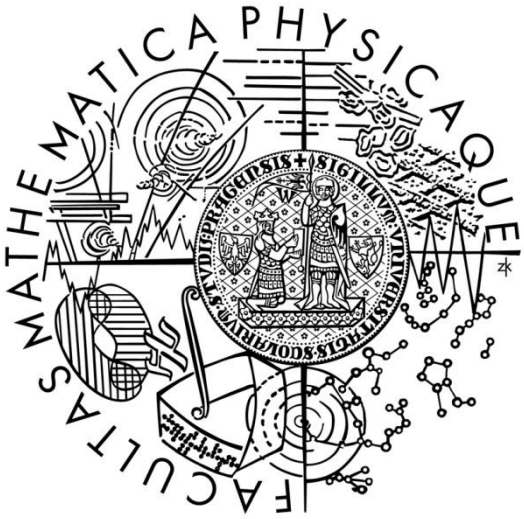
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Miloslav Beňo
Implementing the Dynamic Languages using DLR Technology
Department of Software Engineering
Supervisor: *RNDr. Filip Zavoral, Ph.D.*
Study Program: Computer Science, Software Systems

I would like to thank Filip Zavoral for his professional guidance and patience. Also I have to thank Tomas Matousek from Microsoft for his invaluable advices and comments related to DLR. An acknowledgement belongs also to my colleagues and co-workers Jakub Misek and Tomas Petricek for consultations and comments.
Above all, I would like to thank my parents for their support throughout my studies.

I declare that I have written this thesis by myself and that I have used only the cited resources. I agree with lending of this master thesis.

Miloslav Beňo

In Prague

Contents

Title: *Implementing the Dynamic Languages using DLR Technology*
Author: *Miloslav Beňo*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Filip Zavoral, Ph.D.*
Supervisor's e-mail address: *zavoral@ksi.mff.cuni.cz*

Abstract: *The Microsoft .NET Framework was from the beginning designed to support broad range of languages on a Common Language Runtime (CLR). CLR provides shared services such as garbage collection, JIT and tools integration. The other benefit is that these languages can work together and use libraries written in any of them as well as .NET Base class library (BCL).*

The CLR didn't have the support for dynamic languages. Their dynamic nature makes the compilation uneasy and places high demands on the language runtime. Unlike static languages as C# which don't require runtime support other than CLR itself. How difficult was it to make the dynamic language on .NET can be seen in the open-source implementation of PHP language on .NET called Phalanger. Its code is really complex and hard to survey. This is a serious problem for an open-source project, because it's hard to contribute.

The new Dynamic Language Runtime (DLR) makes a difference. It adds a lot of support for dynamic languages on .NET, that makes implementing the dynamic languages much easier and it also enables the interoperability between the dynamic languages built on DLR and standard static languages on .NET.

This work focuses on features of PHP dynamic language and discusses how they can be implemented in DLR. A part of this work is a pilot implementation of PHP language on DLR; the target of this implementation is to prove some new concepts, find advantages and disadvantages that DLR brings and serves as an example for implementing the dynamic language on DLR.

Keywords- dynamic language, DLR, Phalanger, PHP, .NET, CLR

Název: *Implementace dynamických jayzků nad DLR technologií*
Autor: *Miloslav Beňo*
Katedra: *Department of Software Engineering*
Vedoucí diplomové práce: *RNDr. Filip Zavoral, Ph.D.*
e-mail vedoucího: *zavoral@ksi.mff.cuni.cz*

Abstrakt: The Microsoft .NET Framework byl od jeho počátku vytvořen tak, aby podporoval široké spektrum jazyků nad *Common Language Runtime (CLR). CLR poskytuje technologie jako garbage collection, JIT nebo integrované vývojové nástroje. Další výhodou je, že tyto jazyky spolu mohou komunikovat a využívat knihovny napsané v kterémkoliv z nich a rovněž tak .NET Base class library (BCL).*

CLR nemělo podporu dynamických jazyků. Jejich dynamická povaha dělá z kompilace nelehký úkol a klade velké nároky na runtime jazyka. Narozdíl od statických jazyků jako je C#, který nepotřebuje jiný runtime než je přítomen v CLR samotném. Jak těžké bylo vytvořit dynamický jazyk nad .NET je možné vidět na open-source projektu Phalanger. Jeho kód je velmi komplexní a je obtížné do něj proniknout. To je vážný problém pro open-source projekt, jelikož je těžké se na něm začít podílet.

Nový Dynamic Language Runtime (DLR) přináší změnu. Přidává mnoho podpory pro dynamické jazyky nad .NET, což dělá implementaci dynamických jazyků znatelně snažší a rovněž umožňuje interoperabilitu mezi dynamickými jazyky vytvořenými nad DLR a standartními statickými jazyky nad .NET.

Tato práce se zaměřuje na vlastnosti PHP dynamického jazyka a diskutuje jak jej lze naimplementovat nad DLR. Částí této práce je pilotní implementace PHP jazyka nad DLR, jejímž cílem je ověřit si některé nové koncepty, najít výhody a nevýhody které DLR prináší a taktéž slouží jako příklad implementace dynamického jazyka nad DLR.

Klíčová slova- dynamický jazyk, DLR, Phalanger, PHP, .NET, CLR

2. Introduction

From the beginning the Microsoft .NET Framework [**1**] was designed for supporting various programming languages on a Common Language Runtime (CLR) [**2**] and allowing them to interoperate between each other. Beside this key value, the CLR also provides number of shared services as garbage collection, Just-In-Time compilation, and integrated tools for debugging, profiling and common security model. The .NET languages can also use the power of .NET Base class library (BCL).

The .NET simplifies the implementation of new languages, because a lot of difficult engineering work is already done. The compiler doesn't have to work with the processor's instructions, just with Microsoft Intermediate Language (MSIL) [**3**] which is a higher level set of instructions.

Although the .NET makes implementing a programming language easier, implementing a dynamic language (see 3.2) was still very difficult. The CLR didn't have any support for them. The .NET was originally designed for static languages as C# or Visual Basic. And unlike them the dynamic languages as PHP [**4**], Python [**5**] or Ruby [**6**] need broad support of the language runtime, because of their dynamic nature.

The dynamic languages have become very popular during the past years. The reason for this might be that they are more flexible and less restrictive than static languages. Because of this, it's more efficient to use them for certain applications for a price of higher possibility of runtime bugs. Developers want to use their preferable dynamic language and have .NET interoperability for building applications and providing scripting for applications.

When .NET lacked support of dynamic language the Phalanger project (see 3.3) was created. It's compiler of PHP language for .NET platform could serve as one of the examples of dynamic languages on .NET. It's a very complex project that had to solve all the aspects and difficulties of dynamic language compiler. Its complexity makes it hard to contribute and being open-source project is a big problem.

Nowadays the Dynamic Language Runtime's (DLR) (see 0.) appears to make implementing the dynamic languages a much easier job than it was before. Its goal is to

enable a common playground of dynamic languages on .NET. The DLR adds small sets of the key features to the CLR on .NET platform and set of services designed for the needs of dynamic languages. With these features it's much easier to implement the efficient dynamic language on .NET platform. More importantly, it enables interoperability between dynamic languages that use DLR as well as between static languages that already exists on .NET.

By using DLR the dynamic language automatically gets support of tools and integration with libraries and platforms. The true benefit here is sharing. It lets language implementers to focus just on their language and its semantics rather than building bunch of services for their language. So it's not necessary for example to build a garbage collector or to create development tools from scratch. Furthermore every time the DLR or .NET improves, the language implementation will benefit from this without any work.

This works focus on features, especially dynamic features, of PHP language and discuss how they can be implemented on top of the DLR. The implementation concepts are compared to the ones used in Phalanger.

The part of this work is PHPp (see 3.5) - a pilot implementation of PHP language on DLR. It has been made to try and compare these new concepts with the old ones, examine performance gains, find difficulties coming from using DLR etc. It also serves as an example of using DLR to implement the dynamic language.

Completed PHP language implementation on DLR would make whole project easier to survey and contribute. It would improve the performance, but more importantly it enables interoperability between other dynamic languages on DLR. It actually brings more benefits (see 3.4).

3. Overview

This section presents important technologies related to this work, explains dynamic languages and how they differs from static languages. Also presents motivations why to implement a language on DLR.

3.1. Common Language Infrastructure (CLI)

The core aspects of the Microsoft .NET lies in the Common Language Infrastructure (CLI) [**7**]. The purpose of CLI is to provide a language-neutral platform for application development and execution, including functions for exception handling, garbage collection, security, and interoperability. Microsoft's implementation of the CLI is called the Common Language Runtime or CLR.

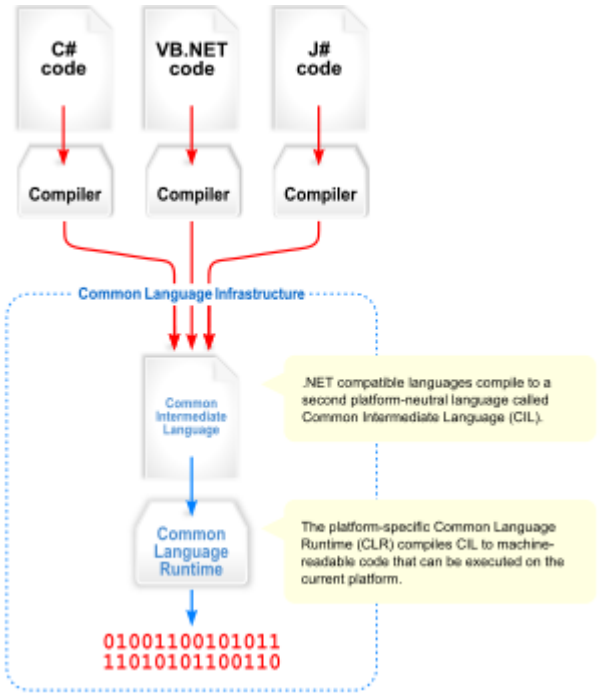


Figure 1. Visual overview of the Common Language Infrastructure (CLI)

The CLR provides the appearance of application virtual machine with Microsoft Intermediate Language (MSIL) instructions. The MSIL is a universal assembler-like language independent on the hardware. The MSIL is compiled by CLR during execution to the processors instructions of the machine where the program is actually running. The language compilers just generate MSIL and don't need to consider the capabilities of the specific CPU that will execute the program.

The MSIL is designed to describe the code of a static language. The Listing 1 illustrates the situation where two integer numbers are added. The instruction "add" takes two integer values from the stack, adds them and puts the result back into the stack. During

the compilation process it was known that x is an integer, not something else. That is different than in the dynamic languages.

C#	MSIL
`int x = 5;`	`L_0000: ldc.i4.5` `L_0001: stloc.0`

`int res = x + 2;`	`L_0002: ldloc.0`
	`L_0003: ldc.i4.2`
	`L_0004: add`
	`L_0005: stloc.1`

`Console.WriteLine(res);`	`L_0006: ldloc.1` `L_0007: call void [mscorlib]System.Console::WriteLine(int32)`

Listing 1. C# code and coresponding MSIL

3.2. Dynamic languages

The dynamic language [**8**] is a high-level programming language which behavior is known just at time of the execution, not during the compilation in contrast to the static languages. The behavior could be altered by adding new code, by extending objects and definitions, or by modifying the type system, all during the program execution.

Most of the dynamic languages are dynamically typed, but not all of them. A language is called dynamically typed when the most of the type checking is performed during runtime, not during the compilation process. User can for example generate the types during runtime. The dynamic typing is more flexible than static typing, since static type [**9**] checkers can consistently reject the code that would actually work. On the other hand the static type checking ensures that type errors will not occur during execution of a program.

The dynamic typing is the biggest problem during implementation of the dynamic language on .NET., because the MSIL is a static language and requires knowledge of the variable type during the compilation. However, the types of the variables in the dynamic language are known only at runtime and can be changed at any time. The problem here is how a compiler should declare the variables in MSIL when the type isn't known.

```
x = 20;

if (test)
   y = 10;
else
   y = "14.5";

res = x + y;
```

Listing 2. Example of the dynamic typing

Listing 2 shows the example of the dynamic code. The problem of dynamic typing system is illustrated on this example. How should be variable y typed? It could be either integer or string, depending on the value of test.

This isn't the only problem. Other issue is the semantics of the operations. The operation + might be either arithmetical addition or string concatenation, which are obviously different instructions in MSIL. And the result in this case could be float or string.

The language implementers have to deal with these problems, but usual solutions (4.4.1) aren't efficient. How to deal with them and how DLR will make it simpler and more efficient is explained at 4.4.2.

3.3. Phalanger

The Phalanger [**10**] [**11**] is a PHP language compiler for .NET Platform. It was started on Charles University in Prague in 2003 and it became first-class .NET language. It's possible to not just compile the existing PHP web application into verifiable managed .NET assemblies, but also create console applications, windows application and newly also Silverlight [**12**] applications. And on a top of that, it's possible to use it in Visual studio 2005 and 2008[1] with intellisense [**13**] and other tools coming from integration with this IDE.

In the version 1.0 primary goal was to be able to compile any existing PHP applications in to the MSIL. The 2.0 version added the goal to allow interoperability between PHP and .NET world (see .NET Interoperability5.5). This means that in the version 2.0 it is possible to use most of the .NET objects right from the PHP code. That opened many possibilities for this language. But this is one way interoperability, the both ways it's little bit more complicated, but still possible in version 2.0. Hover it becomes much easier with the new feature build into Phalanger called DuckTyping (see 5.5).

[1]Phalanger integration for Visual Studio 2010 is being developed in these moments.

The Phalanger can run real-world PHP applications with a minor or no modifications. It provides a robust platform for PHP applications with many advantages over the traditional implementation of PHP language compiler. The Phalanger benefits greatly from being built on top of .NET CLR. The execution of applications is faster when compared to standard PHP interpreter[2]. It's also easily configurable using the ASP.NET configuration systems, more reliable and secure due to the very well tested managed environment of ASP.NET. Examples include Wordpress, MediaWiki, phpMyAdmin and phpBB. The huge benefit is also lot of libraries and tools implemented on .NET that Phalanger can use.

Phalanger became an open source project in 2.0 version and it was released under Microsoft Shared Source Permissive License (it allows commercial usage, modification, and redistribution, see the license for details).

3.4. Reasons for implementing PHP language on DLR

The Phalanger project doesn't use DLR or any other compiler framework. Its implementation has to deal with all the problems which come from PHP's dynamic nature itself. This makes its code very complex and hardly readable.

PHP is a scripting language aimed particularly at rapid development of simple server-side HTML-generating scripts. Its dynamic nature makes the compilation an uneasy task and also places high demands on the language runtime. Unlike statically typed languages such as C#, which require no runtime support other than the CLR itself.

Being open-source project, the Phalanger needs support of the community. However, actual implementation makes participation on this project very difficult.

There are many reasons for reimplementing Phalanger's core to use DLR:

- **Clarity of code.** By respecting traditional architecture of DLR languages, more people could easily start to see inside the project.

[2] Newest version of PHP are actually faster than Phalanger in these moments, but Phalanger core team is working on many optimizations. Being compiler it has great performance potential, it can even use DLR in situation where it's most beneficial and combine it with its emited MSIL code.

- **Moved responsibility.** When language uses DLR a lot of hard engineering work is already done. Every time the DLR or .NET will improve the language will benefit from this too without any work.
- **Effectiveness.** The DLR offers services that can improve the efficiency of the language implementing it. As fast dynamic dispatch, call-sites, etc.
- **Interoperability.** The language can interoperate not just with .NET static languages, but also with other dynamic languages based on DLR.
- **Common hosting environment.** The DLR could be hosted in any .NET applications. Therefore the language can be used in the newly developed environments without any work.
- **IDE**. The language will get the implicit colorization, completion, and parameter tips in editing tools hosting DLR.

3.5. PHPp - PHP on top of Dynamic Language Runtime

Phalanger being compiler of PHP language into .NET platform compiles source code into MSIL. Its main purpose was to enable execution of PHP scripts on the Microsoft .NET Platform. By cooperating with ASP.NET technology enables to generate web-pages written in PHP.

As well as primary goal for Phalanger for PHP implementation on DLR is to enable full functionality of existing PHP scripts without modification. The condition is that code doesn't rely on specific features provided by UNIX platform or the Apache server nor on undocumented, obsolete or faulty functionality of the original PHP interpreter. Sometimes is problematic decide if some feature is bug or desired functionality, here could occur differences between PHPp implementation and original one. But most of the time it's necessary copy the buggy behavior because existing application rely on this.

The Phalanger project is currently part of PHPp solution and it can't be modified. This restriction is set up, because in the future version there will not be any Phalanger project included. The PHPp uses from Phalanger following parts: scanner, parser and Phalanger AST. They weren't extracted this time because they are highly dependent on

the rest of the Phalanger. That might seem as a design mistake, but it's because of the dynamic nature of PHP. All the other parts from Phalanger when used in PHPp are moved into its solution.

4. Dynamic Language Runtime

The Dynamic Language Runtime [**14**] [**15**] [**16**] mission is to enable an environment for dynamic languages on top of the .NET CLR. The language implementation should be much easier with DLR, because lot of problems coming from this are already solved and packed in DLR.

The language implementer can just focus on language specificities, as scanner, parser and runtime semantics of his language. The rest is on DLR. To be more specific, it takes care of the MSIL code generation and its optimization, as well as of the dynamic method dispatch, hosting environment and dynamic type system [**9**].

However easier language implementation isn't the key value for using DLR. By using it as a common underlying system the language implementations can easily interoperate with one another. Hence, it won't be a problem to write a library in some dynamic language and use it in another. This powerful feature applies also on statically typed .NET languages. The DLR joins both the dynamic and static worlds together.

The main idea of DLR is that it's possible to implement the dynamic language specificities on top of a generic language-agnostic abstract syntax tree, whose nodes corresponds to a specific functionality that is common to many dynamic languages. In .NET 3.5 the Expressions Trees (ETs) were introduced to model code for LINQ expressions in C# and VB. These expressions were limited; they could not contain control flow, assignment or dynamic dispatch nodes, only simple expression. Expressions Trees in version 2 were extended by previously mentioned capabilities to be able to represent full method bodies.

Other problematic issue when developing the dynamic language without DLR was performance. The dynamic operations are a lot slower than static operations, because they have to solve almost everything during runtime. The DLR helps significantly in this matter by focusing on fast dynamic dispatch capabilities and call site caching. And also every time the DLR or .NET improves, the language implementation will benefit from this without any work.

The DLR doesn't exist just for new .net dynamic languages. It also provides services for already existing languages for fast dynamic dispatch capabilities and for library support. For instance, C# in NET 4.0 comes with a new dynamic keyword that enables use of dynamic objects.

The DLR also provides common hosting environment for dynamic languages. It makes possible to employ dynamic languages in any .NET application.

This section contains some basic information about DLR, description of a few important concepts and its interoperability protocol. It is included to the work, because in the time of starting this work, there wasn't available any documentation other than source code, some blog posts and discussions with creators of DLR (see Appendix B. DLR interoperability protocol schema). Nowadays there is available documentation of DLR [**17**], however not entirely complete.

4.1. Architecture

The main parts that DLR offers are:

- Common hosting model (4.2)
- Shared abstract semantic tree representation (DLR Expression Trees 4.3)
- Unified dynamic type system (4.6)
- Support for fast dynamic operations (4.4)
- Interoperability protocol (4.5)
- Utilities (to make it easier to implement dynamic language, as *DefaultBinder*)

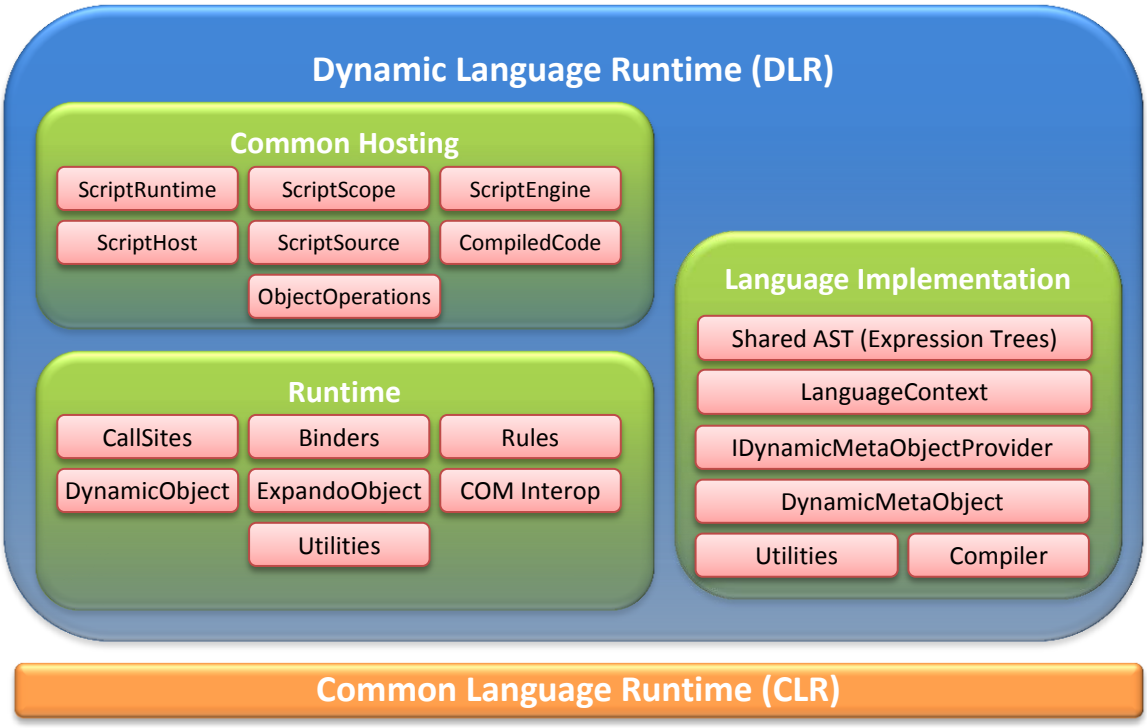


Figure 2. Conceptual architecture diagram of DLR (altered picture originally from [17])

4.2. Common hosting model

One of the advantages of DLR is its common hosting model. The application developers can use DLR in its solutions to provide scripting capabilities, or they can create integrated developments tools, testing solutions or any kind of application that can use or is related to dynamic languages. By incorporating DLR common hosting model, any DLR language can be hosted in this environment. For detailed information about common hosting model and its concepts see [**18**].

4.3. DLR Expression Trees

The DLR Expression Trees [**19**] are expression-based. All of the nodes are based on a one abstract class called *Expression* and have a result value and type. The statements are modeled as expressions having a *void* return type. *Void* is already allowed as a type, indicating there is no return value for an expression. There are a number of reasons for choosing this design for Expression Trees. DLR avoids the dual type hierarchies and being expression-based matches many languages as Lisp, Scheme, Ruby, F# and doesn't harm modeling other languages as PHP.

When building the Expression Trees it's not allowed to create their instances directly, instead it's necessary to use the factory methods. They are included in base class *Expression*. The factory methods can create the nodes with particular node kind and check the restrictions. For example, when creating a *MethodCallExpression* to call *"System.Console.WriteLine(string)",* or when creating an assignment, the factory method checks that the type of the variable is assignable from the type of the right-hand expression.

Some of the other most important nodes in the DLR Trees are:

- ***UnaryExpression*** - Negate, Convert, logical Not, Throw (it existed *ThrowExpression*, but later it was changed to *UnaryExpression* to be consistent with a design)
- ***BinaryExpression*** - comparisons, arithmetic and logical operations, array index
- ***MemberExpression*** - field or property access
- ***MethodCallExpression*** - call to a method specified via *MethodInfo*
- ***NewExpression*** - calling a constructor to create an instance of .NET class
- ***ParameterExpression*** - a value of the variable
- ***ConditionalExpression*** - *condition ? ifTrue : ifFalse*
- ...

Since Expression Trees represent complex programs, statement-like constructs are needed also:

- ***LoopExpression***
- ***ReturnExpression***
- ***TryExpression***
- ***SwitchExpression***
- ***...***

The above mentioned nodes are used to model static behaviors. However the dynamic language needs to have a way to model dynamic behaviors. Because of this Expression Trees contains *DynamicExpression* node.

DynamicExpression represents dynamic operations whose exact semantic isn't known during a compilation and has to be determined during runtime. These nodes are provided just with information where to get the semantics during runtime. The provider of this information is called Binder. In some sense it marks the sub Expression Trees as late-bound.

4.4. Dynamic operations

What makes dynamic languages truly dynamic are the dynamic operations. When compiling static code, there are static operations. The compiler knows the types of the operands and it can emit the exact instruction for this operation. The exact instruction can be emitted only when the types of the operands are known. However in dynamic code the information about the type isn't available during compilation. What to do with operation has to be determined during runtime when operands are known.

4.4.1. Before DLR

```
function dostuff($a,$b)
{
    return $a + $b;
}
```

Listing 3. Php function for adding to variables

Listing 3 shows the function that returns addition of two variables. In PHP it could be the addition of doubles, integers or when using a string as an operand it would convert it into a number and add it with the other operand. Other dynamic languages can even consider + operation as a string concatenation. There are many possible semantics for one simple operation.

Usual way how this dynamic behavior was implemented in .NET was to place a method call instead of particular instruction for this operation. The called method has to decide what action to do.

Since the types of the arguments and return type aren't known, this helper method has to take and return *object* types.

```
public static object Add(
          object a, object b)
{
  if (a is int && b is int)
    return (int)a + (int)b;

  if (a is string && b is string)
    return String.Concat(
          (string)a, (string)b);

  throw new NotImplementedException();
}
```

Listing 4. Example of simple Add helper method

Listing 4 illustrates a simple helper method for adding operation. It has just two semantics. When both arguments are integers it performs an addition on them and when they are strings a concatenation is performed. Otherwise the *NotImplementedException* is thrown.

This simple approach works quite well. Unfortunately there are a few disadvantages. First it's possible to single out only finite number of types to handle the way the int and string are handled. Every type that would need to be supported has to be added to this method and the sequences of if statements would grow. The number of tests to perform before finding the right operation wouldn't be insignificant.

The other problem is caused by using an *object* type. The primitive types have to be boxed [**20**] when they are passed as an *object* argument, which is a slow operation. And there are cases where this could be avoided. For example when a variable and int constant is added. When using the helper method with signature where the second argument is int, the not just boxing and conversion is saved, but also the number of necessary tests inside the helper method.

Implementations of languages could (and mostly had) have many types of helper methods for one single operation. That was resulting in a huge code and finally big assembly file of this language.

Despite of these disadvantages, this was the usual way how the dynamic operations were implemented in .NET dynamic languages, including Phalanger.

4.4.2. DLR approach

The way to express dynamic behavior in DLR Expression Trees is using special node called *DynamicExpression*. This node is placed into the tree on spots where dynamic operation has to be used. Hence, instead of putting the method call to a

```
MSA.Expression.Dynamic(
   Binders.BinaryOperationBinder(Operation),
   typeof(object),
   LeftExpression,
   RightExpression);
```

Listing 5. ExpressionTree for dynamic binary operation

helper method, the *DynamicExpression* node is placed there. It's provided with a reference to a so called Binder and with a list of arguments of the operation as shows Listing 5.

```
// The language's answer for int,int
if (d1 is int && d2 is int) {
    return (int)d1 + (int)d2;
}
// The language's answer for string,string
if (d1 is string && d2 is string) {
    return String.Concat(
       (string)d1, (string)d2);
}
// delegate doesn't know and asks binder
return site.Update(site, d1, d2);
```

Listing 6. CallSite's Target delegate

In general terms, this node is generated by the compiler to indicate the place where a particular operation occurs, but its exact semantics has to be determined during the runtime.

When the DLR generates the code for this node it emits the instance of so called *CallSite* (also known as dynamic site) and in a place of the actual operation it generates invocation of the *CallSite's* Target delegate.

When the delegate is invoked for the first time it just calls the binder and asks it how to perform this operation. There are two important elements in this process. First one is that the delegate could be changed (replaced by better version). And second, the delegate doesn't ask the binder to perform the operation, but asks how to do this operation. Next time when the same operation must be performed, it's not necessary to call the binder again, because *CallSite* learned it and improved the delegate.

Listing 6 shows sample of *CallSite's* Target delegate generated during runtime in C#. (DLR compiles the delegate directly to MSIL). It already knows how to perform the operation on integers and strings. When a new type combination occurs it calls the binder again.

When the binder is asked by the delegate how to perform a certain operation on particular operand types it answers by returning a so called rule. The rule is in DLR compound expression formed from the test and implementation for the operation in Expression Trees. The implementation expression tree represents the operation semantic. And the test expression specify situation in which this implementation can be reused again in the future. For example two rules can be seen at Listing 6. First rule in

the example has the test on both arguments whether they are integers. If test passes the provided implementation is performed. In this case two integers are added.

As could be seen generated delegate's code on Listing 6 is very similar to the code in Listing 4. The difference is that the code for these rules is generated only when particular operation already happened. Usually there is enormous chance when one operation with some operands happened, the next time; operands will have the same type.

Actually the previous paragraph is big simplifications, but illustrates the idea. In reality there is a complex caching mechanism behind it [**21**]. Important is that performance gain is significant, because it usually just performs only one condition and then performs the operation.

This caching system for dynamic operations in DLR is based on an original idea know as polymorphic inline cache [**22**].

Another performance gain comes from a signature of *CallSite's* target delegate. DLR generates the signature which arguments are strongly typed which prevents unnecessary conversions and boxing.

4.4.3. Set of operations

In order to support lot of languages DLR has to offer broad set of operations that has to cover complete set of features for these languages. Hence, an objects from different languages can communicate (4.5) with a set of common operations that can be performed on the objects.

Operation	**Example**	**Description**
GetMember	Object.member	Gets the member of the object. If member doesn't exist some languages can create a new one, return null, throw exception, etc.
SetMember	Object.member = value	Assigns a value to an object's member. If member doesn't exist some languages can create a new member, trow a exception, etc.
DeleteMember	delete object.member	Deletes a member of the object.

GetIndex	object[index]	Access to an indexed element of an object that retrives value.
SetIndex	object[index] = value	Access to an indexed element of an object that assigns value.
DeleteIndex	delete object[index]	Access to an indexed element of an object that deletes the element.
Invoke	a(args)	Invokes invocable object
InvokeMember	Object.a(args)	Invokes a invocable member object
CreateInstance	New X(args)	Creates an instance of an object
Convert	(TargetType)object	Converts an object to a targettype
UnaryOperation	-a	Unary operation
BinaryOperation	a + b	Binary operation

Table 1. Set of DLR interoperability operations

For unary operation and binary operation there is a parameter that specify what kind of operation it really is. This is the set of these operations: Decrement, Increment, Negate, Positive, Not, Add, Subtract, Multiply, Divide, Mod, ExclusiveOr, BitwiseAnd, BitwiseOr, LeftShift, RightShift, Equals, GreaterThan, LessThan, NotEquals, GreaterThanOrEqual, Power, LessThanOrEqual, InPlaceMultiply, InPlaceSubtract, InPlaceExclusiveOr, InPlaceLeftShift, InPlaceRightShift, InPlaceMod, InPlaceAdd, InPlaceBitwiseAnd, InPlaceBitwiseOr, InPlaceDivide, InPlacePower.

The dynamic languages in order to fulfill language semantics can implement these operations. For example *GetMember* or *SetMember* can act very differently depending on the language. If member doesn't exist, some languages can create a new one, return null, throw exception, etc.

4.5. DLR Interoperability protocol

DLR enables that languages can communicate between each other not knowing anything about the other language implementation. The idea is that a type system is based on passing messages to objects. Because all the languages have its own implementation, it's very difficult to think about it on a type level. But it's possible to view all of the languages from perspective of the objects and messages that are sent between them. Particularly the object is any object implementing *IDynamicMetaObjectProvider* interface and messages are the operations between those objects (see 4.4.3).

DLR interoperability protocol[3] consists of these main elements:

- *ExpandoObject* class
- Classes inheriting *DynamicObject* abstract class
- Classes implementing interface *IDynamicMetaObjectProvider* and its complement *DynamicMetaObject* defining the operation semantics for the classes
- Language semantics defined at *DynamicMetaObjectBinder* subclasses in the *fallback* methods

This list is ordered from the highest level to the lowest level that gives most control over operations binding.

4.5.1. DynamicObject and ExpandoObject

There is lot of code in the static languages that look like this: *Customer.Element("address").Element("zipcode").* This is exactly situation where dynamic languages offer a higher level of abstraction in a code, therefore it's more natural and simpler to read and orient inside the code like this. In the static languages isn't possible to add members to the objects during runtime, but for dynamic languages it's easy. In dynamic language it can look like this: *Customer.Address.ZipCode*.

The DLR brings to .NET a namespace *System.Dynamic*. In this namespace there is the class *ExpandoObject* which is effective implementation of a dynamic property bag. Instances of this class can add and delete members at runtime. Because it supports dynamic binding it enables syntax shown above.

In C# in .NET 4.0 there is a new keyword dynamic [**23**]. C# as a static language enables dynamic dispatch capabilities, which makes it some kind of hybrid

```
dynamic customer = new ExpandoObject();
customer.Name = "Karel Nyvlt";
customer.Phone = "555-123-456";
customer.Address = new ExpandoObject();
customer.Address.Street = "Jemenska 5";
customer.Address.City = "Prague";
customer.Address.Country = "CZ";
customer.Address.ZipCode = "16000";
```

Listing 7. Example of using ExpandoObject in C#

[3] DLR Interoperability protocol is often called also IDynamicMetaObjectProvider protocol.

static/dynamic language [**24**]. Because of this feature it's possible to write a code like shown at Listing 7. In this code there is an instance of the class *ExpandoObject* and all the members are dynamically added during runtime.

DynamicObject is a base class for specifying dynamic behavior at runtime. It allows users to override its operations and implement a custom behavior for them (4.4.3). It gives much more control than *ExpandoObject*, but has to be inherited.

Both *DynamicObject* inherited classes and *ExpandoObject* can participate in DLR interoperability protocol, because they implement an interface *IDynamicMetaObjectProvider* (see 4.5.2). They are intended for library authors that want to offer dynamic features for their libraries. However language implementers need more control and to take advantage of a fast dynamic dispatch, therefore they have to use *IDynamicMetaObjectProvider*

4.5.2. IDynamicMetaObjectProvider and DynamicMetaObject

The lower level for implementing dynamic dispatch capabilities on object than *DynamicObject* is the interface *IDynamicMetaObjectProvider*. Actually *DynamicObject* implements *IDynamicMetaObjectProvider*. This interface has only one method – *GetMetaObject()* that returns *DynamicMetaObject*.

Language implementers inherit from *DynamicMetaObject* abstract class to define its custom dynamic operations for their objects implementing *IDynamicMetaObjectProvider*. It defines same set of the operations (4.4.3) as *DynamicObject*. Important difference is that *DynamicObject* performs the operations itself. *DynamicMetaObject* creates rules for the operation (expression tree of the operation and restrictions as explained at 4.4.2). Hence, it enables using call site cache for these operations resulting in a better performance for these dynamic operations than usual solutions in the static languages, which is just a method call for every operation, usually with lot of type checks inside the method.

4.5.3. DynamicMetaObjectBinder and fallback method

To enable interoperability for a language on DLR, the language has to create its binders by deriving them from a class *DynamicMetaObjectBinder* or its subclasses. *DynamicMetaObjectBinder* has the subclasses for all the common operations from the DLR interoperability protocol (see 4.4.3).

For language implementers the most important method to override from *DynamicMetaObjectBinder* is a *Fallback* method. This method defines actual semantics for the operations in the language by means of the rules that are created in these methods equally as in *DynamicMetaObject*.

In DLR it's available *DefaultBinder* which performs the operation according to .NET semantics. It's simple way for language implementers to get started. The languages can use *DefaultBinder* by deriving from it and then override the implementations of the operations to fulfill its language semantics.

4.5.4. DLR interoperability model

When an operation is executed during the runtime, the DLR doesn't know how to perform the operation and asks *DynamicMetaObjectBinder* what to do with the operation. The main principle in DLR is that objects get first chance to perform the operation, because the object can come from another language which can have different semantics. Hence, firstly the operands are taken and if any of them is *DynamicObject* or implements *IDynamicMetaObjectProvider*, the operation is performed by *DynamicObject* itself or the operation is created in *DynamicMetaObject* for object implementing *IDynamicMetaObjectProvider*. If this isn't the case the operation will be performed according to the language semantics defined by *fallback* method of the *DynamicMetaObjectBinder*.

This model is simplified, but it can illustrate the idea of the interoperability protocol. The example can be a class *MyPhpClass* defined in PHP used from IronPython [**25**]. When *GetMember* operation is called dynamically on the instance of this type with operand *__class__* (*myobject.__class__*), firstly it gets a chance to handle the operation PHP object itself. Let's assume that this object doesn't have any member called *__class__*.

Then it gets the chance to answer *fallback* method of the *DynamicMetaObjectBinder* implemented in IronPython and because in IronPython all the classes have implicit member *__class__* (returns the object's type) the binder returns the appropriate rule.

This is very important design because it allows to have unified type system based on standard .NET types and the languages in most cases don't have to have wrappers for its objects. This is explained in more detail in section 4.6.

4.6. Unified type system

One of the main principles of DLR is using standard .NET's object as the root of the type system [**26**]. Hence there shouldn't be any runtime wrappers e.g. *PhpObject*, *PhpArray*, *RubyObject* etc. This resulting in a cleaner design and a better performance in the interoperability between languages, because it isn't necessary to rewrap the objects when they are passed from one language to another. This is possible because in DLR types aren't important, only important fact is that the objects implements *IDynamicMetaObjectProvider* interface and the complement *DynamicMetaObject* that are defining how to perform the operations on these objects.

4.7. DLR Language project structure

The DLR implementations of the languages respect certain architecture model. A developer who has awareness of one DLR language can easily orient in another DLR language implementation.

PHPp project respects this structure and can be seen as an example. It is divided into these main parts:

Directory	**Description**
• Compiler	Contains all necessary classes for compilation phase (transformation of the AST into DLR Expression trees). Including lexical scope, *AstGenerator* etc.
○ AST	This directory includes one file for each AST node.
• Hosting	Classes for language hosting environments.
• Runtime	The actual runtime of the dynamic language. The

	Language*Context* (4.8) class resides here.
○ Binders	Contains binders for twelve standard DLR operations.
○ Operations	Contains static methods for language operations.
○ Types	Classes to represent language type hierarchy and its meta-object to provide descriptions of the operations above them in forms of AST.

Table 2. Language implementation structure

4.8. LanguageContext class

When building a language on DLR, *LanguageContext* class from *Microsoft.Scripting.Runtime* has to be inherited to provide language specific facilities to communicate with DLR and properties of the language. This is the object that represents a language that is implemented on the DLR and supports the DLR's common hosting model.

```
protected override ScriptCode CompileSourceCode(
        SourceUnit sourceUnit,
        CompilerOptions options,
        ErrorSink errorSink)
```

Listing 8. CompileSourceCode method

It contains identification of the language, version information, references to namespaces distributed from the hosted environment, provides binders for the operations, global variables, and many more members that support various higher-level features in the DLR.

Almost all its members have its default behavior; the only one that is purely abstract is *CompileSourceCode* method that returns *ScriptCode* class. Hence, it has to be implemented.

Typical implementation of this method consists of calling a parser to code inside *SourceUnit* which creates an AST of the language. Then *AstGenerator* class is created and this AST is transformed into DLR Expression Tree which is given to the instance of the inherited class from the abstract *ScriptCode*. Usually the language has its own implementation of the *ScriptCode* class.

4.9. DLR Adaptive compilation

Any code that isn't known during compilation has to be compiled during runtime. This is actually desirable when the execution of this code takes more time; in fact compiled code performs more efficiently. But when the block of code has low time demands which is usual case; the compilation takes actually more amount of time than its actual execution.

The DLR comes with an efficient solution for this problem. Because a compiler on the DLR doesn't generate MSIL but expression trees, the DLR can either compile code or interpret it. Hence, the code with low time demands can be interpreted and code which execution will take more time can be compiled.

Let's define code that has low time demands as a code whose number of iterations of some part of the code (loop, function, method) during execution is lower than a given number – compilation threshold. Clearly to find out if code has low time demands is possible only during its execution (some pre- analysis could be possible, but not in the general case). Therefore, in the DLR there is present feature called adaptive compilation.

The adaptive compilation doesn't compile the code (Expression tree) before execution. The code when executed is interpreted and iterations of its code parts (loop, function, method) are counted. When the compilation threshold is reached for some part, the DLR runs compilation of this code on the background thread, while still interpreting the code. When compilation is finished, the DLR switches to the compiled code.

The adaptive compilation doesn't have to be used only for a code that isn't available in time of the compilation; it can be used generally on all the code. This would certainly improve start-up problem caused by compilation of the code. Actually IronRuby [**27**] uses this feature as its default behavior for all its code.

```
internal static Delegate/*!*/ CompileLambda(
       LambdaExpression/*!*/ lambda,
       bool debugMode,
       bool noAdaptiveCompilation,
       int compilationThreshold) {

    if (debugMode) {
        return CompileDebug(lambda);
    } else if (noAdaptiveCompilation) {
        return lambda.Compile();
    } else {
        return lambda.LightCompile(compilationThreshold);
    }
}
```

Listing 9. CompileLambda method

To use this feature it's just necessary to call an extension method [**28**] *LightCompile(this LambdaExpression, int compilationThreashold)* of a class *Microsoft.Scripting.Generation.CompilerHelpers* instead of usual *LambdaExpression.Compile* method. To follow the DLR conventions the *ScriptCode* class should be inherited. *ScriptCode* class represents compiled code that is bound to a specific *LanguageContext*, but not to a specific scope. In inherited class should be implemented a static method that can look like one depictured in Listing 9.

4.10. Summary

The DLR makes much easier to implement an efficient dynamic language on .NET. There isn't necessary to generate directly MSIL code, but just higher-level expression trees. The DLR can not only compile the code to MSIL, it can also interpret it or to use sophisticated solutions as adaptive compilation.

When implementing a dynamic language it's desirable to use static expression nodes when possible because of the performance. But in this case there won't be any performance gain in comparison with the older dynamic languages on .NET that don't use DLR.

However DLR makes real difference in the dynamic code. When isn't possible to use static expression nodes (behavior will be known during runtime), the dynamic expression nodes has to be used. In this situation the performance gain is significant

when used the DLR in comparison to the usual implementation of dynamic operations (4.4.1).

The DLR also comes with common hosting model that allow using any DLR language at the application that can use the dynamic languages as scripting language, etc.

The most important advantage is the DLR interoperability protocol that allows interoperability not only between dynamic languages based on DLR, but also between static .NET languages. This protocol is based on *IDynamicMetaObjectProvider* interface and *DynamicMetaObject* that provides semantics in the forms of rules for the operations on objects implementing this interface. When objects don't provide its own semantics the actual language semantics for dynamic operations is provided by *fallback* method of *DynamicMetaObjectBinder* for a particular operation.

The DLR can be used also on higher levels by library implementers. They can use *ExpandoObject* or *DynamicObject* to bring dynamic capabilities into their libraries; Hence, increasing the level of abstraction for its users.

The DLR creates a common environment for dynamic languages on .NET platform as well as CLR created a common environment for static languages. This all together creates very strong platform that can use the best from any language implemented on directly .NET or DLR.

5. High-level language functionalities

This section describes a main high-level language functionalities for PHP language that can be implemented on the DLR.

5.1. PHP code compilation

PHP has tree possibilities to be implemented on top of the .NET:

- Create interpreter of the PHP language to simulate its behavior in the managed environment.
- Create a front-end compiler targeting the MSIL byte-code, leaving back-end to the JIT as well as a native code generation and optimization.
- Create front-end compiler that targets DLR expression trees, so whole code generation is left to handle by this runtime.

The second one is far better solutions than first one, because MSIL is powerful enough to host many language features of PHP, however there is a problem with dynamic features that couldn't' be compiled directly and have to simulated by the runtime. The Phalanger compiler chooses this way.

The third possibility is the newest one and wasn't available when Phalanger was built. The solution based on this runtime doesn't have to generate MSIL directly, instead generates higher level expression trees. It also brings a capability to host many dynamic features of PHP. This solution is applied in PHPp.

PHP code can be divided into two main categories according its character:

- **Static**

  This code can be handled in a compilation time so the resulting code can run more effectively than a traditional interpreted PHP code. Although operations have dynamic character according to types actually presented during execution. A lot can be known from a static type analysis during the compilation, resulting into pure static code.

- **Dynamic**

  There is no way to know a dynamic code in the compile time (e.g. the code can come from a user into *eval* construct). How to handle this code has to be resolved during runtime.

Phalanger handles well a static code that can be compiled, although all the operations in the actual version are resolved during runtime. There are some optimizations, but good type analysis during compilation would result in much faster code. However a dynamic code requires an execution of a compiler to generate MSIL during the runtime, which is followed by executing the JIT compiler. This isn't very effective, but Phalanger relies on the assumption that an experienced programmer doesn't use these constructs very often, because in most of the cases it's possible to reach the same behavior by using "cleaner" techniques.

The DLR brings important advantages; because a code is represented as an expression tree, it can be either compiled or interpreted. Hence, the code with static character can be pre-compiled and dynamic features can be interpreted during the runtime. It even implements more sophisticated solution called adaptive compilation (see 4.9).

5.2. PHP functions

PHP language offers several hundred functions available to use in PHP scripts. Hence, the alternative PHP implementation has to be able to offer compatible set of functions to be able to run existing PHP applications.

PHP functions could be categorized into tree main categories:

- **PHP language constructs** that work directly with variables, functions, objects etc. (e.g. *eval, echo, require, include*, …)
- **Built-in functions** for strings and array manipulations, file system functions, regular expressions, mathematical functions, …
- **External functions** from PHP extensions used for database access, LDAP, image manipulation, …

PHP language's constructs are used very often; therefore have to be implemented completely and as effectively as possible. They have to be implemented by the compiler even if the construct looks like regular function call. The most of them have to be re-implemented because their tight coupling to the compiler's core.

Built-in function is larger set of functions than the PHP constructs. In Phalanger there is a separate project called PHP.NET Class Library which contains a completely managed implementation of these functions in C#. This Library is well designed and could be extended to implement PHP functions in new versions of PHP. It can be also reused for the DLR version, but it will have to be adapted to the new type system.

The Last category of PHP functions are external functions which are provided by unmanaged dynamically linked libraries (in Windows platform) called PHP extensions. These libraries are loaded into address space of PHP and they communicate with it using a predefined set of functions (called Zend API). Original PHP distribution contains a large amount of these extensions. They could be re-implemented into the PHP Class Library in some .NET language. However the PHP extension could be written by anyone, hence the number of them isn't limited and it's impossible to implement them all into the class library.

Phalanger is implementing a few of the extensions which are used very often and their performance has big impact on lot of PHP applications, for example MySql extension. For all the others PHP 4 extensions[4] Phalanger introduces a model to use them in .NET applications. The model is sufficiently general that could be used from any .NET application. Hence, it's suitable for using in PHPp.

This model has two modes of using PHP extensions:

- **Collocated** – The PHP extension is loaded into the same *AppDomain* as hosting application.
- **Isolated** – The PHP extension is loaded into different *AppDomain* than hosting application.

[4] PHP 5 extensions are not supported right now, because Phalanger's model for extensions doesn't implement lot of functions in new Zend API.

Collocated mode could be used in the trusted PHP extensions, because of risk of loading them into the same application domain (*AppDomain* - is a logical space in its own address space) [**29**] as a hosting application. The advantage is a huge performance gain. It could be from 5 to 10 times faster than the isolated mode.

The isolated mode loads PHP extension into an *AppDomain* of a special project called *ExtManager*. An application that wants to use a PHP extension has to communicate with the *ExtManager* through .NET remoting, because of isolated address spaces. The communication overhead is clearly a performance issue, but the main process is protected from unmanaged exceptions that can occur in the extension (programmed in native code) and could cause a termination of the sever process.

With the *ExtManager* there was introduced a *ShmChannel* communication protocol for .NET remoting based on shared memory. It's much faster alternative than a *TcpChannel* and an *HttpChannel* shipped with .NET, but still can't beat the performance of collocated mode.

5.3. ASP.NET cooperation

In the DLR implementation of PHP language there are two possible approaches to enable cooperation with ASP.NET.

First approach presented in Phalanger uses http handlers in ASP.NET. The cooperation is enabled by an object implementing an interface for responding the requests send to web-server. In IIS it's necessary to associate .php files to be handled by ASP.NET. ASP.NET process hosts web application in its application domains. Phalanger and its http handler exist on one of *AppDomains* of ASP.NET process and handle the requests. This approach is used to simulate behavior of regular PHP web application and could be used by a DLR language as well.

Second approach is to use PHP as .NET language for ASP.NET pages (.aspx), as well as could be used C#, visual basic or any other static .net language. This approach has some important assumptions, the language used with ASP.NET has to generate regular on-disk

assembly with .NET class that inherits from *System.Web.UI.Page* and language has to have its Code Document Object Model (*CodeDOM*) [**30**] provider. Both of them were fulfilled in Phalanger, because it has so called Pure mode (see 5.5), that generates CLR classes to on disk assemblies as regular static .net language. And also it has Phalanger *CodeDOM* provider.

Both assumptions are needed because of model that ASP.NET is designed on. The ASP.NET to be independent of a language uses *CodeDOM* technology. Asp.net pages are parsed and transformed to language independent *CodeDOM* tree. This tree represents .NET class that is inherited from System.Web.UI.Page, but doesn't have any other language-specific assumption. This tree is later passed to an instantiated specific *CodeDOM* provider (which provider is used depends on setting is aspx page). This *CodeDOM* provider generates the target language source code from the *CodeDOM* tree. This source code is connected with a code-behind for aspx page and compiled by the language compiler into .NET assembly (a DLL).

The languages implemented on top of the DLR can't use this original model for ASP.NET pages. In general the dynamic languages can't fulfill requirements to satisfy this model. Even though they can have a class construct they can't easily generate .NET classes, mostly because of lack of strong typing. As a consequence it's impossible to implement the language *CodeDOM* provider needed by ASP.NET. Hence, it was introduced another model to enable use of DLR languages in ASP.NET pages [**31**]. Nowadays it's available only for IronPython, but soon it should work for all DLR languages.

The principle of the new dynamic language extensibility model for ASP.NET is not to use the *CodeDOM*, rather to use feature no-compile page. This feature changes the target of parsing asp.net page file. It doesn't create the *CodeDOM* tree, but a control builder tree, a special data structure that keeps track of everything that it needs to know to create pages. Afterwards the tree instantiates all the controls that are represented by nodes in it. However this no-compile mode makes impossible to use any programmatic code (everything has to be declared), it was hacked a little bit, so the programmatic code is transparently included in special controls and later run on top of the DLR. That was only change into *System.Web.dll* (the main ASP.NET assembly).

5.4. Interactive mode support

An interactive mode is a console application that allows entering a code which is immediately evaluated and a user can see the result. It's helpful for debugging, testing and priceless for language developers that can try actually implemented features.

To support interactive mode a DLR language has to inherit a *CommandLine* class. The class has a lot of members to override. But for example for PHPp needs was necessary to inherit only a Logo member that servers to print information about the language that console shows in the start-up.

Optionally a *ConsoleOptions* class can be inherited to provide specific console starting options.

```
class PhpConsole : ConsoleHost
{
    protected override Type Provider
    {
        get { return typeof(PhpContext); }
    }

    protected override CommandLine CreateCommandLine()
    {
        return new PhpCommandLine();
    }

    [STAThread]
    static int Main(string[] args)
    {
        return new PhpConsole().Run(args);
    }
}
```

Listing 10. PHPp console implementation

The console project could look like the one from PHPp shown at Listing 10. This code is really simple, but it's sufficient for PHPp purposes.

To completely support interactive mode it's necessary to alter parser of the language to return the result of parsing in a form of *ScriptCodeParseResult* shown at Listing 11. It's important because the DLR console can recognize when user press enter weather

```
public enum ScriptCodeParseResult {
    Complete,
    Empty,
    Invalid,
    IncompleteToken,
    IncompleteStatement,
}
```

Listing 11. ScriptCodeParseResult enum

the statement is invalid or just incomplete and sill can be completed correctly. In that case console allows user to enter more code for actual statement on a new line.

5.5. .NET Interoperability

Phalanger introduced both-way interoperability from PHP language to statically typed languages on .NET. To enable a one-way interoperability from .NET to PHP was straightforward, because Phalanger compiles PHP code into MSIL and therefore it can easily create an instances of.NET classes, inherit from them and call .NET methods. Only problem was that .NET supports method overloads and PHP doesn't. Hence, there isn't any defined behavior to resolve which method overload should be called (see 7.4.4).

The other direction interoperability from PHP into .NET is more complicated, because it's crucial to use objects from a language without any type information in the statically typed environment. In Phalanger from the beginning existed so called pure mode [**32**]. In this mode compiler generates CLR classes to on disk assemblies as regular static .NET language. But it's not completely compatible with PHP, it uses more logic known from C# and therefore can be used only for some specific applications. It has several restrictions:

- **No global code can be present.** Hence, every script can contain only top-level declarations of classes and functions. The entry point is a static function called Main in the selected main class.
- **No dynamic inclusions are allowed.** It means that all inclusions are specified globally and unconditionally. Therefore scripts can be merged together during compilation.

Although in the pure mode generated classes can be used from any statically typed language on .NET, it isn't very convenient. A problem is that every method argument including return arguments will be typed as an *object*, because Phalanger can't know during compilation what types will have when used in the runtime. This approach isn't type safe. Therefore new technique based on a principle called duck typing [**33**] [**34**] was introduced.

Duck typing is based on an idea, which says that object is compatible with an interface if it has all its methods and properties required by the interface, regardless whether the object actually implements the interface or not.

Having a class that should be used from .NET, it is necessary to declare an interface in a static .NET language. This interface basically defines types for the PHP class without types. Than in the Phalanger runtime is possible to create an instance of the class implementing the interface. This object can be used in strongly typed way. Nowadays Phalanger is even capable of generating these strongly typed interfaces itself from a PHP code that includes XML comments and can create a strongly typed object transparently from the PHP code.

The DLR comes with a new interoperability model that enables full interoperability with static and dynamic languages. It's not limited only to calling methods, but it's possible to inherit from a class from other language. For example Ruby programmer could take a PHP code (using a PHP implementation on DLR), derive from its classes written in Ruby also using .NET classes and then Python programmer can take it and use in its application completely written in Python. The DLR interoperability model is explained in more detail in section 4.5.

6. Compilation process

This section describes compiler of the PHP language. Being dynamic the compiler is just one part of the language, it also relies on the runtime to perform dynamic operations.

6.1. Architecture

A compiler of the PHP language has to have all important parts as a usual static language compiler [**35**] [**36**]. The compilation goes through series of loosely coupled components: lexer, parser, AST of the language and generation of DLR Expression Trees. The final Expression Tree is highly dependent on the language runtime based on DLR. This is different than a usual static compiler which generates series of instructions for processor, or MSIL in case of .NET.

Because lexer, parser, AST are loosely coupled components, they can be taken from Phalanger and can be reused to build PHP language on top of the DLR. They almost don't have to be modified, only for some exceptions explained in section 6.2.

6.2. Lexer/Parser

To make a lexicalization and parse a PHP source code, the language compiler has to have a lexer and a parser module. They can be the same as in the Phalanger, where the lexicalization module is generated using the modified GPLEX project [**37**] which is an open source generator for lexical scanners in C# that accepts "LEX-like" input format. And the parser is built by the GPPG [**38**]; a project made for generating LALR(1) parsers that accepts a "YACC/BISON-like" input specification and produces a C# output file.

The lexer and the parser module in Phalanger could work without the rest of the project. Hence, they can be extracted and used in an implementation of PHP on the DLR with slight modifications:

- Parser should return *ScriptCodeParseResult* depictured on Listing 11.

- Lexer and parser should be modified to allow a usage of the new PHP syntax which isn't currently implemented in Phalanger e.g. namespaces[5].

Except from these small modifications these components can be reused to build an abstract syntax tree (AST).

6.3. Phalanger AST

Whole PHP source code is internally represented after parsing by Phalanger's abstract syntax tree (AST). On Figure 3 there are base classes of the hierarchy. The base abstract class for all the AST nodes is *AstNode*. The *GlobalCode* is the class that represents the root node for AST. A *LangElement* class contains the source code position information and its most important successors are *Statement* and *Expression*. The difference between these two is that *Statement* doesn't return any value and *Expression* does.

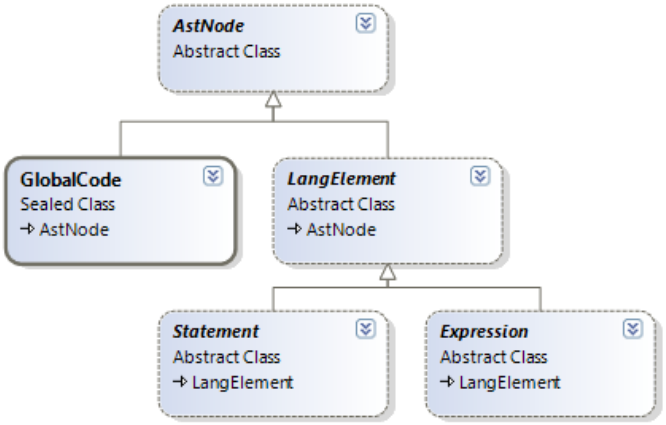


Figure 3. Base Phalanger's AST classes

The *GlobalCode*, *Statement* and *Expression* contain these important virtual methods that are overridden by its successors:

- ***Emit***. This method was used by Phalanger for generating the MSIL code. PHPp doesn't use it.
- ***Analyze***. Method for analyzing and optimizing the AST
- ***VisitMe***. The visitor method that calls the appropriate method in the visitor depending on actual AST node. It's used by Intellisense for Visual Studio integration or for creating the DLR Expression Trees.

[5] Namespaces were actually presented in Phalanger before they were presented by original PHP interpreter. They were necessary for .NET interoperability purposes. However PHP namespaces have different syntax that has to be modified in Phalanger for compatibility purposes.

6.4. Transforming Phalanger AST into Expression trees (DLR AST)

One of the main jobs of language implementations that target DLR is to produce the DLR Trees (also called Expression Trees). The Expression Trees are essentially the DLR representation of programs. Traditional language implementations as well as the DLR languages implement their own language-specific ASTs. To work with the Dynamic Language Runtime it's necessary to transform the language-specific tree into language-agnostic Expression Trees.

There are several reasons for keeping language-specific tree and not creating the expression trees directly from parser.

- **Simplicity.** It's much easier to create the language-specific tree from a parser of a language, because the tree is designed from the syntax. And expression trees might not have a direct equivalent for some AST nodes.
- **Services based on language AST.** For example intellisense services that are specialized on particular language. The expression trees might not have necessary information available.
- **Changing the target platform.** This isn't probably common case, but it can happen.
- **Good design.** It respects a common software design principles—loosely coupled components

The main parts for the translation are:

- ***TransformToDlrVisitor***. Visitor for transforming Phalanger AST into the DLR expression trees.
- ***Scope***. A class for implementing a PHP scope semantics behavior.
- ***DlrAstGenerator***. An instance of this class is included in *TransformToDlrVisitor* and provides helper methods and objects that are useful for more than one AST node. It also includes the *Scope*.
- **AST directory**. This directory includes one file for each Phalanger AST node, where is the implementation of *Transform* method that transform actual node into an expression tree node.

The PHPp have also a custom AST to represent completely PHP programs. This AST comes from Phalanger project. The PHPp uses the visitor to translate the Phalanger's AST into the expression trees. The visitor is implemented as a partial class divided into numerous files, because of big number of AST nodes. Each file corresponds with one Phalanger AST node.

7. Implementing language features

One of the main problems of PHP is a lack of the formal documentation. The documentation is presented on the internet in means of examples and discussions. For a regular user it's sufficient source of information, however for purposes of implementing a compiler it isn't sufficient. Hence, it's necessary to make a lot of discovery and try a lot of experiments on an original PHP interpreter to find out how it behaves. It also helps to look into the source code of the PHP interpreter and Phalanger's source code. But it has to be always checked a compatibility with the new version of the PHP interpreter, because of incompatibility issues with Phalanger.

This chapter presents and explains features of the PHP language and discuss how they can be implemented using the DLR. As an example it's used PHPp.

7.1. Script

PHP scripts contain mix of a HTML code and a PHP code. The code is enclosed in a special type of tags (<? ?>, <?php ?> and <script language="php"></script> "brackets"). The code outside these tags that occurs in php file is taken as a parameter to echo language construct.

In PHP there can be present a global code and global declarations that are explained in chapter 7.1.1. The global code is represented by a *GlobalCode* AST node which is transformed into a *LambdaExpression* with a *BlockExpression* that holds the top-level expressions. The *LambdaExpression* could be compiled by the DLR; the result of compilation is a delegate which can be executed. The signature of the lambda function should have as a parameter *Scope* object that contains reference to a *LanguageContext* object; it usually looks like this *Func<LanguageScope, object, object>.*

The dynamic languages on the DLR should inherit a *ScriptCode* class to represent script in this language. An instance of this the *ScriptCode* is an instance of compiled code that is bound to a specific *LanguageContext* but not to a specific *ScriptScope*. The code can be re-executed multiple times in different scopes. Hosting API counterpart for this class is *CompiledCode* class.

7.1.1. Scope

A *Scope* is an object that encloses the context where values and expressions are associated. It contains declarations or definitions of identifiers. It's used to define a visibility. Various programming languages have various types of scopes.

```
namespace N // namespace scope, merely
            //groups identifiers
{
    class C // class scope,
             //defines/declares member
             //variables and functions
    {
        public void M()//outermost block
              //(method) scope, contains
              //executable statements
        {
            if (condition)
            {
     // inner block scope for
     //conditionally executed
     //statements
            }
        }
    }
}
```

Listing 12. Various scopes declared in the language C#

Listing 12 shows scope semantics of C# language. It has scopes basically in every occurrence of braces { }.

PHP has simpler scope semantics then other languages, which is depictured on Listing 13. There are just two scopes: global and local. Each identifier belongs to just one scope. Global variables aren't available implicitly in the local scopes and have to be explicitly signed with a global keyword.

```
<?php
$a = 1; /* global scope */

function foo()
{
    $a = 2;
    echo $a; /* reference to local scope variable */
}

foo();
echo $a;
?>
/* Output of this script is: 21 */
```

Listing 13. Example of PHP scope semantics

The scope in the dynamic languages could be divided also into two classes depending how they are used:

Static (lexical or analysis) scope is used during a compilation process; it contains all variables that are known during the compilation. It has to allow a static look-up.

- **Dynamic (runtime)** scope is object used during the runtime which contains variables mostly declared during the runtime (not always). But it has to allow a dynamic look-up for these variables during the runtime.

Many other languages than PHP needs a complex hierarchy of *Scope* classes with different properties for methods, closures etc. In PHP as a runtime scope it is necessary only one class *PhpLocalScope* for storing runtime local variables for functions with unoptimized local variables. When it's necessary to use unoptimized variables is explained in section 7.2.1.

```
function foo()
{
 $x = 5;
 $y = 20;
 eval("echo $x+$y;");
 //prints 25
}
```

Listing 14. Eval construct using local scope example

The actual implementation of a runtime scope could be made as a collection of the variables e.g. hash table.

The global scope however has to be dynamic in all cases, because it has to bind host-provided variables for variables from script that includes actually processed one. In the DLR global scope is usually implemented by inheriting a *ScopeExtension* class which is supposed to extend a DLR class *Microsoft.Scripting.Runtime.Scope*[6]. This inherited class from the *ScopeExtension*, let's call it *PhpScopeExtension* should have a reference to top-level scope in the code – a real global scope object of the language.

When transforming an AST into the expression tree the runtime scope isn't actually available, there is only its representation available as the expression tree, it's usually contained in an instance of class called *ScopeBuilder*. This class actually creates the dynamic scope (if it's necessary) as a local hidden variable in a context of a function which is actually transformed. It doesn't contain real runtime variables and values, these entities are represented as the expressions trees. *ScopeBuilder* also usually contains representations of the *LanguageContext*. The real dynamic scope will be available in the runtime and it will be set from the hidden parameters. The appropriate signature of all functions is necessary.

[6] Scope class is sealed therefore can't be inherited

The actual scope during the runtime is set from a *AstGenerator* instance with a pair of methods Enter and Leave that define life span of scope which is entered. In PHP because of the simple scope semantic it's sufficient to have only *EnterFunction* and *LeaveFunction*.

PHPp has implemented only the static scope in the class *Scope*. A *GetOrMakeLocal* method is used every time an identifier is used. When it's the identifier of an variable, the variable is automatically created and returned or just returned when it already existed.

However only global and local scopes exist, it' is necessary to keep the chain of scopes, because more independent local scopes can be nested.

When an actual local scope is finished, a *FinishScope* method is called to get the DLR *LambdaExpression*, which is basically a captured block of code that is similar to a .NET method body. The *LambdaExpression* takes the input thought parameters that are expected to be fully bound.

7.1.2. Declarations

The PHP code can contain these declarations:

- Functions
- Classes
- Interfaces

The declarations stated in the global code are called unconditional and can't be rewritten. When a user tries to rewrite an unconditional declaration the program has to fail with error "Function cannot be redeclared"[7]. All unconditional declarations takes effect before any global code is evaluated. Hence, it's possible to instantiate class or call function declared later in the code.

[7] I've chosen to allow possibility of rewriting the declarations in an interactive mode session for better testing options. Thanks to this it's possible to rewrite declarations if it suites better to the actual needs .

PHP also allows placing declarations into control-flow statements as if-then-else or into a function or a method body. In this case the declaration is considered conditional, because it depends on a runtime evaluation whether or not and when it takes effect. Note that once the declaration is evaluated it can't be undone and it will live as long as global scope.

If script contains conditional declarations and one unconditional declaration, the conditional declarations don't have to be transformed into Expression trees; instead their body has to include one *ThrowExpression.*

All declarations take place in a scope that is available in all the scopes. Whether or not declarations is nested or comes from included script it's always available.

In the DLR conditional declarations could be solved in a number of ways. In PHPp the body of the function is transformed into the *LambdaExpression* which is placed into *Assign* expression node that represents assigning this lambda into a special variable available in all the scopes. When evaluated the compiled lambda is assigned into the variable. Any other assignment into this variable would throw an exception. A function call is translated into obtaining the value of the variable and calling the lambda.

7.1.3. Inclusion

In PHP there are four inclusion statements: *include*, *include_once*, *require*, *require_once* whose behavior differs only in the treatment of a repeated inclusion and in an error handling.

The inclusion allows including a specified file into the actual source code. It can be thought as an include statement appears, the file that the statement is referencing to is placed on a location of the statement. Inclusion as well as declaration can be conditional or unconditional. The behavior is analogous.

Include/*require* constructs with _once suffix means that the specific file can be loaded only once in an execution of a script. If file can't be included, the include statement

throws a warning and continues in the execution. On the other hand *require* construct throws a fatal error and execution of the script is halted.

Included script could be divided into two parts:

- Declarative
- Executive

A declarative part contains all declarations in an included script. If inclusion is unconditional, all unconditional declarations of the included script take effect. They are available after the include statement in all the scopes and can't be rewritten. If inclusion is conditional, all unconditional declarations are conditional as well depending on the condition for the inclusion statement.

All conditional declarations contained in the included script stay conditional with the respect to its condition (Treated in same way as in chapter 7.1.2).

The executive part of the included file is its global code. This code has to be executed in the scope from which is the include construct called. This means it has to have available all the variables from the scope.

A dynamic inclusion can be implemented as a method call of static method e.g. *DoInclude* with the signature requiring a runtime scope object of an including function or a method in case of conditional inclusion or a global scope in a case of unconditional inclusion. Also an including file and an included file have to be part of the signature. The *DoInclude* signature could look like this *(Scope, string includedFile, string currentFileName)->void* considering that the scope argument has also reference to *LanguageContext* object. The *LanguageContext* is important because it contains a method for compiling file and creating lambda out of it.

A DoInclude method resolves *includedFile* path in case it's a relative to an absolute path, it opens the file and proceed analogously as if processing newly opened script, only difference isn't initialized but is distributed from the parameter. Hence, the file is compiled into the lambda and run with passed scope.

In case of *include_once* or *require_once* constructs are executed, another static method has to be called, let's call it *IncludeOnce*. This method has to check whether this file was included before or not. This information could be stored in the global scope.

In PHPp inclusion is treated in a same way as the way described above. Therefore the callback routine is placed in a place of the inclusion statement. When it's called the source file name is evaluated and a content of the file is loaded and transformed into the DLR expression. This DLR expression is compiled into a lambda which is executed on the given scope.

In this section only dynamic inclusion was considered. That means that actual inclusion always takes place during runtime. That's because DLR is a runtime and in the time of writing this work pre-compilation wasn't available. This can be thought as downside of the DLR, because Phalanger gains performance benefit because of static inclusion performed during compilation. Therefore during runtime there aren't any file operations and unnecessary compilation. Nowadays in the DLR this could be compensated by cashing mechanisms and in the future pre-compilation will make possible static inclusion.

7.1.4. Dynamic code execution

This section focus on an execution of a code that isn't known during compilation and therefore it has to be compiled or interpreted during runtime. In PHP this happens in the following cases:

- **Inclusion constructs** as were explained in section 7.1.3.
- ***Eval* construct** evaluates a given string as PHP code on an actual scope.
- ***Assert* function** checks the given assertion expression and takes an appropriate action if its result is FALSE. The actions are defined by *assert_options* function.
- ***Create_function*** creates an anonymous function from the parameters passed, and returns a unique name for it.
- **Inheriting from a class unknown in compile time** is the construction that has to be evaluated during runtime. It's common to inherit from a parent class which is in a different file than a child class. The parent class file is dynamically included.

Therefore the parent class isn't known during compilation, the creation of the child class has to be postponed to the runtime. The same applies to interfaces.

All these cases can be generalized as a special case of using *eval* construct. Table 3 shows the PHP code that is equivalent for the above constructs.

Construct	**Equivalent with eval construct**
`Include $filename`	`eval("?>".file_get_contents("second.php")."<?");`
`Assert(expression);`	`eval("return expression;") == false`

`create_function ($args,$code)`	`function my_create_function($args,$body)`
	`{`
	`global $_dynamicfn;`
	`$_dynamicfn++;`
	`$name = "_dynamicfn".$_dynamicfn;`
	`$a = eval('function`
	`'.$name.'('.$args.'){'.$body.'};');`
	`return $name;`
	`}`

`Class child extends parent`
`{`
`//body`
`}`
	`eval("Class child extends parent`
	`{`
	`//body`
	`}");`

Table 3. Runtime code evaluation constructs

For inclusion statements only include equivalent is mentioned, but all the other versions of the inclusion function need just a slight modification.

All these methods can be implemented this way in PHP, but for better performance is useful implement those as static methods (in C# or the other language in which the implementation is being written) and on the places where they are called place a method call expression. However the declaration of class inheriting from unknown class can be directly transformed into calling *eval* construct.

The generalization of all the above cases allows continuing by discussing only *eval* construct implementation.

A signature of the *eval* construct as well as include needs to have a runtime scope object and a string containing valid PHP code. The function takes the string given as a parameter, runs parser on it and creates AST tree. Then the AST is converted into the DLR expression tree and compiled into a lambda function, which will run on the given scope object.

In Phalanger there was a performance drawback in the runtime code evaluation, because of overhead caused by compilation; this problem can be solved by the DLR adaptive compilation (see 4.9).

7.2. Variables

PHP variables identifiers are always string literals. If used any other type (in case of indirect variable) in place of variable name, it has to be implicitly converted into the string type.

The PHP variable always belongs to one scope, global or local. When used in a global code or included by the global code it belongs to the global scope. If a variable is used in a function or a method or it's used in a global code included there, it belongs to a local scope of a function or a method. Therefore a user function or a method can't access any other local variable of another function or any global variable implicitly (a user can use the global keyword in the function that allows usage of the specified global variable).

An exception from this rule are special auto-global or super-global variables which are automatically accessible from both scopes. They are predefined and user can't create them.

7.2.1. Local variables storage

How local variables can be stored depends on whether a static or a dynamic scope has to be used. The ideal is if only static (lexical) scope would be necessary as is in the static languages. In this case optimized local variables can be used; this means that all the local variables are compiled as CLI local variables of a function. This is optimal situation because those variables can be even stored in CPU's registers. Gladly usually PHP code is written with the good culture and therefore it is possible.

However PHP being dynamic language has to use the runtime scope for local variables in some cases. In this case we are talking about unoptimized local variables, because local variables can't be implemented as CLI local variables, but they have to be inside the

runtime scope. This is clearly slower than real local variables, because runtime scope is in fact just some collection and access into this collection is made every time the variable is used.

The unoptimized local variables have to be used in these situations:

- ***Eval* equivalent constructs** (explained in 7.1.4) take a parameter as a code and runs it on actual scope. Hence, the current runtime scope has to be used as the scope for *eval* contained code, which isn't known during compilation. The scope has to offer the runtime variables for this code.
- **Special library functions** have to have variables available at the runtime as a collection e.g. *compact, extract, get_defined_variables*

There is also a hybrid situation when optimized local variables can be used, but the function also has to use the runtime scope.

- **Indirect variables or functions** are referenced by a string (or any other expression that is converted into string) with a name of the variable evaluated during runtime. When indirect variable is contained in a function the referencing is made by a special custom look-up explained in section 7.2.3. Listing 15 shows the situation when indirect variable is used.

7.2.2. Global variable storage

Global variables can be store only in the runtime scope, because of the following reasons:

- **DLR host-provided variables** – The DLR hosting environment can provide scripts with global variables.
- **Dynamic inclusion** – when a script is included, its global code has to continue on the scope from the code that called the inclusion.
- ***$GLOBALS*** – this super-global variable is an array of global variables.

Hence, a global code can't have optimized local variables.

7.2.3. Indirect variables

In PHP variables can be accessed using two dollar (or more dollars) notion that takes a value of a specified variable, converts it to a string (if isn't already) and the string is used to reference an actual variable that is going to be accessed.

```
function foo()
{
 $x = "neco";
 $$x = 20;
 echo $neco;
 //prints 20
}
```

Listing 15. Indirect variable use example

PHP also allows using this syntax *${expression}* to access a variable. The expression is evaluated during runtime, converted to a string and used to reference the variable.

There is an ambiguity problem in PHP when using double dollar notion with arrays e.g. *$$a[1]* can mean two things. Either accessing array in the variable *$a* or accessing an array in variable that is referenced by the value in the variable $a. This can be specified clearly be using *${}* syntax i.e. *${$a[1]}* for the first case, ${$a}[1] for the other.

When indirect variables are used in a global code or in a function that already has to use the runtime scope – it has unoptimized local variables. The dynamic look-up into the runtime scope is performed. This is a simple situation and could be used for all the situations in general, but to optimize the access when function has optimized local variables the hybrid look-up can be used.

When function has optimized local variables, all of them are stored as CLI local variables, but the indirect variable access can create a new variable during runtime. Hence, the runtime scope has to be available, but in the beginning of the function it is empty and can be accessed only by the indirect variable access. The runtime scope isn't accessed by anything else because the function has optimized local variables. Hence, it doesn't contain any *eval* equivalent construct or special library function call as explained at 7.2.1.

The hybrid look-up means that for each indirect access to a variable a switch statement has to be generated. The switch statement chooses the right CLI local variable according to names and if isn't available, the access to runtime scope is performed.

The hybrid look-up has to be done this way because there isn't any local variable reflection in CLI. However switch statement over strings is highly optimized in .NET.

7.2.4. Auto-global and Super-global variables

The Auto-global variables are automatically available variables which can be provided by the hosting environment e.g. the console hosting environment should provide a script with these auto-global variables:

- *$argc* — The number of arguments passed to the script
- *$argv* — Array of arguments passed to the script

In PHP there is also a special kind of variable called super-global that is available automatically in all the scopes.

The Superglobal variables are always arrays that contain these values:

- $GLOBALS — References all variables available in global scope
- $_SERVER — Server and execution environment information
- $_GET — HTTP GET variables
- $_POST — HTTP POST variables
- $_FILES — HTTP File Upload variables
- $_REQUEST — HTTP Request variables
- $_SESSION — Session variables
- $_ENV — Environment variables
- $_COOKIE — HTTP Cookies

In the DLR the auto-global variables could be just putted into the runtime global scope. Implementing the super-global variables can be made in the *LanguageContext* class.

7.2.5. Types

The PHP language doesn't explicitly require user to work with types, actually this is hidden and the language works with types implicitly. In PHP exists types listed in Table 4.

Type	Representation
int	System.Int32
bool	System.Boolean
double	System.Double
string	System.String
array	See 7.7
object	See 7.5
resource	See 7.7

Table 4. PHP types and its representations

For the .NET interoperability purposes it's also necessary to add *System.long* type and all the operations of the language has to be extended with the semantics for this type.

7.3. Operators

In PHP there are three groups of operators: unary, binary and ternary. Unary and binary operators in PHP are dynamic and to implement them the DLR fast dynamic dispatch mechanism can be used (4.4). However the DLR doesn't support ternary operator in its set of common operations, but PHP ternary operator (*a?b:c*) isn't dynamic and it can be converted to one if statement during compilation.

In Phalanger and in the other dynamic languages which aren't built on DLR, the operators are implemented by static methods. For example an operator plus was implemented by a method *object Add(object,object).* However for optimization reasons it would be ideal to have more overloads; one overload for each type combination of operands. But this would make the source code and final binary file very big. In Phalanger there are just few type combination overloads for the most common ones e.g. *object Add(object,int)* for cases when a user adds a constant i.e. *$a + 1*.

Having a language based on DLR allows generating the *DynamicExpression* node during compilation of an operation. In PHP the node has to be provided with binder for unary operation *PhpUnaryOperationBinder* or *PhpBinaryOperationBinder* for a binary

operation. In *fallback* methods (4.5.3) of these binders there is implemented a mechanism that builds the rules for the operations. Therefore it isn't necessary to have a code for all the operators and all the type combinations of operands; just the mechanism that can generate the efficient rules for these cases.

7.4. Functions

As stated in 7.1.2 a PHP function declaration can be conditional or unconditional. They both can be transformed into the *LambdaExpression*. The *LambdaExpression* should have signature according to formal arguments defined in the function. However the formal arguments are not sufficient, because it's necessary to have a reference to the global runtime scope, because there are declared functions, classes, interfaces and global variables. The other option is that the *LanguageContext* can be present instead of the global runtime scope, assuming there is a reference to *GlobalScope* inside. Hence, the signature of the function can look like this *(GlobalScope, Object*) -> Object or (LanguageContext, Object*) -> Object.* For a method it's necessary to have *this* variable reference to represent an instance of object on which the method is called inside the method, therefore the signature can be *(Instance,GlobalScope, Object*) -> Object.*

The above stated signature would be sufficient in all static cases, therefore when it's clear in the compile time which function will be called e.g. static method or unconditional function. In this case function call could be represented by *MethodCallExpression*. But if the function is conditional (has more versions and in the runtime is decided which one will be called) or it is a method, therefore instance could be unknown during compile time, the function has to be called dynamically.

The dynamic case is solved by generating the *DynamicExpression* during compilation, therefore choosing the method is postponed to the runtime. All compiled *LambdaExpression* should be wrapped in an object (let's call it *PhpFunction*) that implements *IDynamicMetaObjectProvider* interface and its *PhpFunction.Meta* inherited from *DynamicMetaObject* providing the rule for invoke operation that calls this lambda function.

Recall, the rule is compound expression consisting of the test and implementation of the action (see 4.4.2). The test would be sufficient only testing a number of arguments, because the types are always objects[8] and implementation of this call action depends on whether the function is args-aware or –unaware (7.4.2). In the place of the *DynamicExpression* when compiled is created a *CallSite* with target delegate signature according to runtime arguments.

7.4.1. Args-aware and Args-unaware functions

In Phalanger there are used two terms: Args-aware and –unaware function. A function is args-aware if and only if it contains *eval*, *assert*, an inclusion, an indirect function call or a compile-time known call to an arguments-handling function. Otherwise, it is said to be args-unaware. The arguments-handling functions are some functions present in the Phalanger class library e.g. *func_get_arg, func_get_args* and *func_get_arg_count.* These functions have to access arguments of the function as an array.

PhpFunction.Meta will generate the rule for the invoke operation according to the type of function:

- **Args-unaware function**

  Lambda is called with a proper number of arguments.
- **Args-aware function**

Args-aware function can't be just called with a proper number of arguments, because the supplied arguments have to be available in some kind of collection. The obvious solution would be to use an array of objects, but it wouldn't be very efficient, because every call of args-aware function would need to create a new array of objects. A better solution is to use one pre-initialized collection for each call in one script run. Let's call the collection *PhpStack* (as it's called in Phalanger). Hence, the rule will pushes the arguments into the *PhpStack* before the function call. Then the pushed arguments, they

[8] The mechanism that would use also types information of the arguments could be useful in case of late compilation of a function. A function would be compiled with known types arguments and inlined to the target delegate. Hence, it would bring performance gain, because the signature would have exact types and it wouldn't be necessary convert and box arguments next time the function would be called with the same type arguments.

are available inside the function in *PhpStack* and in the end of the function they are all poped. The signature of this function can stay like it was said before *(LanguageContext, Object*) -> Object,* because *PhpStack* is initialized in the *LanguageContext* once in the beginning and it has its reference.

7.4.2. Arguments count

PHP allows calling a function with less or more arguments than is a number of formal arguments of the function. When calling a function with:

- **Fewer arguments than formal arguments**

  The arguments that aren't provided by the function call should be set to zero and its equivalents for other types (false for bool, null for an object,"" for string, 0.0 for double) and warning should be generated for each missing argument
- **More arguments than formal arguments**

  The function is called with the right number of arguments. No warning is generated because all the arguments including the ones exceeding the number of formal arguments can be accessed by arguments-handling functions and If those functions aren't present inside the function there isn't way how to access them. Therefore they can be forgotten.
- **Exact number of arguments**

  The function is called with the right number of arguments.

Hence, *PhpFunction.Meta* generates the rule for the invoke operation also according to the number of arguments supplied for the call.

7.4.3. Locals

Recall that there can be optimized or unoptimized local variables (see 7.2.1). The function's arguments have to be also considered as local variables. In case of optimized locals, when local variables are implemented directly as CLI local variables, arguments of the function are also implicitly local variables.

When the function has unoptimized locals, thus the local variables are stored in the runtime scope, the arguments of the function call have to be deep copied into the

runtime scope to be considered also local variables. This has to be done in the beginning of the function, directly after local runtime scope initialization.

7.4.4. Resolving overloads

The PHP language doesn't support function's or method's overloads. But to enable the interoperability with .NET it's necessary to have a mechanism that finds a proper .NET overload when a function is called. The overload resolution is postponed to the runtime when types of the arguments are known. Hence, the appropriate overload could be selected according to the PHP semantic.

A problem is that there isn't any PHP semantics for the overload resolution, but by examining the semantics of the PHP operations and the implicit conversions it could be assumed the right PHP-like behavior.

It can be said that the PHP operators can be implemented as function overloads with permutation of supported types. And during runtime the proper overload is selected according to the known argument types. From this it could be assumed how overload resolution should work.

The idea of the overload resolution algorithm in a pseudo code is depicted at Figure 4. It's important to rate the implicit conversions for arguments, because just using a first overload that would match with the standard PHP implicit conversion would result in an unexpected behavior. For example: *Console.WriteLine* method has many overloads and first one with only one argument is *Console.WriteLine(bool).* Therefore calling *Console.WriteLine("Hola!")* would implicitly convert string "Hola!" to bool value *true*. As a result "true" would be printed in the screen, which isn't clearly intended behavior.

The lot of important details weren't mentioned e.g. optimization, last argument can be marked as *param*, etc, but the complete overload resolution algorithm for PHP is out of scope of this work[9].

[9]This will be covered in detail in some future publication.

```
ResolveOverloads(Arguments[],Overloads[])
{

    Overloads = getOverloadsWithNArguments(Overloads, Arguments.Length);

    int i = 0;
    foreach (object Overload in Overloads)
    {
        // Rates the implicit conversions necessary to fit the overload
        result = RateConversions(Arguments,Overload);
        if (result < bestResult)
        {
            bestResult = result;
            bestIndex = i;

// No implicit conversion is ne
            if (result == BestConversion)
                break;
        }

        i++;
    }

    convertedArguments = Convert(Overloads[i],Arguments);
    Overloads[i](convertedArguments);
}
```

Figure 4. Pseudo code of overload resolution algorithm for PHP

The important is that this resolution for overloads will be implemented in the rules that come from *NetFunction.Meta* derived from *DynamicMetaObject*. As *PhpFunction* serves to represent PHP functions, *NetFunction* can serve to represent .NET method.

7.5. Objects

PHP is class–based object oriented language. It supports multiple inheritance of interfaces and single inheritance of classes. It also has one special feature; it can add or remove properties (not methods) to an instance during runtime. As stated at 7.1.2 unconditional declaration of a class can't be altered, same as conditional declaration when evaluated during runtime. Hence, the declaration of the class can't be changed, only instances can add or remove fields.

A PhpClass class can be used to represent the class declaration; it contains reference to the *PhpClass* from which inherits and to the objects that represent interfaces, collection for properties and methods. To represent methods it can be used the *PhpFunction* (7.4)

with the signature of the lambda containing reference to the instance of the class. This is to provide $*this* keyword inside the methods.

The PhpClass implementing *IDynamicMetaObjectProvider* and its complement the *PhpClass.Meta* is inherited from *DynamicMetaObject*. The *PhpClass.Meta* provides rules for operation on the class. Most important one is clearly *CreateInstance* operation (when new operator is called on the class). It has to produce a rule which initialize the *PhpObject* for representing instances.

The PhpObject also implements *IDynamicMetaObjectProvider* and has *PhpObject.Meta* inherited from *DynamicMetaObject*. It contains reference to the *PhpClass*, collection for properties which are copied during initialization from the *PhpClass*, storage for data of the instance. Methods can just stay at the *PhpClass* and their calls are forwarded there.

The PhpObject.Meta provides the rules for operations that can be performed on an instance of a class. For example a *GetMember* operation on the *PhpObject* will need the rule that examines the presence of a member with the name given by operation in the collection for properties present in the *PhpObject*. If exist it returns it, if not it prints a notice “Undefined property”.

This demonstrates the principle how the classes of dynamic language can be implemented on the DLR. But implementing the classes efficiently is a complex problem. It’s necessary to use .NET reflection to emit real classes into a dynamic assembly implementing the principles explained in chapter 4.5. However even IronRuby nowadays uses for classes this approach.

7.6. Control-flow statements

PHP includes control flow statements as language elements; however the DLR is purely expression-based. The DLR node always has a return value and type. But it’s possible to model statements just by returning void type.

These are control-flow statements available in PHP:

- While and Do-while
- For
- Foreach
- If
- Switch
- Break and continue
- Return

The behavior is known from other well-known programming languages. Hence, their implementation should be straightforward in the DLR. There is however interesting difference that makes it a little bit more complicated.

PHP optionally allows a user to declare break or continue statement with having a parameter specifying the number of loops or switch statements that should be exited before the script execution continues. The value of the parameter may not even be known at the compile-time because it may be a non-constant expression such as a variable.

During compilation of loop statements a *BranchingStack* class is used to store the list of statements where break or continue can be used in. Each of these statements is represented by a pair of Label objects which are then used as arguments for break and continue AST nodes. The argument of break or continue specifies a level of nesting of the accessed statement, with 0 or 1 being for the nearest. If there is no argument specified the nearest is taken into account. If a constant is specified, the correct statement is selected from the *BranchingStack*. For an unknown expression, a switch statement tree is created, where correct statement is selected and used based on the runtime value.

7.7. Summary

This chapter doesn't completely cover implementation of all the language features of PHP. But it should introduce into the problematic of implementation of a dynamic language, illustrates concepts and presents the problems and their solutions.

This chapter didn't mention following features:

- Variables
  - References
  - Type conversions
  - Operator chaining
- Constants
- Functions
  - Indirect Function Calls
  - Callbacks
  - Arguments passed by a reference
- Objects
  - Constructors
  - Destructors
  - Cloning
  - Conversions to string
  - Getters
  - Setters
  - Callers
  - Serialization
  - Interfaces
- Resource type
- Array type
- Error handling

8. Evaluation

The tests were performed on the following configuration:

- PC
  - Manufacturer - Lenovo
  - Model – ThinkPad T500
- Software
  - Operating System - Microsoft Windows 7 Professional x64
  - CLR Version - 2.0.50727.4200
  - DLR Version : 0.91
- CPU
  - Full Name - Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz
  - Number of Cores - 2
  - RAM – 4 GB

The expressions used in this test were evaluated 20 000 000 times. Figure 5 shows time needed to finish the task.

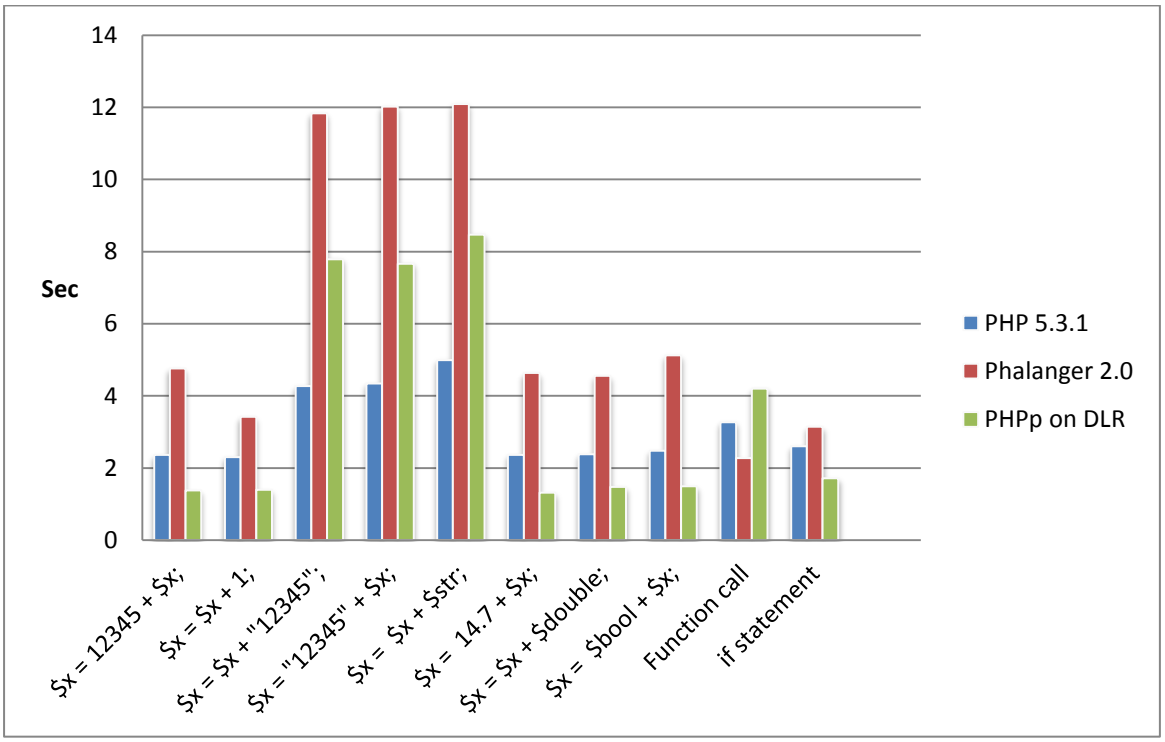


Figure 5. Performance evaluation of PHPp on DLR

As could be seen operations with integer, boolean and double are significantly faster than both Phalanger and PHP 5.3.1. But when string type with numeric value is used as operand traditional PHP is faster. However the DLR approach improved this operation, because PHPp uses same conversion algorithm as Phalanger. This conversion algorithm takes most of the time, therefore by optimizing it the speed can be improved even more.

In a static function call PHPp is the slowest one, because there isn't any optimization at all in this moment. PHPp just uses lambda functions from the DLR to implement function. Phalanger is in this test the fastest one, because of compilation. However when we compare it to the dynamic call it could be seen how DLR is strong on dynamic operations. PHPp is the fastest; it has almost the same result as in the static call.

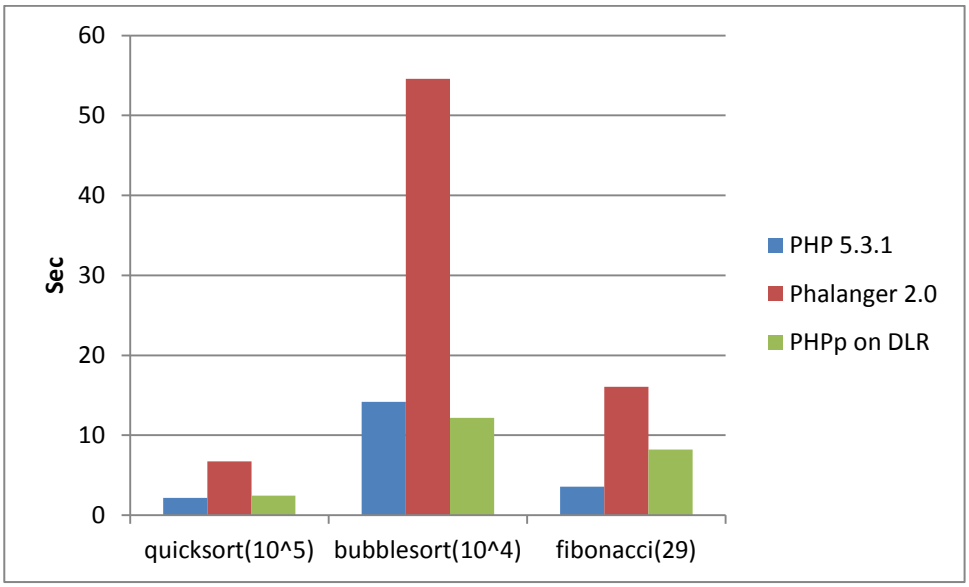


Figure 6. Performance evaluation of algorithms written in PHP

Testing of some algorithms written in PHP shows significant improvement in performance within Phalanger and PHPp. PHPp is considerably faster than Phalanger, in bubblesort test even faster than PHP. Considering that PHPp isn't optimized and there is a big space for optimizations, it is very good result.

Despite of the fact that PHPp isn't made to fully use potential of DLR and also uses old version of DLR[10], shows a real potential in the dynamic operations, which is exactly what it should. Also it has to be considered that building the dynamic language on top of DLR is significantly easier than without it and first of all it brings one invaluable benefit which is interoperability with the static and dynamic languages on .NET.

[10] It could be assumed that performance of the newest version of DLR improved.

9. Conclusion

This thesis focuses on implementing the PHP language on top of the new Dynamic Language Runtime. It describes some of the features and concepts of the DLR and discusses methods and approaches to use it for implementation of the PHP language. It's focused more deeply to the PHP language; however most of the ideas can be used in an implementation of any other dynamic language on the DLR.

The pilot implementation of the PHP language on the DLR called PHPp demonstrates implementation of some of the presented ideas and serves as an example of language implementation on the DLR. Future enhancements of the Phalanger project are planned to use the DLR and they will include many concepts from PHPp project in a step-by-step integration.

The DLR makes implementing efficient DLR language on .NET easier than it was before, because it's not necessary to emit MSIL code instead DLR expression trees are created. Because of that it's much less work for language implementers and the DLR can compile them or interpret them. The DLR can also use more sophisticated methods like adaptive compilation. The most important advantages are the interoperability with static and dynamic languages, a better performance of dynamic operations and common hosting environment.

References

[1] Microsoft Corporation..NET Framework Developer Center. http://msdn.microsoft.com/en-us/netframework/default.aspx

[2] Erik Meijer and John Gough. (2001) Technical Overview of the Common Language Runtime. http://research.microsoft.com/~emeijer/Papers/CLR.pdf

[3] Serge Lidin, *Inside Microsoft®.NET IL Assembler*.: Microsoft Press, 2002.

[4] PHP. www.php.net

[5] Python. Python Programming Language. http://www.python.org/

[6] Ruby. Ruby Programming Language. http://www.ruby-lang.org/en/

[7] ECMA. (2006, June) Standard ECMA-335: Common Language Infrastructure. http://www.ecma-international.org/publications/standards/Ecma-335.htm

[8] Laurence Tratt, "Dynamically Typed Languages," in *Advances in Computers*, 2009, pp. 149–184.

[9] E. Meijer and P. Drayton, "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages," in *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[10] Phalanger project. www.php-compiler.net

[11] J. Benda, T. Matousek, and L. Prosek, "Phalanger: Compiling and running PHP applications on the Microsoft.NET platform.," in *Proceedings of.NET Technologies 2006, the 4th International Conference on.NET Technolgoies*, Plzen, 2006, pp. 11–20.

[12] Microsoft Corporation. Silverlight. http://www.silverlight.net/

[13] Misek J. and Zavoral F, "Syntactic and Semantic Prediction in Dynamic Languages," in *SERA 2009, Studies in Computational Intelligence, Springer Verlag*, 2009.

[14] Microsoft Corporation. Microsoft Dynamic Language Runtime. http://www.codeplex.com/dlr

[15] Bill Chiles. CLR Inside Out: IronPython and the Dynamic Language Runtime. http://msdn.microsoft.com/en-us/magazine/cc163344.aspx

[16] Jim Hugunin. A Dynamic Language Runtime (DLR). http://blogs.msdn.com/b/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx

[17] Bill Chiles and Alex Turner. Dynamic Language Runtime Overview. http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation

[18] Bill Chiles. DLR Hosting Spec. http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation

[19] Bill Chiles. Expression Trees v2 Spec. http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation

[20] Jeffrey Richter. (2000, December) Type Fundamentals. http://msdn.microsoft.com/en-us/magazine/cc301569.aspx

[21] Alex Turner and Bill Chiles. Sites, Binders, and Dynamic Object Interop Spec. http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation

[22] Urs Hölzle, Craig Chambers, and David Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *Proceedings of the European Conference on Object-Oriented Programming*, 1991.

[23] Microsoft Corporation. Using Type dynamic (C# Programming Guide). http://msdn.microsoft.com/en-us/library/dd264736(VS.100).aspx

[24] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin, "Dynamic typing in a statically typed language," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 237–268, April 1991.

[25] Microsoft Corporation. IronPython - the Python programming language for the.NET Framework. http://ironpython.net/

[26] Jim Hugunin. (2007 , May) The One True Object (Part 1). http://blogs.msdn.com/b/hugunin/archive/2007/05/02/the-one-true-object-part-1.aspx

[27] Microsoft Corporation. IronRuby. http://ironruby.net/

[28] Microsoft Corporation. Extension Methods (C# Programming Guide). http://msdn.microsoft.com/en-us/library/bb383977.aspx

[29] Microsoft Corporation. Application Domains Overview. http://msdn.microsoft.com/en-us/library/2bh4z9hs(v=VS.71).aspx

[30] Microsoft Corporation. Using CodeDOM. http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx

[31] Microsoft Corporation. The New Dynamic Language Extensibility Model for ASP.NET. http://www.asp.net/learn/whitepapers/ironpython

[32] Tomas Petricek. (2007, January) Phalanger, PHP for.NET: Introduction for.NET developers. http://www.codeproject.com/KB/cross-platform/phalanger-intro.aspx

[33] Abonyi A., Balas D., Beno M., Misek J., and Zavoral F., "Phalanger Improvements," Department of Software Engineering, Charles University in Prague, Technical report 2009.

[34] Tomas Petricek. Using PHP objects from C# in a type-safe way. http://tomasp.net/blog/ducktyping-in-phalaner.aspx

[35] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*.: Addison-Wesley, Reading, Mass, 1986.

[36] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*. 1979: Addison-Wesley.

[37] John Gough. The Gardens Point Scanner Generator. http://plas.fit.qut.edu.au/gplex/

[38] Wayne Kelly. The Gardens Point Parser Generator. http://plas.fit.qut.edu.au/gppg/

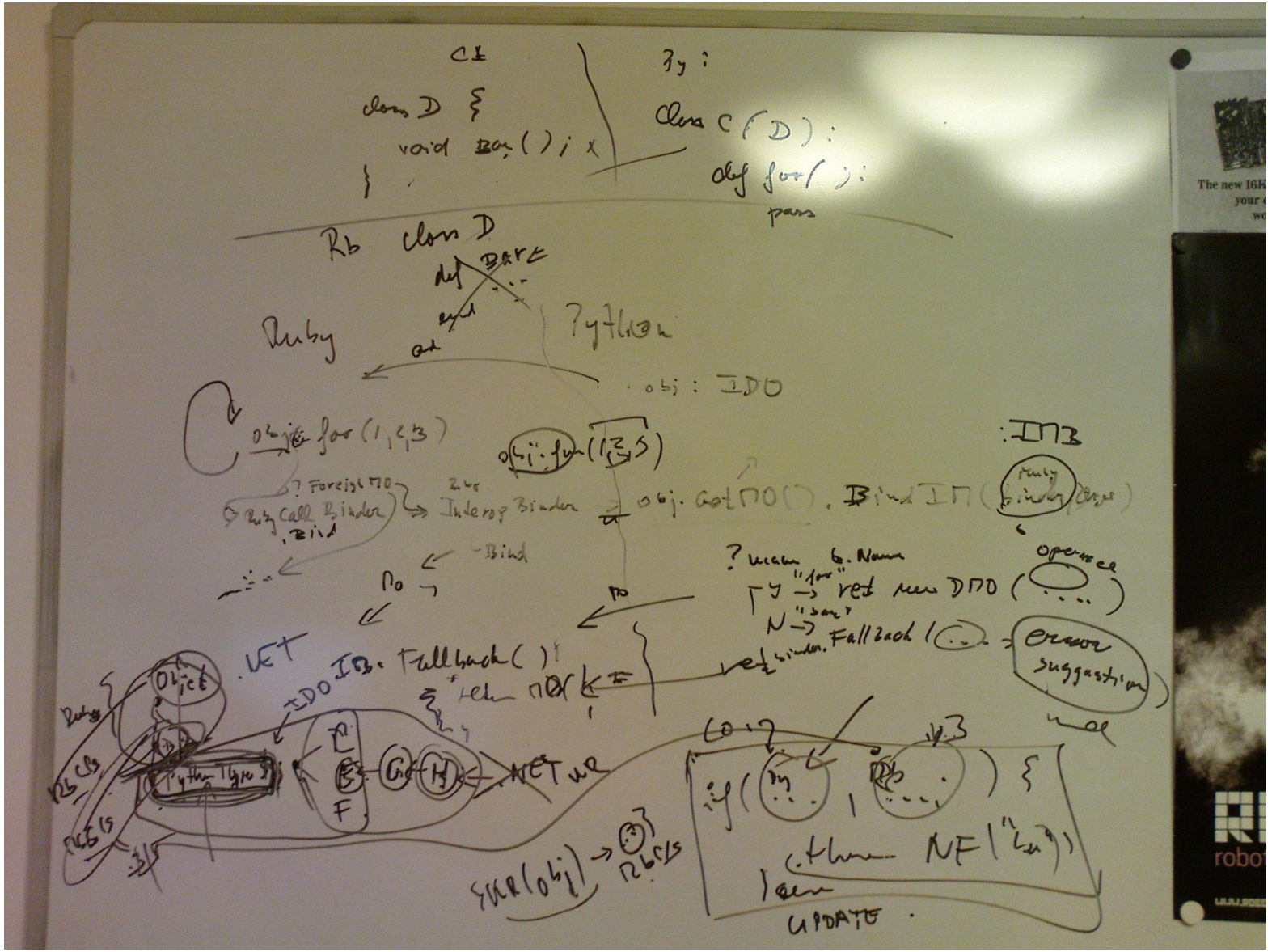
Appendix A. CD content

- **Documents**
  - Implementing the Dynamic Languages using DLR Technology.pdf
  - DLR
    - Dlr-overview.pdf - Dynamic Language Runtime overview
    - Dlr-spec-hosting.pdf - DLR common hosting model specification
    - Expr-tree-spec.pdf - Expression Trees v2 specification
    - Library-authors-introduction.pdf-DLR introduction for library authors
    - Sites-binders-dynobj-interop.pdf- Specification of interoperability protocol
    - Sympl.pdf – Documentation of the example of a dynamic language implemented on the DLR
  - Phalanger
    - User.pdf – Phalanger documentation, user's guide
- **Binaries**
  - Phalanger 2.0 (June 2010)
    - Phalanger_(June_2010).msi - Phalanger 2.0, php compiler installation
    - Phalanger_(June_2010)_VS2008_SP1.msi - Visual Studio integration installation
  - Phpdlr(PHPp)
    - PhpConsole – The interactive mode for PHP on the DLR
- **Source codes**
  - Phalanger - Phalanger project source codes.
  - DLR
    - Newest source code – newest available DLR source code
    - Release – DLR 1.0 source code
  - Phpdlr(PHPp)
    - Language -
      - PhpConsole –interactive mode for PHP on the DLR
      - Phpp – implementation of PHP on DLR
    - Runtime – older DLR's source code
    - Phalanger – older Phalanger's source code, Phpdlr(PHPp) is based on this version

Appendix B. DLR interoperability protocol schema

It was particularly very difficult to write about and work on the constantly changing environment without any good publications. The best source of knowledge was the actual source code of DLR and talks with developers of DLR. But because of this it will be actually one of the first existing publications about DLR. The DLR was released in late April this year.

This is documentation of DLR interoperability protocol from the creators of Microsoft Dynamic Language Runtime available during writing this work.


C#
class D {
void Bar();
}
Py:
class C(D):
def foo():
pass
Rb class D
def Bar
end
Ruby
Python
obj : IDO
C
obj.foo(1,2,3)
? Foreign MO
Ruby Call Binder
.Bind
Interop Binder
obj.GetMO().BindIT(
:IT3
ruby binder
operator
? name, b, Name
"foo"
ret new DMO(
"bar"
N->
ret binder.FallBack(
error suggestion
.NET
IDO IB: FallBack()
Object
Python Type
Ruby Cls
RCls
Err(obj)
RClass
if(
else
UPDATE