `int` field is set to the `Node` object inside the `List`. Repeating the previous access to the object inside the `Node` object inside the `List`, an `int` value is now obtained (`data2`). As shown in Figure 3, in this scenario the IDE is also capable of showing hints with the types of the `var` references inferred by the compiler. In this case, the invocation to the `setData` method of the `node` object implies modifying the inferred type of the `aList` object. To support this functionality, we have implemented an alias analysis algorithm that allows the IDE to know all the objects a reference may be pointing to [25]. This algorithm makes use of inter-procedural flow information [26], and differentiates between different calls to the same method [27].

2.3. Code Completion

The IDE also uses the type information inferred by the compiler to provide code completion (IntelliSense). IntelliSense helps the programmer speeding up the process of coding by reducing misunderstandings, typos, and other common mistakes. Figure 4 shows the previous example where the `number` reference first holds a `string` and then a `double`. It can be seen how IntelliSense provides different messages for the same reference, depending on the different types inferred by the compiler. Moreover, *StaDyn* detects the error in the last line of code, and the IDE underlines the erroneous statement showing the message: *'Length': no suitable member found*. `Length` is not a valid member, even though

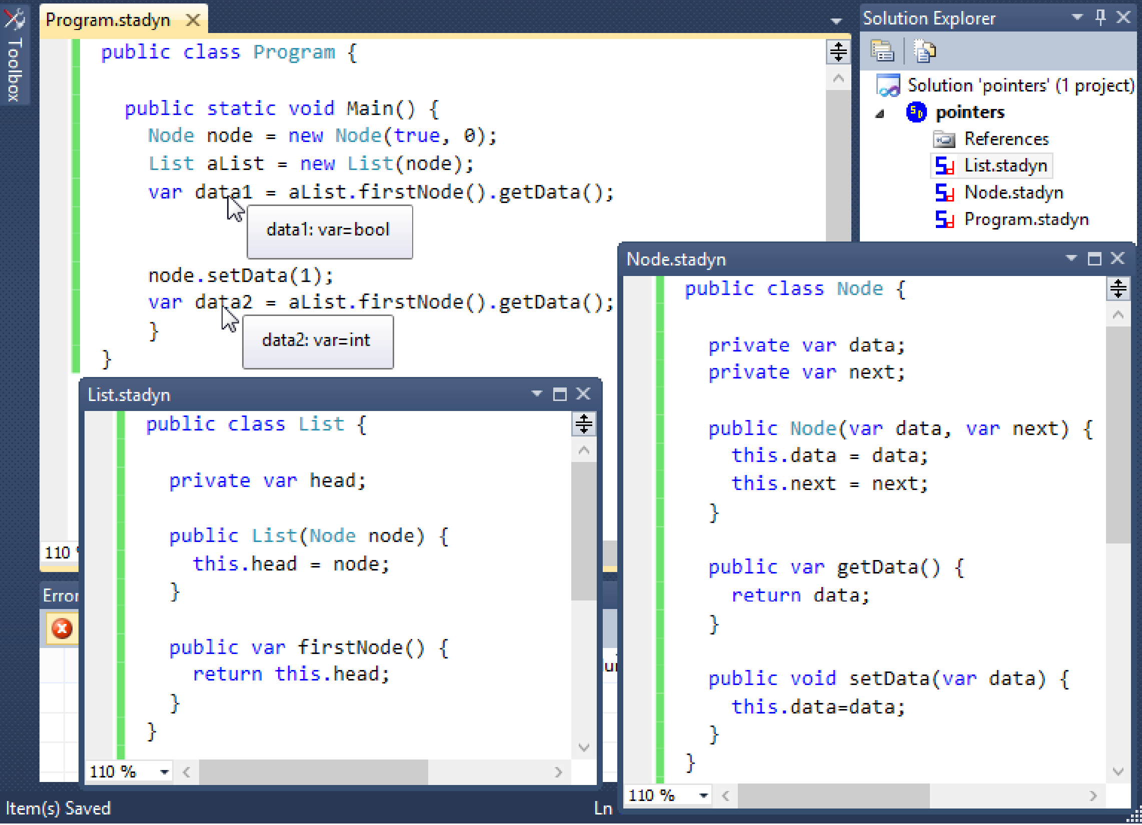

Figure 3: Type hints of polymorphic data structures.

it was valid three lines of code before, because, in the erroneous statement, the new type of `number` is `double`.

2.3.1. Storing Values of Multiple Types

The use of `var` references can lead to situations where a `var` variable has different types depending on the execution flow. In the example code in Figure 5, the `figure` reference may be pointing to either a `Circumference` or a `Rectangle`, depending on the execution flow. The IDE collects concrete type information (opposite to classic abstract type systems) [28], knowing all the possible object types a `var` reference may be pointing to. Instead of declaring a reference with an abstract type that embraces all the possible concrete values, the compiler infers the collection of all the possible concrete types.

The set of messages that can be applied to a collection of concrete types are those accepted by every type in the collection; i.e., the intersection of the message sets. Therefore, Figure 5 shows how, for a given `figure` that can be a `Circumference` or a `Rectangle`,

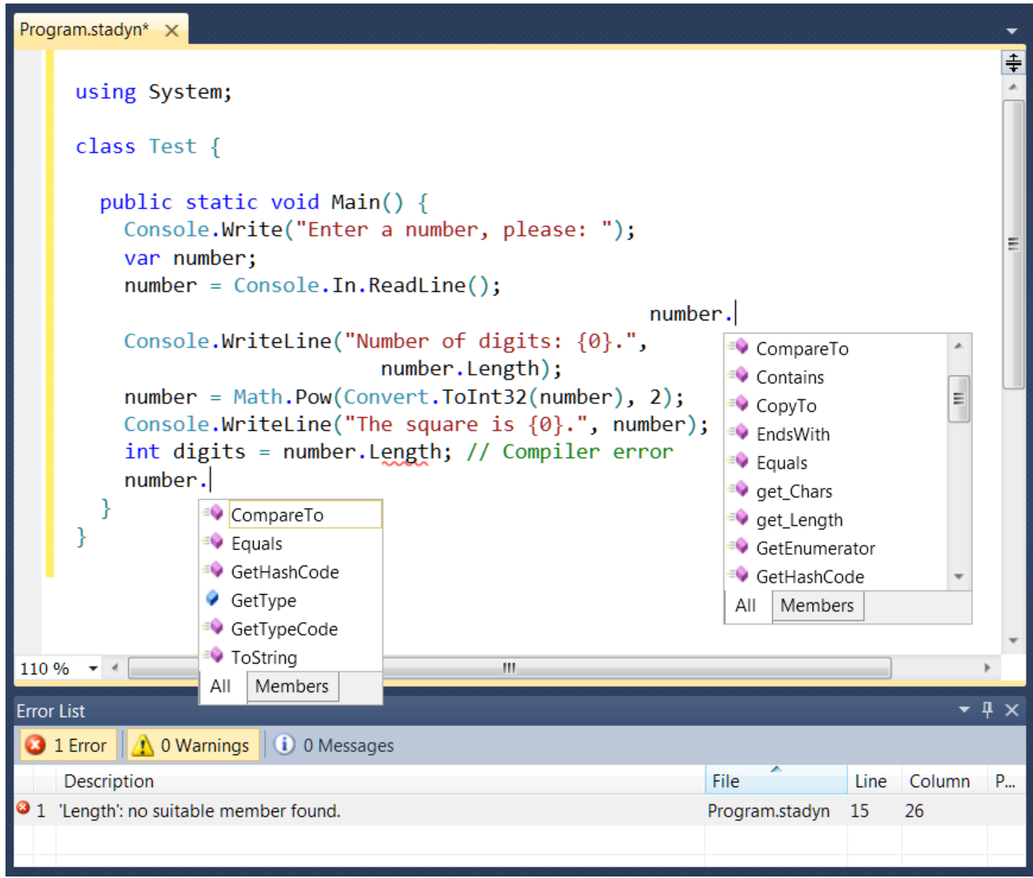

Figure 4: IntelliSense for implicitly typed local variables.

only the `x` and `y` fields provided by both types (and the public methods of `Object`) are offered by IntelliSense.

The key language element we have used to obtain this concrete-type flow-sensitiveness is union types [29]. Concrete types are first obtained by the above-mentioned unification algorithm (applied in assignments and method calls). Whenever a branch is detected, a union type is created with all the possible concrete types inferred. The type of `figure` in Figure 5 is `Circumference` ∨ `Rectangle`, shown by the IDE as a hint, denoting either of these two types [30]. A union type represents the least upper bound of the types it collects (their most specific supertype) [31]. Therefore, the set of messages that can be applied to a union type are those accepted by every type it collects (the intersection).

An outcome of this approach is that IntelliSense provides duck typing for `var` references. Duck typing is a property of dynamic languages that means that an object is

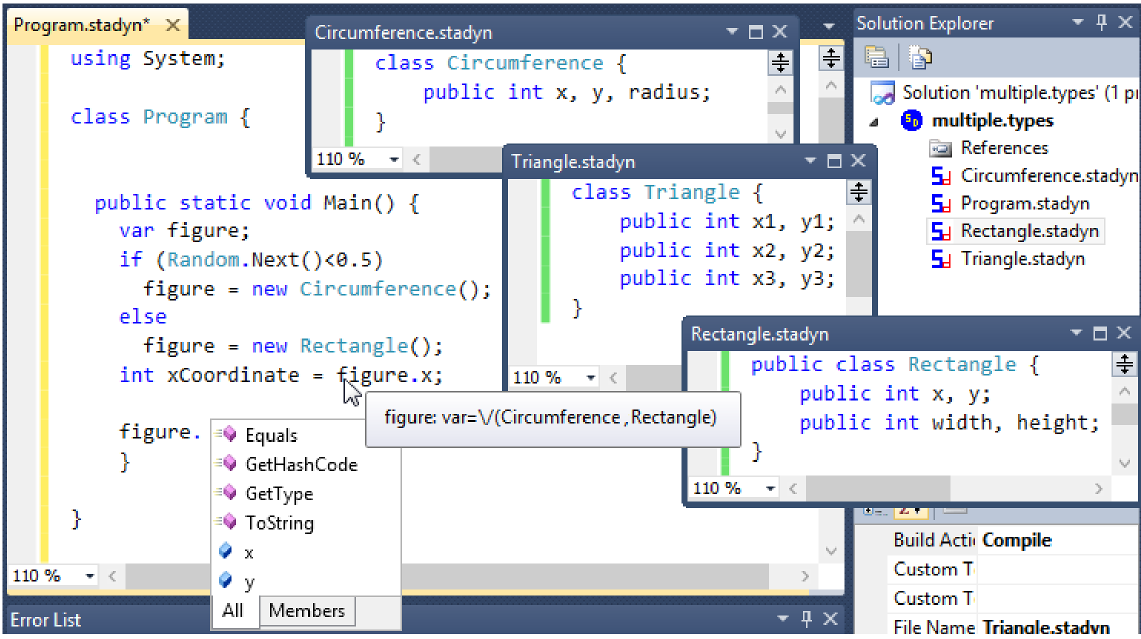

Figure 5: IntelliSense when a reference may hold multiple value types (the constructors of `Circumference`, `Rectangle` and `Triangle` assign random values to their fields; their implementations are omitted for the sake of brevity).

interchangeable with any other object that implements the same dynamic interface, regardless of whether they have a related inheritance hierarchy or not. This is a powerful feature offered by most dynamically typed languages. However, the *StaDyn* IDE offers it with compile-time error detection, code completion, type hints, and runtime performance optimizations. Whenever a `var` reference may point to a set of objects that implement a public `m` method, the `m` message could be safely passed, and hence IntelliSense offers that message to the programmer. These objects do not need to implement a common interface or an (abstract) class with the `m` method.

In the last line of code in the `Program.stadyn` file (Figure 5), the `x` field can be accessed because both `Circumference` and `Rectangle` provide this field. In case the `figure` reference could also be pointing to a third `Triangle` object (as happens in Figure 7), the `x` field would not be offered by IntelliSense, because `Triangle` objects do not provide it.

2.4. Separation of the Dynamism Concern

StaDyn is a hybrid language that allows the programmer to use statically and dynamically typed references. However, the dynamism concern is not explicitly stated in the

source code. `var` references can be either statically or dynamically typed, and its dynamism is specified in a separate XML file [14] transparently managed by the IDE. This makes it possible to customize the trade-off between runtime flexibility of dynamic typing, and runtime performance and robustness of static typing. It is not necessary to modify the application source code to change its dynamism. Therefore, dynamic references could be converted into static ones and vice versa, minimizing the changes in the source code. This feature facilitates the transition from rapid prototyping to efficient software production [32]. It also reduces the changes in the source code, when the programmer requires a more lenient dynamic type system for specific parts of an application. This idea follows the *Separation of Concerns* principle [15] and the *pluggable* type system approach [33].

Figure 6 shows how `var` references can be changed from static to dynamic (and vice versa) with a context menu, right-clicking on a `var` reference. As mentioned, a statically typed union type accepts the intersection of all the types in the union type. However, a more lenient behavior is provided when the reference is dynamic. In that case, it is possible to pass a message to a reference when it is accepted by at least one of the types in the union type; i.e., it accepts the union of all the messages provided by the types in the union type (IntelliSense in the right-hand side of Figure 6) [34]. Therefore, depending on the dynamism of a `var` reference, type checking is more restrictive (static) or lenient (dynamic), but the dynamic semantics of the programming language is not changed (i.e., program execution does not depend on its dynamism). As shown in Figure 6, dynamic references are displayed in red, denoting a kind of caution because dynamic type errors might be produced at runtime (e.g., passing the `radius` message to `figure` when it actually holds a `Rectangle`).

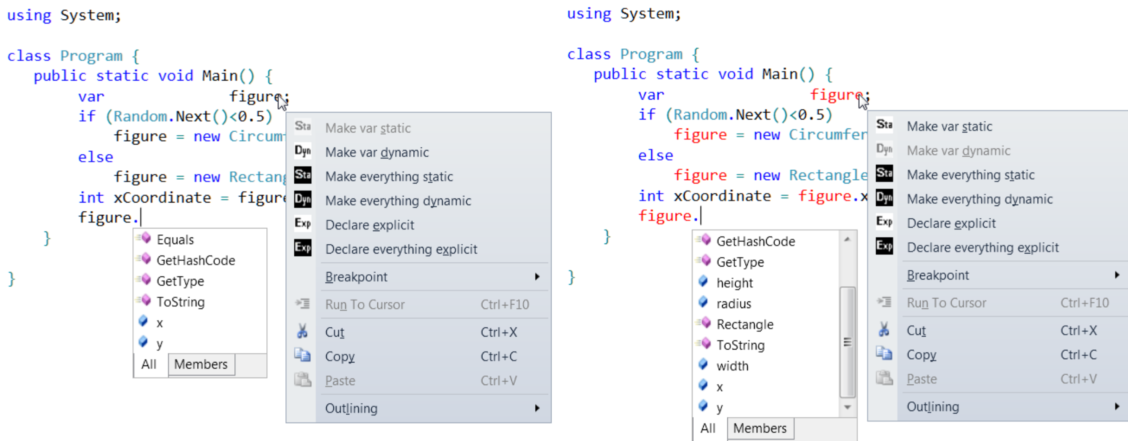

Figure 6: Changing the dynamism concern of `var` references.

2.5. Dynamic References

Another difference with the dynamic languages approach is that *StaDyn* performs compile-time type checking even when dynamic references are used. Setting a reference as dynamic does not imply that any message could be passed to it (unlike dynamic languages); static type-checking is still performed. For example, the IDE shows a compiler error in the last line of code in Figure 7 saying that *the dynamic type Circumference* ∨ *Rectangle* ∨ *Triangle has no valid Y member*. This error is produced because none of these types offers an uppercase `Y` public member. The IDE uses the static type information of dynamically typed code to indicate the appropriate compile-time error. This information is also used to offer code completion for dynamic references (as shown in Figure 7). Unlike *StaDyn*, VS does not provide code completion when the programmer uses `dynamic` references in the C# 4.0 hybrid language.

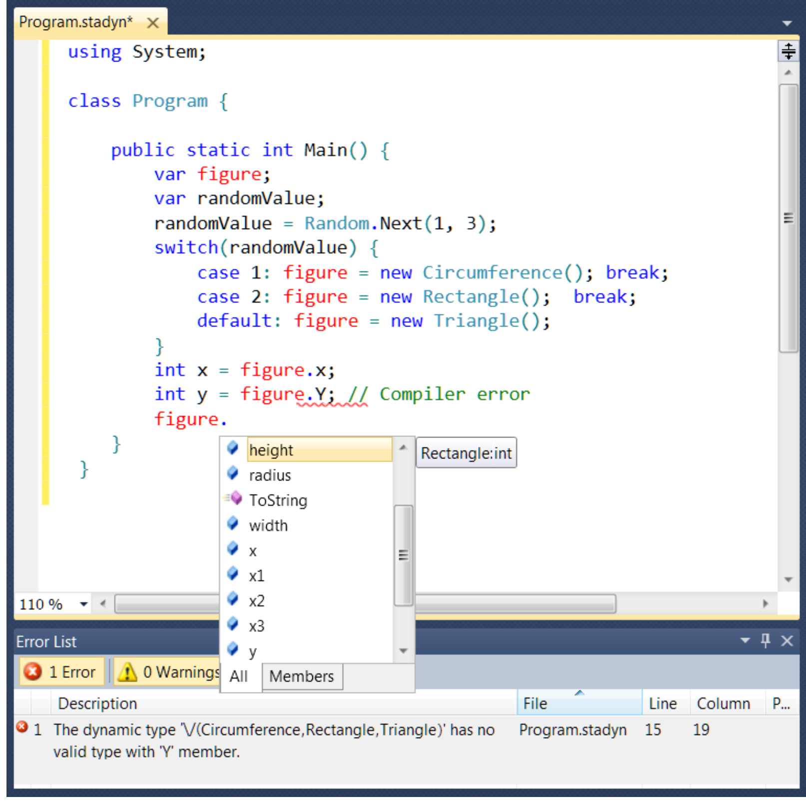

Figure 7: Using compile-time type information of dynamic references.

2.6. Three Different States of Compilation

Separating the dynamism of `var` references facilitates changing dynamically typed code into statically typed one, and vice versa. The *StaDyn* IDE takes advantage of this feature by offering three different modes of compilation: the default one (*managed dynamism*), that takes into consideration the specific dynamism of each single reference; the *everything dynamic* option, that considers every `var` reference as dynamic; and the *everything static* one, that interprets all the `var` references as static.

Figure 8 shows the IDE menu that allows the programmer to select one of the three different modes of compilation. If the programmer is building a prototype, he or she may select the *everything dynamic* option, so that the compiler will interpret all the `var` references in the project as dynamic (their particular dynamism is not changed, it is simply ignored). In this case, we can see how the project icon (bottom-right window) changes from to , and all the `var` references in the project are shown in red. In this state, the subsequent compilations (`Project` | `Build` in the main menu) consider all the `var` references as dynamic. Although compile-time type checking is still performed, the type system is more lenient, being more suitable for rapid prototyping. Therefore, the program in Figure 8 is compiled without any error; even though the `figure` reference was set as static.

The generated program of Figure 8 will not produce any runtime type error because the random number that is generated is always 1 or 2. However, if the programmer, once the prototype has been tested, wants to generate the application with better runtime performance and stronger type checking, he or she can build the project in the *everything static* mode (project icon changes to). Then, all the `var` references will be considered as static, and the IDE will show them in black. In this case, a compilation error is shown saying that `x` is not a valid member of `Triangle`. The programmer should then modify the source code or use the hybrid dynamic and static typing approach (third mode of compilation). If the hybrid option is selected, the programmer changes the dynamism of `figure` to dynamic, and the compilation mode to *managed dynamism* (project icon switches back to). In this case, both the flexibility of dynamic typing and the efficiency and robustness of static typing are being used in the very same application. For instance, `figure` will be consider as dynamic (shown in red) whereas `randomValue` can be declared as static (shown in black).

2.7. Compile-Time Errors of Non-Unified References

In general, implicitly typed (`var`) parameters cannot be unified to a single concrete type. Since they represent any actual type of the corresponding argument, they cannot be inferred the same way as local references or object fields [21]. This issue is shown in the source code of Figure 9. Both the `getX` and the `asString` methods require the parameter

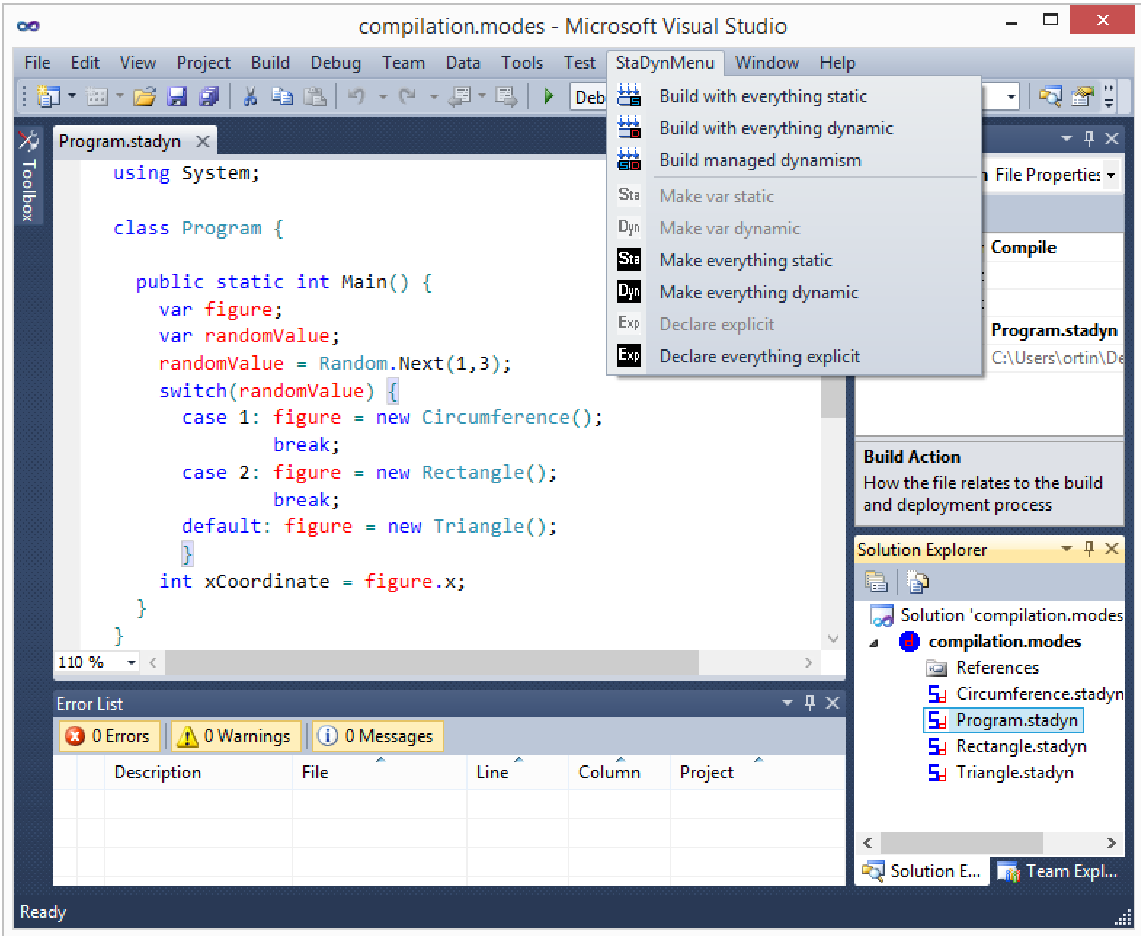

Figure 8: Different modes of compilation.

to implement a specific message (`x` and `ToString`, respectively), returning its value. For the `asString` method, any object could be passed as a parameter because every object in .NET accepts the `ToString` message (including built-in types such as the `33` integer literal). For the `getX` method, however, the parameter should be any object with an `x` field or property (duck typing).

Depending on the type of the actual argument, the IDE shows a different error message. Both `rectangle` and `circumference` can be passed to `getX`, whereas a `Triangle` or an `int` cannot. As shown in Figure 9, the IDE presents two lines for each error of this kind. The first line indicates the operation in the method body that does not fulfill the type system rules, for a given argument: the `x` field is not provided (for the `33` argument). The second line indicates the precise method argument (the `33` integer value has no `x` field) referring

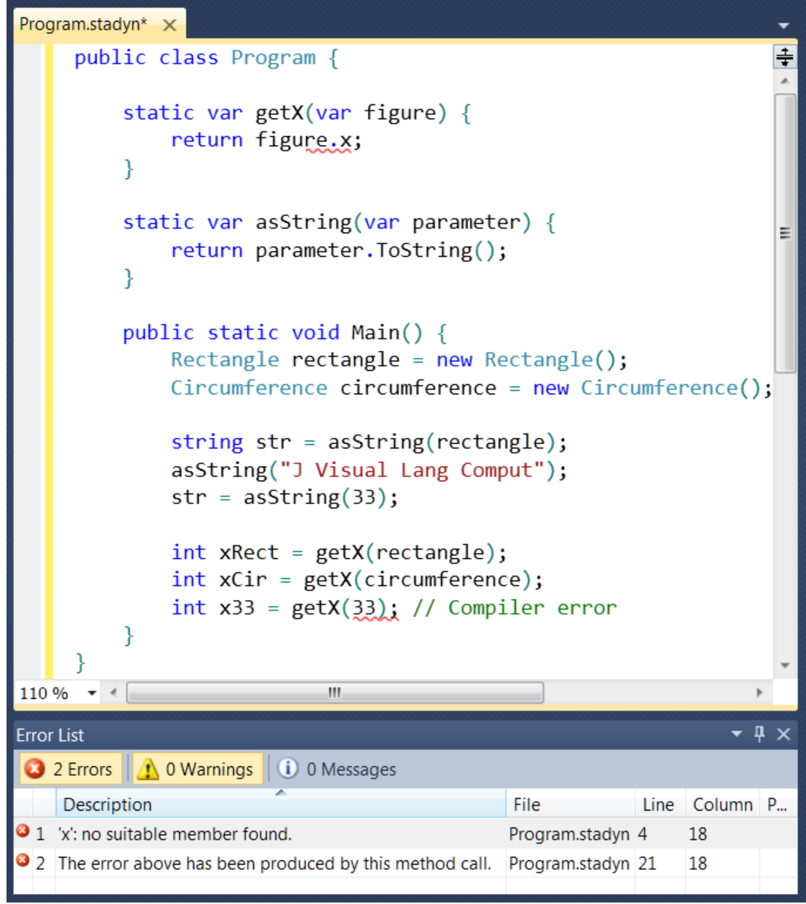

Figure 9: Compile-time errors of non-unified references.

to the previous error message. The two erroneous code sections are also underlined by the IDE.

2.8. Making Explicit the Implicit References

The IDE also uses the compile-time type information inferred by the *StaDyn* compiler to replace implicitly typed `var` declarations with explicit ones. The *declare explicit* option of either the *StaDyn* menu (Figure 8) or the pop-up window that appears right-clicking on a `var` declaration (Figure 10) allow substituting `var` with the explicit type inferred by the compiler. In the example scenario shown in Figure 10, the declaration of the `first` reference is replaced with the explicit `string` type. The *declare everything explicit* option does the same for every `var` reference in the current file. Applying this option to the

`Program` class in Figure 10, the declaration of the `second` reference will be replaced with the `Exception` type, and `wrapped` and `wrapper` with the `Wrapper` class.

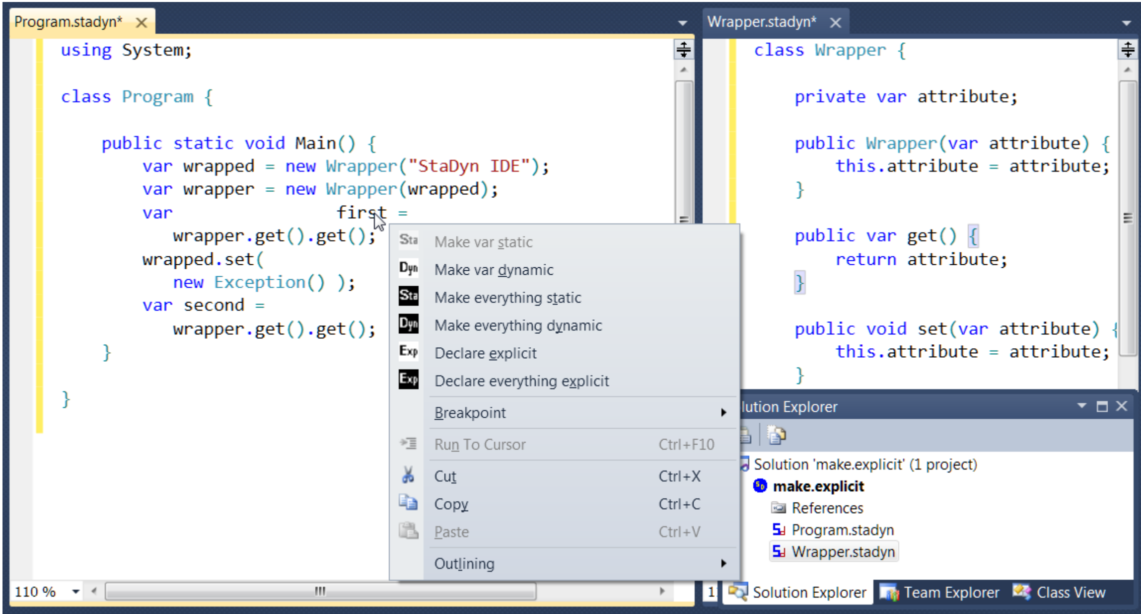

Figure 10: Converting implicitly typed references into explicitly typed ones.

As described in Section 2.3.1, there are references that may be holding different values of unrelated types depending on the execution flow. The compiler internally represents these values with union types, but neither *StaDyn* nor C# offer that type constructor to the programmer. Therefore, the *make explicit* functionality is only applicable when the compiler is able to infer a single type for a `var` reference; otherwise, a message is shown to the programmer. However, for those languages that support union types explicitly (such as Pike [35] or Whiley [36]), the IDE would be able to make the corresponding type substitution.

2.9. Runtime Performance

Although runtime performance is not an IDE feature, it could affect the development process, especially when dynamic typing is used (see Section 4). For this reason, we present a summary of the runtime performances of the existing hybrid languages for the .NET platform: *StaDyn*, C#, Visual Basic, Boo, Cobra and Fantom [37]. We took 58 applications that use dynamic, static and hybrid typing [37].

For fully dynamically typed code, *StaDyn* provides the best runtime performance. On average, it is 2.53, 3.2, 11.68, 14.8 and 29.9 times faster than C#, Boo, VB, Fantom and

Cobra, respectively [37]. These results show the benefit of using the type information inferred by the compiler to optimize the generated code. For hybrid static and dynamic typing code, *StaDyn* also provides the best performance: 5.1, 8.8, 16.7, 53.7 and 127 times higher than C#, Boo, VB, Fantom and Cobra, respectively [37].

The last scenario is when all the types are explicitly declared. In this case, C# is the language that obtains the lowest execution times, being on average 2.5% faster than *StaDyn*, the second fastest implementation. This result is caused by the greater number of optimizations the production C# compiler performs in relation to our language implementation. Compared with the rest of languages, *StaDyn* is, on average, 24%, 48%, 243% and 335% faster than Cobra, VB, Boo, and Fantom, respectively.

Regarding memory consumption, *StaDyn* was the language implementation that required the lowest memory resources for all the scenarios [37].

3. Implementation

Both the programming language and the IDE described in this paper have been implemented in the .NET Framework 4.5 platform, using the C# programming language. Our compiler is a multiple-pass language processor that follows the *Pipes and Filters* architectural pattern [38]. We have used the AntLR language processor tool to implement lexical and syntactic analysis. ASTs have been implemented following the *Composite* design pattern [39], and each pass over the AST implements the *Visitor* design pattern [39].

We have developed the following AST visits: two visitors for the SSA algorithm; two visitors to load types into the type table; one visitor for symbol identification and another one for type inference; and two visitors to generate code. The type system has been implemented following the guidelines described in [40] and the formalization depicted in [21].

We generate .NET intermediate language (IL) and then assemble it to produce the binaries. At present, we use the CLR 2.0 as the unique compiler's back-end. However, we have designed the code generator module following the *Parallel Hierarchies* design pattern [41] to add both the DLR [42] and the ЯRotor [43] back-ends (current work).

The VS IDE has been customized using both the Managed Extensibility Framework (MEF) and the Managed Package Framework (MPF). The previous prototype customized VS 2008 by implementing a collection of Visual Studio extension packages (*VSPackages*) that needed to be installed in the Windows registry. However, VS 2010 and 2012 use the MEF component framework to facilitate the customization of the VS IDE, avoiding hard dependencies among the component parts [44]. MEF has allowed us to create editor plug-ins for syntax highlighting, IntelliSense, type information hints, error messages, and context menus. The editor plug-ins export (provide) and import (consume) MEF component parts [44]. As shown in Figure 11, the editor plug-ins import some editor extension

points exported by VS. When an event related to these extension points occurs, the plug-in invokes the compiler to perform lexing, parsing and type inference of the current project. The type-annotated AST is returned to the plug-in that, using the annotated AST, responds to the event generated. Our plug-ins export this information that is in turn imported by the VS, producing the desired customization.

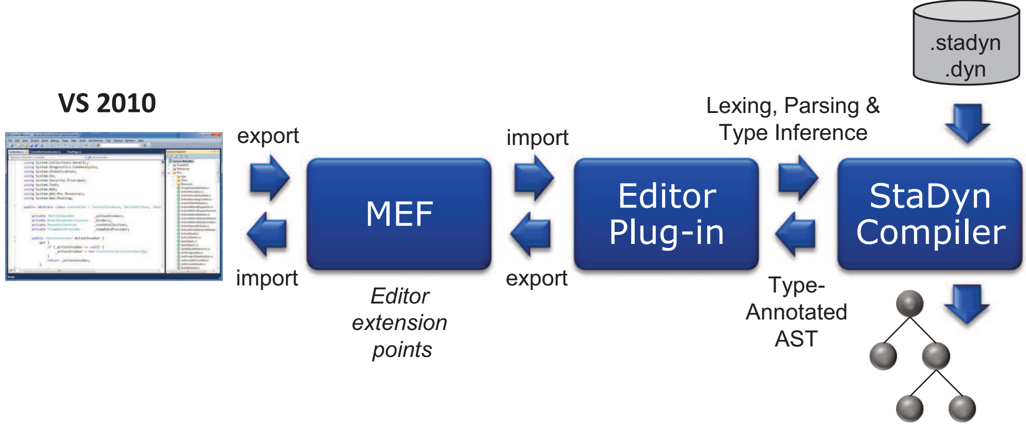

Figure 11: Customization of the VS IDE using the Managed Extensibility Framework.

The Managed Package Framework (MPF) has been used to implement two VS extension packages (VSPackages): the *StaDyn* project and the *StaDyn* menu commands. These two packages were extended from the previous version of the IDE [18].

4. Evaluation

We have conducted a user study to evaluate the *StaDyn* IDE. This evaluation is aimed at showing how the IDE supports programmers in certain tasks, comparing our approach with similar systems. For this purpose, we have designed two experiments (Sections 4.1 and 4.2) conducted by 10 participants, all of them graduate students in Computer Science. All the students are experienced Java programmers, but they barely knew the rest of languages used, including *StaDyn*.

4.1. First Experiment: Programmer's Performance using the StaDyn IDE

For both experiments, we applied the GQM (Goal, Question and Metric) approach to perform the evaluation [45]. The goal of this first experiment is to evaluate the programmer's performance using the *StaDyn* IDE, compared with the command-line plus editor approach.

The procedure used for this first experiment is presented in Figure 12. The participants are given a brief introduction (10 minutes) to the experiment goal. Then, a short lecture (30 minutes) about the *StaDyn* programming language and the compiler command-line options is presented to the students. After this lecture, the participants implement a warm-up program in *StaDyn*, using their favorite editor and the command-line compiler. In this session, they could ask any question to the instructor, learning how to apply the correct language features and compiler options. Once the warm-up program is correctly implemented, they are asked to develop a more difficult application with the same tools. For this second program, the development time is recorded.

The second part of the experiment starts with another lecture about the features of the *StaDyn* IDE (10 minutes). The participants then implement the same warm-up exercise using the IDE. Afterwards, they develop a new different program, recording the elapsed development time. Finally, a questionnaire is used to evaluate in which terms the IDE has facilitated the development process.

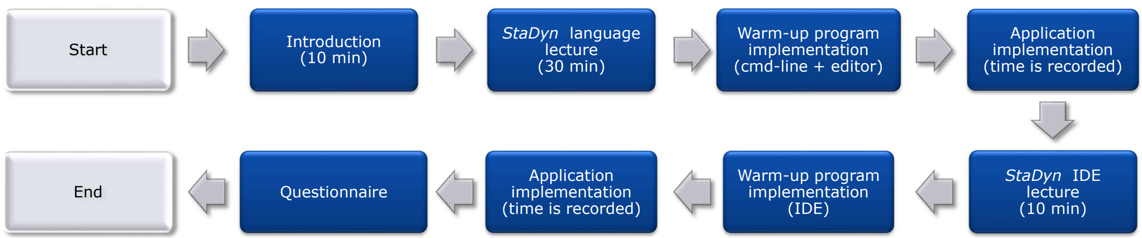

Figure 12: Procedure used for the first experiment.

For each of the two scenarios (command-line plus editor and IDE), we selected a distinct program. If the same program would have been implemented, development time had been affected by the ordering of implementation. However, selecting two different programs requires them to have a similar degree of complexity in order to compare their development times. Consequently, we selected consecutive Euler problems from problem 29 to problem 34 (i.e., 29 and 30, 30 and 31, . . . , 33 and 34) [46]. The first problem was developed with the editor plus command line option, and the second one with the IDE. Therefore, 5 different pairs of programs were created, and each pair was developed by 2 of the 10 participants. To evaluate whether both problems have a similar degree complexity, the questionnaire asks the students if they think so (Q1.2 in Table 1). Euler 25 is the warming-up exercise used in all the experiments.

All the programs, including the warm-up one, were implemented following the next process. First, a rapid prototyping approach was followed, where all the references in the program must be declared as dynamically typed. Then, the prototype must be converted into a robust and efficient program with fully statically typed code. The last step was

ensuring that all the variables in the final program have their types explicitly declared.

In the GQM approach followed [45], different questions were elaborated to cover the goal of evaluating the programmer's performance using the *StaDyn* IDE. For each question in the questionnaire, the students specified their level of agreement in a 5-point Likert scale [47]: strongly disagree (1), disagree (2), neutral (3), agree (4), or strongly agree (5). Such scale has been widely used in the evaluation of usability and preference in human-computer interaction research [48]. Table 1 shows the questions used. The first question (Q1.1) is aimed at covering the experiment goal. The second one (Q1.2), as mentioned, validates the premise that the two selected programs have a similar degree of complexity. The rest of questions are aimed at identifying which IDE features influenced on improving the programmer's performance.

Q1.1	The IDE has significantly improved my performance compared with the command-line plus editor option.
Q1.2	This problem was as complex as the one implemented using the command line options.
The following features of the IDE have been useful for improving my performance:	
Q1.3	Compile-time type error detection of dynamic references.
Q1.4	Code completion of dynamic references.
Q1.5	Separation of the dynamism concern.
Q1.6	The "build with everything dynamic" compilation state of the IDE.
Q1.7	The "build with everything static" compilation state of the IDE.
Q1.8	Make explicit.
Q1.9	Type hints.

Table 1: Questionnaire to evaluate the programmer's performance using the *StaDyn* IDE.

4.1.1. Results

Figure 13 shows the elapsed time required by each participant to develop the two applications. Using the *StaDyn* IDE, all the students developed the program significantly faster than with the command-line plus editor option. On average, development time using the IDE was 38.3% the time employed without it.

Table 2 shows the distribution of the responses of the 10 participants to the questionnaire presented in Table 1. All the students but one strongly agreed that the *StaDyn* IDE significantly improved their performance compared with the command-line plus editor option (Q1.1). All the participants agreed that the two developed programs have a similar degree of complexity (Q1.2).

Regarding the features provided by the IDE, code completion of dynamically typed references (Q1.4) was evaluated as the most useful feature. Type hints (Q1.9) obtained the lowest evaluation, with a median value of *agree* (4) and a standard deviation of 0.67. The rest of features were evaluated with a minimum value of 4.7.

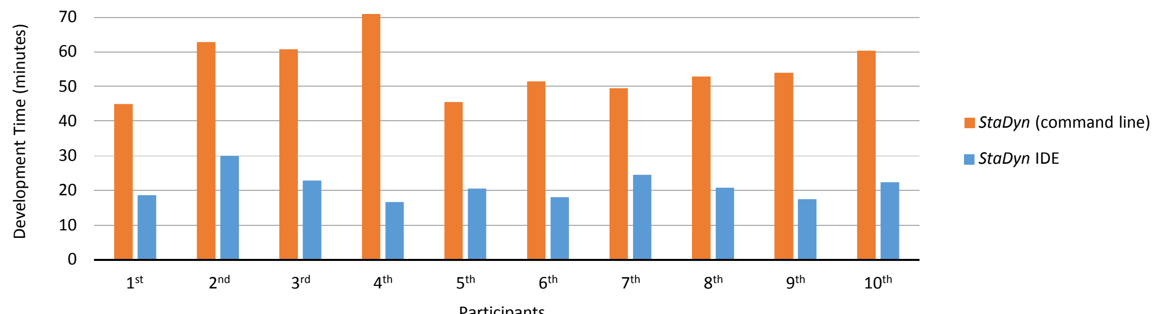

Figure 13: Development time (minutes) used by each participant to implement the two *StaDyn* programs.

	Q1.1	Q1.2	Q1.3	Q1.4	Q1.5	Q1.6	Q1.7	Q1.8	Q1.9
Strongly agree	9	3	8	10	7	9	9	8	4
Agree	1	7	2		3	1	1	2	5
Neutral									1
Disagree									
Strongly disagree									
Average	**4.9**	**4.3**	**4.8**	**5.0**	**4.7**	**4.9**	**4.9**	**4.8**	**4.3**

Table 2: Distribution of the answers to the questionnaire in Table 1.

4.2. Second Experiment: Programmer's Performance using Similar IDEs

The goal of this second experiment is to evaluate the programmer's performance using other IDEs for hybrid static and dynamic typing languages. In particular, we evaluate whether the features provided by each IDE are effective for reducing development time of dynamically and statically typed code, plus the transition from the former to the latter.

We restricted our evaluation to the existing IDEs of hybrid languages for the .NET framework to facilitate the comparison among them [37]. The IDE selected for each language was that recommended by the language developers. Visual Studio 2012 was used for C# –Visual Basic was not included in the experiment, because this IDE provides the same features for both languages. SharpDevelop 4.4 was used for Boo, Xamarin Studio 4.2.3 (former MonoDevelop) for Cobra, and the F4 1.0.2 Eclipse-based IDE for Fantom.

The procedure is based on performing the same experiment for the 5 different IDEs. As done with the *StaDyn* IDE, we start with a lecture about the features of the visual environment to be evaluated (10 minutes). The students first implement the warm-up exercise (Euler 25) to get used to the IDE. Then, they develop a new program and the elapsed development time is recorded. Finally, a questionnaire is answered by the participants to evaluate the IDE. This procedure is applied for all the languages. The questionnaire for the *StaDyn* IDE is passed just after the first experiment (Section 4.1).

In the previous experiment, each participant implemented two consecutive programs

from Euler 29 to 34. In this second experiment, we use the four remaining programs so that each student implements a different program in a different IDE (and language). Each 2 of the 10 participants conducted the same experiment. As in the first experiment, they followed the process of first creating a rapid prototype, and then converting it into a robust explicitly typed application.

Table 3 shows the questionnaire used for this second experiment. The 5-point Likert scale questions, together with the elapsed development time, are aimed at evaluating the programmer's performance using similar IDEs for hybrid static and dynamic typing languages. Q2.1 validates whether the 6 selected programs have a similar degree of complexity. The objective of questions Q2.2 and Q2.4 is evaluating whether each IDE was useful in reducing the development time of dynamic and static typing scenarios; Q2.3 is focused on the transition from the former scenario to the latter. The question Q2.5 evaluates the lack of features provided by the IDEs, and Q2.6 indicates whether runtime performance of dynamically typed code influenced on the elapsed development time.

Q2.1	This problem was as complex as those implemented in the other IDEs.
The IDE provides features that have helped me to	
Q2.2	Reduce the development time of the dynamically typed program.
Q2.3	Perform the transition from dynamically to statically typed code.
Q2.4	Reduce the development time of the statically typed application.
Q2.5	The lack of features provided by the IDE has somehow influenced the development time.
Q2.6	The low runtime performance of the dynamically typed code has increased the development time.

Table 3: Questionnaire to evaluate the programmer's performance using each IDE.

4.2.1. Results

Figure 14 shows the average development time using each IDE. Values are grouped by program (i.e., Euler problem) to reflect any possible influence of the problem on the elapsed development times. The development times using the *StaDyn* IDE have been the lowest for all the programs. On average, the participants employed 33.48%, 75.6%, 90.25% and 92.27% more development time when they used the IDEs for C#, Boo, Cobra and Fantom, respectively.

To assess the similarity among the complexity degree of the problems, we added the Q2.1 question to the questionnaire shown in Table 3. Figure 15 shows the answers of the participants to this question, considering each problem. They identified Euler 31 as the problem with the lowest degree of complexity similarity. This may be the cause of its higher average development time in Figure 14. However, the average value is close to *agree* (3.7), third quartile is 4.5, and first quartile is *neutral* (3). For the rest of programs,

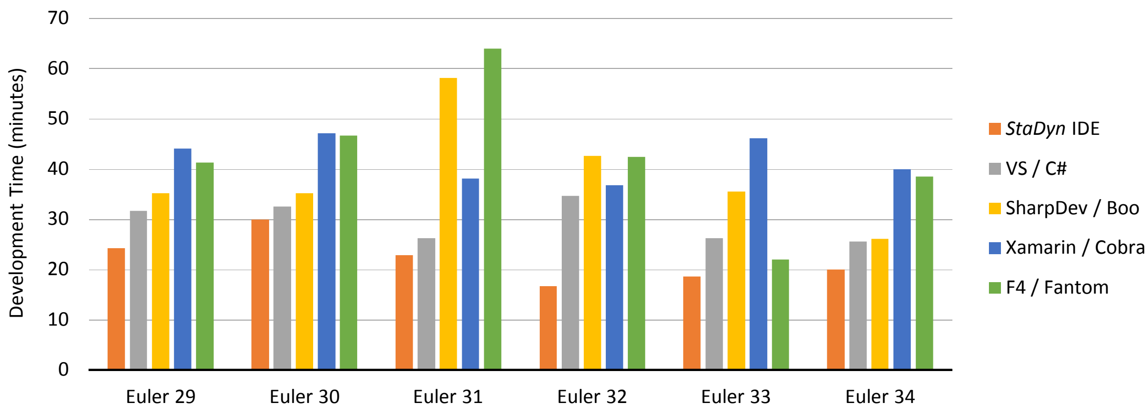

Figure 14: Average development time (minutes) used to implement the 6 selected programs in each IDE.

the students agreed that the problems have a similar degree of complexity (average is above 4, and third quartile is at least *agree*).

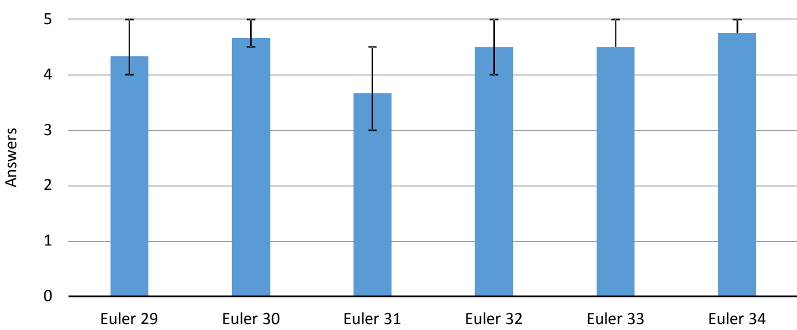

Figure 15: Average responses to Q2.1 (questionnaire in Table 3) grouped by programs; whiskers show first and third quartile.

Figure 16 shows the answers to the rest of questions (questionnaire in Table 3). Q2.2 is aimed at evaluating whether the IDE helps the programmer to reduce development time when creating dynamically typed code. The only IDE that received a positive answer was *StaDyn* (4.25). All the participants strongly agreed that the *StaDyn* IDE facilitates the transition from dynamic to static typing (Q2.3); the rest of IDEs obtained a maximum average evaluation of 1.25 for this question. For statically typed code (Q2.4), the participants agreed that all the IDEs are useful for reducing development time (4 was the smallest average value, obtained by F4 / Fantom).

Q2.5 states that the of lack features provided by each IDE has somehow influenced on the development times. The average value for the *StaDyn* IDE was 1.5, between *strongly*

disagree and *disagree*. For Visual Studio, the average answer was higher than *neutral* (3.5). The participants agreed that the rest of IDEs lack some features, influencing the time they used to develop their programs.

The last question (Q2.6) is related to the language implementation rather than to the IDE. It asks the participants whether the lower runtime performance of dynamically typed code increased the development times. The participants answered that *StaDyn* did not cause this development time increase (1.5). On average, they agreed that the lower performance of Fantom's dynamic typing implementation raised the elapsed development time. The rest of languages obtained an average value between *neutral* and *agree*.

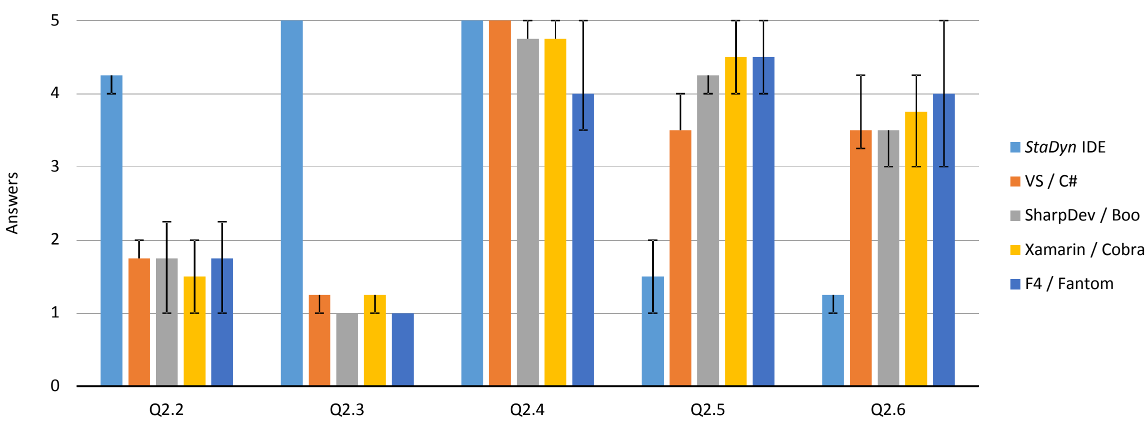

Figure 16: Average answers to the questionnaire shown in Table 3; whiskers show first and third quartile.

5. Related Work

In this section, we analyze the existing approaches related to ours, considering the existing IDEs for hybrid static and dynamic typing languages. A detailed analysis of the programming languages plus some other theoretical approaches are discussed in [17].

Boo is an object-oriented programming language for the CLI with Python inspired syntax [49]. In Boo, a reference may be implicitly declared making the compiler infer its type (references could only have one unique type in the same scope), but fields and parameters cannot be declared without specifying their type. The Boo compiler provides the `ducky` option that interprets the `Object` type as if it was `duck`, i.e. dynamically typed. This approach follows the idea of separating the *dynamism* concern, but does not reduce the number of changes to be done in the source code. SharpDevelop, Boo Explorer and the BooLangStudio plug-in for VS 2008 are the existing IDEs for the Boo programming language. Their features include syntax highlighting, building and debugging services,

and basic IntelliSense capabilities for statically typed references only. SharpDevelop also provides services for converting C# to Boo, three refactoring operations, and some code templates to help the programmer get started.

The Fantom programming language generates both JVM and .NET code, offering dynamic and static typing [50]. Instead of adding a new type, dynamic typing is provided with the `->` dynamic invocation operator. Unlike the dot operator, the dynamic invocation operator does not perform compile-time checking. Fantom does not follow the separation of concerns principle. The existing IDEs for Fantom are F4 (an Eclipse-based IDE), Netbeans FantomIDE (either as a plug-in or as a standalone IDE), and the Fantom bundle for TextMate. F4, the most advanced one, provides code completion for statically typed code, debugging, search for code elements, and useful navigating services. F4 does not provide code completion when the dynamic invocation operator (`->`) is being used.

Cobra is another hybrid typing programming language with different IDEs [51]. The Cobra approach is similar to C# 4.0, offering a `dynamic` type for dynamic typing. The language is compiled to .NET assemblies. Different IDEs are provided: Xamarin Studio (MonoDevelop), Visual Cobra (an extension of VS 2010), a plug-in for SharpDevelop, and Naja (written in Cobra). All of them offer editing, compiling, and syntax highlighting. Xamarin, the IDE recommended by the Cobra developers, provides interactive debugging, tooltips, and some support for code completion. This last feature is not supported for dynamic references, and the *dynamism* concern is not separated.

Objective-C is a programming language that supports both static and dynamic typing, and it is commonly natively compiled [52]. In Objective-C, variables declared with the `id` type are dynamically typed, postponing type checking until runtime. No type information is gathered by the compiler. Apart from advanced editors such as Textmate and BBEdit, Xcode and JetBrains AppCode are two full-featured IDEs for Objective-C. Both include debugging, code navigation, and code completion for statically typed code. When the `id` dynamic type is used, autocomplete only shows the messages provided by any object (`NSObject`).

As we have previously mentioned, Visual Studio offers dynamic typing for both C# and VB. However, they do not infer any type information for dynamic references. As a result, IntelliSense is not provided for these variables (`dynamic` in C# and implicitly typed variables in VB).

The IDE described in this paper is an evolution of an initial prototype [18]. The first prototype only provided some features for local variables when a single type was inferred in its scope[1]. In this new stable version, code completion, type hints, and the modification

[1]The programming language supported `var` fields and parameters, but the IDE did not provide these services for those references.

of the dynamism of implicitly typed references are provided for `var` fields, parameters, and return types, and for local variables with multiple types in the same scope. The three different states of compilation described in Section 2.6 were not provided by the previous version (the previous version allowed compilation with every `var` reference as static or dynamic, but that state was not saved for the following compilations, and the color of the `var` references and IntelliSense did not change accordingly). IntelliSense was not activated automatically; and the appropriate keywords, types, identifiers in scope, and class members were not provided either. The current version also introduces new minor features such as brace, parenthesis and square bracket matching, code completion for built-in types, overloaded methods and expressions with multiple dots, and hints for every `var` expression (not only declarations). For implicitly typed parameters, error messages show both the method body and the specific argument in each erroneous method invocation. The last version is released as 2 different implementations for VS 2010 and 2012, and it is more maintainable and scalable due to the benefits of using the MEF component framework [44]. Finally, this IDE incorporates the last version of the *StaDyn* programming language [37], which includes support of value types, properties, implicitly typed static fields, autoboxing and unboxing of `var` fields, and full support of the + overloaded operator.

6. Conclusions

Gathering type information of dynamically typed code at compile time is a valuable mechanism to improve the features of the programming language IDEs, offering services commonly provided when statically typed code is being processed (such as code completion and early detection of type errors). Moreover, if the programming language is hybrid dynamically and statically typed, specific services aimed at facilitating the transition from dynamically to statically typed code (and vice versa) can also be provided. The Separation of Concerns principle facilitates the implementation of these services. The proposed features have been implemented as part of a production IDE for a research hybrid programming language, obtaining significant benefits on the programmer's performance compared to its counterparts.

Future work will be adding debugging capabilities to our extension of Visual Studio. The current implementation of the *StaDyn* programming language, the binaries and source code of the visual IDE (both the VS 2010 and 2012 implementations), and the benchmark and source code examples presented in this paper are freely available at

`http://www.reflection.uniovi.es/stadyn/download/2014/jvisualang`

Acknowledgments

We would like to thank the anonymous reviewers for their indications, corrections and suggestions to improve the article. We also thank the anonymous participants that participated in the two experiments described in Section 4.

This work was partially funded by Microsoft Research to develop the project entitled *Extending Dynamic Features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution* Request for Proposals. This work was also funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*.

References

[1] D. Thomas, C. Fowler, A. Hunt, Programming Ruby, 2nd Edition, Addison-Wesley, 2004.

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

[3] A. Hunt, D. Thomas, The Pragmatic Programmer: from Journeyman to Master, Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999.

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

[5] D. Crane, E. Pascarello, D. James, AJAX in Action, Manning Publications, Greenwich, 2005.

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

[7] A. Latteier, M. Pelletier, C. McDonough, P. Sabaini, The Zope2 book, `http://docs.zope.org/zope2/zope2book` (2014).

[8] Django Software Foundation, Django, the Web framework for perfectionists with deadlines, `http://openjdk.java.net/projects/mlvm` (2014).

[9] G. M. Bierman, E. Meijer, M. Torgersen, Adding dynamic types to C#, in: European Conference on Object-Oriented Programming (ECOOP), 2010, pp. 76–100.

[10] F. Ortin, P. Conde, D. Fernandez-Lanvin, R. Izquierdo, Runtime performance of invokedynamic: Evaluation through a Java library, IEEE Software (2014) 1–16.

[11] F. Ortin, M. A. Labrador, J. M. Redondo, A hybrid class- and prototype-based object model to support language-neutral structural intercession, Information and Software Technology 56 (2) (2014) 199–219.

[12] E. Meijer, P. Drayton, Static typing where possible dynamic typing when needed: The end of the cold war between programming languages, in: Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004, pp. 1–6.

[13] M. Abadi, L. Cardelli, B. C. Pierce, D. Rémy, Dynamic typing in polymorphic languages, Journal of Functional Programming 5 (1) (1995) 111–130.

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

[15] W. Hürsch, C. Lopes, Separation of Concerns, Technical Report NU-CCS-95-03, Northeastern University, Boston, 1995.

[16] ECMA, ECMA-334 standard: C# language specification 4th edition, `http://www.ecma-international.org/publications/standards/Ecma-334.htm` (2009).

[17] F. Ortin, Type inference to optimize a hybrid statically and dynamically typed language, The Computer Journal 54 (11) (2011) 1901–1924.

[18] F. Ortin, A. Morant, IDE support to facilitate the transition from rapid prototyping to robust software production, in: Proceedings of the 1st Workshop on Developing Tools as Plug-ins, TOPI '11, ACM, New York, NY, USA, 2011, pp. 40–43.

[19] R. Milner, M. Tofte, R. Harper, The definition of Standard ML, MIT Press, Cambridge, MA, USA, 1990.

[20] P. Hudak, S. Peyton Jones, P. Wadler (editors), Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2), ACM SIGPLAN Notices 27 (5).

[21] F. Ortin, M. Garcia, Supporting dynamic and static typing by means of union and intersection types, in: Proceedings of the IEEE Progress in Informatics and Computing (PIC), 2010, pp. 993–999.

[22] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (1978) 348–375.

[23] L. Cardelli, Basic polymorphic typechecking, Science of Computer Programming 8 (2) (1987) 147–172.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490.

[25] A. W. Appel, Modern compiler implementation in ML: basic techniques, Cambridge University Press, New York, NY, USA, 1997.

[26] W. Landi, B. G. Ryder, A safe approximate algorithm for interprocedural aliasing, in: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, PLDI '92, ACM, New York, NY, USA, 1992, pp. 235–248.

[27] M. Emami, R. Ghiya, L. J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, ACM, New York, NY, USA, 1994, pp. 242–256.

[28] J. Plevyak, A. A. Chien, Precise concrete type inference for object-oriented languages, in: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications, OOPSLA '94, ACM, New York, NY, USA, 1994, pp. 324–340.

[29] 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, http://www.cis.upenn.edu/~bcpierce/papers/thesis.ps (1991).

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

[31] A. Aiken, E. L. Wimmers, Type inclusion constraints and type inference, in: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, ACM Press, Copenhagen, Denmark, 1993, pp. 31–41.

[32] F. Ortin, D. Zapico, M. Garcia, A programming language to facilitate the transition from rapid prototyping to efficient software production., in: J. A. M. Cordeiro, M. Virvou, B. Shishkov (Eds.), International Conference on Software and Data Technologies (ICSOFT), SciTePress, 2010, pp. 40–50.

[33] G. Bracha, Pluggable Type Systems, in: Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004, pp. 1–6.

[34] F. Ortin, M. Garcia, Union and intersection types to support both dynamic and static typing, Information Processing Letters 111 (6) (2011) 278–286.

[35] F. Hübinette, The Pike programming language, `http://pike.lysator.liu.se` (2014).

[36] D. J. Pearce, L. Groves, Whiley: a platform for research in software verification, in: Proceedings of the International Conference on Software Language Engineering (SLE), 2013, pp. 1–10.

[37] M. Garcia, Improving the performance and robustness of hybrid statically and dynamically typed programming languages, Ph.D. thesis, Computer Science Department, University of Oviedo, Spain (Jun. 2013).

[38] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-oriented software architecture: a system of patterns, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[39] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, 1995.

[40] F. Ortin, , D. Zapico, J. M. Cueva, Design patterns for teaching type checking in a compiler construction course, IEEE Transactions on Education 50 (3) (2007) 273–283.

[41] F. Ortin, M. Garcia, A type safe design to allow the separation of different responsibilities into parallel hierarchies, in: International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2011, pp. 15–25.

[42] B. Chiles, A. Turner, Dynamic Language Runtime, `http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512` (2014).

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

[44] CodePlex, Microsoft .NET MEF, Managed Extensibility Framework, http://mef.codeplex.com (2014).

[45] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, in: Encyclopedia of Software Engineering, Wiley, 1994, pp. 527–532.

[46] Euler, Project Euler .net, https://projecteuler.net (2014).

[47] R. Likert, A technique for the measurement of attitudes, Archives of Psychology 22 (140) (1932) 1–55.

[48] M. C. Kaptein, C. Nass, P. Markopoulos, Powerful and consistent analysis of Likert-type ratingscales, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, ACM, New York, NY, USA, 2010, pp. 2391–2394.

[49] CodeHaus, Boo, a wrist friendly language for the CLI, http://boo.codehaus.org (2014).

[50] B. Frank, A. Frank, Fantom, the language formerly known as Fan, http://fantom.org (2014).

[51] J. Siegel, D. Frantz, H. Mirsky, R. Hudli, P. de Jong, A. Klein, B. Wilkins, A. Thomas, W. Coles, S. Baker, M. Balick, COBRA fundamentals and programming, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[52] B. J. Cox, Message/object programming: An evolutionary change in programming technology, IEEE Software 1 (1) (1984) 50–61.