CS 164, Fall 2006 **Pyth Run-Time Structures** **P. N. Hilfinger**

Version 2.9, 9 Dec 2006

This document describes the assumptions made by the Pyth run-time about the representation of objects, values of variables, functions, virtual tables, and stack frames. User code generated by the compiler must conform to these assumptions, on pain of extremely obscure errors.

1 Values

Values (and thus variables) in Pyth are 16 bytes long and consist of a pointer and additional data, depending on the dynamic type of the value, as shown in Figure 1.

This representation is rather wasteful due to several interacting considerations:

- All values require dynamic type information.
- We want primitive values (integers, floating-point values) to be "unboxed." That is, we want their numeric values to be stored in variables directly, rather than on the heap. This avoids doing allocation when doing arithmetic.
- Float values require 8 bytes of data beyond that required to hold the dynamic type.
- We want values to be 8-byte aligned, in case they hold Float values (since access to an 8-byte value in memory is significantly faster when the address is divisible by 8.)

Each kind of value starts with an unused segment that serves as part of the padding needed to get values to occupy 16 bytes. Next comes a pointer to an object that contains (at least) information about the dynamic type. Values of type Int and Float contain the actual value; the object pointer is the same for all Ints and for all Floats. Likewise, function values contain the static link and code address needed to call them; their object pointers serve only to give information about their argument types (see §4.

The values in the unused segments are unspecified; they may contain anything at all, and need not be assigned to when copying or forming values. When assigning values to instance variables, in particular, it is important *not* to assign to the first unused area, for reasons that should become obvious in §2.

2 Objects, Classes, and Type Descriptors

All objects pointed to from values consist of a *type tag*— which is a pointer to a *type descriptor*—and zero or more 16-byte value slots containing the instance variables. In the case of predefined classes, the contents of the object, aside from the type-descriptor pointer, are implementation defined, and need not consist of ordinary Pyth values. The type tag is laid on top of the first value in its conveniently available unused slot. If an object has no instance variables, it contains just the type tag. See Figure 2.

The instance variables start with all instance variables inherited from the parent, in the same order. Constant attributes (defined by **def**) are not stored in instances. Put them anywhere in writable static storage (the "data section).

A type descriptor contains various information about a type that is used for a variety of purposes:

Figure 1: Representation of values in Pyth.

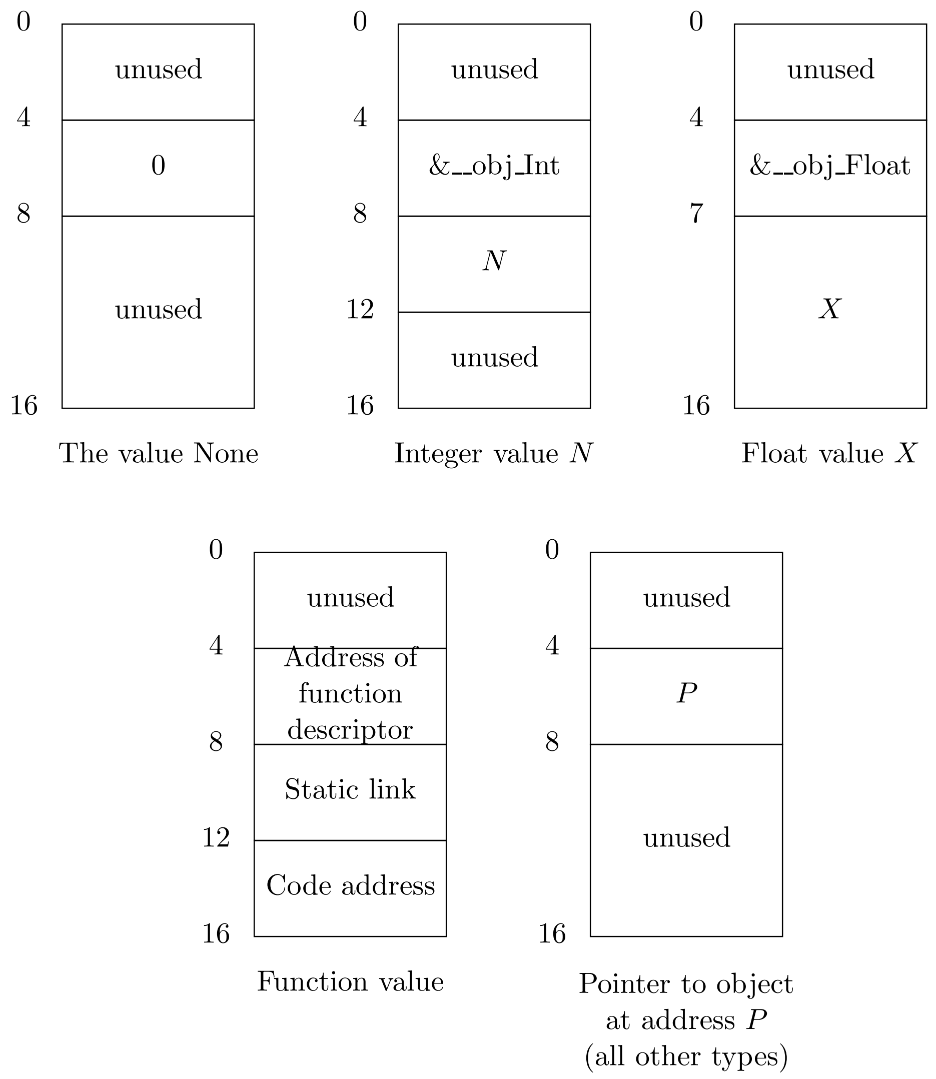
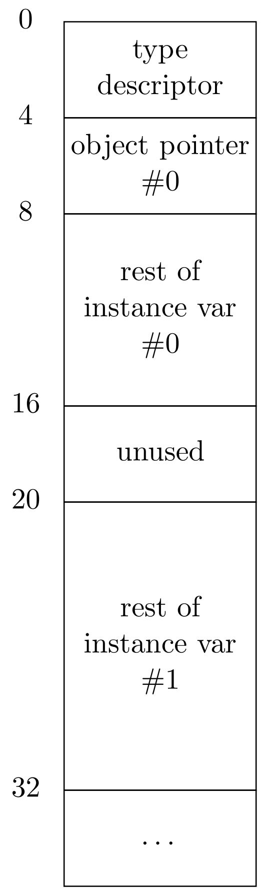


Figure 2: Representation of a user-defined object.



- To implement method calls;
- To indicate dynamic type for the purposes of type checking;
- To indicate the number and types of function arguments for use when calling functions whose static type is `Any`; and
- To give the garbage collector necessary information.

You must create type descriptors for all types, including the pre-defined ones. Suppose that type T has parent type P (0 for type `Any`), contains M instance methods (including inherited ones) whose code addresses are $f_1, f_2, \ldots f_M$; and V instance variables (including inherited ones); In assembly language, the descriptor for type T looks like this:

```
        .section   .rodata
.L1:
        .string    "T"                        # Name of class

        .align     4
.globl __typ_T
__typ_T:
        .long      .L1                        # Pointer to type name.
        .long      __typ_P                    # Parent's descriptor
        .long      V                          # Number of instance variables
        .long      M                          # Number of instance methods
# Method (or virtual) table
        .long      f1
        .long      f2
        ⋮
        .long      fM
```

(The local name `.L1` is arbitrary; any local label will do.) The method table contains only function code addresses, not static links, because methods are never nested, and their static links are ignored. When method k is implemented by a native function (`import` "F"), f_k is simply F. Otherwise, it is the code label of the appropriate method, as generated by your compiler.

Class (static) variables go into static storage. You can give them any assembly-lanuage names you want (that don't clash with other names, of course). Here, we'll just use arbitrary local label names.

According to the rules of Pyth, the class variables include constants defined by `def` (as in "`def pi = 3.14159265`") plus one class variable for each instance variable. For all non-function types other than `Any` and `Void`, there is one distinguished object of that class, called the *class exemplar*. Your code must allocate it in initialized static storage (rather than the heap), and give it the label `__obj_`C, where C is the name of the type. (You can do this for types `Any` and `Void` as well, but the object will never be used). The exemplar will look just like an ordinary object of its type and all its variables will be initialized to the value None. It is this object that is referenced when creating new objects and by references to class variables that correspond to instance variables.

For example, suppose we have defined

```
class A (Object):
   x = 3    # Assigns to the x defined in the class exemplar
   def c1 = 12
A.x = 17    # Same here.
class B (A):
   y = 4
```

Suppose also that your compiler has chosen the labels `.L2`, `.L3`, `.L4`, and `.L5`, respectively, for `A.x`, `B.x`, `B.y`, and `A.c1`. Then we'll have two objects in static storage defined like this (in assembler):

```
        .data
        .align     8
.globl __obj_A
__obj_A:
.L2:                                         # A.x
        .long      __typ_A
        .long      0
        .long      0
        .long      0

L5:                                          # A.c1
        .long      0
        .long      0
        .long      0
        .long      0
.globl __obj_B
__obj_B:
.L3:                                         # B.x
        .long      __typ_B
        .long      0
        .long      0
        .long      0
.L4:                                         # B.y
        .long      0
        .long      0
        .long      0
        .long      0
```

Your code will subsequently set the contents of `.L2` to the Int value 3 and copy it into `.L3`, and will set the contents of `.L5` to the Int value 12.

3 Built-in Types

Again, you create the type descriptors for built-in classes just as for ordinary classes. Objects of the built-in types start with an ordinary type tag, but the rest of their contents are generally not definable in Pyth. They have no program-accessible variable attributes. They may be created and manipulated entirely by their methods, so you don't really need to know the contents, except for Strings, whose layout you need to translate into string literals. This is easily shown by example. The object pointed to by the String value for the literal `"Hello, world"` may be translated

```
        .section   .rodata
        .align     4
.L6:
        .long      __typ_String
        .long      12 # Length
        .string    "Hello, world"
```

For convenience for use with the C library, we use null-terminated string values.

4 Functions, Methods, and Function Descriptors

Besides their declared parameters, Pyth functions take a static link value as their first parameter. Actually, there are two initial parameters: a pointer to space in which to put the return value and a static link. Instance methods and functions at the outer level do not use their static links. The parameters must be there anyway (to avoid confusing the garbage collector), but you need not initialize their storage. For example, after pushing the first declared parameter on the stack (the first parameter is pushed last), you can call an outer-level function with label L with the calling sequence

```
        subl       $4, %esp                  # Unused static link
        leal       N(%esp), %eax             # Return-value address
        pushl      %eax
        call       L
```

where N is the offset from static link of the return-value area.

All Pyth functions maintain a frame pointer, and must push the previous value of the frame pointer and establish a new one as their first instructions, using the usual sequence:

```
pushl %ebp
movl %esp,%ebp
```

undoing this on return with[1]:

```
leave
ret  $4
```

The `$4` here has to do with the pointer to space for the return value (see below).

Local variables, compiler temporaries, and parameters to function calls all go below (at lower addresses than) the saved frame pointer on the stack. We'll call this area (between the saved frame pointer and the top of the stack) *local storage*. You can do what you want with this area (including expanding or contracting it) as long as, just before each function call, local storage consists of an

[1]Where there's a 'leave' there must be an 'enter,' you might think. Quite true, and it does what you'd expect and much more. Turns out the `pushl, movl` sequence is shorter, however—a wonderful example of CISCness.

integral number of valid values, plus (possibly) the static link and return pointer at the top of the stack. Together, these make it possible for the garbage collector to find all roots on the stack.

We use the standard method of returning a structure value from a function in C. The address of the value is the (implicit) first argument of the function. For an n-parameter function, it is convenient to put the 16-byte return value at the stack position that would have held the $n+1^{\text{th}}$ parameter. The calling program must allocate space for this return value. So, if you have an outer-level function declared:

```
def f (x):
    return 3
```

You might call f(42) with

```
        subl       $16, %esp                   # Allocate return variable on stack
        movl       $0, 4(%esp)                 # Initialize to None for garbage collector
# Push argument 42
        subl       $16, %esp                   # Reserve space
        movl       $__obj_Int, 4(%esp)
        movl       $42, 8(%esp)
        subl       $8, %esp                    # Reserve space for implicit arguments
        leal       24(%esp), %eax              # Compute address of return value...
        movl       %eax, (%esp)                # ...  and store it.
        call       f
        addl       $20, %esp
```

That final `addl` operation pops everything off the stack except the return value, which remains at the top of the stack. We pop 20 bytes (value of 42 plus the static link) rather than 24 because the *called* function pops that first argument when it returns, for some obscure reason or another. Inside the code for `f`, the argument `x` will start at offset 16 from the frame base pointer (`%ebp`) and the address of the return variable will be at offset 8 from the frame base pointer. So, you could return the value 3 like this:

```
        movl       8(%ebp), %eax
        movl       $__obj_Int, 4(%eax)
        movl       $3, 8(%eax)
        leave
        ret        $4
```

The arguments passed to a function must always conform to that function's static type. In cases like these:

```
def f (x, y): return x + y
def g (x, y): return x + y
g: (Int, Int) -> Int
print f(3,4), g(3,4)
```

there is no problem, since the compiler knows that `f` and `g` take two parameters and that the types of the actual parameters (Int) is a subtype of the formals (`Any` and `Int`, respectively). But in this case:

```
r = g
print r(3,4)
```

that static type of `r` is `Any`, and the run-time system must decide whether it is legal to call `r`.

The object-pointer part of the value `r` points to this information. Each time you create a function, you generate not only its code, but also a special function-descriptor object. Consider a function, `g`, with n parameters having types $T_1, \ldots, T_n$ and return type T_0. If you have generated the local label `.L7` as its assembler name, its function descriptor has the following format:

```
        .section    .rodata
        .align      4
.L7:
        .long       .+4                         # Pointer to the descriptor proper
        .long       0                           # A null class name indicates a function
        .long       n                           # Argument count
        .long       __typ_T0                    # Return-type descriptor
        .long       __typ_T1                    # Argument type descriptors
        ⋮
        .long       __typ_Tn
```

The reason for the word containing `.+4` (i.e., the address of the next word) is to give the function "object" the same format as ordinary objects, with a type pointer at the beginning. It isn't really necessary for the type pointer to be immediately adjacent to the pointer like this, but it is convenient. If the code of this function is at label `.L8` and you have computed its static link in, say, `%ecx`, then you might push the function value `g` on the stack with:

corrected 12/3/2006

```
        pushl       .L8                         # Code address
        pushl       %ecx                        # Static link
        pushl       .L7                         # Function descriptor
        subl        -4, %esp                    # Reserve unused space
```

If you already have a function value—let's say at `24(%ebp)`—and and want call it, let's say with no arguments to keep things simple, you'd generate code like this:

corrected 12/3/2006

```
        subl        $16, %esp                   # Reserve space for return value
        pushl       32(%ebp)                    # Push the static link from the function
        leal        4(%esp), %eax               # Push address of return value
        pushl %eax
        movl        36(%ebp), %eax              # Load address of code.
        call        *%eax                       # Call function
```

Finally, suppose you have the same function value, but all you know about its static type is that it has type `Any`. Before you call, you must check that the function value you are calling is appropriate. We're providing a runtime function for this purpose, `__checkFunction`. To use it, insert the following code just before the call in the sequence above:

```
pushl      $0                         # Push number of arguments
pushl      28(%ebp)                   # Push the object pointer of the value
call       __checkFunction
```

5 Native Methods

Native methods are pure C functions. The string value in their **import** clause is the name of this C function; the assembler and linker will interpret it correctly as an external name.

6 The Main Program

The Pyth main program is an ordinary function. Since it will be called from the run-time system, its label, `_pyth_main`, must be declared to have external linkage (with `.globl`). It does not need a function descriptor, since it is not available to a Pyth program as a function value.

The main program contains all code that is not contained in a **def**, both inside and outside class definitions. Any variables declared at the outer level are allocated in static storage (the `.data` section), so that the main function has no local variables in the usual sense.

The first duty of the main program is to inform the garbage collector of the addresses of all statically allocated variables, including all exemplar objects (see §2) using two run-time functions provided for that purpose (see §7). Call `__registerVar` with the address (not the contents!) of each global variable that might contain an object pointer. Call `__registerObj` with the address of the exemplar for each user-defined class. Failure to do this will cause bizarre errors when the garbage collector frees storage that is still active.

7 Run-Time Support Functions

The Pyth run-time system provides a number of functions that your code can call to perform necessary functions. They use the normal calling conventions for C functions. You do not push a static link for them. Unless they return type void, you *do* push the address of the place to put the return value. As indicated in the comments, below, some also vary from the rules given for what the stack must look like before a call (for example, the argument to `__createObject` is a single 4-byte pointer, not a 16-byte value). Here, we describe them as if they were ordinary C functions, using the type `PythObject*` to indicate a pointer to an object, `PythType*` to indicate a pointer to a type descriptor, and `PythValue` to indicate a 16-byte value. Where the functions return a `PythValue`, they do so using the convention described in §4, rather than the normal C convention (for those of you who happen to know it). So, for example, to create the Tuple `(3,)`, you could write:

Corrected 12/3/2006

```
subl       $16, %esp                  # Allocate space for return value
movl       $0, 4(%esp)                # ...  and initialize to None
Instructions to push the value 3.
pushl      $1                         # One-item Tuple
leal       20(%esp), %eax             # Return-value address.
pushl      %eax
call       __consTuple
addl       $24, %esp                  # Pop off arguments, leaving return value.
```

```
/** Create a new object consisting of a copy of the object pointed
 *  to by EXEMPLAR (one of the __obj_... objects created for each
 *  class definition). */
PythValue void __createObject (PythObject* exemplar);

/** Checks that TYPE is a supertype VAL's dynamic type, causing an error
 *  if not.  The contents of VAL on the stack are guaranteed not to be
 *  disturbed, so that if the value you wish to check is on top of the
 *  stack, you can simply push the descriptor pointer, call __cast,
 *  and pop the descriptor.  Use this function in cases such as
 *     f (x)    and    y = x
 *  where the static type of x is a supertype (rather than a subtype)
 *  of the formal to f or the static type of y. */
void __cast (PythType* type, PythValue val);

/** Checks that FUNC is object pointer of a function value with N
 *  arguments whose types allow the parameters "...".  Given a call
 *  such as f(E1,E2), where the static type of f is Any, simply set
 *  everything up for a normal call to a function f (push the
 *  arguments, the value that ought to be f's static link, and the
 *  address in which to return the result), then push FUNC (taken
 *  from offset 4 from the value f) and N, call this function, and
 *  then pop FUNC and N from the stack and continue with calling f. */
void __checkFunction (PythObject* func, int n, ...);

/** Cause an error, terminating the program with the given MSG (a
 *  null-terminated string).  */
void __pythError (char* msg);

/** Print the N trailing values (all PythValues) on FILE (or the
 *  standard output if FILE is None). */
void __print (int n, PythValue file, ...);

/** Print the N trailing values (all PythValues) on FILE, (or the
 *  standard output if FILE is None) followed by a newline. */
void __println (int n, PythValue file,  ...);

/** Create a list of the N items (all PythValues) contained in the
 *  trailing arguments. */
PythValue __consList (int n, ...);

/** Create a tuple of the N items (all PythValues) contained in the
 *  trailing arguments. */
PythValue __consTuple (int n, ...);

/** Create a Dictionary initialized with the N pairs of PythValues in
```

```
 *  the trailing parameters.  For example, if X is the PythValue
 *  corresponding to the String "Hello", and Y is the PythValue
 *  corresponding to the Int 42, then __createDict (1, X, Y)
 *  returns { "Hello" : 42 }. */
PythValue __consDict (int n, ...);

/** Returns the constant 1 if X is a "true" value, and 0
 *  otherwise.  False values are None and all those whose
 *  truth method returns False.
 */
int __isTrue (PythValue x);

/* Note: Strings do not need a __cons... method, since they may be
 * constructed directly in static memory. */

/* The two __registerXXX functions below have no visible effects, but
 * must be called to inform the garbage collector of roots that exist
 * in statically allocated storage. */

/** Inform the runtime system that space pointed to by VALP is
 *  a variable in static storage.  (This has no visible effect,
 *  but is needed to allow the garbage collector to find all roots).  */
void __registerVar (PythValue* valp);

/** Inform the runtime system that OBJP is a pointer to a
 *  statically allocated exemplar object. */
void __registerObj (PythValue* objp);
```