UNIVERSITA' DEGLI STUDI DI GENOVA

Dipartimento di Informatica e Scienze dell'Informazione

Anno Accademico 2005/2006

TESI DI LAUREA

Implementing Python in .NET

Relatore

Prof. Massimo Ancona

Correlatore

Prof. Gabriella Dodero

Candidato: Antonio Cuni

A Nicolò

Non c'è cosa più bella che guardarti
negli occhi quando brillano.

Avrei solo voluto
essere io a renderli tali.

Contents

Ringraziamenti

Desidero ringraziare il Prof. Massimo Ancona per avermi sostenuto e dato la possibilità di lavorare a questo progetto.

Un ringraziamento particolare anche ai componenti del progetto *PyPy*: Armin e Samuele per aver sempre risposto con pazienza alle mie numerose domande; Carl Friedrich per la sua amicizia e ospitalità; Holger, Michael, i due Anders, Niklaus, Eric, Maciej, Christian e tutti gli altri per avermi accolto con entusiasmo e aiutato nel mio lavoro.

Inoltre ringrazio la mia famiglia che mi ha sempre sostenuto e che soprattutto mi ha pazientemente sopportato quando ero nervoso ed irascibile negli ultimi mesi. E chiedo scusa al mio nipotino Nicolò per non aver avuto molto tempo da dedicargli.

Desidero ringraziare anche i miei amici, che mi sono sempre stati vicini soprattutto nei momenti difficili: Elisa perché c'è da sempre; Danila perché c'è sempre; Daniela, Cristina, Annalisa e tutti gli altri che non posso citare per motivi di spazio, perché so di poter contare su di loro. Una menzione particolare per Fabrizio e Marco per avermi fatto conoscere i piaceri della Val d'Aosta e del *genepy*.

Infine ringrazio i miei compagni di università, che hanno l'indiscusso merito di avermi sopportato per cinque lunghi anni: Marco, con cui ho condiviso anche i cinque anni delle superiori; Andrea, sempre pronto a rispondere alle mie innumerevoli domande; un altro Marco (per gli amici *John*), sempre pronto a pormi innumerevoli domande; Marianna, Deborah, Daniela, Francesco e tanti altri per avermi tenuto compagnia nelle mie pur non tanto frequenti giornate al DISI.

Acknowledgments

I would like to acknowledge Prof. Massimo Ancona, who supported and gave me the possibility of working for this project.

A special thank to *PyPy* people: Armin and Samuele, who always patiently answered to my many questions; Carl Friedrich for his friendship and hospitality; Holger, Michael, the two Anders, Niklaus, Eric, Maciej, Christian and all the others because they hailed me and because they helped me in my work.

I am also grateful to my family, becayse they always supported me and above all because they patiently tolerated me when I was unquiet and choleric during last months. And I apologize to my little nephew Nicolò because I had little time for him.

I would like to thank my friends too, because they always helped me, especially during hard circumstances: Elisa, because she has always been there; Danila, because she is always there; Daniela, Cristina, Annalisa and all the others I can not cite because of lack of space, because I know I can rely on them. A special mention to Fabrizio and Marco, who made me aware of pleasures of Val d'Aosta and *genepy*.

Finally, I give thanks to my classmates, who unquestionably deserve for having been tolerating me for five long years: Marco, with whom I also have shared the high school years; Andrea, who is always ready to answer to my many questions; another Marco (also known as *John*), who is always asking me many questions; Marianna, Deborah, Daniela, Francesco and many others for keeping company with me during my not so frequent days at DISI.

Preface

Python is a programming language that has became more and more popular over the years. It is a multi-paradigm language. This means that, rather than forcing coders to adopt one particular style of coding, it permits several. Object orientation, structured programming, functional programming, and aspect-oriented programming are all supported.

Python is **dynamically type-checked** and uses garbage collection for memory management. An important feature of Python is dynamic name resolution, which binds method and variable names during program execution.

Python is sometimes referred to as a **scripting language**. In practice, it is used as a dynamic programming language for both application development and occasional scripting.

The programming language itself is specified by the *Python Reference Manual* [6]. There are many implementation of this specifications: the most widely used is known as *Classical Python* (CPython) and can be considered as the reference implementation of the language.

Moreover, there is a bunch of other alternative implementations, each one with its own features: as examples we might cite *Jython* [7], which runs on top of the *Java Virtual Machine*, *IronPython* [8], which integrates in the *.NET Framework* and *Python for Series 60* [9], which runs on *Series 60* mobile phones.

Finally, the *PyPy* project [4] aims at writing a Python implementation in Python itself. The purpose of this thesis is to begin extending *PyPy* in order to obtain a **Python interpreter that runs in the .NET Framework**. We should not consider this project as a mere clone of *IronPython*: although the two projects shares some of the goals, future directions of this project may go beyond *IronPython* features, because it can be extended and reused in many different ways, as we will see in the last chapter.

Chapter 1

An overview of *PyPy*

1.1. What is *PyPy*?

Here is the *mission statement* of the *PyPy* project:

> *PyPy* is an implementation of the Python programming language written in Python itself, flexible and easy to experiment with. Our long-term goals are to target a large variety of platforms, small and large, by providing a compiler toolsuite that can produce custom Python versions. Platform, memory and threading models are to become aspects of the translation process - as opposed to encoding low level details into the language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation aspect - should become robust against language changes.

1.2. Architecture overview

PyPy is composed of two independent subsystems: the *standard interpreter* and the *translation process*.

The **standard interpreter** is the subsystem implementing the Python language, starting from the parser ending to the bytecode interpreter. Note that it can run fine on top of CPython if one is willing to pay for performance penalty for double interpretation.

The **translation process** aims at producing a different (low-level) representation of our standard interpreter. It is composed of four steps:

Flow graph generation a *flow graph* representation of the standard interpreter is produced. A combination of the bytecode interpreter and a *flow object space* performs *abstract interpretation* to record the flow of objects and execution throughout a python program into such a *flow graph*;

Annotation the *annotator* performs type inference on the flow graph;

RTyping the *RTyper* basing on type annotations, turns the flow graph into one using only low-level operations that fit the model of the target platform;

Code generation the selected *backend* compiles the resulting flow graph into the target environment; examples of backends are C, LLVM, Javascript.

1.3. RPython and translation

One of *PyPy*'s now achieved objectives is to enable translation of our **standard interpreter** into a lower-level language. In order for our translation and type inference mechanisms to work effectively, we need to restrict the dynamism of our interpreter-level Python code at some point. In the start-up phase, we are completely free to use all kinds of powerful python constructs, including metaclasses and execution of dynamically constructed strings. However, when the initialization phase finishes, all code objects involved need to adhere to a more static subset of Python: **Restricted Python**, also known as **RPython**.

RPython code is restricted in such a way that the Annotator is able to infer consistent types. How much dynamism we allow in RPython depends on, and is restricted by, the Flow Object Space and the Annotator implementation. The more we can improve this translation phase, the more dynamism we can allow. In some cases, however, it is more feasible and practical to just get rid of some of the dynamism we use in our interpreter level code. It is mainly because of this trade-off situation that the definition of RPython has shifted over time. Although the Annotator is pretty stable now and able to process the whole of *PyPy*, the RPython definition will probably continue to shift marginally as we improve it.

1.4. RPython typesystems

The annotator give us a flow graph whose variables are marked with high level type descriptors, such as `SomeInteger`, `SomeBool` or `SomeList`.

Before generating low level code we need to assign each annotated function a "real" type that can easly fit in the target machine: for example, if we want to generate C source code we might translate `SomeInteger` and `SomeBool` into plain `int` and `SomeList` into a struct containing an array of items and the lenght of that array.

This process is done by the **RTyper** and is called *rtyping*: since different target machines support different primitive operations, the rtyper allow backend writers to choose which **typesystem** to use.

Currently *PyPy* supports two different typesystems:

lltypesystem(Low Level Typesystem) represents RPython objects in terms of structs, pointers and arrays and is suitable for very low level backends such as those targeting C and LLVM;

ootypesystem (Object Oriented Typesystem) represents RPython objects in terms of classes and instances and is suitable for target with object oriented primitives, such as Java or CLI.

1.5. The Big Picture

Figure 1.1 shows how *PyPy*'s subsystems are related.

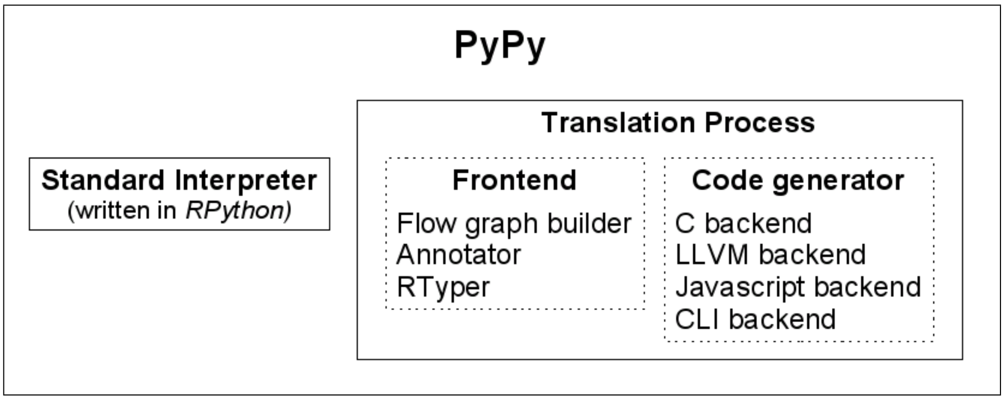

Figure 1.1: *PyPy* subsystems

The goal is to produce a **CLI backend**, i.e. a compiler that accepts RPython programs and produces .NET executables; following *PyPy* naming conventions it has been named *gencli*.

Once the backend works we can run it on top of CPython to compile the *Standard Interpreter* and obtain a .NET Python interpreter. Since *PyPy*'s Standard Interpreter aims to be compatible with CPython ideally it will be possible to run the entire translation chain on top of the just created .NET Python interpreter.

As we saw in section 1.2 the translation process is composed of four steps; since our tool stays at the very end of the chain we should take a look at what is produced by earlier steps in order to understand how the *CLI backend* works. In particular, chapter 2 will examine the *Flow graph generation* and *Annotation* steps, while chapter 3 will examine the *RTyping* step. Once we will have a good knowledge of backends' starting point, 4 will take a deep look at *gencli* internals.

Chapter 2

Flow graph model and annotator model

This chapter is about step 1 and 2 of *PyPy* architecture, as defined in section 1.2. In particular in this chapter we will inspect the *flow graph model* and the *annotator model*.

2.1. The flow graph model

In *PyPy* functions and methods are expressed by flow graphs: they group together bunch of instructions and determine the order they are executed. As an example, look at figure 2.1 shows the flow graph generated from the code in listing 2.1.

Listing 1 Flow graph example

```
def exp(base, n):
    res = 1
    while n > 0:
        res = res*base
        n = n-1
    return res
```

Flow graph are represented by instances of a number of classes that are grouped in the so called *flow graph model*. For each class

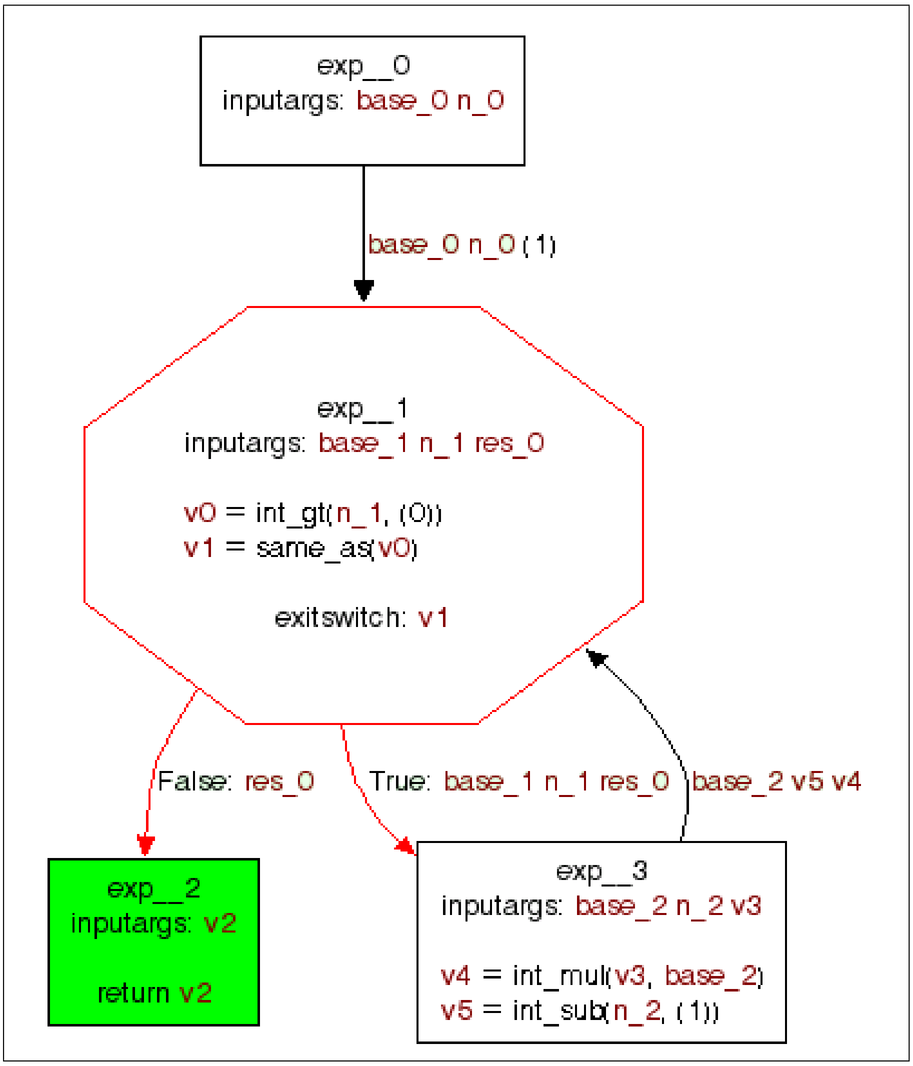

Figure 2.1: Flow graph example

we give a short description and the list of its attributes.

2.1.1. `FunctionGraph`

Flow graphs are composed by blocks and links and are represented by instances of the `FunctionGraph` class.

startblock the first block. It is where the control goes when the function is called. The input arguments of the startblock are the function's arguments. If the function takes a `*args` argument, the `args` tuple is given as the last input argument of the startblock.

returnblock the (unique) block that performs a function return. It is empty, not actually containing any `return` operation; the return is implicit. The returned value is the unique input variable of the returnblock.

exceptblock the (unique) block that raises an exception out of the function. The two input variables are the exception class and the exception value, respectively. (No other block will actually link to the exceptblock if the function does not explicitely raise exceptions.)

2.1.2. `Block`

Basic blocks are represented by instances of the `Block` class; it contains a list of operations and ends in jumps to other basic blocks. All the values that are "live" during the execution of the block are stored in `Variables`. Each basic block uses its own distinct Variables.

inputargs list of fresh, distinct Variables that represent all the values that can enter this block from any of the previous blocks.

operations list of low level operations to be executed sequentially.

exitswitch see below

exits list of `Links` representing possible jumps from the end of this basic block to the beginning of other basic blocks.

Each `Block` ends in one of the following ways:

unconditional jump exitswitch is `None`, exits contains a single `Link`.

conditional jump exitswitch is one of the Variables that appear in the `Block`, and exits contains one or more `Links` (usually 2). Each Link's exitcase gives a concrete value. This is the equivalent of a "switch": the control follows the `Link` whose exitcase matches the run-time value of the exitswitch Variable. It is a run-time error if the Variable doesn't match any exitcase.

exception catching exitswitch is `Constant_exception)`. The first Link has exitcase set to `None` and represents the non-exceptional path. The next Links have exitcase set to a subclass of `Exception`, and are taken when the *last* operation of the basic block raises a matching exception. (Thus the basic block must not be empty, and only the last operation is protected by the handler.)

return or except the returnblock and the exceptblock have operations set to an empty tuple, exitswitch to None, and exits empty.

2.1.3. `Link`

Instances of the `Link` class connect different `Block`s togheter.

prevblock the `Block` that this `Link` is an exit of.

target the target `Block` to which this `Link` points to.

args a list of Variables and Constants, of the same size as the target `Block`'s inputargs, which gives all the values passed into the next block. (Note that each `Variable` used in the prevblock may appear zero, one or more times in the `args` list.)

exitcase see above.

last_exception `None` or a `Variable`; see below.

last_exc_value `None` or a `Variable`; see below.

Note that `args` uses `Variables` from the prevblock, which are matched to the target block's `inputargs` by position, as in a tuple assignment or function call would do.

If the link is an exception-catching one, the `last_exception` and `last_exc_value` are set to two fresh `Variables` that are considered to be created when the link is entered; at run-time, they will hold the exception class and value, respectively. These two new variables can only be used in the same link's `args` list, to be passed to the next block (as usual, they may actually not appear at all, or appear several times in `args`).

2.1.4. `SpaceOperation`

This class represents a recorded (or otherwise generated) basic high level operation, such as `add` or `getitem`.

opname the name of the operation.

args list of arguments. Each one is a `Constant` or a `Variable` seen previously in the basic block.

result a *new* `Variable` into which the result is to be stored.

Note that operations usually cannot implicitly raise exceptions at run-time; so for example, code generators can assume that a `getitem` operation on a list is safe and can be performed without bound checking. The exceptions to this rule are:

1. if the operation is the last in the block, which ends with `exitswitch == Constant(last_exception)`, then the implicit exceptions must be checked for, generated, and caught appropriately

2. calls to other functions, as per `simple_call` or `call_args`, can always raise whatever the called function can raise — and such exceptions must be passed through to the parent unless they are caught as above.

2.1.5. `Variable`

A placeholder for a run-time value. There is mostly debugging stuff here.

name it is good style to use the Variable object itself instead of its `name` attribute to reference a value, although the `name` is guaranteed unique.

2.1.6. `Constant`

A constant value used as argument to a `SpaceOperation`, or as value to pass across a `Link` to initialize an input `Variable` in the target `Block`.

value the concrete value represented by this `Constant`.

key a hashable object representing the value.

A `Constant` can occasionally store a mutable Python object. It represents a static, pre-initialized, read-only version of that object. The flow graph should not attempt to actually mutate such `Constants`.

2.2. The annotator model

The major goal of the annotator is to “annotate” each variable that appears in a flow graph. An “annotation” describes all the possible Python objects that this variable could contain at run-time, based on a whole-program analysis of all the flow graphs — one per function.

An “annotation” is an instance of `SomeObject`. There are subclasses that are meant to represent specific families of objects. Note that these classes are all meant to be instantiated; the classes `SomeXxx` themselves are not the annotations.

In this section we only take a look at *what* the annotator produces, not *how*. For more details on how the annotator works, see [5].

Here is a brief overview of the class involved:

SomeObject it is the base class. An instance `SomeObject()` represents any Python object. It is used for the case where we don’t have enough information to be more precise. In practice, the presence of `SomeObject()` means that we have to make the annotated source code simpler or the annotator smarter.

SomeInteger `SomeInteger()` represents any integer. `SomeInteger(nonneg=True)` represent a non-negative integer (`>=0`).

SomeBool `SomeBool()` represents any boolean.

SomeString : `SomeString()` represents any string; `SomeChar()` a string of length 1.

SomeTuple `SomeTuple([s1,s2,..,sn])` represents a tuple of length `n`. The elements in this tuple are themselves constrained by the given list of annotations. For example, `SomeTuple([SomeInteger(), SomeString()])` represents a tuple with two items: an integer and a string. The lenght of the tuple must be known: we don't try to handle tuples of varying length (the program should use lists instead).

SomeList it stands for a list of homogeneous type (i.e. all the elements of the list are represented by a single common `SomeXxx` annotation).

SomeDict it stands for a homogeneous dictionary (i.e. all keys have the same `SomeXxx` annotation, and so have all values).

SomeInstance stands for an instance of the given class or any subclass of it. For each user-defined class seen by the annotator, we maintain a `ClassDef` describing the attributes of the instances of the class; essentially, a `ClassDef` gives the set of all class-level and instance-level attributes, and for each one, a corresponding `SomeXxx` annotation.

All the `SomeXxx` instances can optionally have a `const` attribute, which means that we know exactly which Python object the `Variable` will contain.

For a large part of operations when encountering `SomeXxx` with `const` set the annotator will do constant propagation and produce results with also 'const' set. This also means that based on `const` truth values the annotator will not flow into code that is not reachable given global constant values. A later graph transformation will remove such dead code.

Chapter 3

Introduction to *ootypesystem*

As we saw in sections 1.2 and 1.4, the goal of the *RTyper* is to turn the high-level, annotated operations of a flow graph into a low-level representation that is suitable for being easily translated by backends because it makes use of types and operations **natively available** on the target platform.

Of course, the exact low-level representation depends on what primitives we might assume the target platform provides: the role of a *PyPy* **typesystem** is to define a set of low-level types and operations to be used for targeting platforms providing a precise set of primitives.

In this chapter we will examine the **Object Oriented Typesystem** (*ootypesystem*), which is tailored for backends that natively supports constructs like classes, exceptions, and so on.

3.1. The target platform

There are plenty of object oriented languages and platforms around, each one with its own native features: they could be statically or dynamically typed, they could support or not things like multiple inheritance, classes and functions as first class order objects, generics, and so on.

The goal of *ootypesystem* is to define a trade-off between all the potential backends that let them to use the native facilities when available while not preventing other backends to work when they aren't.

3.1.1. Types and classes

ootypesystem defines a number of primitive types that are reasonably available on all platforms, as listed in table 3.1.

Bool	boolean values
Signed	signed integers (usually 32 bit)
Unsigned	unsigned integers (usually 32 bit)
SignedLongLong	signed long integers (usually 64 bit)
UnsignedLongLong	unsigned long integers (usually 64 bit)
Float	double precision floating point numbers
Char	ASCII characters
UniChar	Unicode characters
Void	used for constants known at compile time; it will disappear in the generated code

Table 3.1: *ootypesystem* primitive types

The target platform is supposed to support classes and instances with **single inheritance**. Instances of user-defined classes are mapped to the `Instance` type, whose `_superclass` attribute indicates the base class of the instance. At the very beginning of the inheritance hierarchy there is the `Root` class, i.e. the common base class between all instances; if the target platform has the notion of a common base class too, the backend can choose to map the `Root` class to its native equivalent, if any.

Object of `Instance` type can have attributes and methods: attributes are got and set by the `oogetfield` and `oosetfield` operations, while method calls are expressed by the `oosend` operation (see section 3.2.5).

Classes are passed around using the `Class` type: this is a first order class type whose only goal is to allow **runtime instantiation** of the class. Backends that don't support this feature natively, such as Java, may need to use some sort of placeholder instead.

3.1.2. Static vs. dynamic typing

The target platform is assumed to be **statically typed**, i.e. the type of each object is known at compile time.

As usual, it is possibile to convert an object from type to type only under certain conditions; there is a number of **predefined conversion** between primitive types such as from `Bool` to `Signed` or from `Signed` to `Float`. For each one of these conversions there is a corresponding low level operation, such as `cast_bool_to_int` and `cast_int_to_float` (see section 3.2.3).

Moreover it is possibile to cast instances of a class up and down the inheritance hierarchy with the `ooupcast` and `oodowncast` low level operations (see section 3.2.5). **Implicit upcasting** is not allowed, so you really need to do a `ooupcast` for converting from a subclass to a superclass.

With this design **statically typed** backends can trivially insert appropriate casts when needed, while **dynamically typed** backends can simply ignore some of the operation such as `ooupcast` and `oodowncast`. Backends that supports implicit upcasting, such as *CLI* and *Java*, can simply ignore only `ooupcast`.

3.1.3. Exception handling

Since **flow graphs** are meant to be used also for very low level backends such as C, they are **quite unstructured**, as we saw in section 2.1.3.

This means that the target platform doesn't need to have a **native exception handling** mechanism, since at the very least the backend can handle exceptions just like `genc` does.

By contrast we know that most of high level platforms natively support exception handling, so *ootypesystem* is designed to let them to use it. In particular the exception instances are typed with the `Instance` type, so the usual inheritance exception hierarchy is preserved and the native way to catch exception should just work.

3.1.4. Built-in types

It seems reasonable to assume high level platforms to provide built-in facilities for common types such as *lists* or *hashtables*.

String	self-descriptive
StringBuilder	used for dynamic building of string
List	a variable-sized, homogeneous list of object
Dict	a hashtable of homogeneous keys and values
CustomDict	same as dict, but with custom equal and hash functions
DictItemsIterator	a helper class for iterating over the elements of a `Dict`

Table 3.2: *ootypesystem* built-in types

RPython standard types such as `List` and `Dict` are implemented on top of these common types, as shown by table 3.2.

Each of these types is a subtype of `BuiltinADTType` and has set of **ADT (Abstract Data Type)** methods (hence the name of the base class) for being manipulated. Examples of ADT methods are `ll_length` for `List` and `ll_get` for `Dict`.

From the backend point of view instances of built-in types are treated exactly as plain `Instance`s, so usually no special-casing is needed. The backend is supposed to provide a bunch of classes wrapping the native ones in order to provide the right signature and semantic for the ADT methods.

As an alternative, backends can special-case the ADT types to map them directly to the native equivalent, translating the method names on-the-fly at compile time.

3.1.5. Other types

There are few more *ootypesystem* types that don't fit into categories above:

StaticMethod used for representing static methods and plain functions. As for `Class`, it is a first-class-order type: this means that `StaticMethod` objects can be passed around and called with the `indirect_call` instruction (see section 3.2.4).

Meth subclass of `StaticMethod`, used for representing bound methods.

Record used for grouping together a bunch of fields, much similar to C structs.; from the backend point of view the main difference with `Instance` is that `Record`s don't have methods.

3.1.6. Generics

Some target platforms offer native support for **generics**, i.e. classes that can be parametrized on types, not only values. For example, if one wanted to create a list using generics, a possible declaration would be to say `List<T>`, where `T` represented the type. When instantiated, one could create `List<Integer>` or `List<Animal>`. The list is then treated as a list of whichever type is specified.

Each subclass of `BuiltinADTTypes` defines a bunch of type parameters by creating some class level placeholder in the form of `PARAMNAME_T`; then it fills up the `_GENERIC_METHODS` attribute by defining the signature of each of the ADT methods using those placeholders in the appropriace places. As an example, look at listing 2, which shows part of the implementation of the *ootypesystem*'s List type.

Listing 2 Excerpt from `ootype.List`

```
class List(BuiltinADTType):
  # placeholders for types
  SELFTYPE_T = object()
  ITEMTYPE_T = object()

  ...

  def _init_methods(self):
    # 'ITEMTYPE_T' is used as a placeholder for indicating
    # arguments that should have ITEMTYPE type.
    # 'SELFTYPE_T' indicates 'self'

    self._GENERIC_METHODS = frozendict({
      "ll_length": Meth([], Signed),
      "ll_getitem_fast": Meth([Signed], self.ITEMTYPE_T),
      "ll_setitem_fast": Meth([Signed, self.ITEMTYPE_T], Void),
      "_ll_resize_ge": Meth([Signed], Void),
      "_ll_resize_le": Meth([Signed], Void),
      "_ll_resize": Meth([Signed], Void),
    })

  ...
```

Thus backends that support generics can simply look for placeholders for discovering where the type parameters are used. Backends that don't support generics can simply use the `Root` class instead (see section 3.1.1) and insert the appropriate casts where needed. Note that placeholders might also stand for primitive types, which typically require more involved casts: e.g. in Java, making wrapper objects around ints.

3.2. Low-level instructions

After flow graphs have been rtyped, they contain lists of low-level instructions; some of these low-level instructions are the same used by *lltypesystem*, while others are specific to *ootypesystem*, as we will see in this section.

Many low-level instructions are **strongly typed**, i.e. they can operate only with operands of a precise type; these instructions are prefixed with the name of the type. For historical reasons, the type name is not the same as the types we saw in section 3.1.1, as shown by table 3.3. So, for example, the low-level instruction for integer addition is `int_add`.

Bool	`bool`
Signed	`int`
Unsigned	`uint`
SignedLongLong	`llong`
UnsignedLongLong	`ullong`
Float	`float`
Char	`char`
UniChar	`unichar`

Table 3.3: Type names used by instructions

3.2.1. Comparison instructions

As the name suggests, these instructions are used to compare two values: they are composed by a prefix, indicating the type of the operands, and a suffix, that indicates the actual operation: equal to, not equal to, greater than, greater than or equal to, less than, less than or equal to (`eq`, `ne`, `gt`, `ge`, `lt`, `le`, respectively).

`int`, `uint`, `llong`, `ullong`, `float` and `char` provide instructions for all types of comparisons, while `unichar` and `bool` provide instructions for equality and disequality only.

3.2.2. Arithmetic instructions

As for comparison instructions, the arithmentic ones are prefixed by the name of the type which they operate on. All numeric types provide instructions for negation, addition, difference and multiplication (`neg`, `add`, `sub` and `mul`, respectively).

Moreover integer types provide instructions for integer division and modulo (`floordiv`, `mod`), while the `float` type provides an instruction for exact division (`truediv`).

Integer types also provide bitwise operations such as logical not, and, or, xor, left-shifting and right shifting (`invert`, `and`, `or`, `xor`, `lshift` and `rshift`).

Finally, all numeric types provide the `abs` instruction which, as the name suggest, computes the absolute value.

3.2.3. Conversion instructions

Table 3.4 shows instructions used for casting and converting values from type to type; most of them are self-explanatory.

The `is_true` instruction tests the truth value of numeric types in the usualy way: zero is false, non-zero is true, while `same_as` simply renames the variable, with no conversion at all.

3.2.4. Function call

There are two instructions for calling functions:

direct_call call the given statically-known function.

`cast_bool_to_int`
`cast_bool_to_uint`
`cast_bool_to_float`
`cast_char_to_int`
`cast_unichar_to_int`
`cast_int_to_char`
`cast_int_to_unichar`
`cast_int_to_uint`
`cast_int_to_float`
`cast_int_to_longlong`
`cast_uint_to_int`
`cast_float_to_int`
`cast_float_to_uint`
`truncate_longlong_to_int`
`is_true`
`same_as`

Table 3.4: Conversion instructions

indirect_call call the given `StaticMethod` object (see section 3.1.5).

3.2.5. Object oriented instructions

Table 3.5 shows *ootypesystem*-specific instructions:

new	create a new instance of the given statically-known class
runtimenew	create a new instance of the given `Class` object (see section 3.1.1)
oosetfield	set the value of an object's field
oogetfield	get the value of an object's field
oosend	"send a message" to an object, i.e. call a method
ooupcast	self-descriptive
oodowncast	self-descriptive
oois	identity test
ooononnull	return `False` if the object is *null*, `True` otherwise
instanceof	test if an object is an instance of the given class
subclassof	test if a class is a subclass of the given class
ooidentityhash	return the hash code of an object
oostring	convert `char`, `int`, `float` and `instances` to `string`
ooparse_int	convert a `string` to an `int`, given the base

Table 3.5: Object oriented instructions

Chapter 4

The *CLI* backend

As we saw in section 1.5 the goal of *gencli* is to compile RPython programs to the *CLI* virtual machine.

This chapter explains both how *gencli* works and the reasons behind its design, giving the pros and the cons of the alternatives that came up during the development.

Most of the code belonging to *gencli* is located in the `pypy.translator.cli` subpackage, so referred *gencli* modules are located in the `pypy/translator/cli/` subdirectory.

4.1. Target environment and language

The target of *gencli* is the *Common Language Infrastructure* environment as defined by [10].

While in an ideal world we might suppose *gencli* to run fine with every implementation conforming to that standard, we know the world we live in is far from ideal, so extra efforts can be needed to mantain compatibility with more than one implementation.

At the moment of writing the two most popular implementations of the standard are supported: Microsoft **Common Language Runtime (CLR)** [11] and **Mono** [12].

Then we have to choose how to generate the real executables. There are two main alternatives: generating source files in some high level language (such as C#) or generating assembly level code in **Intermediate Language (IL)**.

The *IL* approach is much faster during the code generation phase, because it doesn't need to call a compiler. By contrast the high level approach has two main advantages:

- the code generation part could be easier because the target language supports **high level control structures** such as structured loops;
- the generated executables take advantage of **compiler's optimizations**.

In reality the first point is not an advantage in the *PyPy* context, because as we saw in section 2.1 the flow graph we start from is quite **low level** and Python loops are already expressed in terms of branches (i.e., `goto`s).

About the compiler optimizations we must remember that the flow graph we receive from earlier stages is already optimized: *PyPy* implements a number of optimizations such a **constant propagation** and **dead code removal**, so it's not obvious if the compiler could do more.

Moreover by emitting IL instruction we are not constrained to rely on compiler choices but can directly choose how to map *ootypesystem* operations to **CLI opcodes** (see section 4.6): since the backend often know more than the compiler about the context, we might expect to produce more efficient code by selecting the most appropriate instruction; e.g., we can check for **arithmetic overflow** only when strictly necessary (see section 4.7).

The last but not least reason for choosing the low level approach is **flexibility** in how to get an executable starting from the IL code we generate:

- we can write IL code to a file, then call the `ilasm` **assembler**;
- we can directly generate code on the fly by accessing the facilities exposed by the `System.Reflection.Emit` **API**.

The second point is not feasible yet because at the moment there is no support for accessing system libraries, but in future it could lead to an interesting *gencli* feature, i.e. the ability to **emitting dynamic code** at runtime.

4.2. Handling platform differences

Since our goal is to support both *Mono* and *Microsoft CLR* we have to handle the differences between the twos; in particular the main differences are in the name of the helper tools we need to call:

- we call `ilasm` on CLR and `ilasm2` on Mono to assemble IL files;

- we call `csc` on CLR and `gmcs` on Mono to compile C# files;
- on Mono we need to call the runtime `mono` to execute programs, while on CLR we can start them directly.

The code that handles these differences is located in the `sdk.py` module: it defines an abstract class exposing some methods returning the name of the helpers and one subclass for each of the two supported platforms, as shown by listing 3.

Listing 3 Platform SDK specification

```
class MicrosoftSDK(AbstractSDK):
    RUNTIME = []
    ILASM = 'ilasm'
    CSC = 'csc'
```

3.1

```
class MonoSDK(AbstractSDK):
    RUNTIME = ['mono']
    ILASM = 'ilasm2'
    CSC = 'gmcs'
```

3.2

Then, we choose the default SDK to use based on the platform we are running on: `MicrosoftSDK` on Windows, `MonoSDK` on other platforms.

4.3. Targeting the CLI Virtual Machine

In order to write a CLI backend we have to take a number of decisions. First, we have to choose the typesystem to use: given that CLI natively supports primitives like classes and instances, **ootypesystem** is the most natural choice (see chapter 3).

Once the typesystem has been chosen there is a number of steps we have to do for completing the backend:

- map ootypesystem's types to CLI **Common Type System**'s types;

- map ootypesystem's low level operation to **CLI instructions**;
- map Python **exceptions** to CLI exceptions;
- write a **code generator** that translates a flow graph into a list of CLI instructions;
- write a **class generator** that translates ootypesystem's classes into CLI classes.

4.4. Mapping primitive types

As discussed in section 1.3 the RTyper give us a flow graph annotated with types belonging to *ootypesystem* (see chapter 3): in order to produce CLI code we need to translate these types into their **Common Type System** equivalents.

For **numeric types** the conversion is straightforward, since there is a one-to-one mapping between the two typesystems, so that e.g. `Signed` maps to `int32` and `Float` maps to `float64`.

For **character types** the choice is more difficult: RPython has two distinct types for plain ASCII and Unicode characters (named `Char` and `UniChar`), while .NET only supports Unicode with the `char` type. There are at least two ways to map plain `Char` to CTS:

- map `Char` to `int8` and `UniChar` to `char`, thus mantaining the original distinction between the two types: this has the advantage of being a one-to-one translation, but has the disadvantage that RPython **strings will not be recognized** as .NET strings, since they only would be sequences of bytes;

- map both **Char** and **UniChar** to **char**, so that Python strings will be treated as strings also by .NET: in this case there could be problems with existing Python modules that use strings as sequences of byte, such as the built-in `struct` module, so we need to pay special attention.

We think that **mapping Python strings to .NET strings** is fundamental, so we chose the second option.

The code that implements the **type-mapping** is located in the module `cts.py`.

4.5. Mapping built-in types

As we saw in section 3.1.6, *ootypesystem* defines a set of types that take advantage of **built-in** types offered by the platform.

For the sake of simplicity we decided to write **wrappers** around .NET classes in order to match the signatures required by *ootypesystem*. These wrappers are in *pypylib.dll* (see section 4.10); table 4.1 shows the .NET classes which they are built on top of.

Wrappers exploit inheritance for wrapping the original classes, so, for example, `pypy.runtime.List<T>` is a subclass of `System.Collections.Generic.List<T>` that provides methods whose names match those found in the `_GENERIC_METHODS` of

String	`System.String`
StringBuilder	`System.Text.StringBuilder`
List	`System.Collections.Generic.List<T>`
Dict	`System.Collections.Generic.Dictionary<K, V>`
CustomDict	*not implemented, yet*
DictItemsIterator	`pypy.runtime.DictItemsIterator`

Table 4.1: *gencli* built-in types

`ootype.List`.

The only exception to this rule is the `String` class, which is not wrapped since in .NET we can not subclass `System.String`. Instead, we provide a bunch of `static methods` in *pypylib.dll* that implement the methods declared by `ootype.String._GENERIC_METHODS`, then we call them by explicitly passing the string object in the argument list.

Listing 4 shows an excerpt of both the `List` and the `String` classes: note how the two implementations differ, because on the left we have an **instance method** (hence we use `this`), while on the right we have a plain **static method**.

Listing 4 Wrappers around built-in types

```
public class List<T>:
    System.Collections.\
    Generic.List<T>
{
  public int ll_length()
  {
    return this.Count;
  }
  ...
}
```

4.1

```
public class String
{
  public static int
    ll_strlen(string s)
  {
    return s.Length;
  }
}
```

4.2

4.6. Mapping instructions

PyPy's low level operations are expressed in **Static Single Information** (SSI) form; they looks like listing 5.1, where `v0` and `v1` are the arguments of the operation and `v2` is the result.

By contrast the CLI virtual machine is **stack based**, that means the each operation pops its arguments from the top of the stacks and pushes its result there. The most straightforward way to translate SSI operations into stack based operations is to **explicitly load the arguments and store the result** into the appropriate places.

Listing 5 shows an example of how basic operations are translated: the code produced works correctly but has some inefficiency issue that can be addressed during the optimization phase.

Listing 5 Example of basic operation

```
v2 = int_add(v0, v1)
```

5.1

```
LOAD v0
LOAD v1
int_add
STORE v2
```

5.2

The CLI Virtual Machine is fairly expressive, so the conversion between ***PyPy*'s low level operations** and **CLI instruction** is relatively simple: many operations maps directly to the correspondent instruction, e.g `int_add` and `int_sub` maps to `add` and `sub`.

By contrast some instructions do not have a direct correspondent and have to be rendered as a **sequence of CLI instruc-**

tions: this is the case of the "less-equal" and "greater-equal" family of instructions, that are rendered as "greater" or "less" followed by a boolean "not", respectively.

Finally, there are some instructions that cannot be rendered directly without increasing the complexity of the code generator, such as `int_abs` (which returns the absolute value of its argument). These operations are translated by calling some **helper function** written in C# (see section 4.10).

The code that implements the mapping is in the modules `metavm.py` and `opcodes.py`.

4.7. Mapping exceptions

Both RPython and CLI have its own set of **exception classes**: some of these are pretty similar; e.g., we have `OverflowError`, `ZeroDivisionError` and `IndexError` on the first side and `OverflowException`, `DivideByZeroException` and `IndexOutOfRangeException` on the other side.

The first attempt was to map RPython classes to their corresponding CLI ones: this worked for simple cases, but it would have triggered subtle bugs in more complex ones, because the two **exception hierarchies don't** completely **overlap**.

At the moment we've choosen to build an RPython exception hierarchy completely **independent** from the CLI one, but this means that we can't rely on exceptions raised by **built-in operations**. The currently implemented solution is to do an **exception translation** on-the-fly.

As an example consider the RPython `int_add_ovf` operation, that sums two integers and raises an `OverflowError` exception in case of overflow. For implementing it we can use the built-in `add.ovf` CLI instruction that raises `System.OverflowExcepion` when the result overflows, catch that exception and throw a new one, as shown in listing 4.7.

Listing 6 Exception translation

```
.try
{
    ldarg 'x_0'
    ldarg 'y_0'
    add.ovf
    stloc 'v1'
    leave __check_block_2
}
catch [mscorlib]System.OverflowException
{
    newobj instance void class OverflowError::.ctor()
    throw
}
```

4.7.1. A possible optimization

Though we haven't measured timings yet we can guess that this machinery brings to some performance penalties even in the non-overflow case; a possible optimization is to do the on-the-fly translation only when it is **strictly necessary**, i.e. only when the except clause catches an exception class whose **subclass hierarchy** is compatible with the built-in one.

As an example, consider listing 7.1: since `IndexError` has no subclasses, we can map it to `IndexOutOfBoundException` and **directly catch this one**, as shown by listing 7.2

Listing 7 Exception mapping optimization

```
try:
  return mylist[0]
except IndexError:
  return -1
```

7.1

```
try
{
  ldloc 'mylist'
  ldc.i4 0
  call int32 getitem(MyListType, int32)
  ...
}
catch [mscorlib]
    System.IndexOutOfBoundException
{
  // return -1
  ...
}
```

7.2

By contrast we can't do so if the except clause catches classes that don't directly map to any built-in class, as shown by listing 8.

4.8. Translating flow graphs

As we saw in section 2.1 in *PyPy* function and method bodies are represented by **flow graphs**, so we need to translate them to CLI IL code. Flow graphs are expressed in a format that is very suitable for being translated to low level code, so that phase is quite straightforward, though the code is a bit involed because we need to take care of three different types of blocks.

The code doing this work is located in the `Function.render` method in the file `function.py`.

First of all it **searches for variable** names and types used by each block; once they are collected it emits a `.local` IL statement

Listing 8 Exception translation

```python
try:
  return mylist[0]
except LookupError:
  return -1
```

8.1

```
.try
{
  ldloc 'mylist'
  ldc.i4 0
  .try
  {
    call int32 getitem(MyListType, int32)
  }
  catch [mscorlib]
      System.IndexOutOfBoundException
  {
    // throw a fresh exception
    newobj instance void class
        IndexError::.ctor()
    throw
  }
  ...
}
.catch LookupError
{
  // return -1
  ...
}
```

8.2

used for indicating the virtual machine the number and type of local variables used.

Then it sequentally renders all blocks in the graph, starting from the **start block** (see section 2.1.1); special care is taken for the **return block** which is always rendered at last to meet CLI requirements.

Each block starts with an **unique label** that is used for jumping across, followed by the low level instructions the block is composed of; finally there is some code that jumps to the appropriate next block.

Conditional and unconditional jumps are rendered with their corresponding IL instructions: `br`, `brtrue`, `brfalse`.

Blocks that needs to **catch exceptions** use the native facilities offered by the CLI virtual machine: the entire block is surrounded by a `.try` statement followed by as many `catch` as needed: each catching sub-block then branches to the appropriate block, as shown by listing 9.

Listing 9 Exception handling

```
try:
    # block0
    ...
except ValueError:
    # block1
    ...
except TypeError:
    # block2
    ...

# block3
```

9.1

```
block0:
 .try {
     ...
    leave block3
   }
   catch ValueError {
     ...
     leave block1
   }
   catch TypeError {
     ...
     leave block2
   }
block1:
   ...
   br block3
block2:
   ...
   br block3
block3:
   ...
```

9.2

4.9. Translating classes

As we saw in section 3.1.1, the semantic of *ootypesystem* classes is very similar to the .NET one, so the translation is mostly straightforward.

The related code is located in the module `class_.py`. Rendered classes are composed of four parts:

- fields;

- user defined methods;
- default constructor;
- the `ToString` method, mainly for testing purposes (see section 4.11).

All user defined methods are declared as `virtual`, since *ootypesystem* implicitly assumes method calls to be late bound. As a future optimization we could check if the `virtual` flag is really needed, and drop it if it's not.

The constructor does nothing more than initializing class fields to their default value.

Inheritance is straightforward too, as it is natively supported by CLI. The only noticeable thing is that we map *ootypesystem*'s `Root` class (see section 3.1.1) to the CLI equivalent `System.Object`.

4.10. The Runtime Environment

The **runtime environment** is a collection of helper classes and functions used and referenced by many of the *gencli* submodules. It is written in C#, compiled to a *DLL* (Dynamic Link Library), then linked to generated code at compile-time.

It is composed of two files: a C# source file containing the real code (`src/pypylib.cs`), and a Python module (`rte.py`) which ensures the library is **recompiled whenever the source if modified**, thus preventing bug due to forget to recompile the library.

`pypylib` is composed of three parts:

- a set of helper functions used to implements complex RPython low-level instructions such as `runtimenew` and `ooparse_int` (see section 4.6),;
- a set of helper classes wrapping built-in types, as we saw in section 4.5;
- a set of helpers used by the test framework (see section 4.11).

The first two parts are contained in the `pypy.runtime` namespace, while the third is in the `pypy.test` one.

4.11. Testing *gencli*

As the whole *PyPy*, *gencli* is a test-driven project: there is at least one **unit test** for almost each single feature of the backend. This development methodology allowed us to early discover many subtle bugs and to do some big refactoring of the code with the confidence not to break anything.

We made a big effort on writing good tests: at the moment of writing there are 310 *gencli* unit tests, composed by about one thousand of lines, i.e. one third of the global three thousands lines of code *gencli* is composed of.

The core of the testing framework is in the module `pypy.translator.cli.test.runtest`; one of the most important function of this module is `compile_function()`: it takes a Python function, compiles it to CLI and returns a Python object that runs the just created executable when called.

This way we can test *gencli* generated code just as if it were a simple Python function; we can also directly run the generated executable, whose default name is `main.exe`, from a shell: the function parameters are passed as command line arguments, and the return value is printed on the standard output, as shown by listing 10.

Listing 10 Implicit and explicit execution of CLI code

```
from pypy.translator.cli.test.runtest\
  import compile_function

def foo(x, y):
  return x+y, x*y

f = compile_function(foo, [int, int])
assert f(3, 4) == (7, 12)
```

10.1

```
$ mono main.exe 3 4
(7, 12)
```

10.2

gencli supports only few RPython types as parameters: `int`, `r_uint`, `r_longlong`, `r_ulonglong`, `bool`, `float` and one-length strings (i.e., chars). By contrast, most types are fine for being returned: these include all primitive types, list, tuples and instances.

There are some *gencli* features whose only purpose is to support the test framework. In particular, as we saw in section 4.10, *pypylib.dll* contains some helper functions that formats CLI objects in a way that can be understood by Python, to be used when printing the result value of a function.

Chapter 5

Conclusions and future work

5.1. Current status of *gencli*

At the moment of writing *gencli* is **quite mature** but still not completed: it can successfully compile a large number of test snippet (see section 4.11) and the only two medium-sized RPython programs available: *rpystone* and *richards*, which are used for benchmarking purposes, as we will see in section 5.2.

The only big feature *gencli* lacks is the support for the `CustomDict` built-in type, as we saw in section 4.5. Moreover there are few known bugs that are waiting to be fixed and that could prevent the compilation to be successful, so we have not tried to compile the whole *PyPy* interpreter yet, though it is very likely that *gencli* will be able to compile it in a few months.

Once *gencli* will have been completed, there are at least three directions we might follow to improve it in the near future:

- optimizations;
- integration of application-level code with the .NET runtime;

- integration of RPython-level code with the .NET runtime, i.e. *gencli* as a general .NET compiler.

5.2. Early benchmarks

The *PyPy* distribution comes with two standard benchmarks for measuring performances: **rpystone** and **richards**: the first is an RPython porting of the standard benchmark *pystone* Python benchmark, while the second is based on a Java version of a benchmark originally written by Dr. Martin Richards in *BCPL*.

The main difference between the twos is that *rpystone* is focused on algorithmic performances, while *richards* uses a lot of object oriented features such as inheritance and late-binding. We will see later how this difference impacts *gencli* performances.

The benchmarks have been ran on an box with the *AMD Athlon XP-M 3000+* CPU and 512 MB of RAM, under *Linux* and *Mono 1.1.13.4*. The results are compared to those obtained by *genc* with and without backend optimizations, which *gencli* is not able to take advantage of, yet (see section 5.3.1).

Backend	**Result** (pystone/seconds)	**Factor**
genc	4,926,108	1.0x
genc w/o optimizations	1,592,356	3.1x
gencli	177,429	27.8x

Table 5.1: *rpystone* results

Table 5.1 show the results for *rpystone*: as expected, *genc* is much more performant than *gencli*, especially with optimizations

Backend	**Result** (ms/iteration)	**Factor**
genc	7.43	1.0x
genc w/o optimizations	16.20	2.2x
gencli	28.65	3.8x

Table 5.2: *richards* results

turned on.

The big surprise come when examining table 5.2, which shows result for the *richards* benchmark. *gencli* is much closer to *genc*: about **3.3 times** and **1.7 times** slower that *genc* with optimizations turned on and off respectively. This is a big result, considering that at the moment the code generated by *gencli* is not optimized at all; probably one of the reasons of this great result is that the *Mono Virtual Machine* is tailored for the efficient execution of object oriented features used by *richards*.

5.3. Optimizations

There is a number of way we can improve the speed of the code generated by *gencli*.

5.3.1. Backend optimizazions

Before generating code, low-level backends such as the C and the LLVM ones run the **backend optimization** phase on the rtyped flow graph. This phase is designed to be ran with *lltypesystem*, but we might be able to use some of the optimizazions with *ootypesystem*, too. Available optimizazions include: inlining, constant folding, dead-code removal, tail-recursion optimization.

5.3.2. Stack push/pop optimitazion

The CLI Virtual Machine is a **stack based machine**: this fact doesn't play nicely with the SSI form the flowgraphs are generated in. At the moment *gencli* does a literal translation of the SSI statements, allocating a new local variable for each variable of the flowgraph, as we saw in section 4.6.

For example, consider the RPython code and the corresponding flowgraph in listing 11. Listing 12.1 shows the code as it is generated by *gencli*: as you can see, the results of `add` and `sub` are stored in `v0` and `v1`, respectively, then `v0` and `v1` are reloaded onto stack. These store/load is redundant, since the code would work nicely even without them, as shown by listing 12.2.

Listing 11 RPython snippet and its flow graph

```
def bar(x, y):
    foo(x+y, x-y)
```

11.1

```
inputargs: x_0 y_0
v0 = int_add(x_0, y_0)
v1 = int_sub(x_0, y_0)
v2 = directcall((sm foo), v0, v1)
```

11.2

If we check the native code generated by the **Mono JIT compiler** on *x86* we can see that this redundand code is not optimized, so we might consider to optimize it manually; it should not be so difficult, but it is not trivial becasue we have to make sure that the dropped locals are used only once.

Listing 12 Unoptimized and optimized IL code

```
.locals init (int32 v0,
              int32 v1,
              int32 v2)

block0:
  ldarg 'x_0'
  ldarg 'y_0'
  add
  stloc 'v0'
  ldarg 'x_0'
  ldarg 'y_0'
  sub
  stloc 'v1'
  ldloc 'v0'
  ldloc 'v1'
  call int32 foo(int32, int32)
  stloc 'v2'
```

12.1

```
.locals init (int32 v2)

block0:
  ldarg 'x_0'
  ldarg 'y_0'
  add
  ldarg 'x_0'
  ldarg 'y_0'
  sub
  call int32 foo(int32, int32)
  stloc 'v2'
```

12.2

5.3.3. Mapping RPython exceptions to native CLI exceptions

We have already addressed this optimization in section 4.7.1.

5.4. Integrate the interpreter with the .NET Framework

Once we get the *PyPy* interpreter to run on the *CLI* virtual machine, we will want to integrate it with the surrounding .NET Framwork.

As an example, these are some of the goals we might want to achieve:

- let Python code to access .NET libraries;
- let Python code to be called from the outside by other .NET languages;

- integrate Python classes with .NET classes, e.g., let Python classes to subclass the .NET ones and vice-versa;
- possibility of building stand-alone executables.

They are not easy tasks, mainly because some Python constructs are not directly supported by .NET and vice-versa: for example, .NET doesn't support multiple inheritance and runtime addition/remotion of attributes to a class, while Python doesn't support function overloading. This means that before implementing anything we would need to carefully design how the two languages integrate.

5.5. *gencli* as a .NET compiler

At the moment of writing it's not possible to use *gencli* to, say, compiling an RPython program to a *DLL* that can be easily reused by other .NET applications.

The biggest problem is that names of classes and functions are mangled to assure they are unique, so it's impossible to design a clean interface for users.

Another issue to be considered is the integration with the framework: at the moment is not possible to access system libraries, e.g. to call `System.Console.WriteLine`.

Finally, there is the same problem we saw in section 5.4: RPython and .NET semantics don't completely overlap. Fortunately in this case the problem is much easier to solve, because of the more static-ness of RPython: many constructs that could cause problem are not allowed (e.g., runtime addition/remotion of attributes to a class), but there is still some small issue that need to be addressed, such as how to expose function overloading to RPython programs.

In conclusion, there is still some work to do on *gencli* to make it a "real" .NET compiler, but it should not be so hard to get it done.

Bibliography

[1] John Gough, *Compiling for the .NET Common Language Runtime*, Prentice Hall PTR, 2001.

[2] Don Box, *Essential .NET, Volume I: The Common Language Runtime*, Addison-Wesley Professional, 2002.

[3] David Stutz, Ted Neward, Geoff Shilling, *Shared Source CLI essentials*, O'Reilly, 2003.

[4] The PyPy project. *PyPy homepage*, http://codespeak.net/pypy

[5] PyPy Project. *Translation*, http://codespeak.net/pypy/dist/pypy/doc/translation.html

[6] Guido Van Rossum. *Python Reference Manual*, http://docs.python.org/ref/ref.html

[7] The Jython Project. *Jython homepage*, http://www.jython.org

[8] IronPython. *IronPython homepage*, http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

[9] Nokia Corporation. *Python for Series 60*, http://www.forum.nokia.com/python

[10] ECMA. *Standard ECMA-335: Common Language Infrastructure*. ECMA International, June 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[11] Microsoft. *Microsoft .NET homepage*, http://www.microsoft.com/net/

[12] Mono Project. *Main Page – Mono*, http://www.mono-project.com/