Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages

Erik Meijer and Peter Drayton
Microsoft Corporation

Abstract

This paper argues that we should seek the golden middle way between dynamically and statically typed languages.

1 Introduction

<NOTE to="reader"> *Please note that this paper is still very much work in progress and as such the presentation is unpolished and possibly incoherent. Obviously many citations to related and relevant work are missing. We did however do our best to make it provocative.* </NOTE>

Advocates of static typing argue that the advantages of static typing include earlier detection of programming mistakes (e.g. preventing adding an integer to a boolean), better documentation in the form of type signatures (e.g. incorporating number and types of arguments when resolving names), more opportunities for compiler optimizations (e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically), increased runtime efficiency (e.g. not all values need to carry a dynamic type), and a better design time developer experience (e.g. knowing the type of the receiver, the IDE can present a drop-down menu of all applicable members).

Static typing fanatics try to make us believe that "well-typed programs cannot go wrong". While this certainly sounds impressive, it is a rather vacuous statement. Static type checking is a compile-time abstraction of the runtime behavior of your program, and hence it is necessarily only partially sound and incomplete. This means that programs can still go wrong because of properties that are not tracked by the type-checker, and that there are programs that while they cannot go wrong cannot be type-checked. The impulse for making static typing less partial and more complete causes type systems to become overly complicated and exotic as witnessed by concepts such as "phantom types" [11] and "wobbly types" [10]. This is like trying to run a marathon with a ball and chain tied to your leg and triumphantly shouting that you nearly made it even though you bailed out after the first mile.

Advocates of dynamically typed languages argue that static typing is too rigid, and that the softness of dynamically languages makes them ideally suited for prototyping systems with changing or unknown requirements, or that interact with other systems that change unpredictably (data and application integration). Of course, dynamically typed languages are indispensable for dealing with truly dynamic program behavior such as method interception, dynamic loading, mobile code, runtime reflection, etc.

In the mother of all papers on scripting [16], John Ousterhout argues that statically typed systems programming languages make code less reusable, more verbose, not more safe, and less expressive than dynamically typed scripting languages. This argument is parroted literally by many proponents of dynamically typed scripting languages. We argue that this is a fallacy and falls into the same category as arguing that the essence of declarative programming is eliminating assignment. Or as John Hughes says [8], it is a logical impossibility to make a language more powerful by omitting features. Defending the fact that delaying all type-checking to runtime is a good thing, is playing ostrich tactics with the fact that errors should be caught as early in the development process as possible.

We are interesting in building data-intensive three-tiered enterprise applications [14]. Perhaps surprisingly, dynamism is probably more important for data intensive programming than for any other area where people traditionally position dynamic languages and scripting. Currently, the vast majority of digital data is not fully structured, a common rule of thumb is less then 5 percent [13]. In many cases, the structure of data is only statically known up to some point, for example, a comma separated file, a spreadsheet, an XML document, but lacks a schema that completely describes the instances that a program is working on. Even when the structure of data is statically known, people often generate queries dynamically based on runtime information, and thus the structure of the query results is statically unknown.

Hence it should be clear that there is a big need for languages and databases that can deal with (semi-structured) data in a much more dynamic way then we have today. In contrast to pure scripting languages and statically typed general purpose languages, data intensive applications need to deal seamlessly with several degrees of typedness.

2 When Programmers Say "I Need Dynamic/Static Typing", They Really Mean

Instead of providing programmers with a black or white choice between static or dynamic typing, we should instead strive for softer type systems [4]. That is, static typing where possible, dynamic typing when needed. Unfortunately there is a discontinuity between contemporary statically typed and dynamically typed languages as well as a huge technical and cultural gap between the respective language communities.

The problems surrounding hybrid statically and dynamically typed languages are largely not understood, and both camps often use arguments that cut no ice. We argue that there is no need to polarize the differences, and instead we should focus on leveraging the strengths of each side.

2.1 I want type inference

Requiring explicit type declarations is usually unnecessary and always a nuisance. For instance, no programming language should force programmers to write pleonasms such as:

```
Button b = new Button();
string s = "Doh!";
```

We have known for at least 35 years how to have the compiler infer the (most general) static types of local variables [3]. Not requiring programmers to write types as dynamic languages do is great; but not inferring the types of these variables whenever possible is literally throwing away the baby with the bath water.

Type inference not only allows you to omit type information when declaring a variable; type information can also be used to determine which constructors to call when creating object graphs. In the following example, the compiler infers from the declaration `Button b`, that it should construct a new `Size` object, assign the `Height` and `Width` fields to the integers 20 and 40 respectively, create a new `Button` instance `b` and assign the `Size` object to the `Size` field of `b`:

```
Button b = {
  Size = { Height = 20, Width = 40 }
}
```

This economy of notation essentially relies on the availability of static type information T to seed the type-directed $e <: T \rightsquigarrow e'$ translation of expression e into fully explicit code e'. In other words, static type information actaully allows programs to be more concise then their equivalent dynamically typed counterparts.

2.2 I want contracts

Static typing provides a false sense of safety, since it can only prove the absence of *certain* errors statically [17]. Hence, even if a program does not contain any static type-errors, this does not imply that you will not get any unwanted runtime errors. Current type-checkers basically only track the types of expressions and make sure that you do not assign a variable with a value of an incompatible type, that the arguments to primitive operations are of the right type (but not that the actual operation succeeds), and that the receiver of a method call can be resolved statically (but not that the actual call succeeds).

However in general programmers want to express more advanced contracts about their code [15, 1]. For example, an invariant that the value of one variable is always less than the value of another, a precondition that an argument to a method call is within a certain range, or a postcondition that the result of a method call satisfies some condition.

The compiler should verify as much of a contract P for some expression e as it can statically, say Q, signaling an error only when it can statically prove that the invariants are violated at some point, and optionally deferring the rest of the checking $Q \Rightarrow P$ to runtime via the witness f:

$$\frac{Q(e) \rightsquigarrow e',\ Q \Rightarrow P \rightsquigarrow f}{P(e) \rightsquigarrow f(e')}$$

Notice the correspondence with the type-directed translation rule above. The translation of expression e is driven by the proof that $P(e)$ holds.

2.3 I want (coercive) subtyping

Subtype polymorphism is another technique that facilitates type-safe reuse that has been around for about 35 years [5]. The idea of subtype polymorphism is that anywhere a value of some type, `Dog` say, is expected, you can actually pass a value of any *subtype*, `Poodle` say, of `Dog`. The type `Dog` records the assumptions the programmer makes about values of that type, for instance the fact that all dogs have a `Color` and can `void Bark()`. The fact that `Poodle` is a subtype of `Dog` is a promise that all assumptions about `Dog` are also met by `Poodle`, or as some people sometimes say a `Poodle` *isa* `Dog`.

Subtypes can *extend* their super types by adding new members, for example `IsShaved`, and they can *specialize* their super type by overriding existing members, for example by producing a nasty yepping instead of a normal bark. Sometimes overriding a member completely changes the behavior of that member.

The real power of inheritance, and in particular of overriding, comes from virtual, or late-bound, calls, where the actual member that is invoked is determined by the dynamic type of the receiver object. For example, even though the static type of `d` is dog, it will have a yepping bark since its dynamic type is actually `Poodle`:

```
Dog d = new Poodle();
d.Bark(); // yep, yep
```

In some sense, objects, even without inheritance and virtual methods, are just higher-order functions conveniently packed together in clumps, so all advantages of using higher-order functions [8] immediately carry over to objects.

Coercive subtyping is an extension of subtyping where a subtype relation between types S and T induces a witness that actually

coerces a value of type S into a value of type T [2].

$$\frac{e <: S \rightsquigarrow e',\ S <: T \rightsquigarrow f}{e <: T \rightsquigarrow f(e')}$$

Notice that this is just another instance of rule for contracts above; indeed typing is a very simple form of contracts.

Coercive subtyping is extremely powerful; driven by the type of sub-expressions, the compiler insert coercions to bridge the gap between inferred and required types. Perhaps the most useful example is "auto-boxing" from value types to reference types. In the example below, the compiler infers that the rhs of the assignment has type `int` while the type of the lhs is `object` and hence it inserts a boxing coercion from `int` to `object`:

```
object x = 4711; // new Integer(4711)
```

Type-directed coercions can be applied to arbitrary operations. For example assume that we have a type `int?` that denotes nullable (optional) integers. Based on static type information, the compiler can automatically "lift" addition on normal integers to addition on nullable integers:

```
int? x = null; int? y = 13;
int? z = x+y;
```

The $C\omega$ language takes type-directed lifting to the extreme by radically generalizing member access. In $C\omega$ the `.` is overloaded to lift member access over structural types such as collections, discriminated unions, and tuples. For example, given a collection of buttons `bs` we can get access all their `BackColor` properties using the expression `bs.BackColor`. The compiler translates this into the explicit iteration `{ foreach(b in bs) yield return b.BackColor; }`.

Obviously, type-directed syntactic sugar relies static type information. While in principle it would be possible to do automatic lifting at runtime, this would be quite inefficient. It immediately rules out value types, since these do not carry run-time type information in their unboxed form. On the other hand, we sometimes do want to do truly dynamic member resolution and dispatch. For example consider the following use of reflection in $C^\sharp$ to obtain the `BackColor` member of a button:

```
object b = new Button();
object c = b.GetType()
            .GetField("BuckColor")
            .GetValue(b);
```

Admittedly, this code looks pretty horrible, and even though it seems to be "statically' typed, it really is not. In this particular case `.GetValue("BuckColor")` returns `null` and hence the subsequent `.GetValue(b)` will throw an exception. Wouldn't it be much more convenient if the compiler would use the same type-directed lifting such that we could just us ordinary member access syntax:

```
object b = new Button();
object c = b.BuckColor;
```

Now this is not anymore type unsafe than the previous code, but definitively more concise. Visual Basic.NET provides this mechanism when using `Option Implicit` and a similar extension has been proposed for the Mono $C^\sharp$ compiler.

There is no reason to stop at allowing late bound access to just reflection as in the above example. There are numerous other APIs that use a similar interpretative access/invocation protocol such as ADO.Net, remoting, XPathNavigator, etc. For example, the `SqlDataReader` in ADO.Net class exposes a method `object GetValue(int)` to access the value of a particular column of the current row.

```
...;
SqlDataReader r =
  new SqlCommand(SELECT Name, Age FROM ...", c)
   .ExecuteReader();
while (r.Read()) {
  ...; r.GetValue(0); ...; r.GetValue(1); ...;
}
...;
```

Just like the reflection example, the fact that the datareader API is "statically" typed is a red herring since the API is a statically typed interpretative layer over an basically untyped API. As such, it does not provide any guarantees that well-typed programs cannot go wrong at runtime. So why not let the compiler somehow translate `r.Name` into `r.GetValue(0)` so that you can write a much more natural call:

```
...;
SqlDataReader r =
  new SqlCommand("SELECT Name, Age From ...", c)
    .ExecuteReader();
while (r.Read()) {
  ...; r.Name; ...; r.Age; ...;
}
...;
```

Instead of exposing all kinds of different dynamic interpretative APIs, we should look for ways to expose them in a uniform way via normal member access syntax.

2.4 I want Generics

The principles of abstraction and parametrization apply to types as well as to programs. By allowing types and programs to be parametrized by types, or even type constructors, it becomes possible to create highly reusable libraries, while maintaining the benefits of compile-time type checking [19]. For example, we can define lazy streams of elements of arbitrary type `T` as follows:

```
interface IEnumerator<T> {
  bool MoveNext(); T Current { get; }
}
```

In combination with type-inference, you hardly ever need to write types explicitly. In the example below, the compiler infers that variable `xs` has type `IEnumerator<string>`, and hence that variable `x` has type `string`:

```
var xs = {
  yield return "Hello";
  yield return "World";
};
foreach(x in xs) Console.WriteLine(x);
```

In languages without generics and without subtype polymorphism, you need to write a new sort function for any element type, and any kind of collection. Using generics you can instead write a single function that sorts arbitrary collections of arbitrary element types `void Sort<T>(...)`. There is no need to lose out on type safety. However, in order to sort a collection, we should be able to compare the elements, for example using the `IComparer<T>` interface:

```
interface IComparer<T> { int Compare(T a, T b) }
```

There are several ways to obtain an `IComparer`, first of all we can (a) dynamically downcast an element in the collection to an `IComparer` thereby potentially loosing some static type guarantees, or (b) we can pass an instance of `IComparer` as an additional argument, thereby burden the programmer with threading this additional parameters throughout the program, or (c) we can add a constraint to the type parameter of the `sort` function to constrain it to types `T` that implement `IComparer<T>` only, thereby complicating the type system considerably.

In some sense solutions (a) and (c) are very similar, in both cases the programmer somehow declares the assumption that the type `T` implements `IComparer<T>`. In principle the compiler could infer the constraint for solution (c) from the cast in solution (a). In Haskell, member names uniquely determine the type class in which they are defined, so you do not even need to downcast for the compiler to infer the constraint [6]. Interestingly in Haskell, the instance state and the function members of classes are separated and the compiler passes the implementation of `ICompare<T>` as an additional (hidden) argument to each function that has an `ICompare<T>` constraint, and also automagically inserts instances of the actual implementations.

An alternative solution to passing additional arguments around as in solution (b) is to use dynamically scoped variables or implicit arguments [7, 12], something that many dynamic and scripting languages support. In the example below, `comparer` is dynamically scoped variable that will be used to pass an explicit comparer as an implicit argument.

```
void Sort<T>
  ([implicit]IComparer<T> comparer) {...}

IComparer<T> comparer =
  new CaseInsensitiveComparer();
...
myCollection.Sort();
...
```

Dynamically scoped variables do not require us to completely abandon static typing. In fact, we think that dynamic scoping becomes more understandable if the fact that a function relies on a dynamically scoped variable is apparent in it's type. In this case, the static type of `Sort` includes the fact that it expects a variable `comparer` to be in scope when `Sort` is being called, or else the implicit parameter is propagated and that code itself now recursively relies on a variable `comparer` to be in scope when called. Note that this is similar to the `throws` clause in Java, and in fact, dynamic variables can be implemented in a very similar manner as exception handling.

2.5 I want (unsafe) covariance

Array covariance allows you to pass a value of type `Button[]` say where a value of type `Control[]` is expected. Array covariance gracefully blends the worlds of parametric and subtype polymorphism and this is what makes statically typed arrays useful. There is no such thing as a free lunch, and array covariance comes with a price since each write operation into an array (potentially) requires a runtime check otherwise it would be possible to create a `Button[]` array that actually contains a string:

```
object[] xs = new Button[]{ new Button() };
xs[0] = "Hello World";
```

Without covariance, we would be forced use parametric polymorphism everywhere we want to pass a parametric type, like array, covariantly. This causes type parameters to spread like a highly contagious disease. Another option is to make the static type system more complicated by introducing variance annotations [9] when declaring a generic type or method as in CLR generics (`FooBar<+T> where T: Baz`), when using a generic type as in Java's wild-card types (`FooBar<? extends Baz>`)[21].

Despite the added complexity, variance annotations and wild-card types do not provide the full flexibility of "unsafe" covariance. Before generics was introduced in Java or $C^\sharp$, it was not even possible to define generic collections, and hence there was no static type-checking whatsoever. It is interesting to note that suddenly the balance has swung to the other extreme where people now suddenly lean over backward to make their code statically typed.

Instead of allowing covariance by making the type-system more complicated, we should allow covariance just like arrays by adding a few runtime checks. Note that this is impossible in an implementation of generics that uses erasure like in Java since the runtime checks require the underlying element type.

2.6 I want ad-hoc relationships and prototype inheritance

Statically typed (object-oriented) languages such as $C^\sharp$ and Java force programmers to make premature commitments about inter-entity relationships [18]. For example, at the moment that you define a class `Person` you have to have the divine insight to define all possible relationships that a person can have with any other possible object or keep type open:

```
class Person { ...; Dog myDog; }
```

The reason is that objects embed relationships as pointers (links) within their instance, much like HTML pages embed hyperlinks as nested `<A href="...">` elements. In the relational model, much like in traditional hypertext systems, it is

possible to create relationships after the fact. The bad way (BadDog below) is to embed the relationship to the parent entity when defining the child entity; the good way (Dog) is to introduce an explicit external "link table" MyDogs that relates persons and dogs:

```
table Person{ ...; int PID; }
table BadDog { ...; int PID; int DID; }
table Dog { ...; int DID; }
table MyDogs { ...; int PID; int DID; }
```

The difficulty with the relational approach is that navigating relationships requires a join on PID and DID. Assuming that we have the power in the "." we can simply use normal member access and the compiler will automatically insert the witnessing join between p and the MyDogs table:

```
Person p;
Collection<Dog> ds = p.MyDogs;
```

The link table approach is nice in the sense that it allows the participating types to be sealed, while still allowing the illusion of adding properties to the parent types after the fact. In certain circumstances this is still too static, and we would like to actually add or overwrite members on a per instance basis [22].

```
var p = new Object();
p.Name = "John Doe";
p.Age  = (){
  DateTime.Today - new DateTime(19963,4,18);
};
```

This protype-style programming is not any more unsafe than using a HashTable<string, object>, which as we have concluded before is not any more unsafe than programming against statically typed objects. It is important that new members show up as regular methods when reflection over the object instance, which requires deep execution engine support.

```
object c = p.GetType().GetField("Name")
           .GetValue(p);
```

2.7 I want lazy evaluation

A common misconception is that loose typing yields a strong glue for composing components into applications. The prototypical argument is that since all Unix shell programs consume and produce streams of *bytes*, any two such programs can be connected together by attaching the output of one program to the input of the other to produce a meaningful result. Quite the contrary, the power of the Unix shell lies in the fact that programs consume and produce *lazy streams* of bytes. Examples like `ls | more` work because the `more` command lazily sucks data produced by the `ls` command.

The fact that pipes use bytes as the least common denominator actually diminishes the power of the mechanism since it is practically infeasible to introduce any additional structure, into flat streams of bytes without support for serializing and deserializing the more structured data that is manipulated by the programs internally. So what you really want to glue together applications is lazy streams of structured objects; this is the abstraction provided by lazy lists, Unix pipes, asynchronous messaging systems, etc.

Using XML instead of byte streams as a wire-format is one step forward, but three steps backwards. While XML allows dealing with semi-structured data, which as we argue is what we should strive for, this comes at an enormous expense. XML is a prime example of retarded innovation; it makes the life of the low-level plumbing infrastructure easier by putting the burden on the actual users by letting them parse the data themselves by having them write abstract syntax tree, introducing an alien data model (Infoset) and an overly complicated and verbose type system (XSD) neither of which blends in very well with the paradigm that programmers use to write their actual code.

The strong similarity between the type-system of the CLR and the JVM execution environments makes it possible to define a common schema language, much in the style of Corba or COM IDL, or ASN/1, that maps easily to both environments, together with some standard (binary) encoding of transporting values over the wire. This would be a superior solution to the problem that XML attempts to solve.

2.8 I want higher-order functions, serialization, and code literals

Many people believe that the ability to dynamically `eval` strings as programs is what sets dynamic languages apart from static languages. This is simply not true; any language that can dynamically load code in some form or another, either via DLLs or shared libraries or dynamic class loading, has the ability to do `eval`. The real question is whether your really need runtime code generation, and if so, what is the best way to achieve this.

In many cases we think that people use `eval` as a poor man's substitute for higher-order functions. Instead of passing around a function and call it, they pass around a string and `eval` it. Often this is unnecessary, but it is always dangerous especially if parts of the string come from an untrusted source. This is the classical script-code injection threat.

Another common use of `eval` is to deserialize strings back into (primitive) values, for example `eval("1234")`. This is legitimate and if `eval` would only parse and evaluate values, this would also be quite safe. This requires that (all) values are expressible within in the syntax of the language.

A final use of `eval` that we want to mention is for partial evaluation, multi-stage programming, or meta programming. We argue that in that case strings are not really the most optimal structure to represent programs and it is much better to use programs to represent programs, i.e. C++-style templates, quasiquote/unquote as in Lisp, or code literals as in the various multi-stage programming languages [20].

3 Conclusion

Static typing is a powerful tool to help programmers express their assumptions about the problem they are trying to solve and allows them to write more concise and correct code. Dealing with uncertain assumptions, dynamism and (unexepected) change is becoming increasingly important in a loosely couple

distributed world. Instead of hammering on the differences between dynamically and statically typed languages, we should instead strive for a peaceful integration of static and dynamic aspect in the same language. Static typing where possible, dynamic typing when needed!

4 Acknowledgments

We would like to thank Avner Aharoni, David Schach, William Adams, Soumitra Sengupta, Alessandro Catorcini, Jim Hugunin, Paul Vick, and Anders Hejlsberg, for interesting discussions on dynamic languages, scripting, and static typing.

References

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 3(6), 2004.

[2] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as Implicit Coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[3] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proceedings PLDI'91*. ACM Press, 1991.

[5] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some Features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations*. IEEE Press, 1968.

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[7] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *proceedings PLDI'01*, 2001.

[8] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[9] A. Igarashi and M. Viroli. On Variance-Based Subtyping for Parametric Types. In *Proceedings ECOOP'02*. Springer-Verlag, 2002.

[10] S. P. Jones, G. Washburn, and S. Weirich. Wobbly Types: Type Inference for Generalised Algebraic Data Types.

[11] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, 1999.

[12] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings POPL'00*, 2000.

[13] P. Lyman and H. R. Varian. How Much Information 2003.

[14] E. Meijer, W. Schulte, and G. Bierman. Programming with Circles, Triangles and Rectangles. In *Proceedings of XML 2003*, 2003.

[15] B. Meyer. *Object-Oriented Software Construction (2nd edition)*. Prentice-Hall, Inc., 1997.

[16] J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings OOSPLA'87*, 1987.

[19] D. A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994.

[20] T. Sheard. Accomplishments and Research Challenges in Meta-Programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*. Springer-Verlag, 2001.

[21] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ah, G. Bracha, and N. Gafter. Adding Wildcards to the Java Programming Language. In *Proceedings of the 2004 ACM symposium on Applied computing*. ACM Press, 2004.

[22] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA'87*, 1987.