RUNTIME TYPE COLLECTION AND ITS USAGE IN CODE TRANSPILING

Pavel Krivanek, Richard Uttner

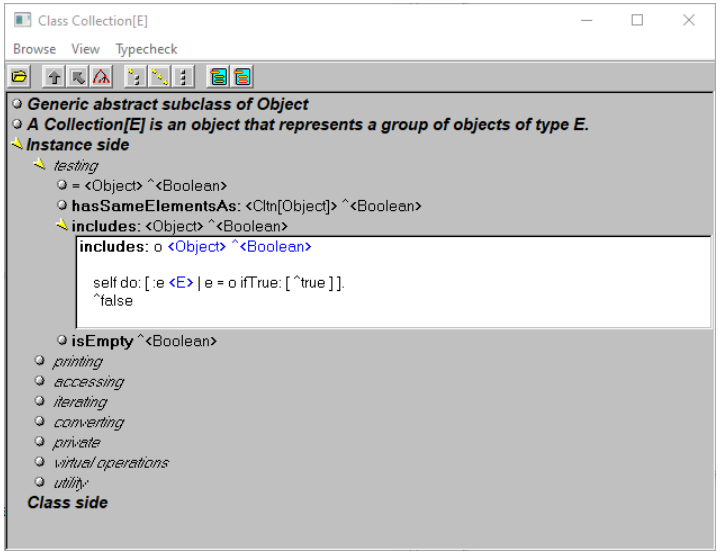
static typing

> error detection
> facilitation of code refactoring
> documentation
> information source for JIT
> code transpiling *)

*) our motivation

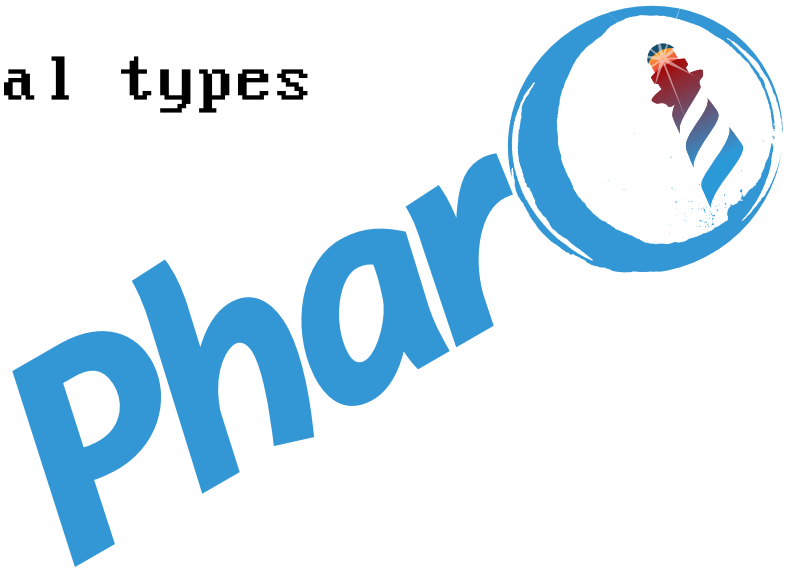
Strongtalk

> optional static typing
> grammar extensions
> fastest Smalltalk of its time
> BSD license



Pharo

> no standard syntax for optional types
> pragmas (Bloc...)
> laborious to annotate



type annotations

> collect during runtime
> inference of the rest *)
> generate annotations
> repeat, keep them up-to-date

*) RoelTyper (https://github.com/RMODINRIA/RoelTyper)

our pragmas

```
occurrencesOf: o

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

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

Strongtalk

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

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

block arguments

> require distinct names

> name prefixes (block number)

```
self addEdge: { parent model. child }
from: [ :each | each first ]
to: [ :each | each second ] ]
```

<blockArg: #_1_each type: #Integer>

<blockArg: #_2_each type: #Point>

class-level annotations

> similar to method-level annotations

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

simple types

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

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

*) unfortunatelly, #Symbol? Is not available

complex types

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

```
<slot: #languagePrioritiesByType type: #(Dictionary of (Dictionary of (Array of String) keys String) keys String)>
```

block types

> assigned to variables:

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

block types

> for literals, is the block "void" or actually used?
> resolving type of an existing object (block)
> select: vs. do:
> required for translation to lambdas

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

argument type specified by blockArg: pragma

```
^ array select: [ :each | each value isNotNil ]
```

```
self selectedItems do: [:item | self deselectItem: item]
```

runtime type collecting

> install **Metalinks** on
  * all types of variable assignments
  * method beginning, return

```
orderedBlocks

    | allBlocks |

    allBlocks := self allBlocks.

    ^ (blockInformation associations collect: [ :assoc |
        (self isWellKnownBlock: assoc key) ifFalse: [
            (allBlocks indexOf: assoc key) -> assoc value ].
    ]) sorted: [ :a :b | a key <= b key ]
```

runtime type collecting

> detect and store object class on each invocation
> slow for complex types (Dictionary...)
> post-processing
> write annotations

```
<returns: #RuntimeTypeCollectorExample generated: true>
<blockArg: #index type: #SmallInteger generated: true>
<blockArg: #num type: #SmallInteger generated: true>
<blockArg: #x type: #SmallInteger generated: true>
<blockArg: #y type: #SmallInteger generated: true>
```

type annotations

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

> fixed by programmer:
<arg: #anObject type: Object>
<arg: #anObject type: Object generated: false>
```

block return value usage detection

> custom block closure subclass

> swap blocks with custom #value, value:… methods returning proxy

> detect assignment of this proxy or calling a message to it

> become:

usage for transpilation

> experiment with translation to C#

* class based object-oriented
* GC
* keyword arguments
* closures (lambdas since 2005)

> notable differences
  * statically typed
    (with type inference)
  * more complex grammar
  * different standard library

unary and binary messages

```
self next.                    Next;
3+4*5                         (3+4)*5
=                             Equals()
```

keyword messages

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

method header in Pharo

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

in C#

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

keyword used as the argument name

Implementation in C#

non-local returns

> ifTrue:, ifFalse:, ifNil:, ifEmpty:, whileTrue:, do:
  * use C# statements

> others are forbidden (Pharo code refactoring needed)
> tools to detect
> exceptions in the future

expressions

> Pharo: uses statement-like expressions without limits
> C#:
  * only expressions like ?:, ??
  * cannot include statements

> during transpilation, mark AST subtree
> others require rewrite of Pharo code

new

```
Dictionary new.

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

> try to detect type from assignment, if present
> explicitly add an extra assignment into
  a variable with defined type

cascade

> no alternative in C#

> crate temporary variables

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

> can be embedded and used inside other expressions (order!)

metaclasses

> use static methods
> no static methods polymorphism in C#
> detect instance creation, translate to constructors, forbid rest
> no polymorphism of constructors in C#

casting

> castAs: #typeName

 * does nothing in Pharo

> application specific hooks like automatic casting to method return type

> C# and collections casting

```
static void ProcessList(List<object> list) { }

List<MyObject> myList = new List<MyObject>();
ProcessList(myList); X
```

etc., etc

> traits as interfaces (but no stateful traits)

> method categories as #region

> comments preserving

> limited extension methods

> conflicts with C# keywords

> presence of primitive object types (makes some general collection methods impossible to implement)

experiment

> 20,000 lines of compilable and working C# code
> readable, non-idiomatic C# code
> Pharo code evolving in the meantime
  * regeneration
> Pharo subset
 * small tools to detect issues
> Pharo as language with optional static typing

value of static typing

> no significant type error in existing code

> but runtime errors after compilation still common

beyond current the experiment

> more complete standard library support
> RoelTyper
> improve non-local returns using exceptions
> metaclasses
> other languages (TypeScript, Java, C++...)
...

thank you for your attention!

https://github.com/pavel-krivanek/Pharo-CSharp

https://github.com/pavel-krivanek/Runtime-Type-Collector