RPython Documentation

Release 4.0.0

The PyPy Project

Oct 09, 2017

Contents

RPython is a translation and support framework for producing implementations of dynamic languages, emphasizing a clean separation between language specification and implementation aspects.

By separating concerns in this way, our implementation of Python - and other dynamic languages - is able to automatically generate a Just-in-Time compiler for any dynamic language. It also allows a mix-and-match approach to implementation decisions, including many that have historically been outside of a user's control, such as target platform, memory and threading models, garbage collection strategies, and optimizations applied, including whether or not to have a JIT in the first place.

CHAPTER 1

General

Goals and Architecture Overview

High Level Goals

Traditionally, language interpreters are written in a target platform language such as C/Posix, Java or C#. Each implementation provides a fundamental mapping between application source code and the target environment. One of the goals of the "all-encompassing" environments, such as the .NET framework and to some extent the Java virtual machine, is to provide standardized and higher level functionalities in order to support language implementers for writing language implementations.

PyPy is experimenting with a more ambitious approach. We are using a subset of the high-level language Python, called *RPython Language*, in which we write languages as simple interpreters with few references to and dependencies on lower level details. The RPython toolchain produces a concrete virtual machine for the platform of our choice by inserting appropriate lower level aspects. The result can be customized by selecting other feature and platform configurations.

Our goal is to provide a possible solution to the problem of language implementers: having to write `l * o * p` interpreters for `l` dynamic languages and `p` platforms with `o` crucial design decisions. PyPy aims at making it possible to change each of these variables independently such that:

- `l`: the language that we analyze can be evolved or entirely replaced;
- `o`: we can tweak and optimize the translation process to produce platform specific code based on different models and trade-offs;
- `p`: we can write new translator back-ends to target different physical and virtual platforms.

By contrast, a standardized target environment - say .NET - enforces `p=1` as far as it's concerned. This helps making `o` a bit smaller by providing a higher-level base to build upon. Still, we believe that enforcing the use of one common environment is not necessary. PyPy's goal is to give weight to this claim - at least as far as language implementation is concerned - showing an approach to the `l * o * p` problem that does not rely on standardization.

The most ambitious part of this goal is to *generate Just-In-Time Compilers* in a language-independent way, instead of

only translating the source interpreter into an interpreter for the target platform. This is an area of language implementation that is commonly considered very challenging because of the involved complexity.

Architecture

The job of the RPython toolchain is to translate *RPython Language* programs into an efficient version of that program for one of the various target platforms, generally one that is considerably lower-level than Python.

The approach we have taken is to reduce the level of abstraction of the source RPython program in several steps, from the high level down to the level of the target platform, whatever that may be. Currently we support two broad flavours of target platforms: the ones that assume a C-like memory model with structures and pointers, and the ones that assume an object-oriented model with classes, instances and methods (as, for example, the Java and .NET virtual machines do).

The RPython toolchain never sees the RPython source code or syntax trees, but rather starts with the *code objects* that define the behaviour of the function objects one gives it as input. It can be considered as "freezing" a pre-imported RPython program into an executable form suitable for the target platform.

The steps of the translation process can be summarized as follows:

- The code object of each source functions is converted to a *control flow graph* by the *flow graph builder*.
- The control flow graphs are processed by the *Annotator*, which performs whole-program type inference to annotate each variable of the control flow graph with the types it may take at run-time.
- The information provided by the annotator is used by the *RTyper* to convert the high level operations of the control flow graphs into operations closer to the abstraction level of the target platform.
- Optionally, *various transformations <optional-transformations>* can then be applied which, for example, perform optimizations such as inlining, add capabilities such as stackless-style concurrency, or insert code for the *garbage collector*.
- Then, the graphs are converted to source code for the target platform and compiled into an executable.

This process is described in much more detail in the *document about the RPython toolchain* and in the paper Compiling dynamic language implementations.

Further reading

- *Getting Started with RPython*: a hands-on guide to getting involved with the PyPy source code.
- PyPy's approach to virtual machine construction: a paper presented to the Dynamic Languages Symposium attached to OOPSLA 2006.
- *The translation document*: a detailed description of our translation process.
- *JIT Generation in PyPy*, describing how we produce a Just-in-time Compiler from an interpreter.
- A tutorial of how to use the *RPython toolchain* to implement your own interpreter.

Frequently Asked Questions

Contents

See also: Frequently ask questions about PyPy.

What is RPython?

RPython is a framework for implementing interpreters and virtual machines for programming languages, especially dynamic languages.

Can RPython compile normal Python programs to C?

No, RPython is not a Python compiler.

In Python, it is mostly impossible to *prove* anything about the types that a program will manipulate by doing a static analysis. It should be clear if you are familiar with Python, but if in doubt see *[BRETT]*.

If you want a fast Python program, please use the PyPy *JIT* instead.

What is this RPython language?

RPython is a restricted subset of the Python language. It is used for implementing dynamic language interpreters within the PyPy toolchain. The restrictions ensure that type inference (and so, ultimately, translation to other languages) of RPython programs is possible.

The property of "being RPython" always applies to a full program, not to single functions or modules (the translation toolchain does a full program analysis). The translation toolchain follows all calls recursively and discovers what belongs to the program and what does not.

RPython program restrictions mostly limit the ability to mix types in arbitrary ways. RPython does not allow the binding of two different types in the same variable. In this respect (and in some others) it feels a bit like Java. Other features not allowed in RPython are the use of special methods (`__xxx__`) except `__init__` and `__del__`, and the use of reflection capabilities (e.g. `__dict__`).

You cannot use most existing standard library modules from RPython. The exceptions are some functions in `os`, `math` and `time` that have native support.

To read more about the RPython limitations read the *RPython description*.

Does RPython have anything to do with Zope's Restricted Python?

No. Zope's RestrictedPython aims to provide a sandboxed execution environment for CPython. *PyPy's RPython* is the implementation language for dynamic language interpreters. However, PyPy also provides a robust sandboxed Python Interpreter.

What's the `"NOT_RPYTHON"` I see in some docstrings?

If you put "NOT_RPYTHON" into the docstring of a function and that function is found while trying to translate an RPython program, the translation process stops and reports this as an error. You can therefore mark functions as "NOT_RPYTHON" to make sure that they are never analyzed.

This method of marking a function as not RPython is outdated. For new code, please use the `@objectmodel.not_rpython` decorator instead.

Couldn't we simply take a Python syntax tree and turn it into Lisp?

It's not necessarily nonsense, but it's not really The PyPy Way. It's pretty hard, without some kind of type inference, to translate this Python:

```
a + b
```

into anything significantly more efficient than this Common Lisp:

```
(py:add a b)
```

And making type inference possible is what RPython is all about.

You could make `#'py:add` a generic function and see if a given CLOS implementation is fast enough to give a useful speed (but I think the coercion rules would probably drive you insane first). – mwh

Do I have to rewrite my programs in RPython?

No, and you shouldn't try. First and foremost, RPython is a language designed for writing interpreters. It is a restricted subset of Python. If your program is not an interpreter but tries to do "real things", like use *any* part of the standard Python library or *any* 3rd-party library, then it is not RPython to start with. You should only look at RPython if you try to *write your own interpreter*. If your goal is to speed up Python code, then look at the regular PyPy, which is a full and compliant Python 2.7 interpreter (which happens to be written in RPython). Not only is it not necessary for you to rewrite your code in RPython, it might not give you any speed improvements even if you manage to.

Yes, it is possible with enough effort to compile small self-contained pieces of RPython code doing a few performance-sensitive things. But this case is not interesting for us. If you needed to rewrite the code in RPython, you could as well have rewritten it in C or C++ or Java for example. These are much more supported, much more documented languages *:-)*

> *The above paragraphs are not the whole truth. It* is *true that there are cases where writing a program as RPython gives you substantially better speed than running it on top of PyPy. However, the attitude of the core group of people behind PyPy is to answer: "then report it as a performance bug against PyPy!".*

Here is a more diluted way to put it. The "No, don't!" above is a general warning we give to new people. They are likely to need a lot of help from some *source, because RPython is not so simple nor extensively documented; but at the same time, we, the pypy core group of people, are not willing to invest time in supporting 3rd-party projects that do very different things than interpreters for dynamic languages — just because we have other interests and there are only so many hours a day. So as a summary I believe it is only fair to attempt to point newcomers at existing alternatives, which are more mainstream and where they will get help from many people.*

If anybody seriously wants to promote RPython anyway, they are welcome to: we won't actively resist such a plan. There are a lot of things that could be done to make RPython a better Java-ish language for example, starting with supporting non-GIL-based multithreading, but we don't implement them because they have little relevance to us. This is open source, which means that anybody is free to promote and develop anything; but it also means that you must let us choose not *to go into that direction ourselves.*

Which backends are there for the RPython toolchain?

Currently, the only backend is *C*. It can translate the entire PyPy interpreter. To learn more about backends take a look at the *translation document*.

Could we use LLVM?

In theory yes. But we tried to use it 5 or 6 times already, as a translation backend or as a JIT backend — and failed each time.

In more details: using LLVM as a (static) translation backend is pointless nowadays because you can generate C code and compile it with clang. (Note that compiling PyPy with clang gives a result that is not faster than compiling it with gcc.) We might in theory get extra benefits from LLVM's GC integration, but this requires more work on the LLVM side before it would be remotely useful. Anyway, it could be interfaced via a custom primitive in the C code. (The latest such experimental backend is in the branch `llvm-translation-backend`, which can translate PyPy with or without the JIT on Linux.)

On the other hand, using LLVM as our JIT backend looks interesting as well — but again we made an attempt, and it failed: LLVM has no way to patch the generated machine code, and is not suited at all to tracing JITs. Even one big method JIT trying to use LLVM has given up for similar reasons; read that blog post for more details.

So the position of the core PyPy developers is that if anyone wants to make an N+1'th attempt with LLVM, they are welcome, and they will receive a bit of help on the IRC channel, but they are left with the burden of proof that it works.

Compiling PyPy swaps or runs out of memory

This is documented (here and here). It needs 4 GB of RAM to run "rpython targetpypystandalone" on top of PyPy, a bit more when running on top of CPython. If you have less than 4 GB free, it will just swap forever (or fail if you don't have enough swap). And we mean *free:* if the machine has 4 GB *in total,* then it will swap.

On 32-bit, divide the numbers by two. (We didn't try recently, but in the past it was possible to compile a 32-bit version on a 2 GB Linux machine with nothing else running: no Gnome/KDE, for example.)

How do I compile my own interpreters?

Begin by reading Andrew Brown's tutorial .

Can RPython modules for PyPy be translated independently?

No, you have to rebuild the entire interpreter. This means two things:

- It is imperative to use test-driven development. You have to exhaustively test your module in pure Python, before even attempting to translate it. Once you translate it, you should have only a few typing issues left to fix, but otherwise the result should work out of the box.
- Second, and perhaps most important: do you have a really good reason for writing the module in RPython in the first place? Nowadays you should really look at alternatives, like writing it in pure Python, using cffi if it needs to call C code.

In this context it is not that important to be able to translate RPython modules independently of translating the complete interpreter. (It could be done given enough efforts, but it's a really serious undertaking. Consider it as quite unlikely for now.)

Why does the translator draw a Mandelbrot fractal while translating?

Because it's fun.

CHAPTER 2

User Documentation

These documents are mainly interesting for users of interpreters written in RPython.

Cross-translating for ARM

Here we describe the setup required and the steps needed to follow to translate an interpreter using the RPython translator to target ARM using a cross compilation toolchain.

To translate an RPython program for ARM we can either translate directly on an ARM device following the normal translation steps. Unfortunately this is not really feasible on most ARM machines. The alternative is to cross-translate using a cross-compilation toolchain.

To cross-translate we run the translation on a more powerful (usually x86) machine and generate a binary for ARM using a cross-compiler to compile the generated C code. There are several constraints when doing this. In particular we currently only support Linux as translation host and target platforms (tested on Ubuntu). Also we need a 32-bit environment to run the translation. This can be done either on a 32bit host or in 32bit chroot.

Requirements

The tools required to cross translate from a Linux based host to an ARM based Linux target are:

- A checkout of PyPy (default branch).
- The GCC ARM cross compiler (on Ubuntu it is the `gcc-arm-linux-gnueabi package`) but other toolchains should also work.
- Scratchbox 2, a cross-compilation engine (`scratchbox2` Ubuntu package).
- A 32-bit PyPy or Python.
- And the following (or corresponding) packages need to be installed to create an ARM based chroot:
  - `debootstrap`
  - `schroot`

  - binfmt-support
  - qemu-system
  - qemu-user-static
- The dependencies above are in addition to the ones needed for a regular translation, listed here.

Creating a Qemu based ARM chroot

First we will need to create a rootfs containing the packages and dependencies required in order to translate PyPy or other interpreters. We are going to assume, that the files will be placed in `/srv/chroot/precise_arm`.

Create the rootfs by calling:

```
mkdir -p /srv/chroot/precise_arm
qemu-debootstrap --variant=buildd --arch=armel precise /srv/chroot/precise_arm/ ␣
↪http://ports.ubuntu.com/ubuntu-ports/
```

Next, copy the qemu-arm-static binary to the rootfs.

```
cp /usr/bin/qemu-arm-static /srv/chroot/precise_arm/usr/bin/qemu-arm-static
```

For easier configuration and management we will create a schroot pointing to the rootfs. We need to add a configuration block (like the one below) to the schroot configuration file in /etc/schroot/schroot.conf.

```
[precise_arm]
directory=/srv/chroot/precise_arm
users=USERNAME
root-users=USERNAME
groups=users
aliases=default
type=directory
```

To verify that everything is working in the chroot, running `schroot -c precise_arm` should start a shell running in the schroot environment using qemu-arm to execute the ARM binaries. Running `uname -m` in the chroot should yield a result like `armv7l`. Showing that we are emulating an ARM system.

Start the schroot as the user root in order to configure the apt sources and to install the following packages:

```
schroot -c precise_arm -u root
echo "deb http://ports.ubuntu.com/ubuntu-ports/ precise main universe restricted" > /
↪etc/apt/sources.list
apt-get update
apt-get install libffi-dev libgc-dev python-dev build-essential libncurses5-dev␣
↪libbz2-dev
```

Now all dependencies should be in place and we can exit the schroot environment.

Configuring scratchbox2

To configure the scratchbox we need to cd into the root directory of the rootfs we created before. From there we can call the sb2 configuration tools which will take the current directory as the base directory for the scratchbox2 environment.

```
cd /srv/chroot/precise_arm
sb2-init -c `which qemu-arm` ARM `which arm-linux-gnueabi-gcc`
```

This will create a scratchbox2 based environment called ARM that maps calls to gcc done within the scratchbox to the arm-linux-gnueabi-gcc outside the scratchbox. Now we should have a working cross compilation toolchain in place and can start cross-translating programs for ARM.

Translation

Having performed all the preliminary steps we should now be able to cross translate a program for ARM. You can use this minimal target to test your setup before applying it to a larger project.

Before starting the translator we need to set two environment variables, so the translator knows how to use the scratchbox environment. We need to set the **SB2** environment variable to point to the rootfs and the **SB2OPT** should contain the command line options for the sb2 command. If our rootfs is in the folder /srv/chroot/precise_arm and the scratchbox environment is called "ARM", the variables would be defined as follows.

```
export SB2=/srv/chroot/precise_arm
export SB2OPT='-t ARM'
```

Once this is set, you can call the translator. For example save this file

```
def main(args):
    print "Hello World"
    return 0

def target(*args):
    return main, None
```

and call the translator

```
pypy ~/path_to_pypy_checkout/rpython/bin/rpython -O2 --platform=arm target.py
```

If everything worked correctly this should yield an ARM binary. Running this binary in the ARM chroot or on an ARM device should produce the output `"Hello World"`.

To translate the full python pypy interpreter with a jit, you can cd into pypy/goal and call

```
pypy <path to rpython>/rpython/bin/rpython -Ojit --platform=arm --
↪gcrootfinder=shadowstack --jit-backend=arm targetpypystandalone.py
```

The gcrootfinder option is needed to work around issue 1377 and the jit-backend works around issue 1376

Logging environment variables

PyPy, and all other RPython programs, support some special environment variables used to tweak various advanced parameters.

Garbage collector

Right now the default GC is (an incremental version of) MiniMark. It has a number of environment variables that can be tweaked. Their default value should be ok for most usages.

PYPYLOG

The `PYPYLOG` environment variable enables debugging output. For example:

```
PYPYLOG=jit:log
```

means enabling all debugging output from the JIT, and writing to a file called `log`. More precisely, the condition `jit` means enabling output of all sections whose name start with `jit`; other interesting names to use here are `gc` to get output from the GC, or `jit-backend` to get only output from the JIT's machine code backend. It is possible to use several prefixes, like in the following example:

```
PYPYLOG=jit-log-opt,jit-backend:log
```

which outputs sections containing to the optimized loops plus anything produced from the JIT backend. The above example is what you need for jitviewer.

The filename can be given as `-` to dump the log to stderr.

As a special case, the value `PYPYLOG=+filename` means that only the section markers are written (for any section). This is mostly only useful for `rpython/tool/logparser.py`.

PYPYSTM

Only available in `pypy-stm`. Names a log file into which the PyPy-STM will output contention information. Can be read with `pypy/stm/print_stm_log.py`.

CHAPTER 3

Writing your own interpreter in RPython

RPython Language

Definition

RPython is a restricted subset of Python that is amenable to static analysis. Although there are additions to the language and some things might surprisingly work, this is a rough list of restrictions that should be considered. Note that there are tons of special cased restrictions that you'll encounter as you go. The exact definition is "RPython is everything that our translation toolchain can accept" :)

Flow restrictions

variables

variables should contain values of at most one type as described in *Object restrictions* at each control flow point, that means for example that joining control paths using the same variable to contain both a string and a int must be avoided. It is allowed to mix None (basically with the role of a null pointer) with many other types: wrapped objects, class instances, lists, dicts, strings, etc. but *not* with int, floats or tuples.

constants

all module globals are considered constants. Their binding must not be changed at run-time. Moreover, global (i.e. prebuilt) lists and dictionaries are supposed to be immutable: modifying e.g. a global list will give inconsistent results. However, global instances don't have this restriction, so if you need mutable global state, store it in the attributes of some prebuilt singleton instance.

control structures

all allowed, `for` loops restricted to builtin types, generators very restricted.

range

`range` and `xrange` are identical. `range` does not necessarily create an array, only if the result is modified. It is allowed everywhere and completely implemented. The only visible difference to CPython is the inaccessibility of the `xrange` fields start, stop and step.

definitions

run-time definition of classes or functions is not allowed.

generators

generators are supported, but their exact scope is very limited. you can't merge two different generator in one control point.

exceptions

fully supported. see below *Exception rules* for restrictions on exceptions raised by built-in operations

Object restrictions

We are using

integer, float, boolean

works.

strings

a lot of, but not all string methods are supported and those that are supported, not necesarilly accept all arguments. Indexes can be negative. In case they are not, then you get slightly more efficient code if the translator can prove that they are non-negative. When slicing a string it is necessary to prove that the slice start and stop indexes are non-negative. There is no implicit str-to-unicode cast anywhere. Simple string formatting using the `%` operator works, as long as the format string is known at translation time; the only supported formatting specifiers are `%s`, `%d`, `%x`, `%o`, `%f`, plus `%r` but only for user-defined instances. Modifiers such as conversion flags, precision, length etc. are not supported. Moreover, it is forbidden to mix unicode and strings when formatting.

tuples

no variable-length tuples; use them to store or return pairs or n-tuples of values. Each combination of types for elements and length constitute a separate and not mixable type.

There is no general way to convert a list into a tuple, because the length of the result would not be known statically. (You can of course do `t = (lst[0], lst[1], lst[2])` if you know that `lst` has got 3 items.)

lists

lists are used as an allocated array. Lists are over-allocated, so list.append() is reasonably fast. However, if you use a fixed-size list, the code is more efficient. Annotator can figure out most of the time that your list is fixed-size, even when you use list comprehension. Negative or out-of-bound indexes are only allowed for the most common operations, as follows:

- *indexing*: positive and negative indexes are allowed. Indexes are checked when requested by an IndexError exception clause.
- *slicing*: the slice start must be within bounds. The stop doesn't need to, but it must not be smaller than the start. All negative indexes are disallowed, except for the [:-1] special case. No step. Slice deletion follows the same rules.
- *slice assignment*: only supports `lst[x:y] = sublist`, if `len(sublist) == y - x`. In other words, slice assignment cannot change the total length of the list, but just replace items.
- *other operators*: `+`, `+=`, `in`, `*`, `*=`, `==`, `!=` work as expected.
- *methods*: append, index, insert, extend, reverse, pop. The index used in pop() follows the same rules as for *indexing* above. The index used in insert() must be within bounds and not negative.

dicts

dicts with a unique key type only, provided it is hashable. Custom hash functions and custom equality will not be honored. Use `rpython.rlib.objectmodel.r_dict` for custom hash functions.

sets

sets are not directly supported in RPython. Instead you should use a plain dict and fill the values with None. Values in that dict will not consume space.

list comprehensions

May be used to create allocated, initialized arrays.

functions

- function declarations may use defaults and `*args`, but not `**keywords`.
- function calls may be done to a known function or to a variable one, or to a method. You can call with positional and keyword arguments, and you can pass a `*args` argument (it must be a tuple).
- as explained above, tuples are not of a variable length. If you need to call a function with a dynamic number of arguments, refactor the function itself to accept a single argument which is a regular list.
- dynamic dispatch enforces the use of signatures that are equal for all possible called function, or at least "compatible enough". This concerns mainly method calls, when the method is overridden or in any way given different definitions in different classes. It also concerns the less common case of explicitly manipulated function objects. Describing the exact compatibility rules is rather involved (but if you break them, you should get explicit errors from the rtyper and not obscure crashes.)

builtin functions

A number of builtin functions can be used. The precise set can be found in rpython/annotator/builtin.py (see `def builtin_xxx()`). Some builtin functions may be limited in what they support, though.

`int, float, str, ord, chr`... are available as simple conversion functions. Note that `int, float, str`... have a special meaning as a type inside of isinstance only.

classes

- methods and other class attributes do not change after startup
- single inheritance is fully supported
- use *rpython.rlib.objectmodel.import_from_mixin(M)* in a class body to copy the whole content of a class *M*. This can be used to implement mixins: functions and staticmethods are duplicated (the other class attributes are just copied unmodified).
- classes are first-class objects too

objects

Normal rules apply. The only special methods that are honoured are `__init__`, `__del__`, `__len__`, `__getitem__`, `__setitem__`, `__getslice__`, `__setslice__`, and `__iter__`. To handle slicing, `__getslice__` and `__setslice__` must be used; using `__getitem__` and `__setitem__` for slicing isn't supported. Additionally, using negative indices for slicing is still not support, even when using `__getslice__`.

Note that the destructor `__del__` should only contain simple operations; for any kind of more complex destructor, consider using instead `rpython.rlib.rgc.FinalizerQueue`.

This layout makes the number of types to take care about quite limited.

Integer Types

While implementing the integer type, we stumbled over the problem that integers are quite in flux in CPython right now. Starting with Python 2.4, integers mutate into longs on overflow. In contrast, we need a way to perform wrap-around machine-sized arithmetic by default, while still being able to check for overflow when we need it explicitly. Moreover, we need a consistent behavior before and after translation.

We use normal integers for signed arithmetic. It means that before translation we get longs in case of overflow, and after translation we get a silent wrap-around. Whenever we need more control, we use the following helpers (which live in rpython/rlib/rarithmetic.py):

ovfcheck()

> This special function should only be used with a single arithmetic operation as its argument, e.g. `z = ovfcheck(x+y)`. Its intended meaning is to perform the given operation in overflow-checking mode.
>
> At run-time, in Python, the ovfcheck() function itself checks the result and raises OverflowError if it is a `long`. But the code generators use ovfcheck() as a hint: they replace the whole `ovfcheck(x+y)` expression with a single overflow-checking addition in C.

intmask()

> This function is used for wrap-around arithmetic. It returns the lower bits of its argument, masking away anything that doesn't fit in a C "signed long int". Its purpose is, in Python, to convert from a Python `long` that resulted from a previous operation back to a Python `int`. The code generators ignore intmask() entirely, as they are doing wrap-around signed arithmetic all the time by default anyway. (We have no equivalent of the "int" versus "long int" distinction of C at the moment and assume "long ints" everywhere.)

r_uint

> In a few cases (e.g. hash table manipulation), we need machine-sized unsigned arithmetic. For these cases there is the r_uint class, which is a pure Python implementation of word-sized unsigned integers that silently wrap around. ("word-sized" and "machine-sized" are used equivalently and mean the native size, which you get using "unsigned long" in C.) The purpose of this class (as opposed to helper functions as above) is consistent typing: both Python and the annotator will propagate r_uint instances in the program and interpret all the operations between them as unsigned. Instances of r_uint are special-cased by the code generators to use the appropriate low-level type and operations. Mixing of (signed) integers and r_uint in operations produces r_uint that means unsigned results. To convert back from r_uint to signed integers, use intmask().

Exception rules

Exceptions are by default not generated for simple cases.:

```
#!/usr/bin/python

    lst = [1,2,3,4,5]
    item = lst[i]    # this code is not checked for out-of-bound access

    try:
        item = lst[i]
    except IndexError:
        # complain
```

Code with no exception handlers does not raise exceptions (after it has been translated, that is. When you run it on top of CPython, it may raise exceptions, of course). By supplying an exception handler, you ask for error checking.

Without, you assure the system that the operation cannot fail. This rule does not apply to *function calls*: any called function is assumed to be allowed to raise any exception.

For example:

```
x = 5.1
x = x + 1.2       # not checked for float overflow
try:
    x = x + 1.2
except OverflowError:
    # float result too big
```

But:

```
z = some_function(x, y)    # can raise any exception
try:
    z = some_other_function(x, y)
except IndexError:
    # only catches explicitly-raised IndexErrors in some_other_function()
    # other exceptions can be raised, too, and will not be caught here.
```

The ovfcheck() function described above follows the same rule: in case of overflow, it explicitly raise OverflowError, which can be caught anywhere.

Exceptions explicitly raised or re-raised will always be generated.

PyPy is debuggable on top of CPython

PyPy has the advantage that it is runnable on standard CPython. That means, we can run all of PyPy with all exception handling enabled, so we might catch cases where we failed to adhere to our implicit assertions.

Generally Useful RPython Modules

Contents

This page lists some of the modules in rpython/rlib together with some hints for what they can be used for. The modules here will make up some general library useful for RPython programs (since most of the standard library modules are not RPython). Most of these modules are somewhat rough still and are likely to change at some point. Usually it is useful to look at the tests in rpython/rlib/test to get an impression of how to use a module.

listsort

The rpython/rlib/listsort.py module contains an implementation of the timsort sorting algorithm (the sort method of lists is not RPython). To use it, make (globally) one class by calling `MySort = listsort.make_timsort_class(lt=my_comparison_func)`. There are also other optional arguments, but usually you give with `lt=...` a function that compares two objects from your lists. You need one class per "type" of list and per comparison function.

The constructor of `MySort` takes a list as an argument, which will be sorted in place when the `sort` method of the `MySort` instance is called.

nonconst

The rpython/rlib/nonconst.py module is useful mostly for tests. The *flow graph builder* and the *annotator* do quite some constant folding, which is sometimes not desired in a test. To prevent constant folding on a certain value, use the `NonConst` class. The constructor of `NonConst` takes an arbitrary value. The instance of `NonConst` will behave during annotation like that value, but no constant folding will happen.

objectmodel

The rpython/rlib/objectmodel.py module is a mixed bag of various functionality. Some of the more useful ones are:

`ComputedIntSymbolic`: Instances of `ComputedIntSymbolic` are treated like integers of unknown value by the annotator. The value is determined by a no-argument function (which needs to be passed into the constructor of the class). When the backend emits code, the function is called to determine the value.

`CDefinedIntSymbolic`: Instances of `CDefinedIntSymbolic` are also treated like integers of unknown value by the annotator. When C code is emitted they will be represented by the attribute `expr` of the symbolic (which is also the first argument of the constructor).

r_dict: An RPython dict-like object. The constructor of r_dict takes two functions: `key_eq` and `key_hash` which are used for comparing and hashing the entries in the dictionary.

instantiate(cls): Instantiate class `cls` without calling `__init__`.

we_are_translated(): This function returns `False` when run on top of CPython, but the annotator thinks its return value is `True`. Therefore it can be used to do different things on top of CPython than after translation. This should be used extremely sparingly (mostly for optimizations or debug code).

cast_object_to_weakaddress(obj): Returns a sort of "weak reference" to obj, just without any convenience. The weak address that it returns is not invalidated if the object dies, so you need to take care yourself to know when the object dies. Use with extreme care.

cast_weakadress_to_object(obj): Inverse of the previous function. If the object died then a segfault will ensue.

UnboxedValue: This is a class which should be used as a base class for a class which carries exactly one integer field. The class should have `__slots__` with exactly one entry defined. After translation, instances of this class won't be allocated but represented by *tagged pointers**, that is pointers that have the lowest bit set.

rarithmetic

The rpython/rlib/rarithmetic.py module contains functionality to handle the small differences in the behaviour of arithmetic code in regular Python and RPython code. Most of them are already described in the *RPython description*.

rbigint

The rpython/rlib/rbigint.py module contains a full RPython implementation of the Python `long` type (which itself is not supported in RPython). The `rbigint` class contains that implementation. To construct `rbigint` instances use the static methods `fromint`, `frombool`, `fromfloat` and `fromdecimalstr`. To convert back to other types use the methods `toint`, `tobool`, `touint` and `tofloat`. Since RPython does not support operator overloading, all the special methods of `rbigint` that would normally start and end with "__" have these underscores left out for better readability (so `a.add(b)` can be used to add two rbigint instances).

rrandom

The rpython/rlib/rrandom.py module contains an implementation of the mersenne twister random number generator. It contains one class `Random` which most importantly has a `random` method which returns a pseudo-random floating point number between 0.0 and 1.0.

rsocket

The rpython/rlib/rsocket.py module contains an RPython implementation of the functionality of the socket standard library with a slightly different interface. The difficulty with the Python socket API is that addresses are not "well-typed" objects: depending on the address family they are tuples, or strings, and so on, which is not suitable for RPython. Instead, `rsocket` contains a hierarchy of Address classes, in a typical static-OO-programming style.

rstrategies

The rpython/rlib/rstrategies module contains a library to implement storage strategies in RPython VMs. The library is language-independent and extensible. More details and examples can be found in the rstrategies documentation.

streamio

The rpython/rlib/streamio.py contains an RPython stream I/O implementation (which was started by Guido van Rossum as sio.py in the CPython sandbox as a prototype for the upcoming new file implementation in Python 3000).

unroll

The rpython/rlib/unroll.py module most importantly contains the function `unrolling_iterable` which wraps an iterator. Looping over the iterator in RPython code will not produce a loop in the resulting flow graph but will unroll the loop instead.

rsre

The implementation of regular expressions we use for PyPy. Note that it is hard to reuse in other languages: in Python, regular expressions are first compiled into a bytecode format by pure Python code from the standard library. This lower-level module only understands this bytecode format. Without a complete Python interpreter you can't translate the regexp syntax to the bytecode format. (There are hacks for limited use cases where you have only static regexps: they can be precompiled during translation. Alternatively, you could imagine executing a Python subprocess just to translate a regexp at runtime...)

parsing

The rpython/rlib/parsing/ module is a still in-development module to generate tokenizers and parsers in RPython. It is still highly experimental and only really used by the Prolog interpreter (although in slightly non-standard ways). The easiest way to specify a tokenizer/grammar is to write it down using regular expressions and simple EBNF format.

The regular expressions are implemented using finite automatons. The parsing engine uses packrat parsing, which has O(n) parsing time but is more powerful than LL(n) and LR(n) grammars.

Regular Expressions

The regular expression syntax is mostly a subset of the syntax of the re module. *Note: this is different from rlib.rsre.* By default, non-special characters match themselves. If you concatenate regular expressions the result will match the concatenation of strings matched by the single regular expressions.

| `R|S` matches any string that *either* matches R or matches S.

***** `R*` matches 0 or more repetitions of R.

+ `R+` matches 1 or more repetitions of R.

? `R?` matches 0 or 1 repetition of R.

(...) Parenthesis can be used to group regular expressions (note that in contrast to Python's re module you cannot later match the content of this group).

{m} `R{m}` matches exactly m repetitions of R.

{m, n} `R{m, n}` matches between m and n repetitions of R (including m and n).

[] Matches a set of characters. The characters to be matched can be listed sequentially. A range of characters can be specified using `-`. For examples `[ac-eg]` matches the characters a, c, d, e and g. The whole set can be inverted by starting it with `^`. So [^a] matches anything except a.

To parse a regular expression and to get a matcher for it, you can use the function `make_runner(s)` in the `rpython.rlib.parsing.regexparse` module. It returns a object with a `recognize(input)` method that returns True or False depending on whether `input` matches the string or not.

EBNF

To describe a tokenizer and a grammar the `rpython.rlib.parsing.ebnfparse` defines a syntax for doing that.

The syntax file contains a sequence or rules. Every rule either describes a regular expression or a grammar rule.

Regular expressions rules have the form:

```
NAME: "regex";
```

NAME is the name of the token that the regular expression produces (it has to consist of upper-case letters), `regex` is a regular expression with the syntax described above. One token name is special-cased: a token called `IGNORE` will be filtered out of the token stream before being passed on to the parser and can thus be used to match comments or non-significant whitespace.

Grammar rules have the form:

```
name: expansion_1 | expansion_2 | ... | expansion_n;
```

Where `expansion_i` is a sequence of nonterminal or token names:

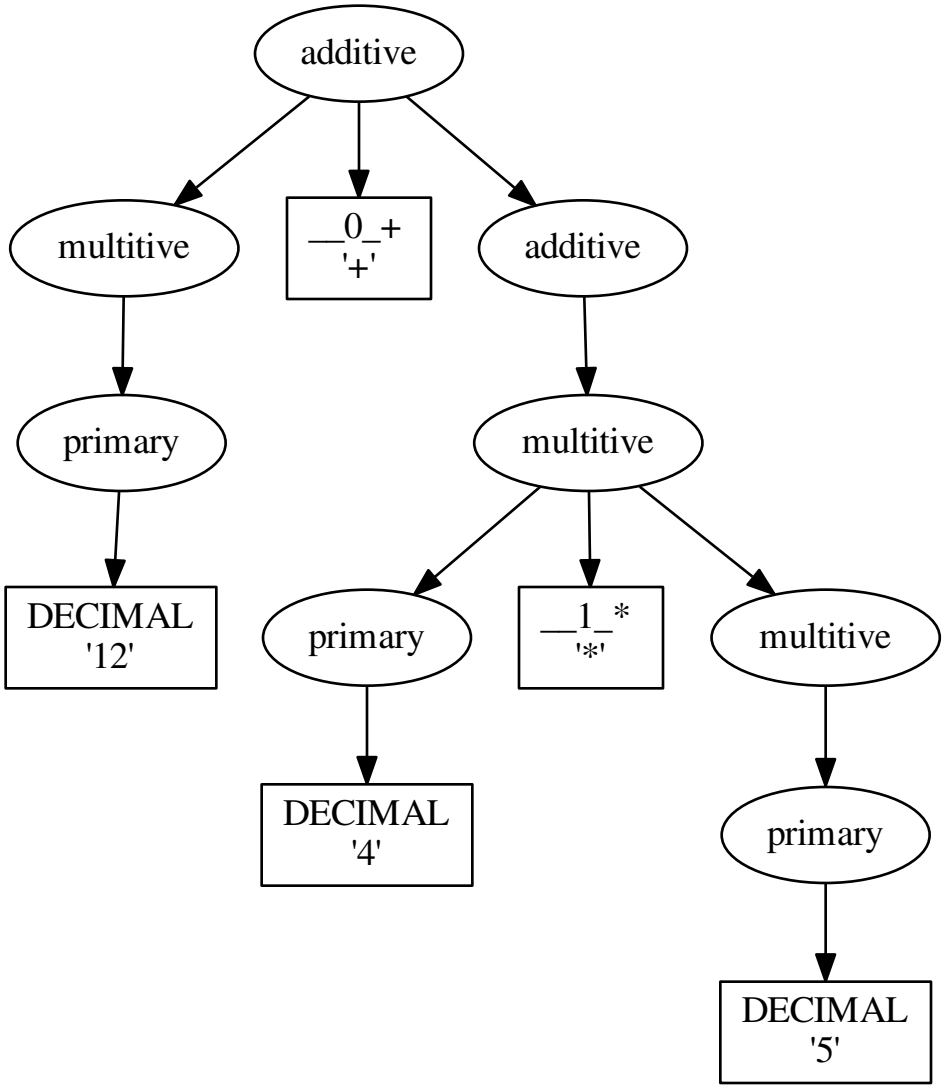
```
symbol_1 symbol_2 symbol_3 ... symbol_n
```

This means that the nonterminal symbol `name` (which has to consist of lower-case letters) can be expanded into any of the expansions. The expansions can consist of a sequence of token names, nonterminal names or literals, which are strings in quotes that are matched literally.

An example to make this clearer:

```
IGNORE: " ";
DECIMAL: "0|[1-9][0-9]*";
additive: multitive "+" additive |
          multitive;
multitive: primary "*" multitive |
           primary;
primary: "(" additive ")" | DECIMAL;
```

This grammar describes the syntax of arithmetic impressions involving addition and multiplication. The tokenizer produces a stream of either DECIMAL tokens or tokens that have matched one of the literals "+", "*", "(" or ")". Any space will be ignored. The grammar produces a syntax tree that follows the precedence of the operators. For example the expression `12 + 4 * 5` is parsed into the following tree:



Parse Trees

The parsing process builds up a tree consisting of instances of `Symbol` and `Nonterminal`, the former corresponding to tokens, the latter to nonterminal symbols. Both classes live in the rpython/rlib/parsing/tree.py module. You can use the `view()` method `Nonterminal` instances to get a pygame view of the parse tree.

`Symbol` instances have the following attributes: `symbol`, which is the name of the token and `additional_info` which is the matched source.

`Nonterminal` instances have the following attributes: `symbol` is the name of the nonterminal and `children` which is a list of the children attributes.

Visitors

To write tree visitors for the parse trees that are RPython, there is a special baseclass `RPythonVisitor` in rpython/rlib/parsing/tree.py to use. If your class uses this, it will grow a `dispatch(node)` method, that calls an appropriate `visit_<symbol>` method, depending on the `node` argument. Here the <symbol> is replaced by the `symbol` attribute of the visited node.

For the visitor to be RPython, the return values of all the visit methods need to be of the same type.

Tree Transformations

As the tree of arithmetic example above shows, by default the parse tree contains a lot of nodes that are not really conveying useful information. To get rid of some of them, there is some support in the grammar format to automatically create a visitor that transforms the tree to remove the additional nodes. The simplest such transformation just removes nodes, but there are more complex ones.

The syntax for these transformations is to enclose symbols in expansions of a nonterminal by [...], <...> or >...<.

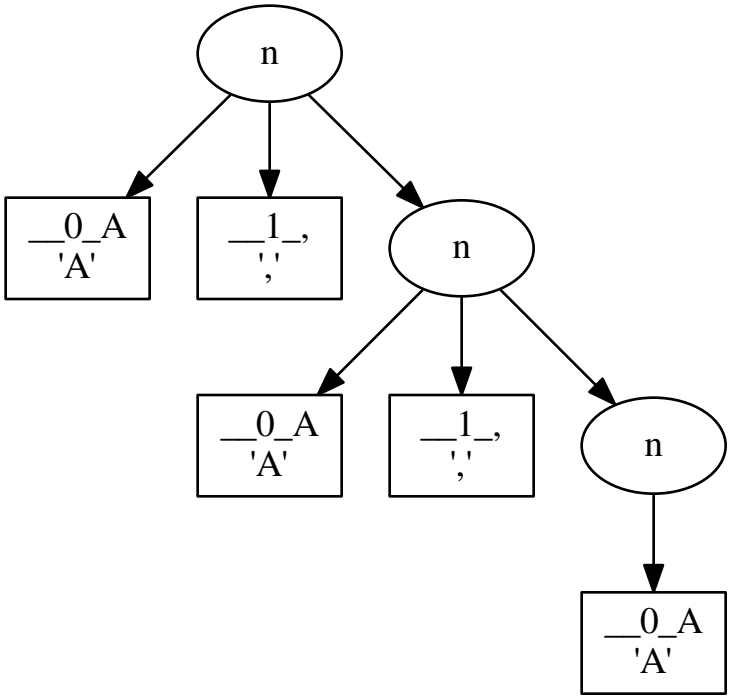
[symbol_1 symbol_2 ... symbol_n]

This will produce a transformer that completely removes the enclosed symbols from the tree.

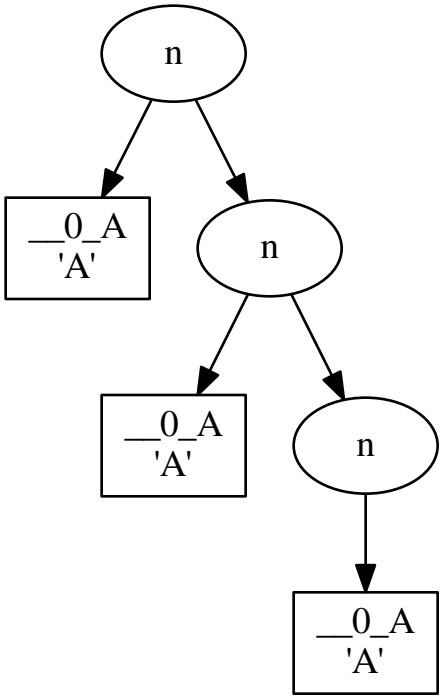
Example:

```
IGNORE: " ";
n: "A" [","] n | "A";
```

Parsing the string “A, A, A” gives the tree:



After transformation the tree has the ”,” nodes removed:



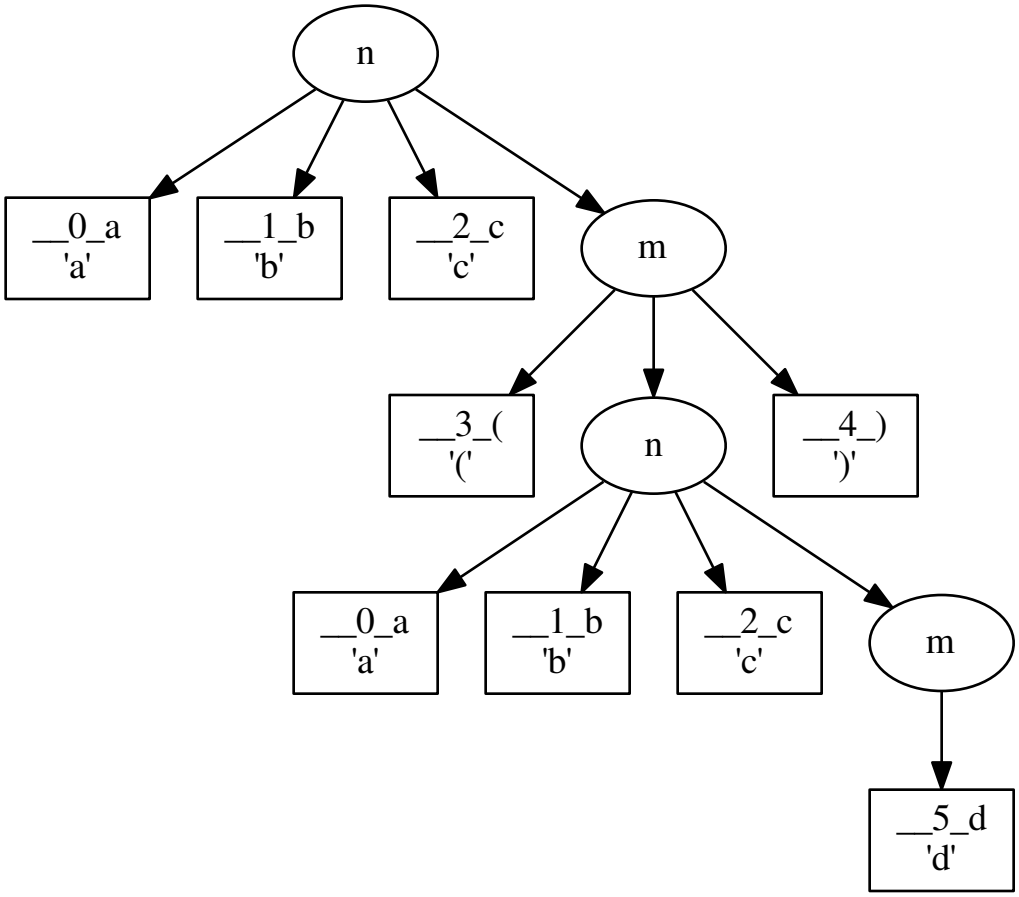
<symbol>

This will replace the parent with symbol. Every expansion can contain at most one symbol that is enclosed by <...>, because the parent can only be replaced once, obviously.

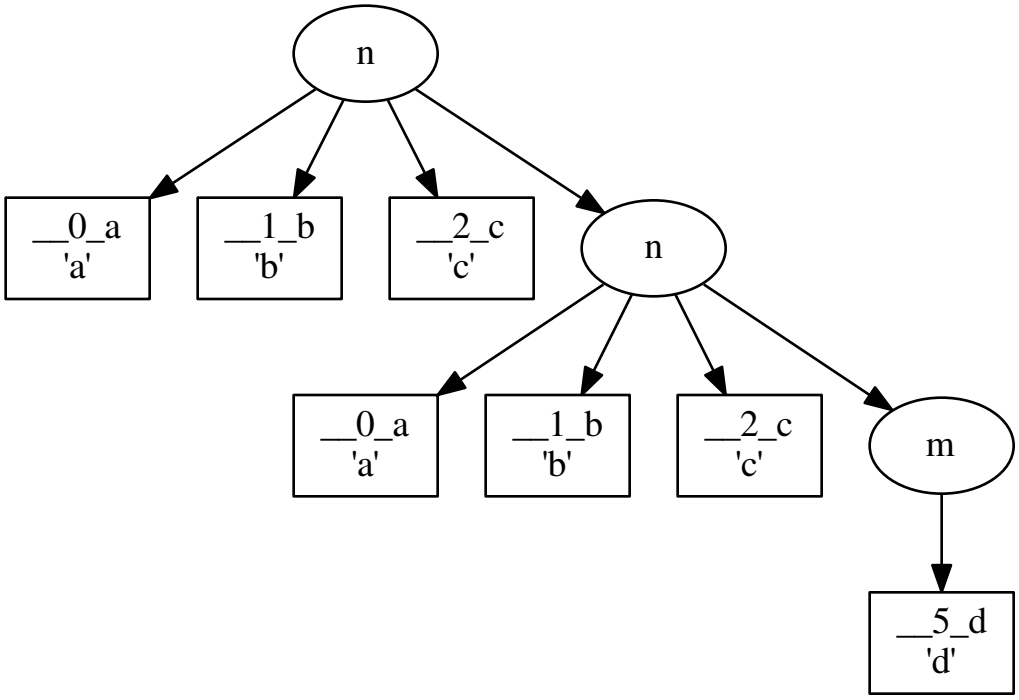
Example:

```
IGNORE: " ";
n: "a" "b" "c" m;
m: "(" <n> ")" | "d";
```

Parsing the string “a b c (a b c d)” gives the tree:



After transformation the tree looks like this:



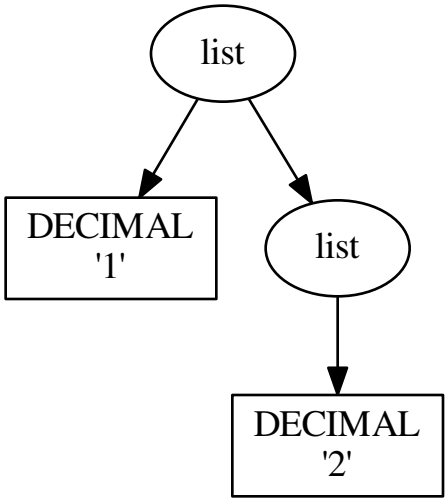
>nonterminal_1 nonterminal_2 ... nonterminal_n<

This replaces the nodes nonterminal_1 to nonterminal_n by their children.

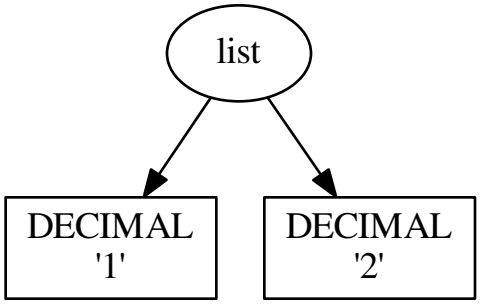
Example:

```
IGNORE: " ";
DECIMAL: "0|[1-9][0-9]*";
list: DECIMAL >list< | DECIMAL;
```

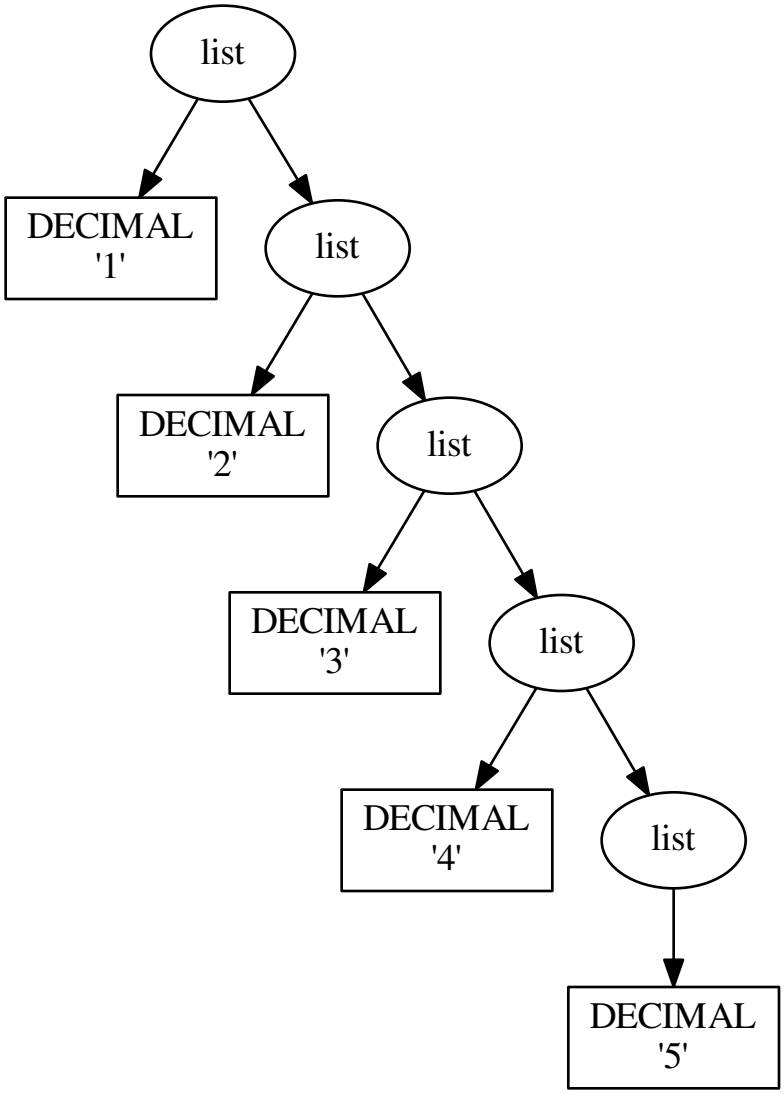
Parsing the string “1 2” gives the tree:



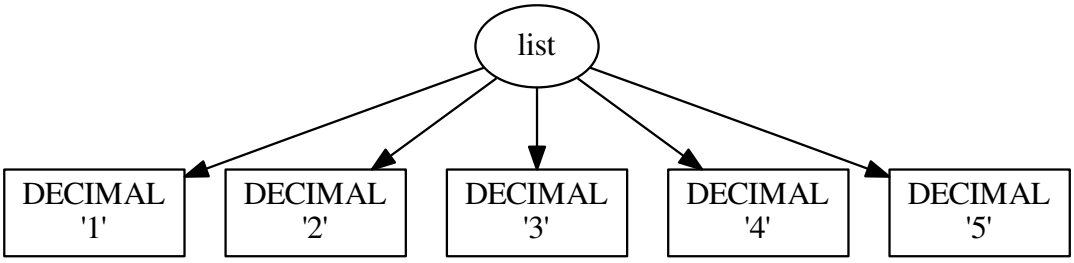
after the transformation the tree looks like:



Note that the transformation works recursively. That means that the following also works: if the string “1 2 3 4 5” is parsed the tree at first looks like this:



But after transformation the whole thing collapses to one node with a lot of children:



Extensions to the EBNF grammar format

There are some extensions to the EBNF grammar format that are really only syntactic sugar but make writing grammars less tedious. These are:

`symbol?`: matches 0 or 1 repetitions of symbol

`symbol*`: matches 0 or more repetitions of symbol. After the tree transformation all these repetitions are children of the current symbol.

`symbol+`: matches 1 or more repetitions of symbol. After the tree transformation all these repetitions are children of the current symbol.

These are implemented by adding some more rules to the grammar in the correct way. Examples: the grammar:

```
s: a b? c;
```

is transformed to look like this:

```
s: a >_maybe_symbol_0_< c | a c;
_maybe_symbol_0_: b;
```

The grammar:

```
s: a b* c;
```

is transformed to look like this:

```
s: a >_star_symbol_0< c | a c;
_star_symbol_0: b >_symbol_star_0< | b;
```

The grammar:

```
s: a b+ c;
```

is transformed to look like this:

```
s: a >_plus_symbol_0< c;
_plus_symbol_0: b >_plus_symbol_0< | b;
```

Full Example

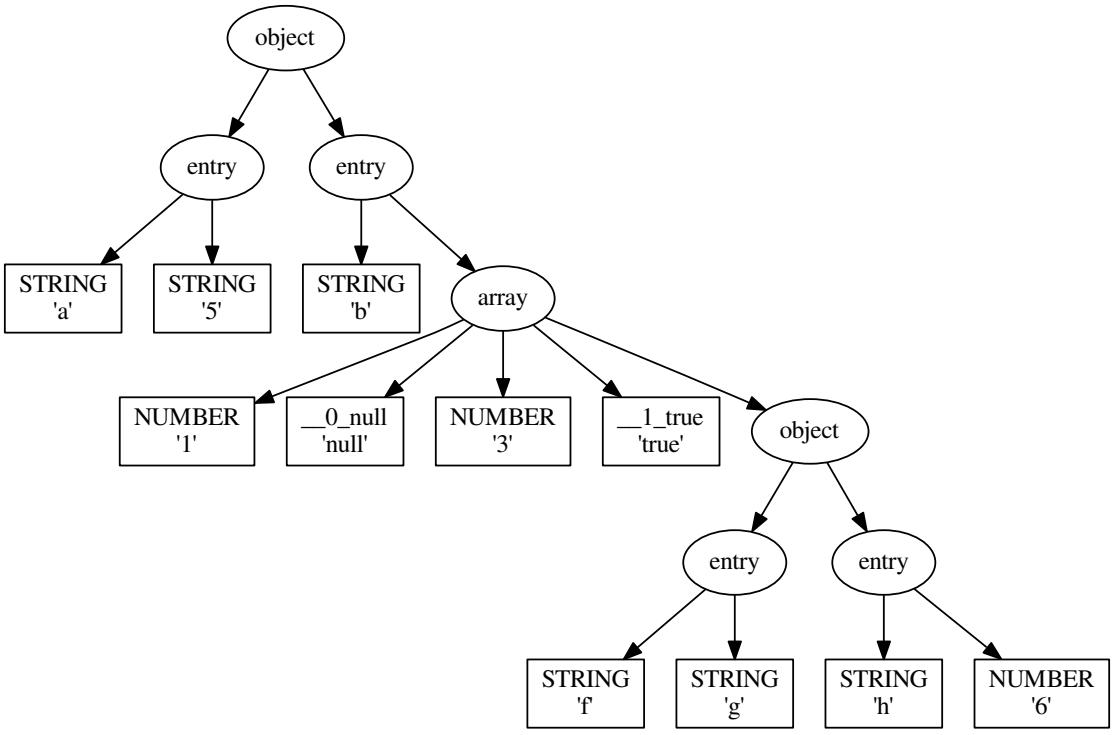
A semi-complete parser for the json format:

```
STRING: "\\"[^\\\\"]*\\"";
NUMBER: "\-?(0|[1-9][0-9]*)(\.[0-9]+)?([eE][\+\-]?[0-9]+)?";
IGNORE: " |\n";
value: <STRING> | <NUMBER> | <object> | <array> | <"null"> |
       <"true"> | <"false">;
object: ["{"] (entry [","])* entry ["}"];
array: ["["] (value [","])* value ["]"];
entry: STRING [":"] value;
```

The resulting tree for parsing the string:

```
{"a": "5", "b": [1, null, 3, true, {"f": "g", "h": 6}]}
```

looks like this:



Foreign Function Interface for RPython

Purpose

This document describes an FFI for the RPython language, concentrating on low-level backends like C. It describes how to declare and call low-level (C) functions from RPython level.

Declaring low-level external function

Declaring external C function in RPython is easy, but one needs to remember that low level functions eat *low level types* (like lltype.Signed or lltype.Array) and memory management must be done by hand. To declare a function, we write:

```
from rpython.rtyper.lltypesystem import rffi

external_function = rffi.llexternal(name, args, result)
```

where:

- name - a C-level name of a function (how it would be rendered)
- args - low level types of args
- result - low level type of a result

You can pass in additional information about C-level includes, libraries and sources by passing in the optional `compilation_info` parameter:

```
from rpython.rtyper.lltypesystem import rffi
from rpython.translator.tool.cbuild import ExternalCompilationInfo

info = ExternalCompilationInfo(includes=[], libraries=[])

external_function = rffi.llexternal(
  name, args, result, compilation_info=info
  )
```

See cbuild for more info on ExternalCompilationInfo.

Types

In rffi there are various declared types for C-structures, like CCHARP (char*), SIZE_T (size_t) and others. Refer to file for details. Instances of non-primitive types must be alloced by hand, with call to lltype.malloc, and freed by lltype.free both with keyword argument flavor='raw'. There are several helpers like string -> char* converter, refer to the source for details.

Registering function as external

Once we provided low-level implementation of an external function, would be nice to wrap call to some library function (like os.open) with such a call. For this, there is a *register_external* routine, located in extfunc.py, which provides nice API for declaring such a functions, passing llimpl as an argument and eventually llfakeimpl as a fake low-level implementation for tests performed by an llinterp.

Projects Using RPython

A very time-dependent list of interpreters written in RPython. Corrections welcome, this list was last curated in Nov 2016

Actively Developed:

- PyPy, Python2 and Python3, very complete and maintained, http://pypy.org

- Pydgin, CPU emulation framework, supports ARM well, jitted, active development, https://github.com/cornell-brg/pydgin
- RSqueak VM, Smalltalk, core complete, JIT working, graphics etc getting there, in active development https://github.com/HPI-SWA-Lab/RSqueak
- Pixie, 'A small, fast, native lisp with "magical" powers', jitted, maintained, https://github.com/pixie-lang/pixie
- Monte, 'A dynamic language inspired by Python and E.' has an rpython implementation, in active development, https://github.com/monte-language/typhon
- Typhon, 'A virtual machine for Monte', in active development, https://github.com/monte-language/typhon
- **Tulip, an untyped functional language, in language design mode, maintained,** https://github.com/tulip-lang/tulip/
- Pycket, a Racket implementation, proof of concept, small language core working, a lot of primitives are missing. Slow development https://github.com/samth/pycket
- Lever, a dynamic language with a modifiable grammar, actively developed, https://github.com/cheery/lever

Complete, functioning, but inactive

- Converge 2, complete, last release version 2.1 in Feb 2015, http://convergepl.org/
- Pyrolog, Prolog, core complete, extensions missing, last commit in Nov 2015, http://bitbucket.org/cfbolz/pyrolog
- PyPy.js, compiles PyPy to Javascript via emscripten, with a custom JIT backend that emits asm.js code at runtime, http://pypyjs.org

Inactive (last reviewed Sept 2015):

- Topaz, Ruby, major functionality complete, library missing, inactive http://topazruby.com
- Rapydo, R, execution semantics complete, most builtins missing, inactive, http://bitbucket.org/cfbolz/rapydo
- Hippy, PHP, proof of concept, inactive, http://morepypy.blogspot.de/2012/07/hello-everyone.html
- Scheme, no clue about completeness, inactive, http://bitbucket.org/pypy/lang-scheme/
- PyGirl, Gameboy emulator, works but there is a bug somewhere, does not use JIT, unmaintained, http://bitbucket.org/pypy/lang-gameboy
- Javascript, proof of concept, inactive, http://bitbucket.org/pypy/lang-js
- An implementation of Notch's DCPU-16, https://github.com/AlekSi/dcpu16py/tree/pypy-again
- Haskell, core of the language works, but not many libraries, inactive http://bitbucket.org/cfbolz/haskell-python
- IO, no clue about completeness, inactive https://bitbucket.org/pypy/lang-io
- Qoppy, an implementation Qoppa, which is a scheme without special forms: https://github.com/timfel/qoppy
- XlispX, a toy Lisp: https://bitbucket.org/rxe/xlispx
- RPySOM, an RPython implementation of SOM (Simple Object Model) https://github.com/SOM-st/RPySOM
- SQPyte, really experimental implementation of the SQLite bytecode VM, jitted, probably inactive, https://bitbucket.org/softdevteam/sqpyte
- Icbink, an implementation of Kernel, core complete, naive, no JIT optimizations yet, on hiatus https://github.com/euccastro/icbink

CHAPTER 4

RPython internals

Glossary

annotator The component of the *RPython toolchain* that performs a form of *type inference* on the flow graph. See *The Annotation Pass* in the documentation.

backend Code generator that converts an *RPython Language* program to a *target language* using the *RPython toolchain*.

compile-time In the context of the *JIT*, compile time is when the JIT is generating machine code "just in time".

external function Functions that we don't want to implement in Python for various reasons (e.g. they need to make calls into the OS) and whose implementation will be provided by the backend.

garbage collection framework Code that makes it possible to write *RPython's garbage collectors* in Python itself.

guard a small test that checks if assumptions the JIT makes during tracing are still true

JIT *just in time compiler*.

llinterpreter Piece of code that is able to interpret flow graphs. This is very useful for testing purposes, especially if you work on the *RPython* Typer.

lltypesystem A *C-like type model* that contains structs and pointers. A *backend* that uses this type system is also called a low-level backend. The C backend uses this typesystem.

low-level helper A function that the *RTyper* can use a call to as part of implementing some operation in terms of the target *type system*.

ootypesystem An object oriented type model containing classes and instances. A *backend* that uses this type system is also called a high-level backend. The JVM and CLI backends all use this typesystem.

prebuilt constant In *RPython* module globals are considered constants. Moreover, global (i.e. prebuilt) lists and dictionaries are supposed to be immutable ("prebuilt constant" is sometimes abbreviated to "pbc").

promotion *JIT* terminology. *promotion* is a way of "using" a *run-time* value at *compile-time*, essentially by deferring compilation until the run-time value is known. See if *the jit docs* help.

RPython *RPython Language*, a limited subset of the Python language. The limitations make *type inference* possible. It is also the language that the PyPy interpreter itself is written in.

RPython toolchain *The Annotation Pass*, *The RPython Typer*, and various *backend*s.

rtyper Based on the type annotations, *The RPython Typer* turns the flow graph into one that fits the model of the target platform/*backend* using either the *lltypesystem* or the *ootypesystem*.

run-time In the context of the *JIT*, run time is when the code the JIT has generated is executing.

specialization A way of controlling how a specific function is handled by the *annotator*. One specialization is to treat calls to a function with different argument types as if they were calls to different functions with identical source.

transformation Code that modifies flowgraphs to weave in translation aspects

translation-time In the context of the *JIT*, translation time is when the PyPy source is being analyzed and the JIT itself is being created.

translator *Tool* based on the PyPy interpreter which can translate sufficiently static Python programs into low-level code.

type system The RTyper can target either the *lltypesystem* or the *ootypesystem*.

type inference Deduces either partially or fully the type of expressions as described in this type inference article on Wikipedia. The *RPython toolchain*'s flavour of type inference is described in *The Annotation Pass* section.

Getting Started with RPython

Contents

Warning: Please read this FAQ entry first!

RPython is a subset of Python that can be statically compiled. The PyPy interpreter is written mostly in RPython (with pieces in Python), while the RPython compiler is written in Python. The hard to understand part is that Python is a meta-programming language for RPython, that is, "being valid RPython" is a question that only makes sense on the live objects **after** the imports are done. This might require more explanation. You start writing RPython from `entry_point`, a good starting point is rpython/translator/goal/targetnopstandalone.py. This does not do all that much, but is a start. Now if code analyzed (in this case `entry_point`) calls some functions, those calls will be followed. Those followed calls have to be RPython themselves (and everything they call etc.), however not entire module files. To show how you can use metaprogramming, we can do a silly example (note that closures are not RPython):

```
def generator(operation):
    if operation == 'add':
        def f(a, b):
            return a + b
    else:
        def f(a, b):
            return a - b
    return f

add = generator('add')
sub = generator('sub')

def entry_point(argv):
    print add(sub(int(argv[1]), 3) 4)
    return 0
```

In this example `entry_point` is RPython, `add` and `sub` are RPython, however, `generator` is not.

The following introductory level articles are available:

- Laurence Tratt – Fast Enough VMs in Fast Enough Time.
- How to write interpreters in RPython and part 2 by Andrew Brown.

Trying out the translator

The translator is a tool based on the PyPy interpreter which can translate sufficiently static RPython programs into low-level code (in particular it can be used to translate the full Python interpreter). To be able to experiment with it you need to download and install the usual (CPython) version of:

- Pygame
- Dot Graphviz

To start the interactive translator shell do:

```
cd rpython
python bin/translatorshell.py
```

Test snippets of translatable code are provided in the file rpython/translator/test/snippet.py, which is imported under the name `snippet`. For example:

```
>>> t = Translation(snippet.is_perfect_number, [int])
>>> t.view()
```

After that, the graph viewer pops up, that lets you interactively inspect the flow graph. To move around, click on something that you want to inspect. To get help about how to use it, press 'H'. To close it again, press 'Q'.

Trying out the type annotator

We have a type annotator that can completely infer types for functions like `is_perfect_number` (as well as for much larger examples):

```
>>> t.annotate()
>>> t.view()
```

Move the mouse over variable names (in red) to see their inferred types.

Translating the flow graph to C code

The graph can be turned into C code:

```
>>> t.rtype()
>>> lib = t.compile_c()
```

The first command replaces the operations with other low level versions that only use low level types that are available in C (e.g. int). The compiled version is now in a `.so` library. You can run it say using ctypes:

```
>>> f = get_c_function(lib, snippet.is_perfect_number)
>>> f(5)
0
>>> f(6)
1
```

A slightly larger example

There is a small-to-medium demo showing the translator and the annotator:

```
python bin/rpython --view --annotate translator/goal/bpnn.py
```

This causes `bpnn.py` to display itself as a call graph and class hierarchy. Clicking on functions shows the flow graph of the particular function. Clicking on a class shows the attributes of its instances. All this information (call graph, local variables' types, attributes of instances) is computed by the annotator.

To turn this example to C code (compiled to the executable `bpnn-c`), type simply:

```
python bin/rpython translator/goal/bpnn.py
```

Translating Full Programs

To translate full RPython programs, there is the script `bin/rpython` in rpython/bin/. Examples for this are a slightly changed version of Pystone:

```
python bin/rpython translator/goal/targetrpystonedalone
```

This will produce the executable "targetrpystonedalone-c".

The largest example of this process is to translate the full Python interpreter. There is also an FAQ about how to set up this process for *your own interpreters*.

There are several environment variables you can find useful while playing with the RPython:

`PYPY_USESSION_DIR` RPython uses temporary session directories to store files that are generated during the translation process(e.g., translated C files). `PYPY_USESSION_DIR` serves as a base directory for these session dirs. The default value for this variable is the system's temporary dir.

`PYPY_USESSION_KEEP` By default RPython keeps only the last `PYPY_USESSION_KEEP` (defaults to 3) session dirs inside `PYPY_USESSION_DIR`. Increase this value if you want to preserve C files longer (useful when producing lots of lldebug builds).

Sources

- rpython/translator contains the code analysis and generation stuff. Start reading from translator.py, from which it should be easy to follow the pieces of code involved in the various translation phases.
- rpython/annotator contains the data model for the type annotation that can be inferred about a graph. The graph "walker" that uses this is in rpython/annotator/annrpython.py.
- rpython/rtyper contains the code of the RPython typer. The typer transforms annotated flow graphs in a way that makes them very similar to C code so that they can be easy translated. The graph transformations are controlled by the code in rpython/rtyper/rtyper.py. The object model that is used can be found in rpython/rtyper/lltypesystem/lltype.py. For each RPython type there is a file rxxxx.py that contains the low level functions needed for this type.
- rpython/rlib contains the *RPython standard library*, things that you can use from rpython.

RPython directory cross-reference

Here is a fully referenced alphabetical two-level deep directory overview of RPython:

Directory	explanation/links
rpython/annotator/	*type inferencing code* for *RPython* programs
rpython/config/	handles the numerous options for RPython
rpython/flowspace/	the *flow graph builder* implementing abstract interpretation
rpython/rlib/	a *"standard library"* for *RPython* programs
rpython/rtyper/	the *RPython Typer*
rpython/rtyper/lltypesystem/	the *low-level type system* for C-like backends
rpython/memory/	the *garbage collector* construction framework
rpython/tool/algo/	general-purpose algorithmic and mathematic tools
rpython/translator/	*translation* backends and support code
rpython/translator/backendopt/	general optimizations that run before a backend generates code
rpython/translator/c/	the *GenC backend*, producing C code from an RPython program (generally via the *rtyper*)
rpython/translator/jvm/	the Java backend
rpython/translator/tool/	helper tools for translation
dotviewer/	*graph viewer*

JIT documentation

abstract When a interpreter written in RPython is translated into an executable, the executable contains a full virtual machine that can optionally include a Just-In-Time compiler. This JIT compiler is **generated automatically from the interpreter** that we wrote in RPython.

This JIT Compiler Generator can be applied on interpreters for any language, as long as the interpreter itself is written in RPython and contains a few hints to guide the JIT Compiler Generator.

Content

Motivating JIT Compiler Generation

Contents

This is a non-technical introduction and motivation for RPython's approach to Just-In-Time compiler generation.

Motivation

Overview

Writing an interpreter for a complex dynamic language like Python is not a small task, especially if, for performance goals, we want to write a Just-in-Time (JIT) compiler too.

The good news is that it's not what we did. We indeed wrote an interpreter for Python, but we never wrote any JIT compiler for Python in PyPy. Instead, we use the fact that our interpreter for Python is written in RPython, which is a nice, high-level language – and we turn it *automatically* into a JIT compiler for Python.

This transformation is of course completely transparent to the user, i.e. the programmer writing Python programs. The goal (which we achieved) is to support *all* Python features – including, for example, random frame access and debuggers. But it is also mostly transparent to the language implementor, i.e. to the source code of the Python interpreter. It only needs a bit of guidance: we had to put a small number of hints in the source code of our interpreter. Based on these hints, the *JIT compiler generator* produces a JIT compiler which has the same language semantics as the original interpreter by construction. This JIT compiler itself generates machine code at runtime, aggressively optimizing the user's program and leading to a big performance boost, while keeping the semantics unmodified. Of course, the interesting bit is that our Python language interpreter can evolve over time without getting out of sync with the JIT compiler.

The path we followed

Our previous incarnations of PyPy's JIT generator were based on partial evaluation. This is a well-known and much-researched topic, considered to be very promising. There have been many attempts to use it to automatically transform an interpreter into a compiler. However, none of them have lead to substantial speedups for real-world languages. We believe that the missing key insight is to use partial evaluation to produce just-in-time compilers, rather than classical ahead-of-time compilers. If this turns out to be correct, the practical speed of dynamic languages could be vastly improved.

All these previous JIT compiler generators were producing JIT compilers similar to the hand-written Psyco. But today, starting from 2009, our prototype is no longer using partial evaluation – at least not in a way that would convince paper reviewers. It is instead based on the notion of *tracing JIT,* recently studied for Java and JavaScript. When compared to all existing tracing JITs so far, however, partial evaluation gives us some extra techniques that we already had in our previous JIT generators, notably how to optimize structures by removing allocations.

The closest comparison to our current JIT is Tamarin's TraceMonkey. However, this JIT compiler is written manually, which is quite some effort. Instead, we write a JIT generator at the level of RPython, which means that our final JIT

does not have to – indeed, cannot – be written to encode all the details of the full Python language. These details are automatically supplied by the fact that we have an interpreter for full Python.

Practical results

The JIT compilers that we generate use some techniques that are not in widespread use so far, but they are not exactly new either. The point we want to make here is not that we are pushing the theoretical limits of how fast a given dynamic language can be run. Our point is: we are making it **practical** to have reasonably good Just-In-Time compilers for all dynamic languages, no matter how complicated or non-widespread (e.g. Open Source dynamic languages without large industry or academic support, or internal domain-specific languages). By practical we mean that this should be:

- Easy: requires little more efforts than writing the interpreter in the first place.
- Maintainable: our generated JIT compilers are not separate projects (we do not generate separate source code, but only throw-away C code that is compiled into the generated VM). In other words, the whole JIT compiler is regenerated anew every time the high-level interpreter is modified, so that they cannot get out of sync no matter how fast the language evolves.
- Fast enough: we can get some rather good performance out of the generated JIT compilers. That's the whole point, of course.

Alternative approaches to improve speed

> **NOTE** Please take the following section as just a statement of opinion. In order to be debated over, the summaries should first be expanded into full arguments. We include them here as links; we are aware of them, even if sometimes pessimistic about them :-)

There are a large number of approaches to improving the execution speed of dynamic programming languages, most of which only produce small improvements and none offer the flexibility and customisability provided by our approach. Over the last 6 years of tweaking, the speed of CPython has only improved by a factor of 1.3 or 1.4 (depending on benchmarks). Many tweaks are applicable to PyPy as well. Indeed, some of the CPython tweaks originated as tweaks for PyPy.

IronPython initially achieved a speed of about 1.8 times that of CPython by leaving out some details of the language and by leveraging the large investment that Microsoft has put into making the .NET platform fast; the current, more complete implementation has roughly the same speed as CPython. In general, the existing approaches have reached the end of the road, speed-wise. Microsoft's Dynamic Language Runtime (DLR), often cited in this context, is essentially only an API to make the techniques pioneered in IronPython official. At best, it will give another small improvement.

Another technique regularly mentioned is adding types to the language in order to speed it up: either explicit optional typing or soft typing (i.e., inferred "likely" types). For Python, all projects in this area have started with a simplified subset of the language; no project has scaled up to anything close to the complete language. This would be a major effort and be platform- and language-specific. Moreover maintenance would be a headache: we believe that many changes that are trivial to implement in CPython, are likely to invalidate previous carefully-tuned optimizations.

For major improvements in speed, JIT techniques are necessary. For Python, Psyco gives typical speedups of 2 to 4 times - up to 100 times in algorithmic examples. It has come to a dead end because of the difficulty and huge costs associated with developing and maintaining it. It has a relatively poor encoding of language semantics - knowledge about Python behavior needs to be encoded by hand and kept up-to-date. At least, Psyco works correctly even when encountering one of the numerous Python constructs it does not support, by falling back to CPython. The PyPy JIT started out as a metaprogrammatic, non-language-specific equivalent of Psyco.

A different kind of prior art are self-hosting JIT compilers such as Jikes. Jikes is a JIT compiler for Java written in Java. It has a poor encoding of language semantics; it would take an enormous amount of work to encode all the details

of a Python-like language directly into a JIT compiler. It also has limited portability, which is an issue for Python; it is likely that large parts of the JIT compiler would need retargetting in order to run in a different environment than the intended low-level one.

Simply reusing an existing well-tuned JIT like that of the JVM does not really work, because of concept mismatches between the implementor's language and the host VM language: the former needs to be compiled to the target environment in such a way that the JIT is able to speed it up significantly - an approach which essentially has failed in Python so far: even though CPython is a simple interpreter, its Java and .NET re-implementations are not significantly faster.

More recently, several larger projects have started in the JIT area. For instance, Sun Microsystems is investing in JRuby, which aims to use the Java Hotspot JIT to improve the performance of Ruby. However, this requires a lot of hand crafting and will only provide speedups for one language on one platform. Some issues are delicate, e.g., how to remove the overhead of constantly boxing and unboxing, typical in dynamic languages. An advantage compared to PyPy is that there are some hand optimizations that can be performed, that do not fit in the metaprogramming approach. But metaprogramming makes the PyPy JIT reusable for many different languages on many different execution platforms. It is also possible to combine the approaches - we can get substantial speedups using our JIT and then feed the result to Java's Hotspot JIT for further improvement. One of us is even a member of the JSR 292 Expert Group to define additions to the JVM to better support dynamic languages, and is contributing insights from our JIT research, in ways that will also benefit PyPy.

Finally, tracing JITs are now emerging for dynamic languages like JavaScript with TraceMonkey. The code generated by PyPy is very similar (but not hand-written) to the concepts of tracing JITs.

Further reading

The description of the current RPython JIT generator is given in *PyJitPl5* (draft).

PyJitPl5

This document describes the fifth generation of the RPython JIT generator.

Implementation of the JIT

The JIT's *theory* is great in principle, but the actual code is a different story. This section tries to give a high level overview of how RPython's JIT is implemented. It's helpful to have an understanding of how the *RPython translation toolchain* works before digging into the sources.

Almost all JIT specific code is found in rpython/jit subdirectories. Translation time code is in the codewriter directory. The metainterp directory holds platform independent code including the the tracer and the optimizer. Code in the backend directory is responsible for generating machine code.

JIT hints

To add a JIT to an interpreter, RPython only requires two hints to be added to the target interpreter. These are jit_merge_point and can_enter_jit. jit_merge_point is supposed to go at the start of opcode dispatch. It allows the JIT to bail back to the interpreter in case running machine code is no longer suitable. can_enter_jit goes at the end of a application level loop. In the Python interpreter, this is the JUMP_ABSOLUTE bytecode. The Python interpreter defines its hints in pypy/module/pypyjit/interp_jit.py in a few overridden methods of the default interpreter loop.

An interpreter wishing to use the RPython JIT generator must define a list of *green* variables and a list of *red* variables. The *green* variables are loop constants. They are used to identify the current loop. Red variables are for everything else used in the execution loop. For example, the Python interpreter passes the code object and the instruction pointer

as greens and the frame object and execution context as reds. These objects are passed to the JIT at the location of the JIT hints.

JIT Generation

After the RTyping phase of translation, where high level Python operations are turned into low-level ones for the backend, the translation driver calls apply_jit() in metainterp/warmspot.py to add a JIT compiler to the currently translating interpreter. apply_jit() decides what assembler backend to use then delegates the rest of the work to the WarmRunnerDesc class. WarmRunnerDesc finds the two JIT hints in the function graphs. It rewrites the graph containing the jit_merge_point hint, called the portal graph, to be able to handle special JIT exceptions, which indicate special conditions to the interpreter upon exiting from the JIT. The location of the can_enter_jit hint is replaced with a call to a function, maybe_compile_and_run in warmstate.py, that checks if current loop is "hot" and should be compiled.

Next, starting with the portal graph, codewriter/*.py converts the graphs of the interpreter into JIT bytecode. Since this bytecode is stored in the final binary, it's designed to be concise rather than fast. The bytecode codewriter doesn't "see" (what it sees is defined by the JIT's policy) every part of the interpreter. In these cases, it simply inserts an opaque call.

Finally, translation finishes, including the bytecode of the interpreter in the final binary, and interpreter is ready to use the runtime component of the JIT.

Tracing

Application code running on the JIT-enabled interpreter starts normally; it is interpreted on top of the usual evaluation loop. When an application loop is closed (where the can_enter_jit hint was), the interpreter calls the maybe_compile_and_run() method of WarmEnterState. This method increments a counter associated with the current green variables. When this counter reaches a certain level, usually indicating the application loop has been run many times, the JIT enters tracing mode.

Tracing is where JIT interprets the bytecode, generated at translation time, of the interpreter interpreting the application level code. This allows it to see the exact operations that make up the application level loop. Tracing is performed by MetaInterp and MIFrame classes in metainterp/pyjitpl.py. maybe_compile_and_run() creates a MetaInterp and calls its compile_and_run_once() method. This initializes the MIFrame for the input arguments of the loop, the red and green variables passed from the jit_merge_point hint, and sets it to start interpreting the bytecode of the portal graph.

Before starting the interpretation, the loop input arguments are wrapped in a *box*. Boxes (defined in metainterp/history.py) wrap the value and type of a value in the program the JIT is interpreting. There are two main varieties of boxes: constant boxes and normal boxes. Constant boxes are used for values assumed to be known during tracing. These are not necessarily compile time constants. All values which are "promoted", assumed to be constant by the JIT for optimization purposes, are also stored in constant boxes. Normal boxes contain values that may change during the running of a loop. There are three kinds of normal boxes: BoxInt, BoxPtr, and BoxFloat, and four kinds of constant boxes: ConstInt, ConstPtr, ConstFloat, and ConstAddr. (ConstAddr is only used to get around a limitation in the translation toolchain.)

The meta-interpreter starts interpreting the JIT bytecode. Each operation is executed and then recorded in a list of operations, called the trace. Operations can have a list of boxes they operate on, arguments. Some operations (like GETFIELD and GETARRAYITEM) also have special objects that describe how their arguments are laid out in memory. All possible operations generated by tracing are listed in metainterp/resoperation.py. When a (interpreter-level) call to a function the JIT has bytecode for occurs during tracing, another MIFrame is added to the stack and the tracing continues with the same history. This flattens the list of operations over calls. Most importantly, it unrolls the opcode dispatch loop. Interpretation continues until the can_enter_jit hint is seen. At this point, a whole iteration of the application level loop has been seen and recorded.

Because only one iteration has been recorded the JIT only knows about one codepath in the loop. For example, if there's a if statement construct like this:

```
if x:
    do_something_exciting()
else:
    do_something_else()
```

and `x` is true when the JIT does tracing, only the codepath `do_something_exciting` will be added to the trace. In future runs, to ensure that this path is still valid, a special operation called a *guard operation* is added to the trace. A guard is a small test that checks if assumptions the JIT makes during tracing are still true. In the example above, a GUARD_TRUE guard will be generated for `x` before running `do_something_exciting`.

Once the meta-interpreter has verified that it has traced a loop, it decides how to compile what it has. There is an optional optimization phase between these actions which is covered future down this page. The backend converts the trace operations into assembly for the particular machine. It then hands the compiled loop back to the frontend. The next time the loop is seen in application code, the optimized assembly can be run instead of the normal interpreter.

Optimizations

The JIT employs several techniques, old and new, to make machine code run faster.

Virtuals and Virtualizables

A *virtual* value is an array, struct, or RPython level instance that is created during the loop and does not escape from it via calls or longevity past the loop. Since it is only used by the JIT, it can be "optimized out"; the value doesn't have to be allocated at all and its fields can be stored as first class values instead of deferencing them in memory. Virtuals allow temporary objects in the interpreter to be unwrapped. For example, a W_IntObject in the PyPy interpreter can be unwrapped to just be its integer value as long as the object is known not to escape the machine code.

A *virtualizable* is similar to a virtual in that its structure is optimized out in the machine code. Virtualizables, however, can escape from JIT controlled code.

Other optimizations

Most of the JIT's optimizer is contained in the subdirectory `metainterp/optimizeopt/`. Refer to it for more details.

More resources

More documentation about the current JIT is available as a first published article:

- Tracing the Meta-Level: PyPy's Tracing JIT Compiler

Chapters 5 and 6 of Antonio Cuni's PhD thesis contain an overview of how Tracing JITs work in general and more informations about the concrete case of PyPy's JIT.

The blog posts with the JIT tag might also contain additional information.

Trace Optimizer

Traces of user programs are not directly translated into machine code. The optimizer module implements several different semantic preserving transformations that either allow operations to be swept from the trace or convert them to operations that need less time or space.

The optimizer is in *rpython/jit/metainterp/optimizeopt/*. When you try to make sense of this module, this page might get you started.

Before some optimizations are explained in more detail, it is essential to understand how traces look like. The optimizer comes with a test suit. It contains many trace examples and you might want to take a look at it (in *rpython/jit/metainterp/optimizeopt/test/*.py*). The allowed operations can be found in *rpython/jit/metainterp/resoperation.py*. Here is an example of a trace:

```
[p0,i0,i1]
label(p0, i0, i1)
i2 = getarray_item_raw(p0, i0, descr=<Array Signed>)
i3 = int_add(i1,i2)
i4 = int_add(i0,1)
i5 = int_le(i4, 100) # lower-or-equal
guard_true(i5)
jump(p0, i4, i3)
```

At the beginning it might be clumsy to read but it makes sense when you start to compare the Python code that constructed the trace:

```
from array import array
a = array('i',range(101))
sum = 0; i = 0
while i <= 100: # can be seen as label
    sum += a[i]
    i += 1
    # jumps back to the while header
```

There are better ways to compute the sum from `[0..100]`, but it gives a better intuition on how traces are constructed than `sum(range(101))`. Note that the trace syntax is the one used in the test suite. It is also very similar to traces printed at runtime by *PYPYLOG*. The first line gives the input variables, the second line is a `label` operation, the last one is the backwards `jump` operation.

These instructions mentioned earlier are special:

- the input defines the input parameter type and name to enter the trace.
- `label` is the instruction a `jump` can target. Label instructions have a `JitCellToken` associated that uniquely identifies the label. Any jump has a target token of a label.

The token is saved in a so called *descriptor* of the instruction. It is not written explicitly because it is not done in the tests either. But the test suite creates a dummy token for each trace and adds it as descriptor to `label` and `jump`. Of course the optimizer does the same at runtime, but using real values. The sample trace includes a descriptor in `getarrayitem_raw`. Here it annotates the type of the array. It is a signed integer array.

High level overview

Before the JIT backend transforms any trace into machine code, it tries to transform the trace into an equivalent trace that executes faster. The method *optimize_trace* in *rpython/jit/metainterp/optimizeopt/__init__.py* is the main entry point.

Optimizations are applied in a sequence one after another and the base sequence is as follows:

```
intbounds:rewrite:virtualize:string:earlyforce:pure:heap:unroll
```

Each of the colon-separated name has a class attached, inheriting from the *Optimization* class. The *Optimizer* class itself also derives from the *Optimization* class and implements the control logic for the optimization. Most of the optimizations only require a single forward pass. The trace is 'propagated' into each optimization using the method *propagate_forward*. Instruction by instruction, it flows from the first optimization to the last optimization. The method *emit_operation* is called for every operation that is passed to the next optimizer.

A frequently encountered pattern

To find potential optimization targets it is necessary to know the instruction type. Simple solution is to switch using the operation number (= type):

```
for op in operations:
    if op.getopnum() == rop.INT_ADD:
        # handle this instruction
        pass
    elif op.getopnum() == rop.INT_FLOOR_DIV:
        pass
    # and many more
```

Things get worse if you start to match the arguments (is argument one constant and two variable or vice versa?). The pattern to tackle this code bloat is to move it to a separate method using *make_dispatcher_method*. It associates methods with instruction types:

```
class OptX(Optimization):
    def prefix_INT_ADD(self, op):
        pass # emit, transform, ...

dispatch_opt = make_dispatcher_method(OptX, 'prefix_',
                                      default=OptX.emit_operation)
OptX.propagate_forward = dispatch_opt

optX = OptX()
for op in operations:
    optX.propagate_forward(op)
```

`propagate_forward` searches for the method that is able to handle the instruction type. As an example *INT_ADD* will invoke *prefix_INT_ADD*. If there is no function for the instruction, it is routed to the default implementation (`emit_operation` in this example).

Rewrite optimization

The second optimization is called 'rewrite' and is commonly also known as strength reduction. A simple example would be that an integer multiplied by 2 is equivalent to the bits shifted to the left once (e.g. `x * 2 == x << 1`). Not only strength reduction is done in this optimization but also boolean or arithmetic simplifications. Other examples would be: `x & 0 == 0`, `x - 0 == x`

Whenever such an operation is encountered (e.g. `y = x & 0`), no operation is emitted. Instead the variable y is made equal to 0 (= `make_equal_to(op.result, 0)`). The variables found in a trace are instances of Box classes that can be found in *rpython/jit/metainterp/history.py*. *OptValue* wraps those variables again and maps the boxes to the optimization values in the optimizer. When a value is made equal, the two variable's boxes are made to point to the same *OptValue* instance.

NOTE: this OptValue organization is currently being refactored in a branch.

Pure optimization

Is interwoven into the basic optimizer. It saves operations, results, arguments to be known to have pure semantics.

"Pure" here means the same as the `jit.elidable` decorator: free of "observable" side effects and referentially transparent (the operation can be replaced with its result without changing the program semantics). The operations marked as ALWAYS_PURE in *resoperation.py* are a subset of the NOSIDEEFFECT operations. Operations such as new, new array, getfield_(raw/gc) are marked as NOSIDEEFFECT but not as ALWAYS_PURE.

Pure operations are optimized in two different ways. If their arguments are constants, the operation is removed and the result is turned into a constant. If not, we can still use a memoization technique: if, later, we see the same operation on the same arguments again, we don't need to recompute its result, but can simply reuse the previous operation's result.

Unroll optimization

A detailed description can be found the document Loop-Aware Optimizations in PyPy's Tracing JIT

This optimization does not fall into the traditional scheme of one forward pass only. In a nutshell it unrolls the trace _once_, connects the two traces (by inserting parameters into the jump and label of the peeled trace) and uses information to iron out allocations, propagate constants and do any other optimization currently present in the 'optimizeopt' module.

It is prepended to all optimizations and thus extends the Optimizer class and unrolls the loop once before it proceeds.

Vectorization

- *Vectorization*

What is missing from this document

- Guards are not explained
- Several optimizations are not explained

Further references

- Allocation Removal by Partial Evaluation in a Tracing JIT
- Loop-Aware Optimizations in PyPy's Tracing JIT

Virtualizables

Note: this document does not have a proper introduction as to how to understand the basics. We should write some. If you happen to be here and you're missing context, feel free to pester us on IRC.

Problem description

The JIT is very good at making sure some objects are never allocated if they don't escape from the trace. Such objects are called `virtuals`. However, if we're dealing with frames, virtuals are often not good enough. Frames can escape and they can also be allocated already at the moment we enter the JIT. In such cases we need some extra object that can still be optimized away, despite existing on the heap.

Solution

We introduce virtualizables. They're objects that exist on the heap, but their fields are not always in sync with whatever happens in the assembler. One example is that virtualizable fields can store virtual objects without forcing them. This is very useful for frames. Declaring an object to be virtualizable works like this:

```
class Frame(object):
   _virtualizable_ = ['locals[*]', 'stackdepth']
```

And we use them in `JitDriver` like this:

```
jitdriver = JitDriver(greens=[], reds=['frame'], virtualizables=['frame'])
```

This declaration means that `stackdepth` is a virtualizable **field**, while `locals` is a virtualizable **array** (a list stored on a virtualizable). There are various rules about using virtualizables, especially using virtualizable arrays that can be very confusing. Those will usually end up with a compile-time error (as opposed to strange behavior). The rules are:

- A virtualizable array must be a fixed-size list. After it is initialized (e.g. in `Frame.__init__`) you cannot resize it at all. You cannot assign a different list to the field, or even pass around the list. You can only access `frame.array[index]` directly.
- Each array access must be with a known positive index that cannot raise an `IndexError`. Using `index = jit.hint(index, promote=True)` might be useful to get a constant-number access. This is only safe if the index is actually constant or changing rarely within the context of the user's code.
- If you initialize a new virtualizable in the JIT, it has to be done like this (for example if we're in `Frame.__init__`):

  ```
  self = hint(self, access_directly=True, fresh_virtualizable=True)
  ```

  that way you can populate the fields directly.
- If you use virtualizable outside of the JIT – it's very expensive and sometimes aborts tracing. Consider it carefully as to how do it only for debugging purposes and not every time (e.g. `sys._getframe` call).
- If you have something equivalent of a Python generator, where the virtualizable survives for longer, you want to force it before returning. It's better to do it that way than by an external call some time later. It's done using `jit.hint(frame, force_virtualizable=True)`
- Your interpreter should have a local variable similar to `frame` above. It must not be modified as long as it runs its `jit_merge_point` loop, and in the loop it must be passed directly to the `jit_merge_point()` and `can_enter_jit()` calls. The JIT generator is known to produce buggy code if you fetch the virtualizable from somewhere every iteration, instead of reusing the same unmodified local variable.

Vectorization

To find parallel instructions the tracer must provide enough information about memory load/store operations. They must be adjacent in memory. The requirement for that is that they use the same index variable and offset can be expressed as a a linear or affine combination.

Command line flags:

- –jit vec=1: turns on the vectorization for marked jitdrivers (e.g. those in the NumPyPy module).
- –jit vec_all=1: turns on the vectorization for any jit driver. See parameters for the filtering heuristics of traces.

Features

Currently the following operations can be vectorized if the trace contains parallel operations:

- float32/float64: add, substract, multiply, divide, negate, absolute
- int8/int16/int32/int64 arithmetic: add, substract, multiply, negate, absolute
- int8/int16/int32/int64 logical: and, or, xor

Reduction

Reduction is implemented:

- sum, prod, any, all

Constant & Variable Expansion

Packed arithmetic operations expand scalar variables or contants into vector registers.

Guard Strengthening

Unrolled guards are strengthend on an arithmetical level (See GuardStrengthenOpt). The resulting vector trace will only have one guard that checks the index.

Calculations on the index variable that are redundant (because of the merged load/store instructions) are not removed. The backend removes these instructions while assembling the trace.

In addition a simple heuristic (enabled by –jit vec_all=1) tries to remove array bound checks for application level loops. It tries to identify the array bound checks and adds a transitive guard at the top of the loop:

```
label(...)
...
guard(i < n) # index guard
...
guard(i < len(a))
a = load(..., i, ...)
...
jump(...)
# becomes
guard(n < len(a))
label(...)
guard(i < n) # index guard
...
a = load(..., i, ...)
...
jump(...)
```

Future Work and Limitations

- The only SIMD instruction architecture currently supported is SSE4.1
- Packed mul for int8,int64 (see PMUL). It would be possible to use PCLMULQDQ. Only supported by some CPUs and must be checked in the cpuid.

- Loop that convert types from int(8|16|32|64) to int(8|16) are not supported in the current SSE4.1 assembler implementation. The opcode needed spans over multiple instructions. In terms of performance there might only be little to non advantage to use SIMD instructions for this conversions.
- For a guard that checks true/false on a vector integer regsiter, it would be handy to have 2 xmm registers (one filled with zero bits and the other with one every bit). This cuts down 2 instructions for guard checking, trading for higher register pressure.
- prod, sum are only supported by 64 bit data types
- isomorphic function prevents the following cases for combination into a pair: 1) getarrayitem_gc, getarrayitem_gc_pure 2) int_add(v,1), int_sub(v,-1)

PyPy's assembler backends

Draft notes about the organization of assembler backends in the PyPy JIT, in 2016

input: linear sequence of instructions, called a "trace".

A trace is a sequence of instructions in SSA form. Most instructions correspond to one or a few CPU-level instructions. There are a few meta-instructions like *label* and debugging stuff. All branching is done with guards, which are instructions that check that a condition is true and exit the trace if not. A failing guard can have a new trace added to it later, called a "bridge". A patched guard becomes a direct *Jcond* instruction going to the bridge, with no indirection, no register spilling, etc.

A trace ends with either a *return* or a *jump to label*. The target label is either inside the same trace, or in some older one. For historical reasons we call a "loop" a trace that is not a bridge. The machine code that we generate is organized as a forest of trees; the trunk of the tree is a "loop", and the branches are all bridges (branching off the trunk or off another branch).

- every trunk or branch that ends in a *jump to label* can target a label from a different tree, too.
- the whole process of assembling a loop or a branch is basically single-threaded, so no synchronization issue there (including to patch older generated instructions).
- the generated assembler has got a "frame" in %rbp, which is actually not on the stack at all, but is a GC object (called a "jitframe"). Spilling goes there.
- the guards are *Jcond* to a very small piece of generated code, which is basically pushing a couple of constants on the stack and then jumping to the general guard-recovery code. That code will save the registers into the jitframe and then exit the whole generated function. The caller of that generated function checks how it finished: if it finished by hitting a guard, then the caller is responsible for calling the "blackhole interpreter". This is the part of the front-end that recovers from failing guards and finishes running the frame (including, possibly, by jumping again into generated assembler).

Details about the JITting process:

- front-end and optimization pass
- rewrite (includes gc related transformation as well as simplifactions)
- assembler generation

Front-end and optimization pass

Not discussed here in detail. This produces loops and bridges using an instruction set that is "high-level" in some sense: it contains intructions like "new"/"new_array", and "setfield"/"setarrayitem"/"setinteriorfield" which describe

the action of storing a value in a precise field of the structure or array. For example, the "setfield" action might require implicitly a GC write barrier. This is the high-level trace that we send to the following step.

Rewrite

A mostly but not completely CPU-independent phase: lowers some instructions. For example, the variants of "new" are lowered to "malloc" and a few "gc_store": it bumps the pointer of the GC and then sets a few fields explicitly in the newly allocated structure. The "setfield" is replaced with a "cond_gc_wb_call" (conditional call to the write barrier) if needed, followed by a "gc_store".

The "gc_store" instruction can be encoded in a single MOV assembler instruction, but is not as flexible as a MOV. The address is always specified as "some GC pointer + an offset". We don't have the notion of interior pointer for GC objects.

A different instruction, "gc_store_indexed", offers additional operands, which can be mapped to a single MOV instruction using forms like *[rax+8*rcx+24]*.

Some other complex instructions pass through to the backend, which must deal with them: for example, "card marking" in the GC. (Writing an object pointer inside an array would require walking the whole array later to find "young" references. Instead of that, we flip a bit for every range of 128 entries. This is a common GC optimization.) Setting the card bit of a GC object requires a sequence of assembler instructions that depends too much on the target CPU to be expressed explicitly here (moreover, it contains a few branches, which are hard to express at this level).

Assembly

No fancy code generation technique, but greedy forward pass that tries to avoid some pitfalls

Handling instructions

- One by one (forward direction). Each instruction asks the register allocator to ensure that some arguments are in registers (not in the jitframe); asks for a register to put its result into; and asks for additional scratch registers that will be freed at the end of the instruction. There is a special case for boolean variables: they are stored in the condition code flags instead of being materialized as a 0/1 value. (They are materialized later, except in the common case where they are only used by the next *guard_false* or *guard_true* and then forgotten.)
- Instruction arguments are loaded into a register on demand. This makes the backend quite easy to write, but leads do some bad decisions.

Linear scan register allocation

Although it's always a linear trace that we consider, we don't use advanced techniques for register allocation: we do forward, on-demand allocation as the backend produces the assembler. When it asks for a register to put some value into, we give it any free register, without consideration for what will be done with it later. We compute the longevity of all variables, but only use it when choosing which register to spill (we spill the variable with the longest longevity).

This works to some extend because it is well integrated with the earlier optimization pass. Loops are unrolled once by the optimization pass to allow more powerful optimizations—the optimization pass itself is the place that benefits the most, but it also has benefits here in the assembly pass. These are:

- The first peeling initializes the register binding on the first use.
- This leads to an already allocated register of the trace loop.
- As well as allocated registers when exiting bridges

[Try to better allocate registers to match the ABI (minor to non benefit in the current state)]

More complex mappings

Some instructions generate more complex code. These are either or both of:

- complex instructions generating some local control flow, like "cond_gc_wb_call" (for write barriers), "call_assembler" (a call followed by a few checks).
- instructions that invoke custom assembler helpers, like the slow-path of write barriers or the slow-path of allocations. These slow-paths are typically generated too, so that we are not constrained by the usual calling conventions.

GC pointers

Around most CALL instructions, we need to record a description of where the GC pointers are (registers and stack frame). This is needed in case the CALL invokes a garbage collection. The GC pointers can move; the pointers in the registers and stack frame are updated by the GC. That's a reason for why we don't have explicit interior pointers.

GC pointers can appear as constants in the trace. We are busy changing that to use a constant table and *MOV REG, (%RIP+offset)*. The "constant" in the table is actually updated by the GC if the object move.

Vectorization

Optimization developed to use SIMD instructions for trace loops. Primary idea was to use it as an optimization of micro numpy. It has several passes on the already optimized trace.

Shortly explained: It builds dependencies for an unrolled trace loop, gathering pairs/packs of operations that could be executed in parallel and finally schedules the operations.

What did it add to the code base:

- Dependencies can be constructed
- Code motion of guards to relax dependencies
- Scheduler to reorder trace
- Array bound check removal (especially for unrolled traces)

What can it do:

- Transform vector loops (element wise operations)
- Accumulation (*reduce([...],operator,0)*). Requires Operation to be associative and commutative
- SSE 4.1 as "vector backend"

We do not

- Keep tracing data around to reoptimize the trace tree. (Once a trace is compiled, minimal data is kept.) This is one reason (there are others in the front-end) for the following result: JIT-compiling a small loop with two common paths ends up as one "loop" and one bridge assembled, and the bridge-following path is slightly less efficient. This is notably because this bridge is assembled with two constraints: the input registers are fixed (from the guard), and the output registers are fixed (from the jump target); usually these two sets of fixed registers are different, and copying around is needed.

- We don't join trace tails: we only assemble *trees*.
- We don't do any reordering (neither of trace instructions nor of individual assembler instructions)
- We don't do any cross-instruction optimization that makes sense only for the backend and can't easily be expressed at a higher level. I'm sure there are tons of examples of that, but e.g. loading a large constant in a register that will survive for several instructions; moving out of loops *parts* of some instruction like the address calculation; etc. etc.
- Other optimization opportunities I can think about: look at the function prologue/epilogue; look at the overhead (small but not zero) at the start of a bridge. Also check if the way guards are implemented makes sense. Also, we generate large-ish sequences of assembler instructions with tons of *Jcond* that are almost never followed; any optimization opportunity there? (They all go forward, if it changes anything.) In theory we could also replace some of these with a signal handler on segfault (e.g. *guard_nonnull_class*).

a GCC or LLVM backend?

At least for comparison we'd like a JIT backend that emits its code using GCC or LLVM (irrespective of the time it would take). But it's hard to map reasonably well the guards to the C language or to LLVM IR. The problems are: (1) we have many guards, we would like to avoid having many paths that each do a full saving-all-local-variables-that-are-still-alive; (2) it's hard to patch a guard when a bridge is compiled from it; (3) instructions like a CALL need to expose the local variables that are GC pointers; CALL_MAY_FORCE need to expose *all* local variables for optional off-line reconstruction of the interpreter state.

- *Overview*: motivating our approach
- *Notes* about the current work in PyPy
- *Optimizer*: the step between tracing and writing machine code
- *Virtualizable*: how virtualizables work and what they are (in other words how to make frames more efficient).
- *Assembler backend*: draft notes about the organization of the assembler backends

Architecture specific notes

Here you can find some architecture specific notes.

IBM Mainframe S390X

Our JIT implements the 64 bit version of the IBM Mainframe called s390x. Note that this architecture is big endian.

Currently supported ISAs:

- z13 (released January 2015)
- zEC12 (released September 2012)
- z196 (released August 2010)
- z10 (released February 2008)

To check if all the necessary CPU facilities are installed on the subject machine, please run the test using a copy of the pypy source code:

```
$ ./pytest.py rpython/jit/backend/zarch/test/test_assembler -v -k 'test_facility'
```

In addition you can run the auto encoding test to check if your Linux GCC tool chain is able to compile all instructions used in the JIT backend:

```
$ ./pytest.py rpython/jit/backend/zarch/test/test_auto_encoding.py -v
```

Translating

Specifically check for these two dependencies. On old versions of some Linux distributions ship older versions.

- libffi (version should do > 3.0.+).
- CPython 2.7.+.

The RPython Toolchain

Contents

This document describes the toolchain that we have developed to analyze and "compile" RPython programs (like PyPy itself) to various target platforms.

It consists of three broad sections: a slightly simplified overview, a brief introduction to each of the major components of our toolchain and then a more comprehensive section describing how the pieces fit together. If you are reading this document for the first time, the *Overview* is likely to be most useful, if you are trying to refresh your PyPy memory then the *How It Fits Together* is probably what you want.

Overview

The job of the translation toolchain is to translate RPython programs into an efficient version of that program for one of various target platforms, generally one that is considerably lower-level than Python. It divides this task into several steps, and the purpose of this document is to introduce them.

To start with we describe the process of translating an *RPython* program into C (which is the default and original target). The RPython translation toolchain never sees Python source code or syntax trees, but rather starts with the *code objects* that define the behaviour of the function objects one gives it as input. The *flow graph builder* works through these code objects using abstract interpretation to produce a control flow graph (one per function): yet another representation of the source program, but one which is suitable for applying type inference and translation techniques and which is the fundamental data structure most of the translation steps operate on.

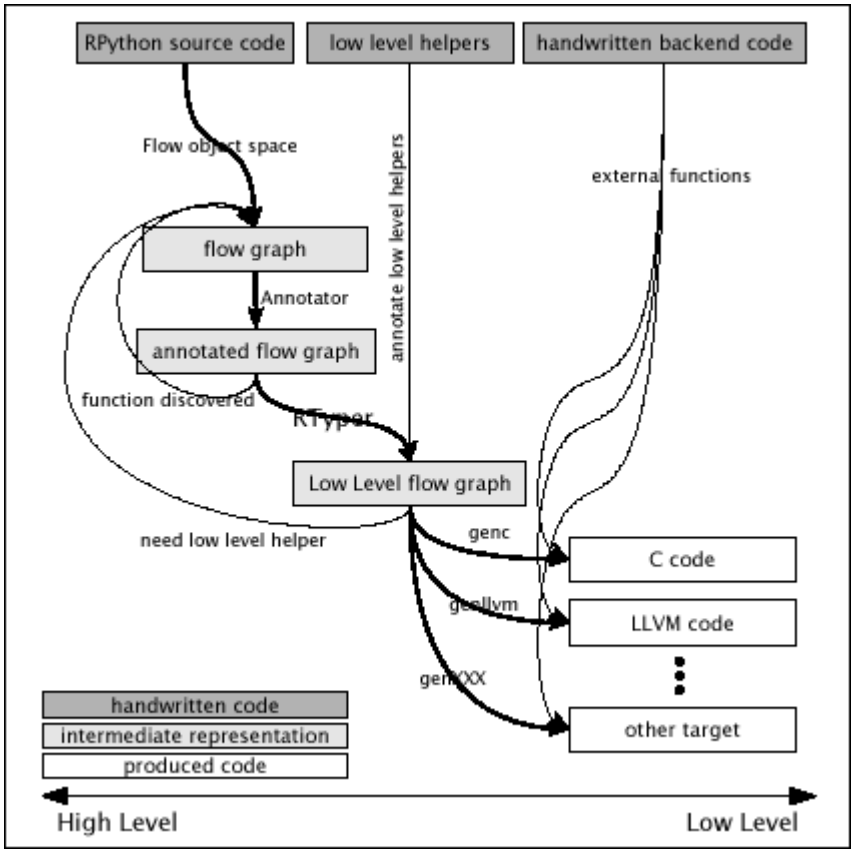
It is helpful to consider translation as being made up of the following steps (see also the figure below):

1. The complete program is imported, at which time arbitrary run-time initialization can be performed. Once this is done, the program must be present in memory as a form that is "static enough" in the sense of *RPython*.
2. The *Annotator* performs a global analysis starting from an specified entry point to deduce type and other information about what each variable can contain at run-time, *building flow graphs* as it encounters them.
3. The *RPython Typer* (or RTyper) uses the high-level information inferred by the Annotator to turn the operations in the control flow graphs into low-level operations.
4. After the RTyper there are several optional *optimizations* which can be applied and are intended to make the resulting program go faster.
5. The next step is *preparing the graphs for source generation*, which involves computing the names that the various functions and types in the program will have in the final source and applying transformations which insert explicit exception handling and memory management operations.
6. *The C backend* (colloquially known as "GenC") produces a number of C source files (as noted above, we are ignoring the other backends for now).
7. These source files are compiled to produce an executable.

(although these steps are not quite as distinct as you might think from this presentation).

There is an *interactive interface* called rpython/bin/translatorshell.py to the translation process which allows you to interactively work through these stages.

The following figure gives a simplified overview (PDF color version):



Building Flow Graphs

Introduction

The task of the flow graph builder (the source is at rpython/flowspace/) is to generate a control-flow graph from a function. This graph will also contain a trace of the individual operations, so that it is actually just an alternate representation for the function.

The basic idea is that if an interpreter is given a function, e.g.:

```
def f(n):
  return 3*n+2
```

it will compile it to bytecode and then execute it on its VM. Instead, the flow graph builder contains an abstract interpreter which takes the bytecode and performs whatever stack-shuffling and variable juggling is needed, but merely records any actual operation performed on a Python object into a structure called a basic block. The result of the operation is represented by a placeholder value that can appear in further operations.

For example, if the placeholder `v1` is given as the argument to the above function, the bytecode interpreter will call `v2 = space.mul(space.wrap(3), v1)` and then `v3 = space.add(v2, space.wrap(2))` and return `v3` as the result. During these calls, the following block is recorded:

```
Block(v1):      # input argument
  v2 = mul(Constant(3), v1)
  v3 = add(v2, Constant(2))
```

Abstract interpretation

`build_flow()` works by recording all operations issued by the bytecode interpreter into basic blocks. A basic block ends in one of two cases: when the bytecode interpreters calls `is_true()`, or when a joinpoint is reached.

- A joinpoint occurs when the next operation is about to be recorded into the current block, but there is already another block that records an operation for the same bytecode position. This means that the bytecode interpreter has closed a loop and is interpreting already-seen code again. In this situation, we interrupt the bytecode interpreter and we make a link from the end of the current block back to the previous block, thus closing the loop in the flow graph as well. (Note that this occurs only when an operation is about to be recorded, which allows some amount of constant-folding.)
- If the bytecode interpreter calls `is_true()`, the abstract interpreter doesn't generally know if the answer should be True or False, so it puts a conditional jump and generates two successor blocks for the current basic block. There is some trickery involved so that the bytecode interpreter is fooled into thinking that `is_true()` first returns False (and the subsequent operations are recorded in the first successor block), and later the *same* call to `is_true()` also returns True (and the subsequent operations go this time to the other successor block).

(This section to be extended...)

The Flow Model

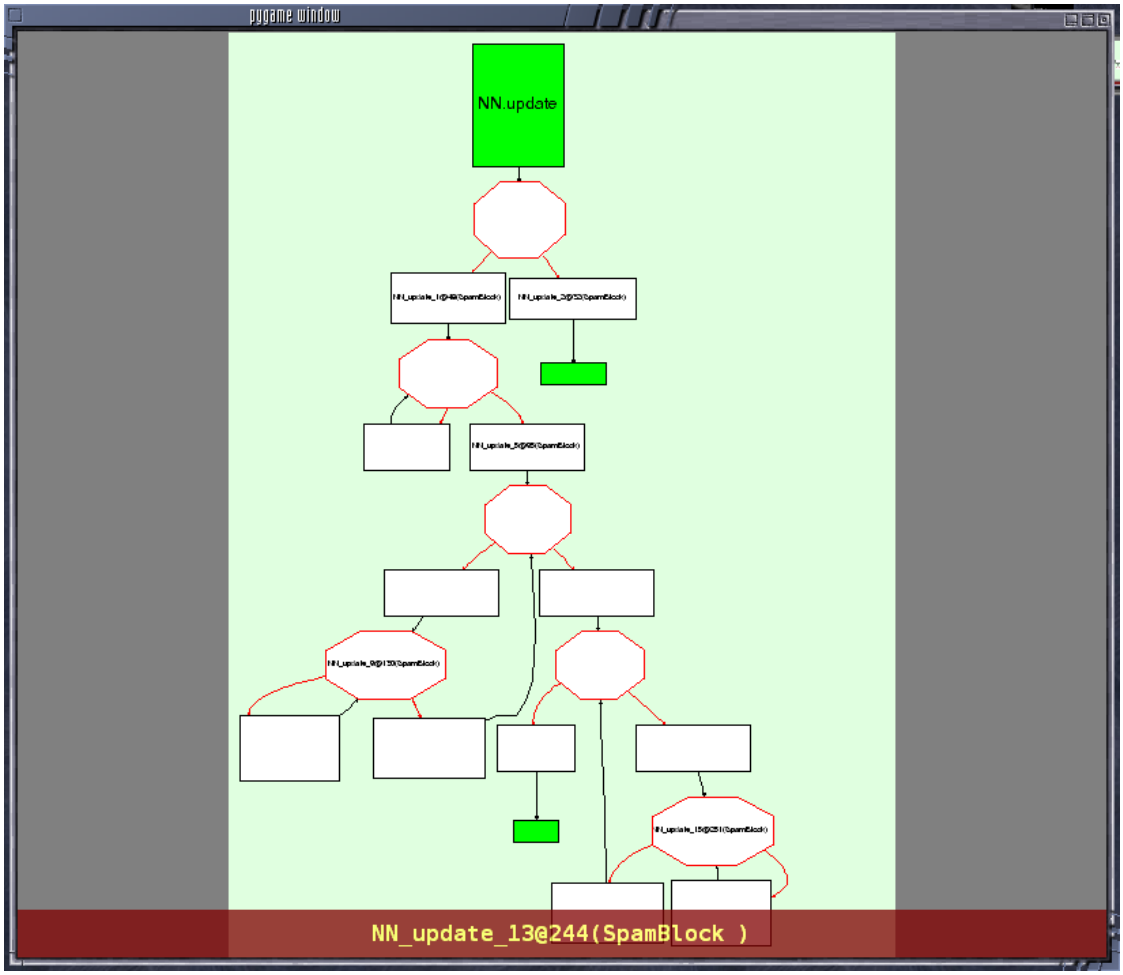
Here we describe the data structures produced by `build_flow()`, which are the basic data structures of the translation process.

All these types are defined in rpython/flowspace/model.py (which is a rather important module in the PyPy source base, to reinforce the point).

The flow graph of a function is represented by the class `FunctionGraph`. It contains a reference to a collection of `Block`s connected by `Link`s.

A `Block` contains a list of `SpaceOperation`s. Each `SpaceOperation` has an `opname` and a list of `args` and `result`, which are either `Variable`s or `Constant`s.

We have an extremely useful PyGame viewer, which allows you to visually inspect the graphs at various stages of the translation process (very useful to try to work out why things are breaking). It looks like this:



It is recommended to play with `python bin/translatorshell.py` on a few examples to get an idea of the structure of flow graphs. The following describes the types and their attributes in some detail:

`FunctionGraph` A container for one graph (corresponding to one function).

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 explicitly raise exceptions.)

`Block` A basic block, containing a list of operations and ending 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 SpaceOperations.

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(last_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.

`Link` A link from one basic block to another.

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`).

`SpaceOperation` A recorded (or otherwise generated) basic operation.

opname the name of the operation. `build_flow()` produces only operations from the list in `rpython.flowspace.operation`, but later the names can be changed arbitrarily.

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.

`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.

`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.

The Annotation Pass

We describe briefly below how a control flow graph can be "annotated" to discover the types of the objects. This annotation pass is a form of type inference. It operates on the control flow graphs built by the Flow Object Space.

For a more comprehensive description of the annotation process, see the corresponding section of our EU report about translation.

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 a subclass of `SomeObject`. Each subclass that represents a specific family of objects.

Here is an overview (see `pypy/annotation/model/`):

- `SomeObject` is the base class. An instance of `SomeObject()` represents any Python object, and as such usually means that the input program was not fully RPython.
- `SomeInteger()` represents any integer. `SomeInteger(nonneg=True)` represent a non-negative integer (`>=0`).
- `SomeString()` represents any string; `SomeChar()` a string of length 1.
- `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 result of the annotation pass is essentially a large dictionary mapping `Variable`s to annotations.

All the `SomeXxx` instances are immutable. If the annotator needs to revise its belief about what a Variable can contain, it does so creating a new annotation, not mutating the existing one.

Mutable Values and Containers

Mutable objects need special treatment during annotation, because the annotation of contained values needs to be possibly updated to account for mutation operations, and consequently the annotation information reflown through the relevant parts of the flow graphs.

- `SomeList` stands for a list of homogeneous type (i.e. all the elements of the list are represented by a single common `SomeXxx` annotation).
- `SomeDict` stands for a homogeneous dictionary (i.e. all keys have the same `SomeXxx` annotation, and so have all values).

User-defined Classes and Instances

`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 (`pypy.annotation.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.

Instance-level attributes are discovered progressively as the annotation progresses. Assignments like:

```
inst.attr = value
```

update the ClassDef of the given instance to record that the given attribute exists and can be as general as the given value.

For every attribute, the ClassDef also records all the positions where the attribute is *read*. If, at some later time, we discover an assignment that forces the annotation about the attribute to be generalized, then all the places that read the attribute so far are marked as invalid and the annotator will restart its analysis from there.

The distinction between instance-level and class-level attributes is thin; class-level attributes are essentially considered as initial values for instance-level attributes. Methods are not special in this respect, except that they are bound to the instance (i.e. `self = SomeInstance(cls)`) when considered as the initial value for the instance.

The inheritance rules are as follows: the union of two `SomeInstance` annotations is the `SomeInstance` of the most precise common base class. If an attribute is considered (i.e. read or written) through a `SomeInstance` of a parent class, then we assume that all subclasses also have the same attribute, and that the same annotation applies to them all (so code like `return self.x` in a method of a parent class forces the parent class and all its subclasses to have an attribute `x`, whose annotation is general enough to contain all the values that all the subclasses might want to store in `x`). However, distinct subclasses can have attributes of the same names with different, unrelated annotations if they are not used in a general way through the parent class.

The RPython Typer

See *The RPython Typer*.

Backend Optimizations

The point of the backend optimizations are to make the compiled program run faster. Compared to many parts of the PyPy translator, which are very unlike a traditional compiler, most of these will be fairly familiar to people who know how compilers work.

Function Inlining

To reduce the overhead of the many function calls that occur when running the PyPy interpreter we implemented function inlining. This is an optimization which takes a flow graph and a callsite and inserts a copy of the flow graph into the graph of the calling function, renaming occurring variables as appropriate. This leads to problems if the original function was surrounded by a `try: ... except: ...` guard. In this case inlining is not always possible. If the called function is not directly raising an exception (but an exception is potentially raised by further called functions) inlining is safe, though.

In addition we also implemented heuristics which function to inline where. For this purpose we assign every function a "size". This size should roughly correspond to the increase in code-size which is to be expected should the function be inlined somewhere. This estimate is the sum of two numbers: for one every operations is assigned a specific weight, the default being a weight of one. Some operations are considered to be more effort than others, e.g. memory allocation and calls; others are considered to be no effort at all (casts...). The size estimate is for one the sum of the

weights of all operations occurring in the graph. This is called the "static instruction count". The other part of the size estimate of a graph is the "median execution cost". This is again the sum of the weight of all operations in the graph, but this time weighted with a guess how often the operation is executed. To arrive at this guess we assume that at every branch we take both paths equally often, except for branches that are the end of loops, where the jump back to the end of the loop is considered more likely. This leads to a system of equations which can be solved to get approximate weights for all operations.

After the size estimate for all function has been determined, functions are being inlined into their callsites, starting from the smallest functions. Every time a function is being inlined into another function, the size of the outer function is recalculated. This is done until the remaining functions all have a size greater than a predefined limit.

Malloc Removal

Since RPython is a garbage collected language there is a lot of heap memory allocation going on all the time, which would either not occur at all in a more traditional explicitly managed language or results in an object which dies at a time known in advance and can thus be explicitly deallocated. For example a loop of the following form:

```
for i in range(n):
    ...
```

which simply iterates over all numbers from 0 to n - 1 is equivalent to the following in Python:

```
l = range(n)
iterator = iter(l)
try:
    while 1:
        i = iterator.next()
        ...
except StopIteration:
    pass
```

Which means that three memory allocations are executed: The range object, the iterator for the range object and the StopIteration instance, which ends the loop.

After a small bit of inlining all these three objects are never even passed as arguments to another function and are also not stored into a globally reachable position. In such a situation the object can be removed (since it would die anyway after the function returns) and can be replaced by its contained values.

This pattern (an allocated object never leaves the current function and thus dies after the function returns) occurs quite often, especially after some inlining has happened. Therefore we implemented an optimization which "explodes" objects and thus saves one allocation in this simple (but quite common) situation.

Escape Analysis and Stack Allocation

Another technique to reduce the memory allocation penalty is to use stack allocation for objects that can be proved not to life longer than the stack frame they have been allocated in. This proved not to really gain us any speed, so over time it was removed again.

Preparation for Source Generation

This, perhaps slightly vaguely named, stage is the most recent to appear as a separate step. Its job is to make the final implementation decisions before source generation – experience has shown that you really don't want to be doing *any* thinking at the same time as actually generating source code. For the C backend, this step does three things:

- inserts explicit exception handling,

- inserts explicit memory management operations,
- decides on the names functions and types will have in the final source (this mapping of objects to names is sometimes referred to as the "low-level database").

Making Exception Handling Explicit

RPython code is free to use exceptions in much the same way as unrestricted Python, but the final result is a C program, and C has no concept of exceptions. The exception transformer implements exception handling in a similar way to CPython: exceptions are indicated by special return values and the current exception is stored in a global data structure.

In a sense the input to the exception transformer is a program in terms of the *lltypesystem* with exceptions and the output is a program in terms of the bare lltypesystem.

Memory Management Details

As well as featuring exceptions, RPython is a garbage collected language; again, C is not. To square this circle, decisions about memory management must be made. In keeping with PyPy's approach to flexibility, there is freedom to change how to do it. There are three approaches implemented today:

- reference counting (deprecated, too slow)
- using the Boehm-Demers-Weiser conservative garbage collector
- using one of our custom *exact GCs implemented in RPython*

Almost all application-level Python code allocates objects at a very fast rate; this means that the memory management implementation is critical to the performance of the PyPy interpreter.

You can choose which garbage collection strategy to use with **:config:'translation.gc'**.

The C Backend

rpython/translator/c/

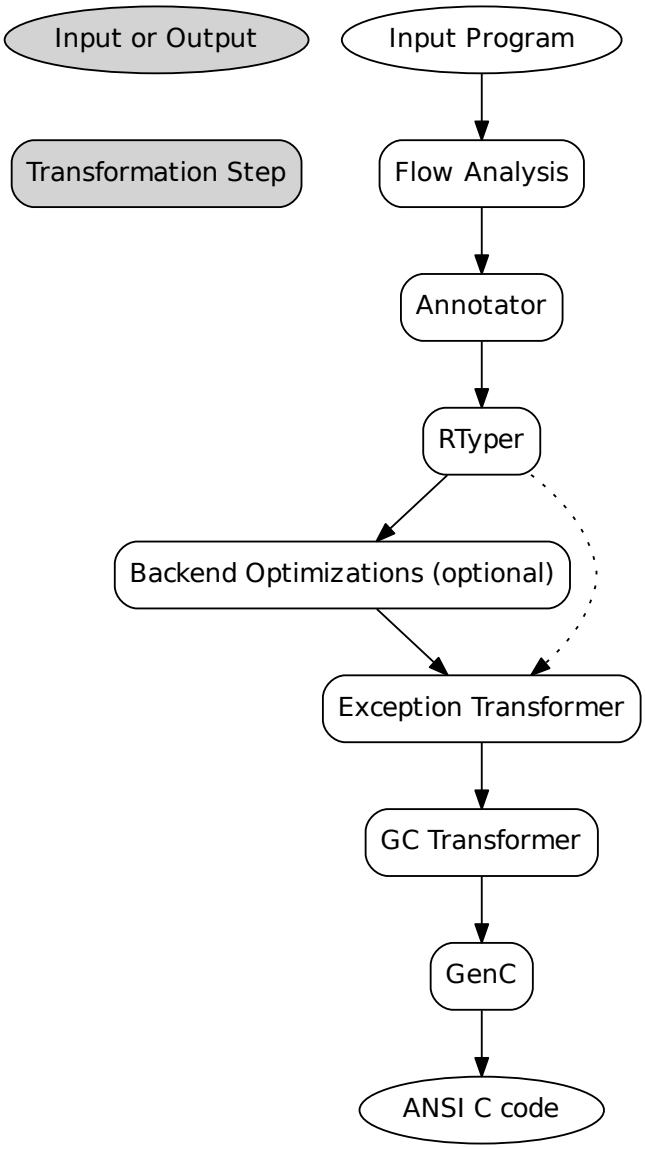
This is currently the sole code generation backend.

A Historical Note

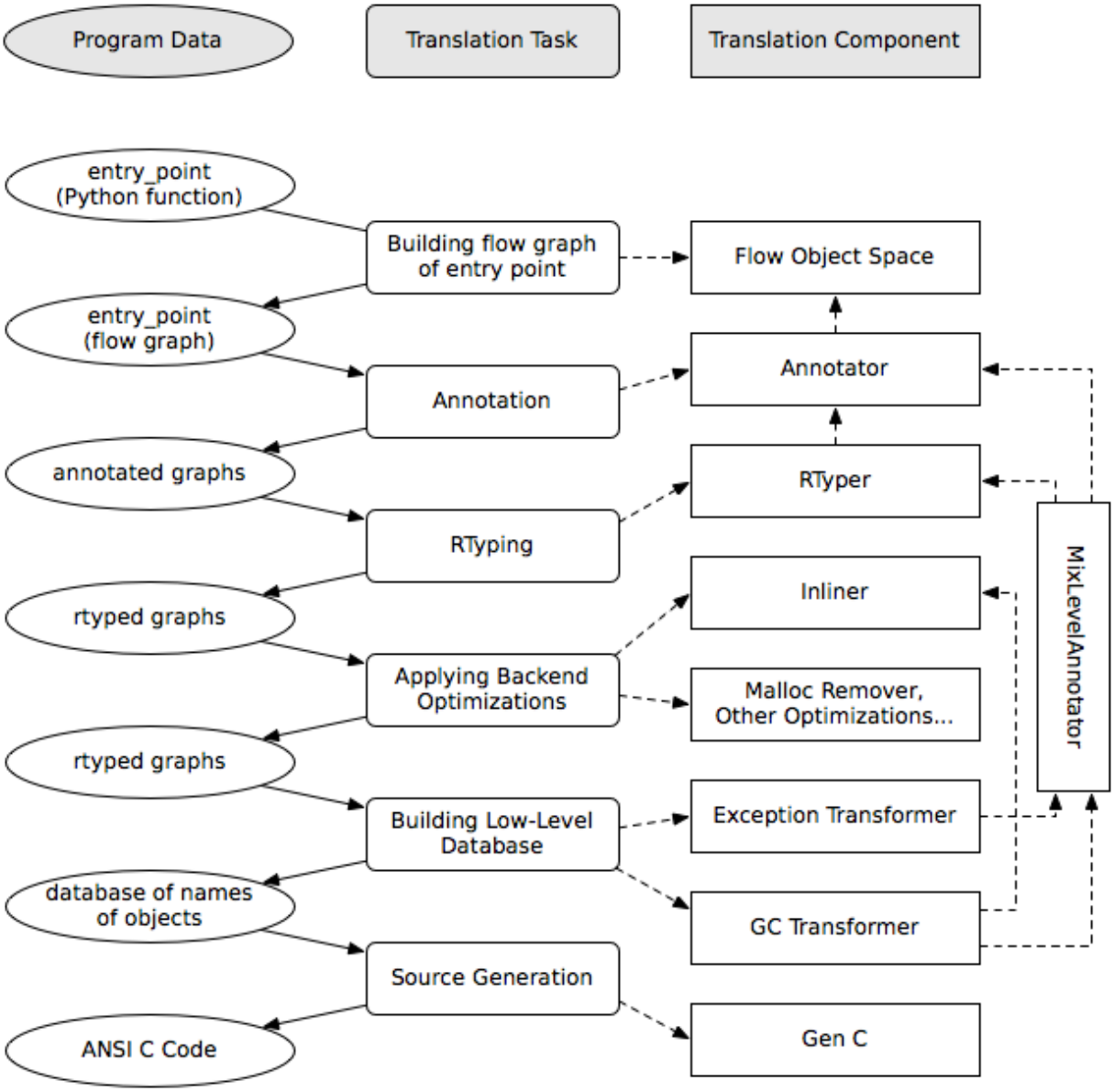
As this document has shown, the translation step is divided into more steps than one might at first expect. It is certainly divided into more steps than we expected when the project started; the very first version of GenC operated on the high-level flow graphs and the output of the annotator, and even the concept of the RTyper didn't exist yet. More recently, the fact that preparing the graphs for source generation ("databasing") and actually generating the source are best considered separately has become clear.

How It Fits Together

As should be clear by now, the translation toolchain of PyPy is a flexible and complicated beast, formed from many separate components.



A detail that has not yet been emphasized is the interaction of the various components. It makes for a nice presentation to say that after the annotator has finished the RTyper processes the graphs and then the exception handling is made explicit and so on, but it's not entirely true. For example, the RTyper inserts calls to many *low-level helper*s which must first be annotated, and the GC transformer can use inlining (one of the *backend optimizations*) of some of its small helper functions to improve performance. The following picture attempts to summarize the components involved in performing each step of the default translation process:



A component not mentioned before is the "MixLevelAnnotator"; it provides a convenient interface for a "late" (after RTyping) translation step to declare that it needs to be able to call each of a collection of functions (which may refer to each other in a mutually recursive fashion) and annotate and rtype them all at once.

The RPython Typer

Contents

The RPython Typer lives in the directory rpython/rtyper/.

Overview

The RPython Typer is the bridge between the *Annotator* and the code generators. The annotations of the *Annotator* are high-level, in the sense that they describe RPython types like lists or instances of user-defined classes.

To emit code we need to represent these high-level annotations in the low-level model of the target language; for C, this means structures and pointers and arrays. The Typer both determines the appropriate low-level type for each annotation and replaces each high-level operation in the control flow graphs with one or a few low-level operations. Just like low-level types, there is only a fairly restricted set of low-level operations, along the lines of reading or writing from or to a field of a structure.

In theory, this step is optional; a code generator might be able to read the high-level types directly. Our experience, however, suggests that this is very unlikely to be practical. "Compiling" high-level types into low-level ones is rather more messy than one would expect. This was the motivation for making this step explicit and isolated in a single place. After RTyping, the graphs only contain operations that already live on the level of the target language, making the job of the code generators much simpler.

Example: Integer operations

Integer operations are the easiest. Assume a graph containing the following operation:

```
v3 = add(v1, v2)
```

annotated:

```
v1 -> SomeInteger()
v2 -> SomeInteger()
v3 -> SomeInteger()
```

then obviously we want to type it and replace it with:

```
v3 = int_add(v1, v2)
```

where – in C notation – all three variables v1, v2 and v3 are typed `int`. This is done by attaching an attribute `concretetype` to v1, v2 and v3 (which might be instances of Variable or possibly Constant). In our model, this `concretetype` is `rpython.rtyper.lltypesystem.lltype.Signed`. Of course, the purpose of replacing the operation called `add` with `int_add` is that code generators no longer have to worry about what kind of addition (or concatenation maybe?) it means.

The process in more details

The RPython Typer has a structure similar to that of the *Annotator* both consider each block of the flow graphs in turn, and perform some analysis on each operation. In both cases the analysis of an operation depends on the annotations of its input arguments. This is reflected in the usage of the same `__extend__` syntax in the source files (compare e.g. rpython/annotator/binaryop.py and rpython/rtyper/rint.py).

The analogy stops here, though: while it runs, the Annotator is in the middle of computing the annotations, so it might need to reflow and generalize until a fixpoint is reached. The Typer, by contrast, works on the final annotations that the Annotator computed, without changing them, assuming that they are globally consistent. There is no need to reflow: the Typer considers each block only once. And unlike the Annotator, the Typer completely modifies the flow graph, by replacing each operation with some low-level operations.

In addition to replacing operations, the RTyper creates a `concretetype` attribute on all Variables and Constants in the flow graphs, which tells code generators which type to use for each of them. This attribute is a *low-level type*, as described below.

Representations

Representations – the Repr classes – are the most important internal classes used by the RTyper. (They are internal in the sense that they are an "implementation detail" and their instances just go away after the RTyper is finished; the code generators should only use the `concretetype` attributes, which are not Repr instances but *low-level types*.)

A representation contains all the logic about mapping a specific SomeXxx() annotation to a specific low-level type. For the time being, the RTyper assumes that each SomeXxx() instance needs only one "canonical" representation. For example, all variables annotated with SomeInteger() will correspond to the `Signed` low-level type via the `IntegerRepr` representation. More subtly, variables annotated SomeList() can correspond either to a structure holding an array of items of the correct type, or – if the list in question is just a range() with a constant step – a structure with just start and stop fields.

This example shows that two representations may need very different low-level implementations for the same high-level operations. This is the reason for turning representations into explicit objects.

The base Repr class is defined in rpython/rtyper/rmodel.py. Most of the rpython/`r*.py` files define one or a few subclasses of Repr. The method getrepr() of the RTyper will build and cache a single Repr instance per SomeXxx() instance; moreover, two SomeXxx() instances that are equal get the same Repr instance.

The key attribute of a Repr instance is called `lowleveltype`, which is what gets copied into the attribute `concretetype` of the Variables that have been given this representation. The RTyper also computes a `concretetype` for Constants, to match the way they are used in the low-level operations (for example, `int_add(x, 1)` requires a `Constant(1)` with `concretetype=Signed`).

In addition to `lowleveltype`, each Repr subclass provides a set of methods called `rtype_op_xxx()` which define how each high-level operation `op_xxx` is turned into low-level operations.

Low-Level Types

The RPython Typer uses a standard low-level model which we believe can correspond rather directly to various target languages such as C. This model is implemented in the first part of rpython/rtyper/lltypesystem/lltype.py.

The second part of rpython/rtyper/lltypesystem/lltype.py is a runnable implementation of these types, for testing purposes. It allows us to write and test plain Python code using a malloc() function to obtain and manipulate structures and arrays. This is useful for example to implement and test RPython types like 'list' with its operations and methods.

The basic assumption is that Variables (i.e. local variables and function arguments and return value) all contain "simple" values: basically, just integers or pointers. All the "container" data structures (struct and array) are allocated in the heap, and they are always manipulated via pointers. (There is no equivalent to the C notion of local variable of a `struct` type.)

Here is a quick tour:

```
>>> from rpython.rtyper.lltypesystem.lltype import *
```

Here are a few primitive low-level types, and the typeOf() function to figure them out:

```
>>> Signed
<Signed>
>>> typeOf(5)
<Signed>
>>> typeOf(r_uint(12))
<Unsigned>
>>> typeOf('x')
<Char>
```

Let's say that we want to build a type "point", which is a structure with two integer fields "x" and "y":

```
>>> POINT = GcStruct('point', ('x', Signed), ('y', Signed))
>>> POINT
<GcStruct point { x: Signed, y: Signed }>
```

The structure is a `GcStruct`, which means a structure that can be allocated in the heap and eventually freed by some garbage collector. (For platforms where we use reference counting, think about `GcStruct` as a struct with an additional reference counter field.)

Giving a name ('point') to the GcStruct is only for clarity: it is used in the representation.

```
>>> p = malloc(POINT)
>>> p
<* struct point { x=0, y=0 }>
>>> p.x = 5
>>> p.x
5
>>> p
<* struct point { x=5, y=0 }>
```

`malloc()` allocates a structure from the heap, initializes it to 0 (currently), and returns a pointer to it. The point of all this is to work with a very limited, easily controllable set of types, and define implementations of types like list in this elementary world. The `malloc()` function is a kind of placeholder, which must eventually be provided by the code generator for the target platform; but as we have just seen its Python implementation in rpython/rtyper/lltypesystem/lltype.py works too, which is primarily useful for testing, interactive exploring, etc.

The argument to `malloc()` is the structure type directly, but it returns a pointer to the structure, as `typeOf()` tells you:

```
>>> typeOf(p)
<* GcStruct point { x: Signed, y: Signed }>
```

For the purpose of creating structures with pointers to other structures, we can declare pointer types explicitly:

```
>>> typeOf(p) == Ptr(POINT)
True
>>> BIZARRE = GcStruct('bizarre', ('p1', Ptr(POINT)), ('p2', Ptr(POINT)))
>>> b = malloc(BIZARRE)
>>> b.p1
<* None>
>>> b.p1 = b.p2 = p
>>> b.p1.y = 42
>>> b.p2.y
42
```

The world of low-level types is more complicated than integers and GcStructs, though. The next pages are a reference guide.

Primitive Types

Signed a signed integer in one machine word (a `long`, in C)

Unsigned a non-signed integer in one machine word (`unsigned long`)

Float a 64-bit float (`double`)

Char a single character (`char`)

Bool a boolean value

Void a constant. Meant for variables, function arguments, structure fields, etc. which should disappear from the generated code.

Structure Types

Structure types are built as instances of `rpython.rtyper.lltypesystem.lltype.Struct`:

```
MyStructType = Struct('somename',  ('field1', Type1), ('field2', Type2)...)
MyStructType = GcStruct('somename',  ('field1', Type1), ('field2', Type2)...)
```

This declares a structure (or a Pascal `record`) containing the specified named fields with the given types. The field names cannot start with an underscore. As noted above, you cannot directly manipulate structure objects, but only pointer to structures living in the heap.

By contrast, the fields themselves can be of primitive, pointer or container type. When a structure contains another structure as a field we say that the latter is "inlined" in the former: the bigger structure contains the smaller one as part of its memory layout.

A structure can also contain an inlined array (see below), but only as its last field: in this case it is a "variable-sized" structure, whose memory layout starts with the non-variable fields and ends with a variable number of array items. This number is determined when a structure is allocated in the heap. Variable-sized structures cannot be inlined in other structures.

GcStructs have a platform-specific GC header (e.g. a reference counter); only these can be dynamically malloc()ed. The non-GC version of Struct does not have any header, and is suitable for being embedded ("inlined") inside other structures. As an exception, a GcStruct can be embedded as the first field of a GcStruct: the parent structure uses the same GC header as the substructure.

Array Types

An array type is built as an instance of `rpython.rtyper.lltypesystem.lltype.Array`:

```
MyIntArray = Array(Signed)
MyOtherArray = Array(MyItemType)
MyOtherArray = GcArray(MyItemType)
```

Or, for arrays whose items are structures, as a shortcut:

```
MyArrayType = Array(('field1', Type1), ('field2', Type2)...)
```

You can build arrays whose items are either primitive or pointer types, or (non-GC non-varsize) structures.

GcArrays can be malloc()ed. The length must be specified when malloc() is called, and arrays cannot be resized; this length is stored explicitly in a header.

The non-GC version of Array can be used as the last field of a structure, to make a variable-sized structure. The whole structure can then be malloc()ed, and the length of the array is specified at this time.

Pointer Types

As in C, pointers provide the indirection needed to make a reference modifiable or sharable. Pointers can only point to a structure, an array or a function (see below). Pointers to primitive types, if needed, must be done by pointing to a structure with a single field of the required type. Pointer types are declared by:

```
Ptr(TYPE)
```

At run-time, pointers to GC structures (GcStruct, GcArray) hold a reference to what they are pointing to. Pointers to non-GC structures that can go away when their container is deallocated (Struct, Array) must be handled with care: the bigger structure of which they are part of could be freed while the Ptr to the substructure is still in use. In general, it is a good idea to avoid passing around pointers to inlined substructures of malloc()ed structures. (The testing implementation of rpython/rtyper/lltypesystem/lltype.py checks to some extent that you are not trying to use a pointer to a structure after its container has been freed, using weak references. But pointers to non-GC structures are not officially meant to be weak references: using them after what they point to has been freed just crashes.)

The malloc() operation allocates and returns a Ptr to a new GC structure or array. In a refcounting implementation, malloc() would allocate enough space for a reference counter before the actual structure, and initialize it to 1. Note that the testing implementation also allows malloc() to allocate a non-GC structure or array with a keyword argument `immortal=True`. Its purpose is to declare and initialize prebuilt data structures which the code generators will turn into static immortal non-GC'ed data.

Function Types

The declaration:

```
MyFuncType = FuncType([Type1, Type2, ...], ResultType)
```

declares a function type taking arguments of the given types and returning a result of the given type. All these types must be primitives or pointers. The function type itself is considered to be a "container" type: if you wish, a function contains the bytes that make up its executable code. As with structures and arrays, they can only be manipulated through pointers.

The testing implementation allows you to "create" functions by calling `functionptr(TYPE, name, **attrs)`. The extra attributes describe the function in a way that isn't fully specified now, but the following attributes *might* be present:

_callable a Python callable, typically a function object.

graph the flow graph of the function.

Opaque Types

Opaque types represent data implemented in a back-end specific way. This data cannot be inspected or manipulated.

There is a predefined opaque type `RuntimeTypeInfo`; at run-time, a value of type `RuntimeTypeInfo` represents a low-level type. In practice it is probably enough to be able to represent GcStruct and GcArray types. This is useful if we have a pointer of type `Ptr(S)` which can at run-time point either to a malloc'ed `S` alone, or to the `S` first field of a larger malloc'ed structure. The information about the exact larger type that it points to can be computed or passed around as a `Ptr(RuntimeTypeInfo)`. Pointer equality on `Ptr(RuntimeTypeInfo)` can be used to check the type at run-time.

At the moment, for memory management purposes, some back-ends actually require such information to be available at run-time in the following situation: when a GcStruct has another GcStruct as its first field. A reference-counting back-end needs to be able to know when a pointer to the smaller structure actually points to the larger one, so that it can also decref the extra fields. Depending on the situation, it is possible to reconstruct this information without having to store a flag in each and every instance of the smaller GcStruct. For example, the instances of a class hierarchy can be implemented by nested GcStructs, with instances of subclasses extending instances of parent classes by embedding the parent part of the instance as the first field. In this case, there is probably already a way to know the run-time class of the instance (e.g. a vtable pointer), but the back-end cannot guess this. This is the reason for which `RuntimeTypeInfo` was originally introduced: just after the GcStruct is created, the function attachRuntimeTypeInfo() should be called to attach to the GcStruct a low-level function of signature `Ptr(GcStruct) -> Ptr(RuntimeTypeInfo)`. This function will be compiled by the back-end and automatically called at run-time. In the above example, it would follow the vtable pointer and fetch the opaque `Ptr(RuntimeTypeInfo)` from the vtable itself. (The reference-counting GenC back-end uses a pointer to the deallocation function as the opaque `RuntimeTypeInfo`.)

Implementing RPython types

As hinted above, the RPython types (e.g. 'list') are implemented in some "restricted-restricted Python" format by manipulating only low-level types, as provided by the testing implementation of malloc() and friends. What occurs then is that the same (tested!) very-low-level Python code – which looks really just like C – is then transformed into a flow graph and integrated with the rest of the user program. In other words, we replace an operation like `add` between two variables annotated as SomeList, with a `direct_call` operation invoking this very-low-level list concatenation.

This list concatenation flow graph is then annotated as usual, with one difference: the annotator has to be taught about malloc() and the way the pointer thus obtained can be manipulated. This generates a flow graph which is hopefully completely annotated with SomePtr() annotation. Introduced just for this case, SomePtr maps directly to a low-level pointer type. This is the only change needed to the Annotator to allow it to perform type inference of our very-low-level snippets of code.

See for example rpython/rtyper/rlist.py.

HighLevelOp interface

In the absence of more extensive documentation about how RPython types are implemented, here is the interface and intended usage of the 'hop' argument that appears everywhere. A 'hop' is a HighLevelOp instance, which represents a single high-level operation that must be turned into one or several low-level operations.

`hop.llops` A list-like object that records the low-level operations that correspond to the current block's high-level operations.

hop.genop(opname, list_of_variables, resulttype=resulttype) Append a low-level operation to `hop.llops`. The operation has the given opname and arguments, and returns the given low-level resulttype. The arguments should come from the `hop.input*()` functions described below.

hop.gendirectcall(ll_function, var1, var2...) Like hop.genop(), but produces a `direct_call` operation that invokes the given low-level function, which is automatically annotated with low-level types based on the input arguments.

hop.inputargs(r1, r2...) Reads the high-level Variables and Constants that are the arguments of the operation, and convert them if needed so that they have the specified representations. You must provide as many representations as the operation has arguments. Returns a list of (possibly newly converted) Variables and Constants.

hop.inputarg(r, arg=i) Same as inputargs(), but only converts and returns the ith argument.

hop.inputconst(lltype, value) Returns a Constant with a low-level type and value.

Manipulation of HighLevelOp instances (this is used e.g. to insert a 'self' implicit argument to translate method calls):

hop.copy() Returns a fresh copy that can be manipulated with the functions below.

hop.r_s_popfirstarg() Removes the first argument of the high-level operation. This doesn't really changes the source SpaceOperation, but modifies 'hop' in such a way that methods like inputargs() no longer see the removed argument.

hop.v_s_insertfirstarg(v_newfirstarg, s_newfirstarg) Insert an argument in front of the hop. It must be specified by a Variable (as in calls to hop.genop()) and a corresponding annotation.

hop.swap_fst_snd_args() Self-descriptive.

Exception handling:

hop.has_implicit_exception(cls) Checks if hop is in the scope of a branch catching the exception 'cls'. This is useful for high-level operations like 'getitem' that have several low-level equivalents depending on whether they should check for an IndexError or not. Calling has_implicit_exception() also has a side-effect: the rtyper records that this exception is being taken care of explicitly.

hop.exception_is_here() To be called with no argument just before a llop is generated. It means that the llop in question will be the one that should be protected by the exception catching. If has_implicit_exception() was called before, then exception_is_here() verifies that *all* except links in the graph have indeed been checked for with an has_implicit_exception(). This is not verified if has_implicit_exception() has never been called – useful for 'direct_call' and other operations that can just raise any exception.

hop.exception_cannot_occur() The RTyper normally verifies that exception_is_here() was really called once for each high-level operation that is in the scope of exception-catching links. By saying exception_cannot_occur(), you say that after all this particular operation cannot raise anything. (It can be the case that unexpected exception links are attached to flow graphs; e.g. any method call within a `try:finally:` block will have an Exception branch to the finally part, which only the RTyper can remove if exception_cannot_occur() is called.)

The LLInterpreter

The LLInterpreter is a simple piece of code that is able to interpret flow graphs. This is very useful for testing purposes, especially if you work on the RPython Typer. The most useful interface for it is the `interpret` function in the file rpython/rtyper/test/test_llinterp.py. It takes as arguments a function and a list of arguments with which the function

is supposed to be called. Then it generates the flow graph, annotates it according to the types of the arguments you passed to it and runs the LLInterpreter on the result. Example:

```
def test_invert():
    def f(x):
        return ~x
    res = interpret(f, [3])
    assert res == ~3
```

Furthermore there is a function `interpret_raises` which behaves much like `py.test.raises`. It takes an exception as a first argument, the function to be called as a second and the list of function arguments as a third. Example:

```
def test_raise():
    def raise_exception(i):
        if i == 42:
            raise IndexError
        elif i == 43:
            raise ValueError
        return i
    res = interpret(raise_exception, [41])
    assert res == 41
    interpret_raises(IndexError, raise_exception, [42])
    interpret_raises(ValueError, raise_exception, [43])
```

Garbage Collection in RPython

Contents

Introduction

The overview and description of our garbage collection strategy and framework can be found in the EU-report on this topic. Please refer to that file for an old, but still more or less accurate, description. The present document describes the specific garbage collectors that we wrote in our framework.

Garbage collectors currently written for the GC framework

Reminder: to select which GC you want to include in a translated RPython program, use the `--gc=NAME` option of `translate.py`. For more details, see the overview of command line options for translation.

The following overview is written in chronological order, so the "best" GC (which is the default when translating) is the last one below.

Mark and Sweep

Classical Mark and Sweep collector. Also contained a lot of experimental and half-unmaintained features. Was removed.

Semispace copying collector

Two arenas of equal size, with only one arena in use and getting filled with new objects. When the arena is full, the live objects are copied into the other arena using Cheney's algorithm. The old arena is then cleared. See rpython/memory/gc/semispace.py.

On Unix the clearing is done by reading `/dev/zero` into the arena, which is extremely memory efficient at least on Linux: it lets the kernel free the RAM that the old arena used and replace it all with allocated-on-demand memory.

The size of each semispace starts at 8MB but grows as needed when the amount of objects alive grows.

Generational GC

This is a two-generations GC. See rpython/memory/gc/generation.py.

It is implemented as a subclass of the Semispace copying collector. It adds a nursery, which is a chunk of the current semispace. Its size is computed to be half the size of the CPU Level 2 cache. Allocations fill the nursery, and when it is full, it is collected and the objects still alive are moved to the rest of the current semispace.

The idea is that it is very common for objects to die soon after they are created. Generational GCs help a lot in this case, particularly if the amount of live objects really manipulated by the program fits in the Level 2 cache. Moreover, the semispaces fill up much more slowly, making full collections less frequent.

Hybrid GC

This is a three-generations GC.

It is implemented as a subclass of the Generational GC. The Hybrid GC can handle both objects that are inside and objects that are outside the semispaces ("external"). The external objects are not moving and collected in a mark-and-sweep fashion. Large objects are allocated as external objects to avoid costly moves. Small objects that survive for a long enough time (several semispace collections) are also made external so that they stop moving.

This is coupled with a segregation of the objects in three generations. Each generation is collected much less often than the previous one. The division of the generations is slightly more complicated than just nursery / semispace / external; see the diagram at the start of the source code, in rpython/memory/gc/hybrid.py.

Mark & Compact GC

Killed in trunk. The following documentation is for historical purposes only.

Inspired, at least partially, by Squeak's garbage collector, this is a single-arena GC in which collection compacts the objects in-place. The main point of this GC is to save as much memory as possible (to be not worse than the Semispace), but without the peaks of double memory usage during collection.

Unlike the Semispace GC, collection requires a number of passes over the data. This makes collection quite slower. Future improvements could be to add a nursery to Mark & Compact in order to mitigate this issue.

During a collection, we reuse the space in-place if it is still large enough. If not, we need to allocate a new, larger space, and move the objects there; however, this move is done chunk by chunk, and chunks are cleared (i.e. returned to the OS) as soon as they have been moved away. This means that (from the point of view of the OS) a collection will never cause an important temporary growth of total memory usage.

More precisely, a collection is triggered when the space contains more than N*M bytes, where N is the number of bytes alive after the previous collection and M is a constant factor, by default 1.5. This guarantees that the total memory usage of the program never exceeds 1.5 times the total size of its live objects.

The objects themselves are quite compact: they are allocated next to each other in the heap, separated by a GC header of only one word (4 bytes on 32-bit platforms) and possibly followed by up to 3 bytes of padding for non-word-sized objects (e.g. strings). There is a small extra memory usage during collection: an array containing 2 bytes per surviving object is needed to make a backup of (half of) the surviving objects' header, in order to let the collector store temporary relation information in the regular headers.

Minimark GC

This is a simplification and rewrite of the ideas from the Hybrid GC. It uses a nursery for the young objects, and mark-and-sweep for the old objects. This is a moving GC, but objects may only move once (from the nursery to the old stage).

The main difference with the Hybrid GC is that the mark-and-sweep objects (the "old stage") are directly handled by the GC's custom allocator, instead of being handled by malloc() calls. The gain is that it is then possible, during a major collection, to walk through all old generation objects without needing to store a list of pointers to them. So as a first approximation, when compared to the Hybrid GC, the Minimark GC saves one word of memory per old object.

There are a number of environment variables that can be tweaked to influence the GC. (Their default value should be ok for most usages.)

In more detail:

- The small newly malloced objects are allocated in the nursery (case 1). All objects living in the nursery are "young".
- The big objects are always handled directly by the system malloc(). But the big newly malloced objects are still "young" when they are allocated (case 2), even though they don't live in the nursery.
- When the nursery is full, we do a minor collection, i.e. we find which "young" objects are still alive (from cases 1 and 2). The "young" flag is then removed. The surviving case 1 objects are moved to the old stage. The dying case 2 objects are immediately freed.
- The old stage is an area of memory containing old (small) objects. It is handled by rpython/memory/gc/minimarkpage.py. It is organized as "arenas" of 256KB or 512KB, subdivided into "pages" of 4KB or 8KB. Each page can either be free, or contain small objects of all the same size. Furthermore at any point in time each object location can be either allocated or freed. The basic design comes from `obmalloc.c` from CPython (which itself comes from the same source as the Linux system malloc()).
- New objects are added to the old stage at every minor collection. Immediately after a minor collection, when we reach some threshold, we trigger a major collection. This is the mark-and-sweep step. It walks over *all* objects (mark), and then frees some fraction of them (sweep). This means that the only time when we want to free objects is while walking over all of them; we never ask to free an object given just its address. This allows some simplifications and memory savings when compared to `obmalloc.c`.

- As with all generational collectors, this GC needs a write barrier to record which old objects have a reference to young objects.
- Additionally, we found out that it is useful to handle the case of big arrays specially: when we allocate a big array (with the system malloc()), we reserve a small number of bytes before. When the array grows old, we use the extra bytes as a set of bits. Each bit represents 128 entries in the array. Whenever the write barrier is called to record a reference from the Nth entry of the array to some young object, we set the bit number `(N/128)` to 1. This can considerably speed up minor collections, because we then only have to scan 128 entries of the array instead of all of them.
- As usual, we need special care about weak references, and objects with finalizers. Weak references are allocated in the nursery, and if they survive they move to the old stage, as usual for all objects; the difference is that the reference they contain must either follow the object, or be set to NULL if the object dies. And the objects with finalizers, considered rare enough, are immediately allocated old to simplify the design. In particular their `__del__` method can only be called just after a major collection.
- The objects move once only, so we can use a trick to implement id() and hash(). If the object is not in the nursery, it won't move any more, so its id() and hash() are the object's address, cast to an integer. If the object is in the nursery, and we ask for its id() or its hash(), then we pre-reserve a location in the old stage, and return the address of that location. If the object survives the next minor collection, we move it there, and so its id() and hash() are preserved. If the object dies then the pre-reserved location becomes free garbage, to be collected at the next major collection.

The exact name of this GC is either *minimark* or *incminimark*. The latter is a version that does major collections incrementally (i.e. one major collection is split along some number of minor collections, rather than being done all at once after a specific minor collection). The default is *incminimark*, as it seems to have a very minimal impact on performance and memory usage at the benefit of avoiding the long pauses of *minimark*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[BRETT] Brett Cannon, Localized Type Inference of Atomic Types in Python, http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.3231

Index