Continuations and Stackless Python

Or "How to change a Paradigm of an existing Program"

Christian Tismer
Virtual Photonics GmbH
mailto:tismer@tismer.com

Abstract

In this paper, an implementation of "Stackless Python" (a Python which does not keep state on the C stack) is presented. Surprisingly, the necessary changes affect just a small number of C modules, and a major rewrite of the C library can be avoided. The key idea in this approach is a paradigm change for the Python code interpreter that is not easy to understand in the first place. Recursive interpreter calls are turned into tail recursion, which allows deferring evaluation by pushing frames to the frame stack, without the C stack involved.

By decoupling the frame stack from the C stack, we now have the ability to keep references to frames and to do non-local jumps. This turns the frame stack into a tree, and every leaf of the tree can now be a jump target. While exploring this idea, we will recognize ordinary function calls and returns to be just special cases of so-called continuations, and we will learn about them as a really simple idea that covers all kinds of program flow.

Instead of an implementation of coroutines and generators as C extensions, we will see how they can be expressed in Python, using the continuation module. Since the theory of continuations is not very broadly known, a small introduction is given.

1 Introduction

There have been a few attempts to implement generators and coroutines in Python. These were either of low performance, since they were implemented using threads, or limited by Python's recursive interpreter layout, which prevented switching between frames liberally. Changing Python to become non-recursive was considered a major, difficult task. Actually it was true, and I had to change truth before I could continue. ☺

2 Continuations

2.1 What is a Continuation?

Many attempts to explain continuations can be found in the literature[5-10], more or less hard to understand. The following is due to Jeremy Hylton, and I like it the best. Imagine a very simple series of statements:

```
x = 2; y = x + 1; z = x * 2
```

In this case, the continuation of `x=2` is `y=x+1; z=x*2`. You might think of the second and third assignments as a function (forgetting about variable scope for the moment). That function is the continuation. In essence, every single line of code has a continuation that represents the entire future execution of the program.

Perhaps you remember the phrase "*Goto is considered harmful*". Functional programming's answer turns bad into good and makes that goto into a key idea: A continuation is a means of capturing the control flow of the program and manipulating it; it is a primitive idea that can be used to synthesize any form of control flow.

Although we do not intend to re-invent the existing control structures, a simple example might help.

```
def looptest(n):
    this = continuation.current()
    k = this.update(n)
    if k:
        this(k-1)
    else:
        del this.link
```

Without going into details, our continuation `this` is prepared to jump into exactly the situation where the boxed assignment takes place. The call `this(k-1)`

Generators using Continuations

```
import continuation
Killed = 'Generator.Killed'
get_caller = continuation.caller

class Generator:
  def __init__(self, func, *args, **kw):
    self.func = func
    self.args = args
    self.kw = kw
    self.done = 0
    self.killed = 0
    self.producer = self._start

  def _start(self, dummy=None):
    if not self.killed:
      try:
        apply(self.func, (self,) +
                      self.args, self.kw)
        raise EOFError, "no more values"
      finally:
        self.kill()

  def put(self, value):
    if self.killed:
      raise TypeError,
        'put() called on killed generator'
    self.producer = get_caller()
    self.consumer(value)

  def get(self):
    if self.killed:
      raise TypeError,
        'get() called on killed generator'
    self.consumer = get_caller()
    self.producer()

  def kill(self):
    if self.killed:
      raise TypeError, 'kill() called on
   killed generator'
    hold = self.func, self.args
    self.__dict__.clear()
    self.func, self.args = hold
    self.killed = 1

  def clone(self):
    return Generator(self.func, self.args)

def count(g, start=1, limit=0):
  # just to measure switching time
  i = start
  while i != limit:
    g.put(i)
    i = i+1
    #if i==42: g.kill()
  g.done = 1
```

moves us immediately back into the context of that assignment, just with a new value for `k`. This results in repeated execution of this piece of code, forming a while loop.

In comparison to our first example, we did not use a continuation at the bounds of a statement, but it was located in the middle of a function call, in the situation of parameter passing. This kind of continuation is most useful since we can provide a parameter and get a result back. We will see more about this in the next chapter. In fact, continuations are not limited to statements or function calls. Every opcode of the (virtual) machine has a continuation.

"The current continuation at any point in the execution of a program is an abstraction of the *rest of the program"* [5]. Another wording is "what to do next "[6]

But in terms of Python frames, a continuation is nothing more than a frame object, together with its linked chain of calling frames. In order to make this callable, handy, and to protect frames from being run more than once at a time, we wrap them into continuation objects, which take care about that. Before discussing the details in chapter 4, let's now build a really useful example.

3 Generators in Python

Instead of a direct implementation of coroutines and generators, I decided to use the most general approach: Implement continuations as first class callable objects, and express generators and coroutines in Python.

3.1 Generators expressed with Threads

Few people might know that there is a generator implementation using threads. It can be found in the source distribution under demo/threads/Generator.py and has not been changed since 1994, when there was a longer discussion of generators on C.L.P [2]. A slightly modified version for Python 1.5.2 can be found in my distribution for comparison. The basic idea is to prove `get()` and `put()`functions which communicate with a consumer and a producer thread.

3.2 Generators done with Continuations

Now we implement Generators using continuations.

Instead of communicating with threads, get() and put() jump directly into the context of the consumer/producer function. The jumps are expressed by calls of continuations, which leave the current context and jump over to the context that is stored in the continuation object.

Let us look at the `put` method. The call of `get_caller` returns a continuation object, which is a snapshot of `put`'s current caller. This is saved away in `self.producer` and will be used later by the get method. A call of this object performs by default a jump to the saved context.

It is crucial to understand how these objects exchange values: A continuation objects always accepts a value as its argument, and it returns a value as its result. It is just the question *where or when* the value is returned. When `get_caller()` is executed in `put()`, it catches the calling frame in a situation, where it has been supplied with a value for the call.

In our `count()` example, this is the expression `g.put(i)`, and the continuation is saved as `producer()` function, waiting to run the next loop cycle. But the passed value is given to the `consumer()` function that was captured in the context of a `get()` call, and the value appears there as the function result. I admit that this is a little hard to understand. Basically, you have understood the idea of a *coroutine* switch, and in fact, this generator is made of two coroutines. More important: You have understood how continuations act as the building block for new control structures.

4 Continuation Module

The continuation module was always my real target application, and it has been rewritten the fifth time now. This was like iteration, since I had to learn what continuations are, what they need, and how Stackless Python must be to support it, without including it. Meanwhile it has grown quite complicated and has undergone many optimizations, and there is no chance to cover it all in this article.

4.1 The Problem

After you learned what a continuation is and how easy this all is to understand, I have to disappoint you a little. The implementation isn't easy at all, due to the fact that Pythons frames are not safe when used as continuations. Python frames hold besides some other things the local variables, exception state, current code location and the evaluation stack for expressions.

And that is the problem. Whenever a frame is run, its state changes, especially the expression stack and the code location. But we wanted to **save** a continuation for later re-use. What we have so far is just enough for the little coroutine idea which I sketched in the beginning. But what happens if we are able to return to the same frame twice, intended or not? (Figure 1) We would need a copy of a frame. But a copy would give a problem if some other structure were pointing to the frame. This was one of the moments where I considered giving up. ☺

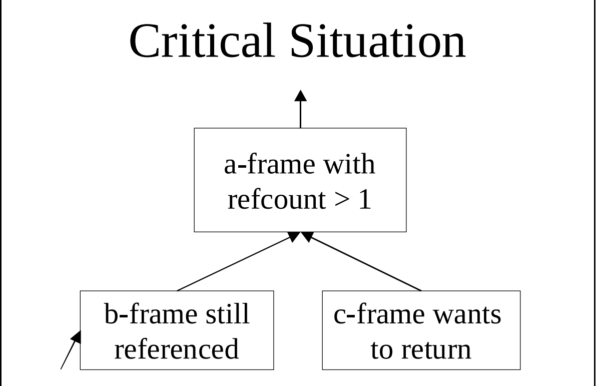


Figure 1: Multiple Return

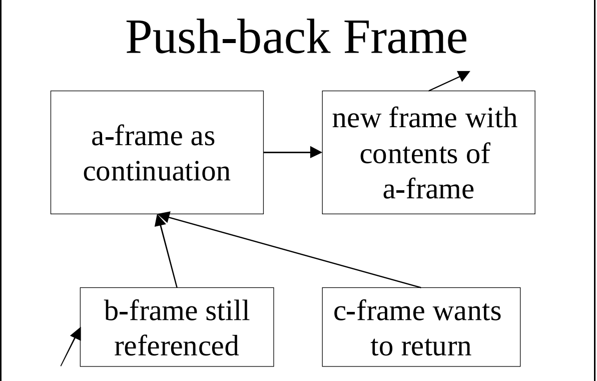


Figure 2: Frame Mutation

4.2 How it Works

In order to solve this problem, I introduced another kind of frames, continuation frames. The idea is simple: While being of the same structure as an ordinary frame, it has a different execute function. It is named `throw_continuation`, and that's exactly its only purpose: Restore the state of a real frame and run it. (Figure 2)

The other half of the solution is not to insert a copy of the real frame in front of it into the frame chain, but to create it **behind** the real frame, then turn the real frame into a continuation frame, and make the pushed back version into the real one.

This was fine for the first three versions of continuation module. But it turned quickly into a problem to keep track of chain growth, and normalization of the resulting mixture of real frames and continuation frames was necessary.

Furthermore, I needed more control over the frame linkage and be able to predict necessary continuation frame creation as much as possible. So here comes the third half of the solution…(Figure 3)

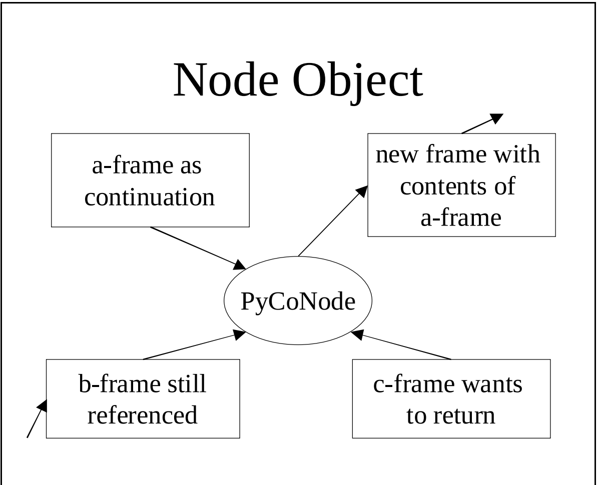


Figure 3: Nodes Collect References

This structure adds a level of indirection, but the cost is very low. Adding a new continuation frame is not much more than the old pushback scheme, together with some pointer and refcount adjustment.

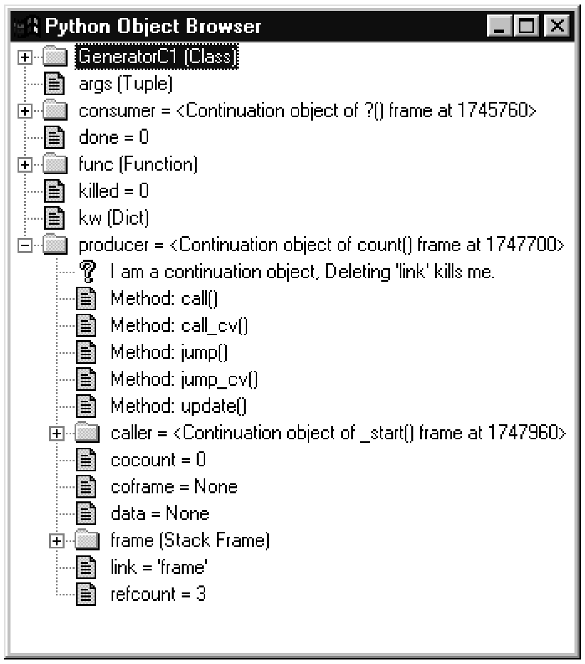
4.3 How fast? Is it fast? Size?

It is. Some more advanced versions of the `get` and `put` functions in our example run three times faster than the threads implementation. The following recursive call of one(!) frame has the same speed as the proper implementation via functions:

```
def rectest2(n):
    this = continuation.current()
    call = this.call
    k = this.update(n)
    if k:
        call(k-1)
    else:
        del this.link
```

On size: The size of our continuation is just one frame, some 400 bytes maybe. You can easily create 10000 of these objects. Threads come at the cost of an extra C stack each, which is a megabyte on some machines.

We close the continuation story with a look through PythonWin's Browser.



```
GeneratorC1 (Class)
args (Tuple)
consumer = <Continuation object of ?() frame at 1745760>
done = 0
func (Function)
killed = 0
kw (Dict)
producer = <Continuation object of count() frame at 1747700>
    I am a continuation object, Deleting 'link' kills me.
    Method: call()
    Method: call_cv()
    Method: jump()
    Method: jump_cv()
    Method: update()
    caller = <Continuation object of _start() frame at 1747960>
    cocount = 0
    coframe = None
    data = None
    frame (Stack Frame)
    link = 'frame'
    refcount = 3
```

Figure 4: Continuation Browser View

5 Stackless Python

After we had continuations as the real fruit of this work, let's come to Stackless Python and its implementation. Continuations were the reason to build Stackless Python; without it there is no chance to implement them in a machine independent manner.

5.1 What does it mean to be stackless?

The standard Python code interpreter is written in C. Every Python action is somehow performed by some C code. Whenever a piece of Python code is executed, a new incarnation of the interpreter loop is created and executed by a recursive call.

First of all, let's see what it means to have a C stack. Consider a Python function `a()`, which calls a

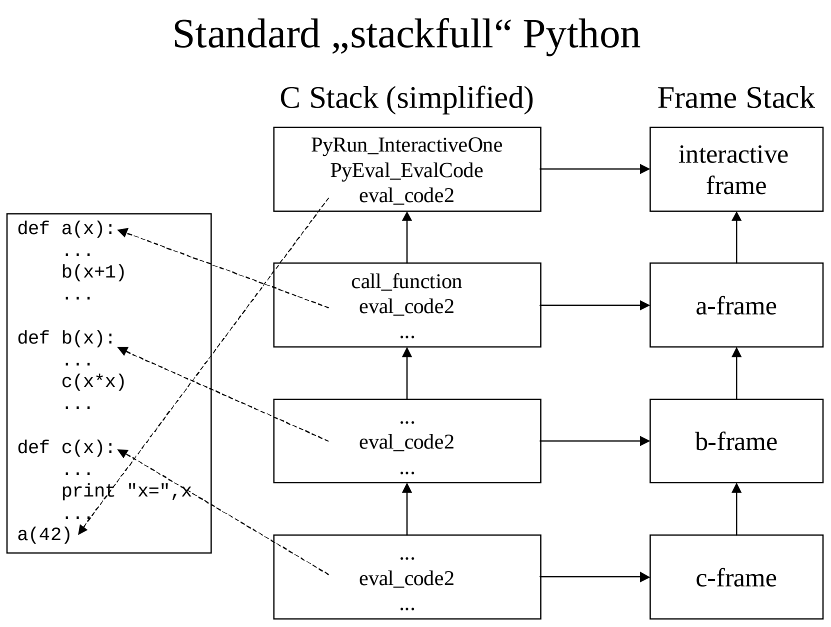


Figure 5: Standard Nested Call

Python function `b()`, which calls a Python function `c()`. In the context of function `c()`, the C interpreters of all three functions are still alive. They are keeping their state on the C stack. When the Python functions come to an end or an exception is raised, the C functions are popped off the C stack. This is called "unwinding the stack".

In that sense, Python is not so different from other C programs, which are usually all stack based. But this is not the full story, since Python does quite a lot more than using the C stack.

Every running piece of Python code also has an associated Frame object. A Frame object is something like a running instance of a code object. Frames are used to hold local and global variables, to maintain the value stack of the byte code interpreter, and some other housekeeping information. (Figure 5)

These Frames are chained together in a last-in/first-out manner. They make up a stack by themselves. And they do this in a way quite similar to the C stack.

5.2 Why the C stack should vanish

The C stack limits recursion depth. The C stack has a fixed size, and therefore the stack cannot grow infinitely. The frame stack is not limited. I think it is good to keep recursion depth limited, but not as a matter of an implementation detail. This should be a user option. The C stack should be reserved for C modules that really need it.

The C stack holds references to objects. Well, `eval_code` is clean in this respect, but other C functions may pile up on the stack as well. References to function parameters are kept on the C stack until a function returns.

The C stack holds execution state. Information of the frame's current program and stack counter is hidden in variables on the C stack and cannot be modified. The C stack therefore limits the possible order of execution.

Removing the C stack is cheap. As I try to show in this paper, the implementation effort is much smaller than one would think. Execution speed is nearly the same. But the new possibilities of having an explicit stack are many, and they cannot even be tried in the moment.

Coroutines can be incredibly fast. As we will see in the following, by decoupling the frames from the C stack, coroutines can be implemented so fast, that it might become feasible to use them for quite a number of problems. Switching between two coroutine frames needs just a built-in function call that is much cheaper than calling a Python function.

PERL has already no longer a C stack. They will have some good reasons for this. Python has no reason to be limited and stay behind.

To conclude: I want to provide the same level of flexibility for the C API as we have already in Python, open Python for new, efficient algorithmic approaches, at low implementation and runtime cost.

5.3 Targets Evolution and History

Everything started with a discussion of coroutines in Python on the python-dev mailing list, initiated by Sam Rushing. In turn, a series of requirements materialized as consequences.

Get rid of the C stack. This became clear quite quickly. In order to be able to rearrange frames, the C stack is in the way.

Allow for pluggable interpreters. After a first implementation of a tail-recursive code interpreter, I realized that *tail recursion* (chapter 6.2) is necessary but not sufficient in order to make map, filter and reduce stackless. Instead, we have to build tiny custom interpreters. As a consequence, every extension module can provide its own interpreter for any frame.

Do the minimum changes to existing code. It was never clear (and still is not) whether the Stackless Python patches would make it into the distribution at some time. Furthermore, Python will undergo a lot of changes in the future. The fewer changes are done now, the easier it will be to incorporate them into future versions. Alternatively, keeping running my own parallel python version becomes less effort.

Stay compatible to compiled extension modules. Stackless Python should be a drop-in replacement for the official distribution. Every existing compiled extension module should work, provided it does not rely on the exact frame layout. The proof of concept was to replace python15.dll and have PythonWin on top of it.

Learn how stackless extension modules work. Although my major goal was to provide stackless behavior for Python code, the requirement to figure out how to make stackless-aware extension modules came up quickly. Suppose that a fast XML parser is written in Python, which makes use of rapid context switching. In order to write a much faster version in C, we need to be able to model the same stackless behavior. Fortunately, the proof of concept comes for free by stackless map.

Implement first class continuations. As a proof of concept, and to learn how all of this must work, I had not only to write an appropriate Python version, but also check the strength of the concept by implementing first class continuations. In fact, a number of improvements had to be done in order to get continuations to work.

Keep continuations out of the core. While the Python core was changed to support continuations, it became clear that the majority of existing and future code would probably make no use of continuations. Therefore, the implementation had to have the smallest possible runtime impact on the Python core, and continuations had to be a dynamic module. The evolution of the continuation module resulted in several changes to Stackless Python, which has reached version 1.0 at the time of writing. It appears to be stable and is used in production code.

6 The Paradigm Shift

We will see in the next chapter, how Stackless Python was developed. Since this was a long, iterative process, the key ideas appear in the order of invention which is not optimal for the readers understanding. The essentials can be summarized as the following 3 axioms:

6.1 Time of Frame Execution

The paradigm in `ceval.c` for running a Python function is to build a frame for the code object, to put all the parameters in place and then run `eval_code2` and wait for it to return.

The transformation: Making sure to run all frames in their correct order does not imply that we must call the interpreter function from the current C-stack nesting level. If we can avoid any C-stack related post-processing, unwinding the stack is possible before the frame execution. This implies the next shift:

6.2 Lifetime of Parameters

The function parameters in standard Python are kept alive by the caller. This means that the caller of `eval_code2` has to wait until the call is finished. After the call the reference to the parameter tuple is removed.

Thinking in frames instead of recursive calls, it is obvious that parameters should be kept alive as long as the frame exists. Therefore, we put a reference to the parameter tuple into a frame field. This reference will be automatically removed at the best possible time: when the frame is disposed.

Since we removed the last cleanup task from the recursive C function call, there is nothing left do do for it, and it may return to its caller **before** the frame is run. This pattern of a function invocation as the last action of the caller is also known as **tail recursion**. Tail recursive calls are logical identical to jumps, and unwinding our C stack before running the next frame is our C equivalent of directly jumping back into the toplevel frame dispatcher.

6.3 Third System State

Standard Python has the throughout semantics of function calls that either a `PyObject` is returned, or `NULL` which signals an exception. These are the two essential system states while returning from a frame.

By introduction of a special object as a return value with a different meaning, we can control the C-stack and request to unwind it, before the next frame is run. Since this object is compatible with all other Python objects, this protocol change isn't visible to most of the involved code. Only the C functions which deal with running a new frame needed to be changed. This is the *third system state*.

Return Value	**System State**
`NULL`	An error occurred
`Py_UnwindToken`	Dispatch a frame
Other `PyObject`	Return this value

The above three ideas were central, necessary and sufficient for Stackless Python. All other details are consequences from these axioms.

7 Implementing Stackless Python

During the implementation, a number of rules showed up as being essential. They are quite simple:

1. Avoid recursive calls into the interpreter.

2. Store all runtime information in frames.

3. Allow frames to be restarted. Just **add** to the frame structure, don't break anything.

4. Store the interpreter function in the frames.

5. Have one central frame handler, turn eval_code2 into just one interpreter

6. Provide new functions but keep backward compatible stub functions.

Rule 1 is obvious but looks hard to fulfill.

Rule 2 is obvious, too. The frame chain becomes our true stack. No information should be hidden in the C stack while a Python function is being called.

Rule 3 is one of the key ideas. We come to it soon.

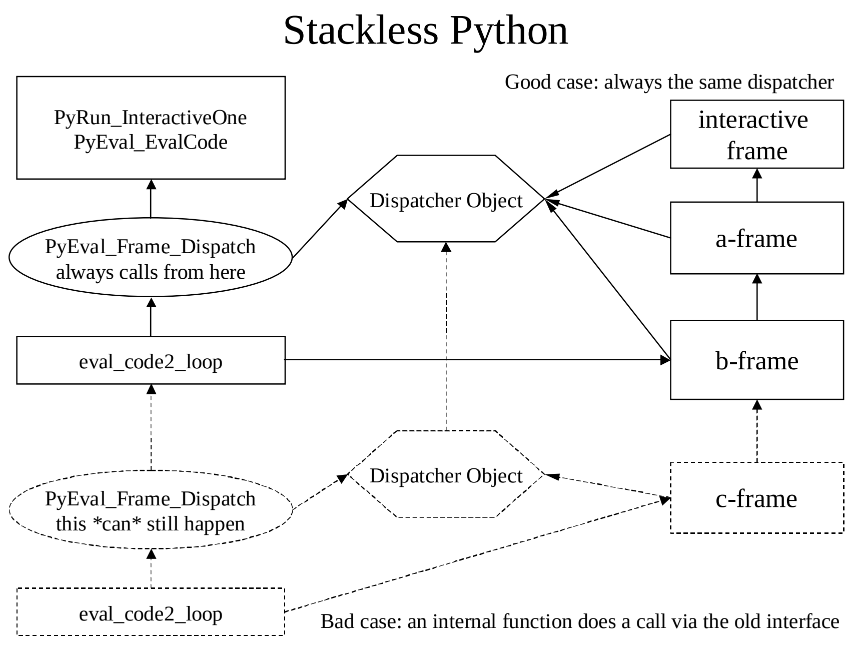
Rule 4 is just a consequent move of information into frames. If a frame is to know everything about the running code, then it also should know which the interpreter is. As a side effect, frames can call each other at wilt, without having to know more than that this is just a runable frame.

Rule 5 is a consequence of splitting responsibilities. The central frame handler is the very minimum that we need. It does nothing more than juggling the frame calls and passing results. With a single concept of frames, we can run any kind of interpreters together, as long as they obey the Python Object Protocol.

Well, Rule 6 is clear. We want to be still standard Python with the standard C API just extended.

7.1 Problem analysis

Let's have a look into the code (Figure 8). The first file to inspect is `ceval.c`. We compare the old and the new version. If you try to figure out what happens on a Python function call, you will first think it is impossible to do that without the C stack.



Stackless Python
Good case: always the same dispatcher
PyRun_InteractiveOne
PyEval_EvalCode
interactive frame
Dispatcher Object
a-frame
PyEval_Frame_Dispatch always calls from here
eval_code2_loop
b-frame
Dispatcher Object
c-frame
PyEval_Frame_Dispatch this *can* still happen
eval_code2_loop
Bad case: an internal function does a call via the old interface

Figure 6: Minimizing Recursive Calls

Have a look at function `eval_code2`, at the case `CALL_FUNCTION(`Figure 8`)`. Many different cases are handled here; other functions are called, like `PyEval_CallObjectWithKeywords`. There, a number of decisions are made, which in turn cause other function calls, and finally, we end up somewhere deeply nested, with either a call to `call_builtin` or `call_function`. For getting a result, either a C function is called, or a new interpreter incarnation handles the call, finally via `eval_code2` again.

In order to do this call, the calling function somehow prepares parameters, keeps a reference to the parameters, lets the evaluation happen, releases the parameters and returns the result.

My biggest fear was that I would have to rewrite all of this. But this is not true.

7.2 Problem solution

If we just avoid doing the final call to `eval_code2`, we are almost done. Please have a look into the old version of `call_function`, and compare it to the new version. Instead of calling the interpreter, we end up with the above piece of code: Prepare a frame to be run, but don't run it.

`eval_code2_setup` is a function that just prepares a new frame, in order to be run later. The frame holds all references to parameters, and the new field `f_hold_ref` takes the role to keep a reference to `arg`. Instead of actually performing the call now, `call_function` returns a special object, `Py_UnwindToken` as its value. This value is passed as a result through the actual pile of invoked C functions until it is caught in the calling interpreter (yes, we are back at the `CALL_FUNCTION` opcode). The interpreter checks this value and knows that a call is pending, so the current frame must be left, and a new frame has been put onto the stack, which should be called now.

Why is this possible, and why does it appear to be so easy?

Python is already nearly stackless. All the functions try to do nothing more than just to prepare for the next interpreter call, which will do the actual evaluation. After as code object and its parameters have been checked for errors, the runtime error checking is already deferred to the responsible interpreter. However, this does not mean that we need to call the interpreter immediately. We can defer this until we have finished the currently active C functions! Do you get the idea?

In other words: Almost all calls into the interpreter

The Frame Dispatcher
"Mario"

```
PyObject *
PyEval_Frame_Dispatch()
{
  PyObject * result;
  PyFrameObject *f;
  PyThreadState *tstate = PyThreadState_GET();
  PyDispatcherObject *self;

  f = tstate->frame;
  if (f==NULL) return NULL;

  self = PyDispatcher_New(f);
  if (self == NULL)
    return NULL;

  /* the initial frame belongs to us */
  Py_INCREF(self);
  Py_XDECREF(f->f_dispatcher);
  f->f_dispatcher = self;
  result = f->f_temp_val;
  f->f_temp_val = NULL;

  while (1) {
    result = f->f_execute(f, result) ;
    f = tstate->frame;
    if (result == Py_UnwindToken) {
      /* this is actually the topmost frame. */
      /* pick an optional return value */
      result = f->f_temp_val;
      f->f_temp_val = NULL;
      /* and mark the frame as our own */
      if (f->f_dispatcher != self) {
        Py_INCREF(self);
        Py_XDECREF(f->f_dispatcher);
        f->f_dispatcher = self;
      }
    }
    else if (f==NULL
            || f->f_dispatcher != self)
      break;
  }
  self->d_back->d_alive = 1;
/* always possible since one always exists */
  self->d_alive = 0;
  Py_DECREF(self);
  return result;
}
```

Frame Protocol:
After some frame's f_execute has been run, we always refer to the topmost tstate frame. If a frame returns the `Py_UnwindToken` object, this indicates that a different frame will be run that now belongs to the current dispatcher. The f_temp_val field holds the temporary return value since `f_execute`'s return value was occupied. Otherwise, we will bail out whenever the result becomes `NULL` or a different dispatcher is detected.

turn out to be tail recursive. After parameters have been checked and everything is prepared, there is no need to actually call the next function. It is ok to generate a frame on the frame stack, which is ready to be run by the next interpreter involved, but there is no need to do this while we are still in the current pile of active C functions. We can leave before we call.

Whenever a function believes that it is done by running eval_code2 and returning a value, it defers all error handling to "the" interpreter. What I do is nothing more than to defer the whole call to just "one" interpreter, which has the same semantics. It is just a special form of tail recursion taking place. It does not matter where the result value is checked for an exception immediately from a deeply nested function call, or later, after unwinding to the top-level interpreter incarnation. The sequence of necessary actions is driven by the frame chain and nothing else.

By following this principle, it was possible to make stackless versions of all of the interpreter functions. The conversion works like this:

When a C function calls the interpreter (the former eval_code2), it does not immediately run the new stack frame. Instead, the new frame object is created and a special token, Py_UnwindToken, is returned. Py_UnwindToken is a special return value that indicates that the current interpreter should get out of the way immediately (which means to return).

Another approach to explain this: The interpreter functions are (almost) all working with PyObjects. Return values can be NULL, or a proper PyObject. NULL has the semantics of an error that has to be handled, and it will cause the interpreters to unwind the stack by an exception. This gives us just a two-valued logic: Return values are either PyObjects or NULL. By introducing the Py_UnwindToken, I extended this to three-valued logic.

Since all these values can be passed through the existing interpreter functions, I saved a major rewrite and had just to take care to catch the right places to change.

If you are still with me, now the time has come to change your understanding of functions, stacks and return values.

7.3 The Frame Dispatcher

Let's have a look into the new, central "interpreter" function in ceval.c Actually it is no interpreter, but a function that repeatedly evaluates the top of the frame stack, using that frame's execute function.

Just try to understand the central loop. The dispatcher picks the topmost frame, the current result, and calls the frame's `f_execute` function. This dispatcher loop is the place where we always return to, regardless for what reason we ran a frame or why we left a frame.

This loop will continue to evaluate frames, until it reaches some end condition. It doesn't care about the frame stack that might grow as Python functions are called, or shrink when Python functions return or raise exceptions. It will simply run the topmost frame, whatever is there, whatever interpreter it uses. Calling frames and returning from frames is no longer different. It reduces to leaving a frame and executing another one. If the frame chain grows, it is a call, and if it shrinks, it is a return. The dispatcher does not care about that.

You might wonder about the `result` variable, which looks a little funny. This is indeed the result that the execution of a frame came up with. We wouldn't even need to examine this variable, since the executing interpreters know whether they expect a result and how to handle it. The reason why we do check for a `Py_UnwindToken` is just to assign our ownership (see below) to a probably newly created frame, and to handle the special case of passing a return value

You might also wonder thy this dispatcher object `self` is needed. Every existing dispatcher creates a dispatcher object, in order to keep track of recursive calls. By inserting a reference from the frame to this object, we associate the frame's ownership.

This is a necessary contribution for being stackless and backward compatible at once. Not every recursive interpreter call can be easily avoided: An `__init__` call for instance has different semantics than "normal" frames, since it is expected to return `None` as result. This cannot easily be turned into *tail recursion*. In this case, the recursive call makes good sense. By monitoring the lifetime of dispatchers, we can track down whether a frame is a valid jump target or not.

7.4 Extended Frame Compatibility

Two frames are compatible iff control flow can be transferred without corrupting the C-stack.

My early assumption was that compatible frames must necessarily share the same dispatcher. This leads to a couple of restrictions: Continuations, which are created from frames belonging to a different dispatcher would never be valid jump targets. You would never be able to save a continuation in an instance's `__init__` function and call it later from the main module. The same is true for continuations which are saved during an import or which even come from a different thread. And most important, extension modules that are not stackless-aware often call back into Python functions and crate incompatible frames.

But there is a way out: Dispatcher objects are keeping track of their running dispatcher function. Whenever the dispatcher returns, it marks its death in its dispatcher object. It is important to note that by the death of a frame's dispatcher, we know for sure that there is no longer an associated C function on the stack that holds a reference to it, thereby forbidding its re-use.

The resulting rule is simple: Wait until a dispatcher is done with its frames, and they become valid jump targets. Before we jump to such a frame, we remove the reference to the dead dispatcher and assign our own. **Dead dispatcher's frames are compatible.**

In other words: By waiting until a call to another dispatcher returns, we can treat this call again like a tail recursion which is equivalent to a jump back into our active dispatcher. Just a very long tail ☺.

8 The New C API

In order to stay backward compatible, all the known recursive interpreter functions needed to be retained. To avoid duplication of all the code, the following technique was applied:

The old function header was copied; the new function got a "_nr" appended to its name. The new function was changed to be non-recursive. If the old version was short enough or had to be changed anyway, code was indeed copied. An example for this is the well-known `eval_code`:

The difference is obvious: While the backward compatible version creates a ready-to-run frame (which is put onto the frame stack by `eval_code2_setup`) and runs it to the end by `PyEval_Frame_Dispatch`, the other just leaves that frame and returns `Py_UnwindToken`.

An example where the original version was expressed in terms of the new version, we have a look at

`PyEval_CallObjectWithKeywords`:
The old version was turned into a call to the new, followed by a dispatcher call. Note that in the case of `PyEval_CallObjectWithKeywords`, the "new" function is exactly the same code as the original one. The difference is just that the used internal functions of `ceval.c`, `call_builtin` and `call_function` have changed their semantics to be non-recursive versions. `PyEval_CallObjectWithKeywords_nr` does not know that it now can return something special, since a `Py_UnwindToken` is just another PyObject. So the "new" function in this case is the new version of `PyEval_CallObjectWithKeywords` that just takes care that no `Py_UnwindToken` can leak out to a C function that is not stackless-aware.

The backward compatible version has exactly the same semantics as in the original code. It runs some code and returns either a PyObject or a NULL, indicating an exception. Extension modules that wish to implement stackless C functions in a similar fashion as shown here, will use the new function instead.

New stackless functions in `ceval.c`:

```
PyEval_CallObjectWithKeywords_nr
PyEval_CallObject_nr
PyEval_EvalCode_nr
PyEval_Frame_Dispatch
```

New stackless functions in `pythonrun.c`:

```
PyRun_SimpleFile_nr
PyRun_String_nr
PyRun_File_nr
```

New stackless functions in `bltinmodule.c`:

```
builtin_map_nr             done
builtin_eval_nr           to do
builtin_filter_nr         to do
builtin_reduce_nr         to do
builtin_apply_nr           done
```

The `builtin_xxx` functions finally replaced their originals, since they are not used elsewhere. `builtin_eval` and `builtin_apply` are straightforward to implement, some are not done yet.

The major problem is functions that cannot easily be converted since they are not tail recursive preparations of a single `eval_code` call, but repetitive calls are performed. To obtain a stackless version of these, it is necessary to define their own interpreter function. I implemented this as a proof of concept just for `builtin_map`.

9 Alternatives to "The Token"

The approach to introduce a special object to represent a third system state is considered not clean sometimes. It should be noted that there are other equivalent approaches. A special flag could be set in the current thread state or the current frame to indicate the special situation of unwinding. My choice was directed by execution speed, since comparison of the current result against a constant value is fast. For sure there is the need of a third system state, by whatever it is expressed.

10 Future directions

A stackless implementation of the SGMLOP module is under consideration. This leads to fast XML parsers, which are no longer driven by callbacks into Python, but using coroutines and quick context switches between instantiated frames. This parsing style may appear not only as faster, but also as not less intuitive than callbacks. Tiny threads will be implemented with Stackless Python and Continuation module. Generators and coroutines will get direct support in the C code, after they have been implemented in Python. Since their layout is not as obvious as for continuations, we should play with different prototypes and choose what fits best.

11 Acknowledgements

I wish to thank all people who helped me to turn this idea into reality. Special thanks apply to

Andrea Tismer for standing a husband who has been hacking day and night for half a year

Jeremy Hylton who provided outstanding help to give this paper its final touch

Tim Peters for all his input and involvement

Sam Rushing for inspiration and the reason at all

Axel Roepke for long and constructive discussions

Jean-Claude Wippler for helping to get the final bits of the very first version done.

Guido for the promised hug :-)

Aaron Watters for being my shepherd for this paper. Actually he never got something, but I hacked on my code since he suggested "meaty examples"

Francois Ladouceur for the proofreading, which didn't happen for the same reasons ☺

References

1. Richard Kelsey, William Clinger, And Jonathan Rees (Editors), *Revised^5 Report on the Algorithmic Language Scheme*, http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html, February 1998
2. Tim Peters, *Coroutines in Python*, http://www.pns.cc/stackless/coroutines.tim.peters.html, May 1994
3. Guido van Rossum, *[Python-Dev] 'stackless' python?*, http://www.pns.cc/stackless/continuations.guido.van.rossum.html, May 1999
4. Sam Rushing, *Coroutines In Python*, http://www.nightmare.com/~rushing/copython/index.html, Nov. 1999
5. Dorai Sitaram, *Teach Yourself Scheme in Fixnum Days*, http://www.cs.rice.edu/~dorai/t-y-scheme/, September 1998
6. Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992
7. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes, *Essentials of Programming Languages*, MIT Press, 1993
8. Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, *Continuations and Coroutines*, Computer Languages, 11(3/4): 143-153, 1986.
9. Guy L. Steele. *Rabbit: a compiler for Scheme*, MIT AI Tech Report 474. 1978.
10. Strachey and Wadsworth, *Continuations: A mathematical semantics which can deal with full jumps*. Technical monograph PRG-11, Programming Research Group, Oxford, 1974.

Appendix: Code Examples

```
/* Backward compatible interface */

PyObject *
PyEval_EvalCode(co, globals, locals)
  PyCodeObject *co;
  PyObject *globals;
  PyObject *locals;
{
  PyFrameObject *frame;
  frame=eval_code2_setup(co,
        globals, locals,
        (PyObject **)NULL, 0,
        (PyObject **)NULL, 0,
        (PyObject **)NULL, 0,
        (PyObject *)NULL);
  if (frame != NULL)
    return PyEval_Frame_Dispatch();
  else
    return NULL;
}


PyObject *
PyEval_EvalCode_nr(co, globals, locals)
  PyCodeObject *co;
  PyObject *globals;
  PyObject *locals;
{
  PyFrameObject *frame;
  frame = eval_code2_setup(co,
        globals, locals,
        (PyObject **)NULL, 0,
        (PyObject **)NULL, 0,
        (PyObject **)NULL, 0,
        (PyObject *)NULL);
  if (frame != NULL)
    return Py_UnwindToken;
  else
    return NULL;
}
```

```
PyObject *
PyEval_CallObjectWithKeywords(func, arg, kw)
  PyObject *func;
  PyObject *arg;
  PyObject *kw;
{
  PyObject *retval =
    PyEval_CallObjectWithKeywords_nr(
        func, arg, kw);
  if (retval == Py_UnwindToken) {
    retval = PyEval_Frame_Dispatch();
  }
  return retval;
}

PyObject *
PyEval_CallObjectWithKeywords_nr(func, arg, kw)
  PyObject *func;
  PyObject *arg;
  PyObject *kw;
{
    ternaryfunc call;
    PyObject *result;

  if (arg == NULL)
    arg = PyTuple_New(0);
  else if (!PyTuple_Check(arg)) {
    PyErr_SetString(PyExc_TypeError,
        "argument list must be a tuple");
    return NULL;
  }
  else
  ...
  ...
```

Figure 7: PyEval_CallObjectWithKeywords

old case CALL_FUNCTION:

```
    {
            [ declarations and initializations ]
      if (PyFunction_Check(func)) {
              [ prepare Python function call ]
        x = eval_code2(
          (PyCodeObject *)co, globals,
          (PyObject *)NULL, stack_pointer-n, na,
          stack_pointer-2*nk, nk, d, nd, class);
      }
      else {
              [ prepare builtin function call ]
        x = PyEval_CallObjectWithKeywords(
          func, args, kwdict);
        Py_DECREF(args);
        Py_XDECREF(kwdict);
      }
      Py_DECREF(func);
      while (stack_pointer > pfunc) {
        w = POP();
        Py_DECREF(w);
      }
      PUSH(x);
      if (x != NULL) continue;
      break;
    }
```

old function call_function()

```
                    [ preparations ]
  result = eval_code2(
    (PyCodeObject *)PyFunction_GetCode(func),
    PyFunction_GetGlobals(func),
    (PyObject *)NULL,
    &PyTuple_GET_ITEM(arg, 0), PyTuple_Size(arg),
    k, nk,
    d, nd,
    class);

  Py_DECREF(arg);
  PyMem_XDEL(k);

  return result;
}
```

new function call_function()

```
                    [ preparations ]
f = eval_code2_setup(
    (PyCodeObject *)PyFunction_GetCode(func),
    PyFunction_GetGlobals(func),
   (PyObject *)NULL,
    &PyTuple_GET_ITEM(arg, 0), PyTuple_Size(arg),
    k, nk,
    d, nd,
    class);
  if (f != NULL) {
    f->f_hold_ref = arg;  /* the decref will
              happen on frame disposal */
    result = Py_UnwindToken;
  }
  else result = NULL;

  PyMem_XDEL(k);
  return result;
}
```

new case CALL_FUNCTION:

```
    {
            [ declarations and initializations ]
      if (PyFunction_Check(func)) {
              [ prepare Python function call ]
        x = (PyObject *)eval_code2_setup(
          (PyCodeObject *)co, globals,
          (PyObject *)NULL, stack_pointer-n, na,
          stack_pointer-2*nk, nk, d, nd, class);
        if (x != NULL)
          x = Py_UnwindToken;
      }
      else {
              [ prepare builtin function call ]
        f->f_stackpointer = stack_pointer;
        x = PyEval_CallObjectWithKeywords_nr(
          func, args, kwdict);
        Py_DECREF(args);
        Py_XDECREF(kwdict);
      }
      Py_DECREF(func);
      while (stack_pointer > pfunc) {
        w = POP();
        Py_DECREF(w);
      }

      /* stackless postprocessing */

      if (x == Py_UnwindToken) {
        why = WHY_CALL;
        break;
      }
      else if (f->f_callguard != NULL) {
        /* also protect normal calls 990712 */
        f->f_stackpointer = stack_pointer;
        f->f_next_instr = next_instr;
        f->f_temp_val = x;
        f->f_statusflags |= WANTS_RETURN_VALUE;
        err = f->f_callguard(f, 2);
        if(err) {
          if(err==-42) {
            return Py_UnwindToken;
          }
          else
            break;
        }
        f->f_statusflags &= ~WANTS_RETURN_VALUE;
      }
      PUSH(x);
      if (x != NULL) continue;
      break;
    }
```

Source code and a Python 1.5.2 compatible build are available at: http://www.tismer.com/research/stackless

Figure 8: CALL_FUNCTION