Runtime type collecting and transpilation to a static language

Pavel Krivanek[1], Richard Uttner[2,†]

[1] Nidea s.r.o., Smejkalova 1033/136, 616 00, Brno Zabovresky, Czechia
[2] Projector Software GmbH, Mathildenstr. 34, 72072 Tübingen, Germany

Abstract
This study delves into incorporating static typing into Pharo, a dynamic language derived from Smalltalk. Utilizing Pharo's pragmas, it introduces type annotations at both method and class levels, drawing from Strongtalk's approach but without altering Pharo's grammar. A novel experiment detailed here involves annotating Pharo code using these annotations and runtime type collection, then transpiling it to C#, highlighting the syntactic and conceptual adaptations required. Despite the complexities, the experiment successfully translates Pharo code into compilable C# code, underscoring Pharo's potential for optional static typing. The exploration also suggests future directions, including improved type inference, while reassessing static typing's role in error detection.

Keywords
Pharo, static typing, type annotations, runtime type collection, transpilation, IWST[*]

1. Introduction

Pharo, as a descendant of the Smalltalk programming language, benefits from the flexibility and expressiveness provided by strong dynamic typing. However, static type checking offers advantages such as error detection, facilitation of code refactoring and documentation. Additionally, a JIT compiler can utilize it or static typing can be used for transpiling code into other languages.

In the context of integrating static typing into dynamically typed languages, Strongtalk [1] is a notable example, enhancing Smalltalk-80 with optional static type annotations to improve performance and robustness. However, Strongtalk required significant grammar changes. Figure 1 illustrates an example of Strongtalk code. Pharo, unlike Smalltalk-80, has pragmas similar to those introduced by Strongtalk, allowing for type annotations without modifying its grammar.

We performed a limited experiment in which we introduced type annotations in the form of pragmas [2] into Pharo, used runtime type collecting to annotate existing Pharo code, and then transpiled it into a static programming language.

2. Type Annotations

Type annotations are categorized into two types: method-level and class-level.

2.1. Method-level Type Annotations

Method-level type annotations describe the types of:

- Method return value
- Arguments
- Temporary variables
- Block arguments

† Authors contributed equally.
[*] *IWST 2024: International Workshop on Smalltalk Technologies, July 9--11, 2024, Lille, France*

- Block temporary variables

```
occurrencesOf: o <Object> ^<Int>

        | c <Int> |
        c := 0.
        self do: [ :e <E> | e = o ifTrue:[ c := c + 1 ]].
        ^c
```

Figure 1: An example code in Strongtalk with type annotations for a method argument, a method return value, a temporary variable and a block argument.

```
occurrencesOf: o

        <arg: #o type: Object>
        <returns: #Integer>
        <var: #c type: #Integer>
        <blockArg: #e type: #Object generated: true>

        | c |
        c := 0.
        self do: [ :e | e = o ifTrue:[ c := c + 1 ]].
        ^c
```

Figure 2: An example Pharo code with type annotations for a method argument, a method return value, a temporary variable and a block argument.

Besides the type information, the method-level annotations include:

- Information on whether the return value of a given block is supposed to be used

Figure 2 shows the alternative to the Strongtalk code written using type annotating pragmas.

Method-level type annotations for block arguments and block temporary variables have one problem - the names of these entities may be the same in multiple blocks. A typical example is a method with several blocks where each of them uses the block argument named each. We used two strategies to solve this problem. The first requires each block argument and temporary variable in the scope of a method

to have a distinct name, which means that before adding type annotations to the code, every method that does not pass this rule needs to be refactored. The second strategy involves adding a block identification into the annotating pragma. We used a prefix that denotes the block order in the method:

```
<blockArg: #_1_each type: #Integer>
<blockArg: #_2_each type: #Point>
```

The latter approach avoids refactoring but complicates readability and further code modifications. Therefore, we preferred using unique names within the method's lexical scope.

2.2. Class-level Type Annotations

Class-level type annotations are used for type descriptions of instance variables (more generally, slots) and other class elements, and we placed them into special methods named _slotTypes. Like the method-level type annotations, they include pragmas.

```
_slotTypes
  <slot: #commandContext type: #CommandContext>
  <slot: #labelString type: #String>
```

Our type annotations allow to describe a single simple type (class) associated with a variable, multiple distinct types, or more complex types like Collections, Dictionaries, Associations, or Blocks.

Notice that the part of the pragma that describes the type is composed of a symbol or a literal array of symbols. If the variable can hold only one class, a single symbol with the class name is used. If it can hold more distinct classes, a literal array describes it.

```
<var: #temp type: #Symbol>
<var: #temp type: #(Symbol Number)>
<var: #temp type: #(Symbol UndefinedObject)>
```

Given the commonality of `UndefinedObject` as an alternative type, we offer a shortcut for it expressed as two colons. Here we would rather have preferred a question mark, but were limited by Pharo's symbol parsing.

```
<var: #temp type: #Symbol::>
```

Complex types use literal arrays comprising the type class name, additional symbols such as `of`, `key` or `value` describing the purpose of the subsequent type description, and embedded type descriptions.

```
#(Array of Symbol)
#(Dictionary of Symbol keys Object)
#(Association key Symbol value Number)
#(Array of (Number String))
#(Array of (Array of String))
```

Type descriptions for block closures are the most complex. They describe the types of all arguments and the return type of the block. Blocks need not have arguments or return values, so these parts can be omitted.

```
#FullBlockClosure
#(FullBlockClosure returning Integer)
#(FullBlockClosure:: returning Integer)
#(FullBlockClosure arguments #(String Object) returning Integer)
```

These block type annotations are useful when describing argument types or variables holding the block. However, Pharo code containing block literals does not indicate if the return value (the result of the last expression) is to be used, as in select:, or ignored, as in do:. Thus, the method with such a block can contain a pragma specifying this information, which is important in some cases like code translation.

```
<block: 1 returnsValue: false>
```

An example of such a method is described in the following chapter.

We developed a simple system to regenerate type annotations while allowing programmers to manually modify them without losing these changes. Each automatically generated type annotation pragma includes an additional argument named `generated:` with `true` as the default value.

```
<arg: #anObject type: Object generated: true>
```

When generating type annotations, the system creates only new pragmas or those with `generated: true`.

Other popular dynamically typed languages often implement type annotations by extending the language syntax and standardizing existing language features such as function annotations (Python 3 [3]), using existing grammar constructs like special forms (Common Lisp [4]), or creating a new language derived from the original one (TypeScript [5] derived from ECMAScript).

3. Runtime Type Collecting

Annotating existing Pharo code with types is a laborious task. Previous attempts, such as RoelTyper [6], resolved a relatively low number of types. Thus, we explored an alternative approach leveraging advances in Pharo's reflectivity infrastructure.

We collect runtime type information automatically during program execution by inserting watchpoints and generate type annotations in post-processing. This method requires most of the existing code to be executed in a manner providing relevant type information. Fortunately, in Pharo, the test-driven style of development is very popular, so projects with a high test coverage ratio are not unlikely. Ideally, runtime type collecting requires 100% code coverage with relevant tests.

Our technical solution involves installing various Metalinks [7] into each method to resolve types. Metalinks in Pharo allow developers to attach additional behavior and metadata to methods dynamically at runtime, enabling fine-grained control and non-intrusive modification of method execution. For instance, to detect method argument types, a Metalink is installed at the method's beginning. For temporary variable types, a Metalink is installed at all assignments to the variable within the method. Method return types are detected by wrapping the entire method execution with a Metalink.

Each Metalink has an associated object describing the collected types. Each invocation of the Metalink updates this collection with the current types of values written into variables. After executing all relevant code, the collected type information is post-processed and written as type pragmas described above.

This straightforward process is hindered by technical limitations in Pharo's Metalinks implementation, such as incorrect handling of some Metalink combinations. For example, Metalinks for method return values need to be installed as standalone and require additional code execution.

Block closure literals require special treatment. Resolving their arguments is not significantly different from resolving the method argument types; however, handling return values poses a significant challenge. As mentioned above, in Pharo, the return value of a block evaluation is the result of the last expression. The block itself does not provide any information about whether this resulting value is actually meaningful and will be used. If the block is not constructed with the intent to use its return value, the type provided by runtime collecting can produce an arbitrary value, which does not help in describing the block type information because the actual expected return type is void.

Consider the following code:

```
self critical: [
  logFileStream ifNotNil: [
    logFileStream close.
    logFileStream := nil
  ].
]
```

For the outermost block closure, the argument of the method `critical:`, we cannot be sure at the first glance how the method `critical:` actually handles the block. We may assume it evaluates the block, but we do not know if it only evaluates it and throws the result away or processes it further. This information is required when we, for example, try to translate this code into a different language that must explicitly specify a return statement for anonymous functions replacing the block. Always returning the result type of the last statement is not possible because the value may sometimes vary significantly. Moreover, in Pharo, the usage of a block return value is sometimes depending on the actual context.

We addressed this issue by returning a special proxy object instead of the actual block result value. When we detect during type collecting that the variable or argument type is a block, we construct a custom block closure of a custom class inheriting from the standard `FullBlockClosure`

class, set it up based on the original closure, and swap their identities using standard `become:`. Thus, we ensure that we can handle standard evaluation messages of this block like `value`, `value:` etc. These custom methods, when called, perform the standard block evaluation and wrap the returned result in a custom proxy object. When a message is sent to the proxy or if it is assigned to a variable – which needs to be resolved by some installed Metalink – the block is marked as a block returning a value of the given result type. The proxy is then replaced by the actual result object (using `become:`, again), so this mechanism is performed at most once.

The type collecting process can be executed several times on a given code: As the type collector always starts reading all available type annotations, it will merge the new collected annotations with the existing ones if they are marked for regeneration.

4. Translation into a Static Language

Our experiment aimed to explore the possibility of transpiling parts of an existing business application from Pharo to C# for architectural reasons, leveraging existing C# libraries and code while benefiting from Pharo's unique features. C# has several properties that facilitate the transpiling of Pharo code, notably being a class-based object-oriented language with automatic memory management based on garbage collection. Additionally, C#'s optional named arguments make the translation of Pharo keywords easier than it would normally be for C-syntax based languages, thus keeping the output code closer to its original source. Modern versions of C# support anonymous functions and some other language properties similar to Pharo.

However, C# has significant lexical, syntactical, and semantic differences from Pharo, particularly in expressions and control structures. Its class metamodel also differs notably.
The C# grammar, being quite complicated, suggests that the direct transpiling of grammatically simpler Pharo code to C# should be straightforward, at least from a syntactic perspective. Unfortunately, this is only partially true. While in Pharo, practically everything is a message call or an assignment, constructs that are easy to transpile, some C# language features introduce difficulties, such as:

- The absence of non-local returns
- A notable difference between statements and expressions
- Different object construction mechanisms
- The absence of metaclasses and cascade
- Missing polymorphism of constructors and static methods
- Only stateless interfaces
- The presence of primitive non-object types
- Limited extension methods
- A wide set of reserved words

Each of these issues presents its own challenges and potential solutions. In some cases, we decided to limit constructs allowed in the transpiled Pharo code, so we must admit that we are able to transpile only a subset of the Pharo language.

We aimed to generate readable, maintainable C# code with a direct relation to the original Pharo code. On the other hand, we wanted the Pharo code to be the main source of information. The C# code was repeatedly generated from it while the Pharo code was still being modified and improved.

The transpilation process itself is not innovative. We start with Pharo code AST. Using a transpiling visitor, we generate a new abstract syntax tree for a C# subset (because we do not need all C# features) and finally, using another visitor, we generate C# code by visiting these nodes. The generated C# code uses a small supporting library providing equivalents to Pharo standard library along with other utility functions.

4.1. Messages

Pharo unary messages are straightforward to translate into C#. They simply follow standard dot notation and skip the this keyword if possible. All message names are modified to use the regular C# customary to start with uppercase letter. So

```
self next.
```

becomes

```
Next;
```

Binary message sends are interpreted as operators for common messages like `+`, `<` and so on. Some binary messages need to be reinterpreted, such as `=` as `Equals()`.

Keyword messages are generally hard to translate into other languages if readability shall be preserved. Fortunately, with C# optional named arguments, this task is manageable with only small limitations.

As an example, let us choose a keyword message `chooseFrom:title:` with two keyword parts. The method header with argument names looks like:

```
chooseFrom: aList title: aString
```

When transpiling to C#, we can keep the first argument name (`aList`) unchanged. We rename every other argument to match the corresponding keyword part name, `aString` to `title` in our case. The original argument name is mentioned as a comment and in the beginning of the method. Then, we define the new variable with the same name as it was in Pharo and assign it from the the argument name we created. Thus the original argument name can be used inside of the method without further change, as shown in the following example:

```
public long ChooseFrom(object aList, string title /* aString */ )
{
    var aString = title;
    …
}
```

This approach proves its utility when we perform a method call of this message with some arguments, as it closely resembles the message calls as done in the original Pharo code:

```
ChooseFrom(someList, title: actualTitle);
```

However, this approach has some small limitations. Keyword part names need to be distinct so it may require renaming of some methods (like `with:with:`) before transpiling. The reason behind is that C# uses named arguments primarily to support different order of arguments. We created a simple non-GUI tool in form of specialized class-side methods to detect such cases in advance.

4.2. Non-local returns

In Pharo, when a return statement is used inside a block closure, it causes program flow to exit from the whole method, not only from a given closure. In C#, anonymous methods, also called lambda expressions, are the closest corresponding construct to Pharo blocks. The main difference in behaviour is that a return statement in C# only exits the execution of the current anonymous function, not the method where it is defined.

The difference in usage of non-local returns in Pharo and in C# stems from the fact that Pharo uses combination of message sends and block closures for constructs that are expressed using special grammar control structures in C#, like if-statements.

Well-known Pharo methods such as `ifTrue:`, `ifFalse:`, `ifNil:`, `ifEmpty:`, `whileTrue:`, and `do:` are translated directly into corresponding C# statements which use regular code blocks `{…}` instead of lambdas. So the statement

```
aBoolean ifTrue: [
    self doSomeAction.
    ^ 0 ]
```

is translated into

```
if (aBoolean)
{
    DoSomeAction;
    return 0;
}
```

Non-local returns that cannot be translated into statements are currently forbidden by our transpiler. We have a small non-GUI tool to detect such cases in the code. The other alternative, which generates slower code but is more general and does not require so many code modifications, is to use exceptions.

4.3. Expressions

Pharo does not have any limits on structure of expressions. So instead of `aBoolean` in the example above, another complex expression containing statement-like message can be used. This is not true for C#. Some simple C# statements like `if-else` statement or `null` checks have an alternative expression syntax. For example, the ternary expressions (`?:`) or the null-coalescing operators (`??`). The transpiling visitor marks the AST subtrees that need to be expressions and tries to use these alternatives. If this is not possible, it reports an error. In that case, the Pharo code needs to be rewritten.

4.4. Object construction mechanisms

Pharo usually instantiates a class by sending a message to it. Such message then directly or indirectly invokes a VM primitive that creates an object. C# does not have the concept of metaclasses. For objects construction, the operator new is used. During building of the object, its constructors are called.

When the C# object is created, it sometimes requires generic type information, for example:

```
new Dictionary<string, int>();
```

Pharo does not provide such information; the equivalent expression is simply `Dictionary new`. Our transpiler needs this information so it expects that the created object is assigned to some variable, which is usually the correct assumption. If the `new` message is on the right side of an assignment, the transpiler can derive generic types from the type annotations belonging to the variable the new object is assigned to.

If this mechanism cannot be used, an error is reported. In such case, the solution is to create an additional temporary variable with type annotation and assign the new object during creation to it.

The base C# class that is used as the root class for classes generated by the transpiler always calls the `Initialize` method from its constructor to automatically mimic default Pharo behavior.

4.5. Cascade

Cascade is a heavily used Pharo construct that does not have a corresponding equivalent in C#. To translate a cascade, we need to first create a temporary variable with a unique name and assign the base cascade object to it. This variable needs to be typed in C#, but fortunately, automatic type inference is sufficient in most cases. Because this assignment is a statement in C# but in Pharo, the cascade is a general expression, there are additional complications in deciding where to and how to evaluate the generated statement. The resultant code then looks, for example, like this:

```
var cascade = new Dictionary<string><string>();
cascade.At("uid", put: uid);
cascade.At("label", put: label);
return cascade;
```

Moreover, cascades may be embedded. We pay special attention to their evaluation order.

4.6. Metaclasses and polymorphism

The metaclasses concept in Pharo is very powerful and heavily used, but does not have a direct mapping to C#. In Pharo, class-side methods often play a role of object constructing methods, and they are polymorphic. We translate them into static methods in C# calling specially generated constructors. We do not allow overrides because C# does not support static methods polymorphism. Moreover, C# does not support polymorphism of constructors.

The only more regular solution that could address this issue and allow to transpile more Pharo code without need to adopt the code-base in advance, is to create a real object playing the role of the class. This solution would break the direct relation between original transpiled code and, as a consequence, lead to new issues. However, we plan to explore this approach in future development, as we expect it to be the best way to avoid large refactorings.

4.7. Other complications

While traits in Pharo and interfaces in C# are comparable, traits in Pharo are stateful [8] such that they can contain slots. As such slots cannot be translated into C#, we do not support stateful traits in our transpiler.

A surprising complication stems from the way how C# handles the nullability of primitive value types like `int`. For such types, the nullability means that they are wrapped in an additional structure that keeps information whether the value is set or not. If not, the actual value is set to the default value, not `null`. This makes general implementation of some trivial Pharo collection messages generally impossible (messages of type `Dictionary >> atOrNil:`).

We tried to mimic many Pharo standard library methods using extension methods in C#. When it was not possible because of C# limitations, like in the case of some Object extensions, we used static methods in ECMAScript style (`PharoObject.IsInteger(anObject)`).

The difference between Pharo and C# in zero array indexing proved to be not a real problem because it was handled by extension methods we created to follow the Pharo collections API.

Unlike Pharo, C# has a wide set of reserved keywords that can cause name collisions with the existing Pharo code. C# has a way how to handle special names (using the `@` sign). However, as we encountered some problems with them, we rather changed the Pharo code in advance manually or using refactorings to avoid names that are C# keywords, which was trivial.

For Associations, we first tried to use C# pairs, but later switched to an own C# class which was easier to handle in all use cases we needed.

In relatively rare cases when the generated C# code required type casting, we introduced the Pharo message `castAs: #TypeName` which does nothing in Pharo but adds a casting operator to the C# code. In cases of several messages, we created own application specific hooks performing certain casting operations, mainly for casting to return types of called methods. Thus we could get around some complications, and even more important, avoid that C# performs unnecessary single element casts for big collections, resulting in significant runtime overhead.

5. Results and Further Development

In our limited experiment, we were able to convert several packages of a real-life business application into about 20,000 lines of compilable and working C# code. While the Pharo code required some modifications to make it translatable to the target language, the amount of required changes was relatively low. Because the automatically generated code was compilable by the C#

compiler, it means that the same task could theoretically be done without any translation to C#. This proves that Pharo can easily play the role of a language with optional static typing.

Our experiences during this experiment suggested several directions for further improvements. Combining runtime type collecting with type inference in the style of RoelTyper [6] would be a significant advantage for transpiling code with low code coverage. It showed us that, at least in the case of C#, it would make sense not to maintain such a tight correspondence between the input and output code and to handle constructs like metaclasses and non-local returns in more a verbose and less readable but more general way. We would like to try translation into other languages, such as TypeScript, as well.

One interesting outcome of this experiment was our curiosity about whether static annotation would reveal any serious type errors in the existing code. At least in this case, the result was negative. We have encountered, of course, many reported type errors during compilation, but all of them were related to the C# narrow interpretation of types, even if the Pharo code had valid semantics.

Even though we did not detect bugs in the Pharo code, runtime errors were still common even for the code that was successfully compiled. Most of them, however, were related to some limitations of the translation process or to bugs therein. This suggests that while static typing brings some advantages, its role in error detection for well-designed dynamically typed code should not be overestimated.

The transpiler is publicly available at https://github.com/pavel-krivanek/Pharo-CSharp under MIT lincense.

References

[1] G. Bracha, D. Griswold, Strongtalk: Typechecking Smalltalk in a production environment, in: Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93), ACM Press, New York, NY, USA, 1993, pp. 215–230. doi:10.1145/165854.165893.

[2] S. Ducasse, E. Miranda, A. Plantec, Pragmas: Literal Messages as Powerful Method Annotations, in: Proceedings of the 11th International Workshop on Smalltalk Technologies (IWST 2016), ACM Press, New York, NY, USA, 2016. doi:10.1145/2991041.2991050.

[3] L. Di Grazia, M. Pradel, The evolution of type annotations in Python: An empirical study, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022), ACM Press, New York, NY, USA, 2022, pp. 209–220. doi:10.1145/3540250.3549114

[4] G. L. Steele Jr., Common Lisp: The Language (2nd ed.), Digital Press, 1990.

[5] B. Cherny, Programming TypeScript: Making Your JavaScript Applications Scale, O'Reilly Media, 2019.

[6] F. Pluquet, A. Marot, R. Wuyts, Fast type reconstruction for dynamically typed programming languages, in: Proceedings of the 5th Symposium on Dynamic Languages (DLS '09), ACM Press, New York, NY, USA, 2009, pp. 69–78. doi:10.1145/1640134.1640145

[7] S. Costiou, V. Aranega, M. Denker, Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use, The Art, Science, and Engineering of Programming 4 (2020). doi:10.1145/3567512.3567517

[8] P. Tesone, S. Ducasse, G. Polito, L. Fabresse, N. Bouraqadi, A new modular implementation for stateful traits, Science of Computer Programming 195 (2020). doi:https://doi.org/10.1145/3167132.3167244

[9] N. Bouraqadi, D. Mason, PharoJS: Transpiling Pharo classes to JS—ECMAScript 5 versus ECMAScript 6, presented at the International Workshop on Smalltalk Technologies (IWST 2023), 2023.