Cleanup and new optimizations in WPython 1.1

Cesare Di Mauro

A-Tono s.r.l.

PyCon Quattro 2010 – Firenze (Florence)

May 9, 2010

What’s new in WPython 1.1

- Removed WPython 1.0 alpha PyTypeObject “hack”
- Fixed tracing for optimized loops
- Removed redundant checks and opcodes
- New opcode_stack_effect()
- Specialized opcodes for common patterns
- Constant grouping and marshalling on slices
- Peepholer moved to compiler.c
- Experimental “INTEGER” opcodes family

The PyTypeObject "hack”

```
typedef struct _typeobject {
[...]
  hashfunc tp_relaxedhash; /* Works for lists and dicts too. */
  cmpfunc tp_equal; /* Checks if two objects are equal (0.0 != -0.0) */
} PyTypeObject;

long PyDict_GetHashFromKey(PyObject *key) {
  long hash;
  if (!PyString_CheckExact(key) ||
      (hash = ((PyStringObject *) key)->ob_shash) == -1) {
    hash = key->ob_type->tp_relaxedhash(key);
    if (hash == -1) PyErr_Clear();
  }
  return hash;
}
```

No conditional branch(es): just get it!

Removed it in Wpython 1.0/1.1

```
long _Py_object_relaxed_hash(PyObject *o) {
  if (PyTuple_CheckExact(o))
    return _Py_tuple_relaxed_hash((PyTupleObject *) o);
  else if (PyList_CheckExact(o))
    return _Py_list_relaxed_hash((PyListObject *) o);
  else if (PyDict_CheckExact(o))
    return _Py_dict_relaxed_hash((PyDictObject *) o);
  else if (PySlice_Check(o))
    return _Py_slice_relaxed_hash((PySliceObject *) o);
  else
    return PyObject_Hash(o);
}
```

Conditional branches kill the processor pipeline!

`_Py_object_strict_equal` is even worse!

LOST: performance!

“Hack” used only on compile.c for consts dict…

…but very important for speed!

Tests made on Windows 7 x64, Athlon64 2800+ socket 754, 2GB DDR 400Mhz, 32-bits Python

	PyStone (sec.)	PyBench Min (ms.)	PyBench Avg (ms.)
Python 2.6.4	38901	10444*	10864*
WPython 1.0 alpha	50712	8727**	9082**
WPython 1.0	47914	9022	9541
WPython 1.1	47648	8785	9033
WPython 1.1 (INT “opc.”)	45655	9125	9486

* Python 2.6.4 missed UnicodeProperties. Added from 1.0.

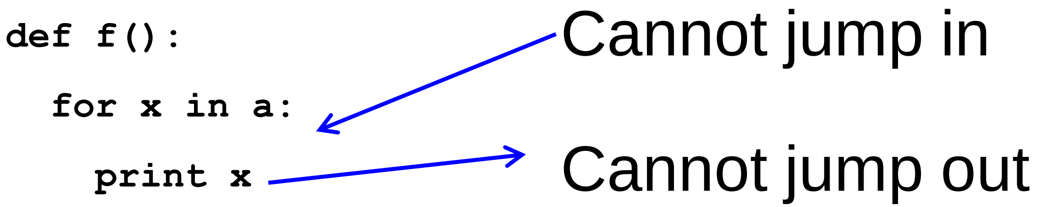
** WPython 1.0 alpha missed NestedListComprehensions and SimpleListComprehensions. Added from 1.0.

Being polite doesn’t pay. “Crime” does!

Tracing on for loops was broken

Optimized for loops don’t have SETUP_LOOPs & POP_BLOCKs

Tracing makes crash on jumping in / out of them!



```
def f():
  for x in a:
    print x
```

Cannot jump in

Cannot jump out

```
2     0 SETUP_LOOP     19 (to 22)
      3 LOAD_GLOBAL     0 (a)
      6 GET_ITER
>>    7 FOR_ITER       11 (to 21)
     10 STORE_FAST      0 (x)
3    13 LOAD_FAST       0 (x)
     16 PRINT_ITEM
     17 PRINT_NEWLINE
     18 JUMP_ABSOLUTE 7
>>   21 POP_BLOCK
>>   22 LOAD_CONST      0 (None)
     25 RETURN_VALUE
```

```
2     0 LOAD_GLOBAL     0 (a)
      1 GET_ITER
   >> 2 FOR_ITER        5 (to 8)
      3 STORE_FAST      0 (x)
3     4 LOAD_FAST       0 (x)
      5 PRINT_ITEM
      6 PRINT_NEWLINE
      7 JUMP_ABSOLUTE   2
   >> 8 RETURN_CONST    0 (None)
```

Now fixed, but was a nightmare!

```
case SETUP_LOOP:
case SETUP_EXCEPT:
case SETUP_FINALLY:
case FOR_ITER:
```

1) Save loop target
2) Check loop target
3) Remove loop target

```
case POP_BLOCK:
case POP_FOR_BLOCK:
case END_FINALLY:
case WITH_CLEANUP:
```

```
/* Checks if we have found a "virtual" POP_BLOCK/POP_TOP
   for the current FOR instruction (without SETUP_LOOP). */
if (addr == temp_last_for_addr) {
  temp_last_for_addr = temp_for_addr_stack[--temp_for_addr_top];
  temp_block_type_top--;
}
```

CPython has “extra” checks…

```
referentsvisit(PyObject *obj, PyObject *list)
{
        return PyList_Append(list, obj) < 0;
}

int
PyList_Append(PyObject *op, PyObject *newitem)
{
        if (PyList_Check(op) && (newitem != NULL))
                return app1((PyListObject *)op, newitem);
        PyErr_BadInternalCall();
        return -1;
}
```

Must be a List

Value is needed

It does the real work!

…remove them!

```
referentsvisit(PyObject *obj, PyObject *list)
{
        return _Py_list_append(list, obj) < 0;
}

int _Py_list_append(PyObject *self, PyObject *v)
{
        Py_ssize_t n = PyList_GET_SIZE(self);
        assert (v != NULL);
[...]
        PyList_SET_ITEM(self, n, v);
        return 0;
}
```

Absolutely sure:

a list and an

object!

Direct call

No checks!

app1 is now `_Py_list_append`

But don’t remove too much!

A too much aggressive unreachable code remover

```
def test_constructor_with_iterable_argument(self):
  b = array.array(self.typecode, self.example)
  # pass through errors raised in next()
    def B():
      raise UnicodeError
      yield None
  self.assertRaises(UnicodeError, array.array, self.typecode, B())
```

Function B was compiled as if it was:

```
def B():
  raise UnicodeError
```

Not a generator! Reverted back to old (working) code…

The old opcode_stack_effect()...

```
static int opcode_stack_effect(int opcode, int oparg)
{
  switch (opcode) {
    case POP_TOP:
      return -1;          /* POPs one element from stack
    case ROT_THREE:
      return 0;           /* Stack unchanged */
    case DUP_TOP:
      return 1;           /* PUSHs one element */
    case LIST_APPEND:
      return -2;          /* Removes two elements */
    case WITH_CLEANUP:
      return -1;          /* POPs one element. Sometimes more */
    case BUILD_LIST:
      return 1-oparg;     /* Consumes oparg elements, pushes one */
```

Several checks

… and the new one!

```
static int opcode_stack_effect(struct compiler *c, struct instr*i, int*stack_retire)
{
  static signed char opcode_stack[TOTAL_OPCODES] = {
    /* LOAD_CONST = 1 */
    /* LOAD_FAST = 1 */
    /* STORE_FAST = -1 */
    0,  0,  0,  0,  0,  0,  1,  1, -1,  0, /*   0 ..  9 */
[...]
  int opcode = i->i_opcode;  int oparg = i->i_oparg;  *stack_retire = 0;
  switch (opcode & 0xff) {
    case BUILD_LIST:
      return 1 - oparg;
[...]
    default:
      return opcode_stack[opcode & 0xff];
}
```

Only special cases checked

Regular cases unchecked

Just get the value!

Removed unneeded copy on for

```
def f():
  for x in[1, 2, 3]:
    pass
```

List of “pure” constants

Python 2.6.4

```
2     0 SETUP_LOOP   23 (to 26)
      3 LOAD_CONST   1 (1)
      6 LOAD_CONST   2 (2)
      9 LOAD_CONST   3 (3)
     12 BUILD_LIST   3
     15 GET_ITER
>>   16 FOR_ITER 6 (to 25)
     19 STORE_FAST   0 (x)
     22 JUMP_ABSOLUTE  16
>>   25 POP_BLOCK
>>   26 LOAD_CONST   0 (None)
     29 RETURN_VALUE
```

WPython 1.0 alpha

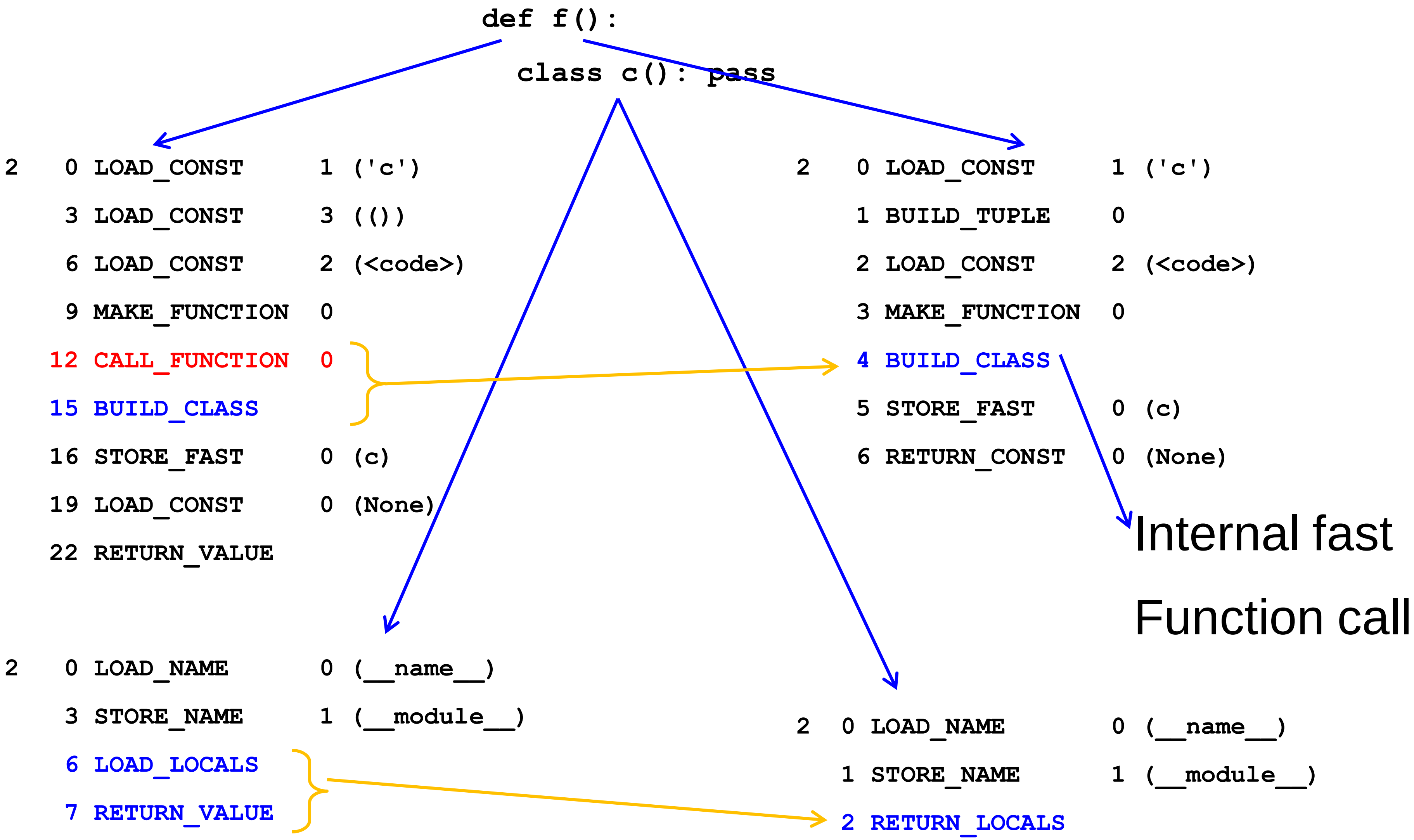
```
2    0 LOAD_CONST     1 ([1, 2, 3])
     1 LIST_DEEP_COPY
     2 GET_ITER
>>   3 FOR_ITER       2 (to 6)
     4 STORE_FAST     0 (x)
     5 JUMP_ABSOLUTE 3
>>   6 RETURN_CONST   0 (None)
```

WPython 1.1

```
2    0 LOAD_CONST     1 ([1, 2, 3])
     1 GET_ITER
>>   2 FOR_ITER       2 (to 5)
     3 STORE_FAST     0 (x)
     4 JUMP_ABSOLUTE 2
>>   5 RETURN_CONST   0 (None)
```

Makes a (deep) copy, before use

Improved class creation



```
def f():
    class c(): pass
```

```
2     0 LOAD_CONST      1 ('c')
      3 LOAD_CONST      3 (())
      6 LOAD_CONST      2 (<code>)
      9 MAKE_FUNCTION   0
     12 CALL_FUNCTION   0
     15 BUILD_CLASS
     16 STORE_FAST      0 (c)
     19 LOAD_CONST      0 (None)
     22 RETURN_VALUE
```

```
2     0 LOAD_CONST      1 ('c')
      1 BUILD_TUPLE     0
      2 LOAD_CONST      2 (<code>)
      3 MAKE_FUNCTION   0
      4 BUILD_CLASS
      5 STORE_FAST      0 (c)
      6 RETURN_CONST    0 (None)
```

Internal fast

Function call

```
2     0 LOAD_NAME       0 (__name__)
      3 STORE_NAME      1 (__module__)
      6 LOAD_LOCALS
      7 RETURN_VALUE
```

```
2    0 LOAD_NAME        0 (__name__)
     1 STORE_NAME       1 (__module__)
     2 RETURN_LOCALS
```

Optimized try on except (last)

```
2       0 SETUP_EXCEPT   8  (to 11)
3       3 LOAD_FAST      0  (x)
        6 POP_TOP
        7 POP_BLOCK
        8 JUMP_FORWARD   11  (to 22)
4  >>  11 POP_TOP
       12 POP_TOP
       13 POP_TOP
5      14 LOAD_FAST      1  (y)
       17 POP_TOP
       18 JUMP_FORWARD   1  (to 22)
       21 END_FINALLY
   >>  22 LOAD_CONST     0  (None)
       25 RETURN_VALUE
```

```
def f(x, y):
  try:
    x
  except:
    y
```

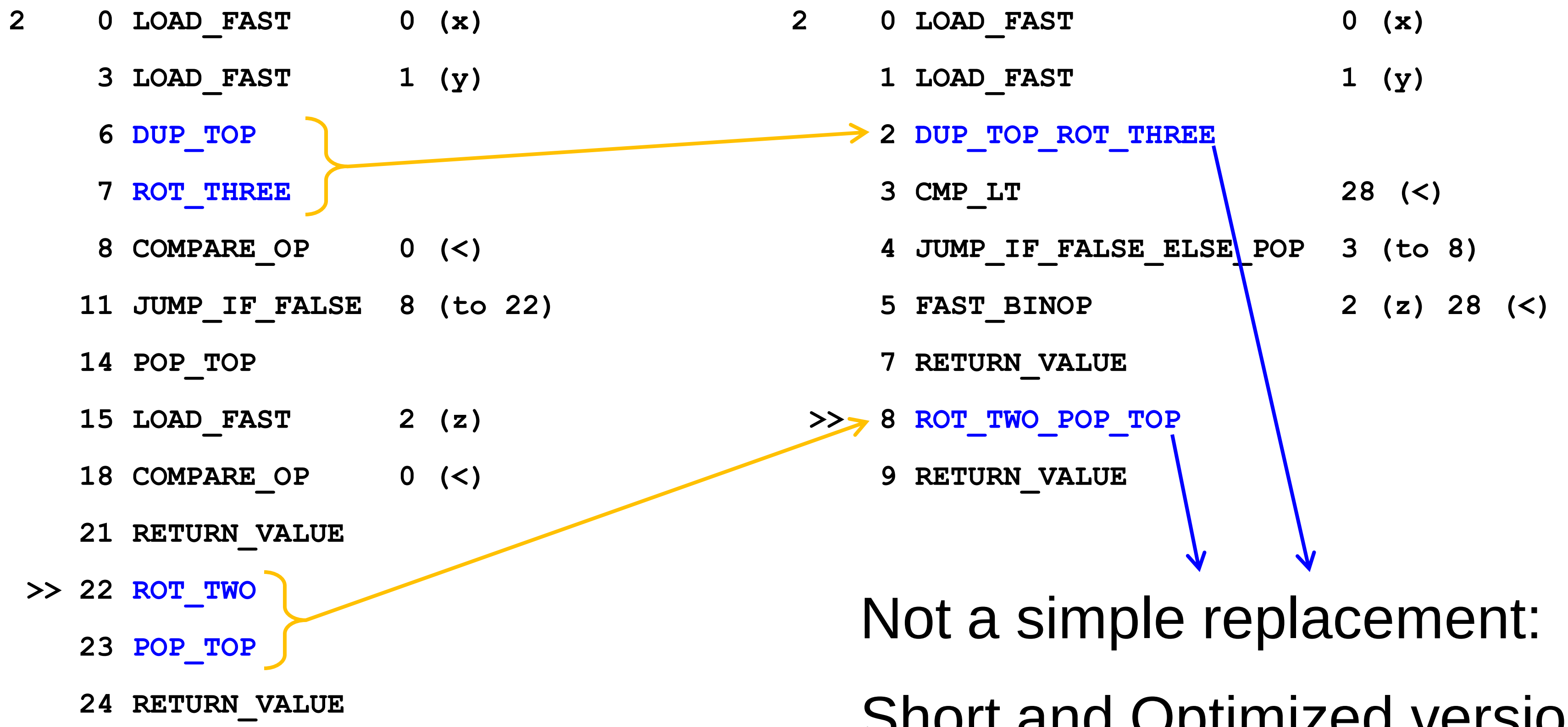
```
2      0 SETUP_EXCEPT   4  (to 5)
3      1 LOAD_FAST      0  (x)
       2 POP_TOP
       3 POP_BLOCK
       4 JUMP_FORWARD   5  (to 10)
4  >>  5 POP_TOP
       6 POP_TOP
       7 POP_TOP
5      8 LOAD_FAST      1  (y)
       9 POP_TOP
  >>  10 RETURN_CONST   0  (None)
```

Unneeded on generic expect clause

(if it was the last one)

Opcodes for multiple comparisons

```
def f(x, y, z):
  return x < y < z
```



```
2       0 LOAD_FAST       0 (x)
        3 LOAD_FAST       1 (y)
        6 DUP_TOP
        7 ROT_THREE
        8 COMPARE_OP      0 (<)
       11 JUMP_IF_FALSE   8 (to 22)
       14 POP_TOP
       15 LOAD_FAST       2 (z)
       18 COMPARE_OP      0 (<)
       21 RETURN_VALUE
  >>   22 ROT_TWO
       23 POP_TOP
       24 RETURN_VALUE
```

```
2       0 LOAD_FAST                  0 (x)
        1 LOAD_FAST                  1 (y)
        2 DUP_TOP_ROT_THREE
        3 CMP_LT                     28 (<)
        4 JUMP_IF_FALSE_ELSE_POP     3 (to 8)
        5 FAST_BINOP                 2 (z) 28 (<)
        7 RETURN_VALUE
   >>   8 ROT_TWO_POP_TOP
        9 RETURN_VALUE
```

Not a simple replacement:

Short and Optimized versions!

Specialized generator operator

```
def f(*Args):
  return (int(Arg) for Arg in Args[1 : ])
```

```
 2   0 LOAD_CONST     1 (<code> <genexpr>)
     3 MAKE_FUNCTION  0
     6 LOAD_FAST      0 (Args)
     9 LOAD_CONST     2 (1)
    12 SLICE+1
    13 GET_ITER
    14 CALL_FUNCTION  1
    17 RETURN_VALUE
```

```
2   0 LOAD_CONST 1 (<code> <genexpr>)
    1 MAKE_FUNCTION 0
    2 FAST_BINOP_CONST 0 (Args) 2 (1) 39 (slice_1)
    4 GET_GENERATOR
    5 RETURN_VALUE
```

Internal fast

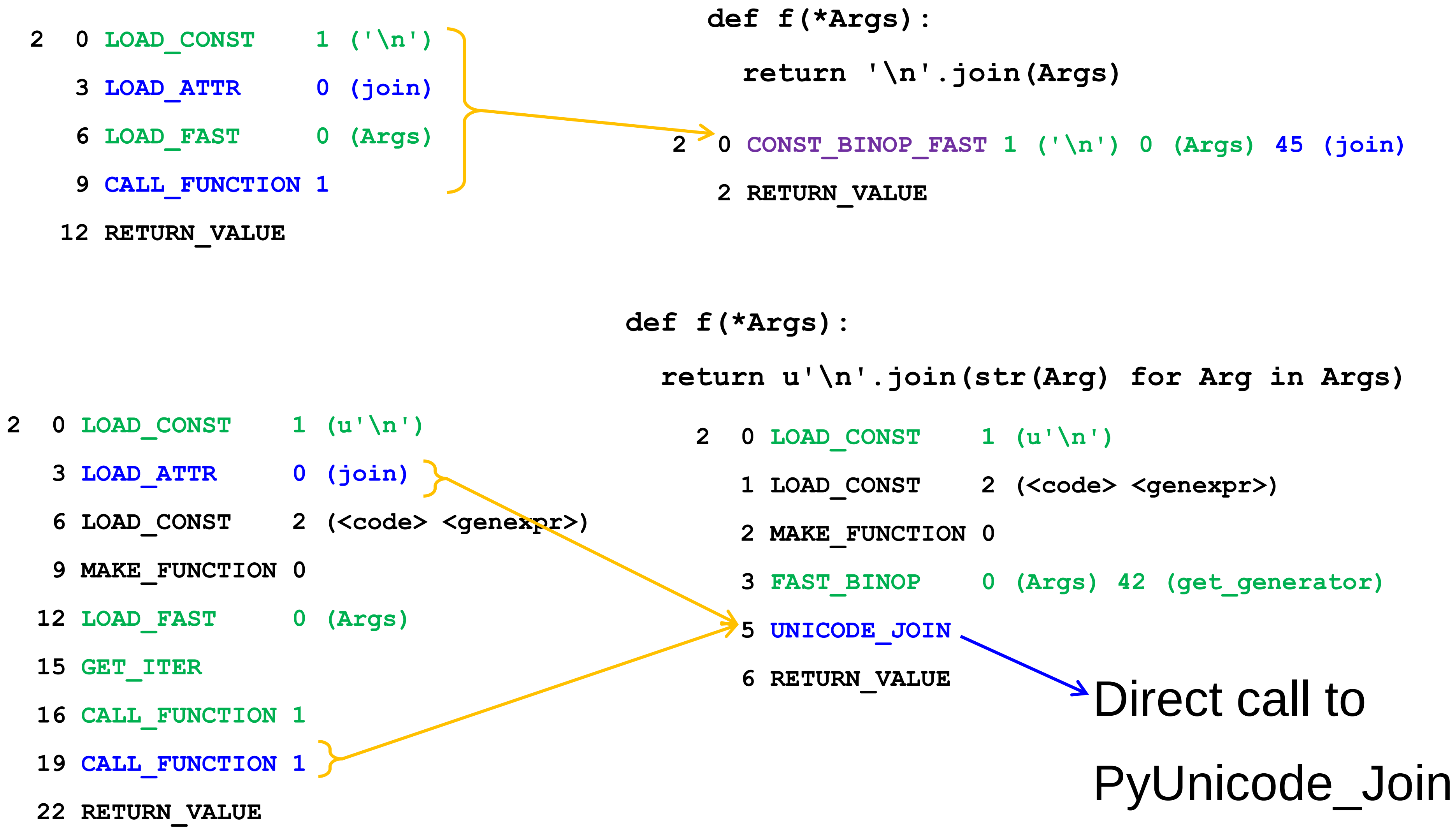
Function call

```
def f(*Args):
  return sum(int(Arg) for Arg in Args)
```

```
2   0 LOAD_GLOBAL    0 (sum)
    3 LOAD_CONST     1 (<code> <genexpr>)
    6 MAKE_FUNCTION  0
    9 LOAD_FAST      0 (Args)
   12 GET_ITER
   13 CALL_FUNCTION  1
   16 CALL_FUNCTION  1
   19 RETURN_VALUE
```

```
2   0 LOAD_GLOBAL 0 (sum)
    1 LOAD_CONST 1 (<code> <genexpr>)
    2 MAKE_FUNCTION 0
    3 FAST_BINOP 0 (Args) 42 (get_generator)
    5 QUICK_CALL_FUNCTION 1 (1 0)
    6 RETURN_VALUE
```

String joins are… binary operators!



```
2   0 LOAD_CONST     1  ('\n')
    3 LOAD_ATTR      0  (join)
    6 LOAD_FAST      0  (Args)
    9 CALL_FUNCTION  1
   12 RETURN_VALUE
```

```
def f(*Args):
  return '\n'.join(Args)
```

```
2   0 CONST_BINOP_FAST 1 ('\n') 0 (Args) 45 (join)
    2 RETURN_VALUE
```

```
def f(*Args):
  return u'\n'.join(str(Arg) for Arg in Args)
```

```
2   0 LOAD_CONST     1  (u'\n')
    3 LOAD_ATTR      0  (join)
    6 LOAD_CONST     2  (<code> <genexpr>)
    9 MAKE_FUNCTION  0
   12 LOAD_FAST      0  (Args)
   15 GET_ITER
   16 CALL_FUNCTION  1
   19 CALL_FUNCTION  1
   22 RETURN_VALUE
```

```
2   0 LOAD_CONST     1  (u'\n')
    1 LOAD_CONST     2  (<code> <genexpr>)
    2 MAKE_FUNCTION  0
    3 FAST_BINOP     0  (Args) 42 (get_generator)
    5 UNICODE_JOIN
    6 RETURN_VALUE
```

Direct call to PyUnicode_Join

Specialized string modulo

```
def f(x, y):
  return '%s and %s' % (x, y)
```

```
2   0 LOAD_CONST   1 ('%s and %s')        2   0 LOAD_CONST   1 ('%s and %s')
    3 LOAD_FAST    0 (x)                      1 LOAD_FAST    0 (x)
    6 LOAD_FAST    1 (y)                      2 LOAD_FAST    1 (y)
    9 BUILD_TUPLE 2                           3 BUILD_TUPLE 2
   12 BINARY_MODULO                           4 STRING_MODULO
   13 RETURN_VALUE                            5 RETURN_VALUE
```

Direct call to

PyString_Format

No checks needed

```
def f(a):
  return u'<b>%s</b>' % a
```

```
2   0 LOAD_CONST   1 (u'<b>%s</b>')      2   0 CONST_BINOP_FAST 1 (u'<b>%s</b>') 0 (a) 44 (%)
    3 LOAD_FAST    0 (a)                      2 RETURN_VALUE
    6 BINARY_MODULO
    7 RETURN_VALUE
```

Improved with cleanup

```
def f(name):
  with open(name):
    pass
```

```
2     0 LOAD_GLOBAL    0 (open)
      3 LOAD_FAST      0 (name)
      6 CALL_FUNCTION  1
      9 DUP_TOP
     10 LOAD_ATTR      1 (__exit__)
     13 ROT_TWO
     14 LOAD_ATTR      2 (__enter__)
     17 CALL_FUNCTION  0
     20 POP_TOP
     21 SETUP_FINALLY  4 (to 28)
3    24 POP_BLOCK
     25 LOAD_CONST     0 (None)
>>   28 WITH_CLEANUP
     29 END_FINALLY
     30 LOAD_CONST     0 (None)
     33 RETURN_VALUE
```

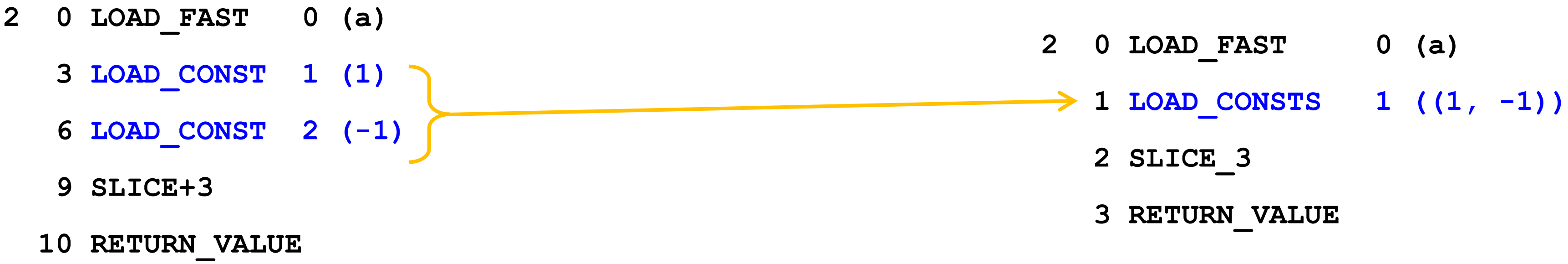
Can be done better

(specialized opcode)

```
2     0 LOAD_GLOB_FAST_CALL_FUNC 0 (open) 0 (name) 1 (1 0)
      2 DUP_TOP
      3 LOAD_ATTR                1 (__exit__)
      4 ROT_TWO
      5 LOAD_ATTR                2 (__enter__)
      6 QUICK_CALL_PROCEDURE     0 (0 0)
      7 SETUP_FINALLY            2 (to 10)
3     8 POP_BLOCK
      9 LOAD_CONST               0 (None)
>>   10 WITH_CLEANUP
     11 RETURN_CONST             0 (None)
```

Constants grouping on slice

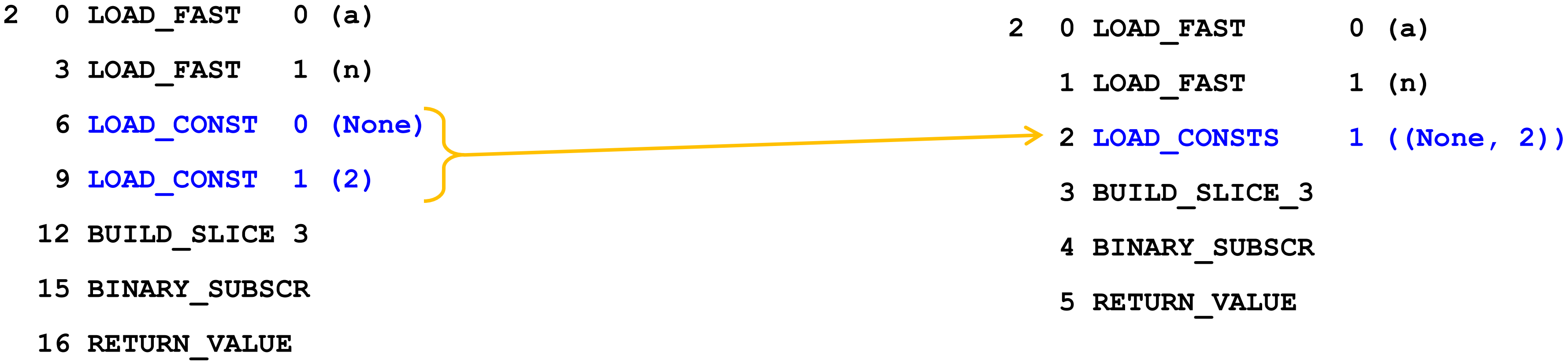
```
def f(a):
  return a[1 : -1]
```



```
2   0 LOAD_FAST    0 (a)
    3 LOAD_CONST   1 (1)
    6 LOAD_CONST   2 (-1)
    9 SLICE+3
   10 RETURN_VALUE
```

```
2   0 LOAD_FAST       0 (a)
    1 LOAD_CONSTS     1 ((1, -1))
    2 SLICE_3
    3 RETURN_VALUE
```

```
def f(a, n):
  return a[n : : 2]
```



```
2   0 LOAD_FAST    0 (a)
    3 LOAD_FAST    1 (n)
    6 LOAD_CONST   0 (None)
    9 LOAD_CONST   1 (2)
   12 BUILD_SLICE  3
   15 BINARY_SUBSCR
   16 RETURN_VALUE
```

```
2   0 LOAD_FAST        0 (a)
    1 LOAD_FAST        1 (n)
    2 LOAD_CONSTS      1 ((None, 2))
    3 BUILD_SLICE_3
    4 BINARY_SUBSCR
    5 RETURN_VALUE
```

The Marshal Matters

```
2    0 LOAD_FAST      0 (a)
     3 LOAD_CONST     1 (1)
     6 LOAD_CONST     2 (10)
     9 LOAD_CONST     3 (-1)
    12 BUILD_SLICE    3
    15 BINARY_SUBSCR
    16 RETURN_VALUE
```

```
def f(a):
  return a[1 : 10 : -1]
```

```
2   0 FAST_BINOP_CONST 0 (a) 1 (slice(1, 10, -1)) 7 ([])
    2 RETURN_VALUE
```

Constant Slice object

Now marshaled

```
2    0 LOAD_FAST       0 (a)
     3 LOAD_CONST      1 (1)
     6 LOAD_CONST      2 (10)
     9 LOAD_CONST      3 (-1)
    12 BUILD_SLICE     3
    15 BINARY_SUBSCR
    16 STORE_FAST      1 (x)
3   19 LOAD_FAST       1 (x)
    22 RETURN_VALUE
```

```
def f(a):
  x = a[1 : 10 : -1]
  return x
```

```
2   0 FAST_SUBSCR_CONST_TO_FAST 0 (a) 1 (slice(1, 10, -1)) 1 (x)
3   2 LOAD_FAST                 1 (x)
    3 RETURN_VALUE
```

Peepholer moved in compile.c

Pros:

- Simpler opcode handling (using instr structure) on average
- Very easy jump manipulation
- No memory allocation (for bytecode and jump buffers)
- Works only on blocks (no boundary calculations)
- No 8 & 16 bits values distinct optimizations
- Always applied (even on 32 bits values code)

Cons:

- Works only on blocks (no global code “vision”)
- Worse unreachable code removing
- Some jump optimizations missing (will be fixed! ;)

Experimental INTEGER opcodes

- Disabled by default (uncomment wpython.h / WPY_SMALLINT_SUPER_INSTRUCTIONS)
- BINARY operations only
- One operand must be an integer (0 .. 255)
- No reference counting
- Less constants usage (to be done)
- Optimized integer code
- Direct integer functions call
- Fast integer “fallback”
- Explicit long or float “fallback”

An example

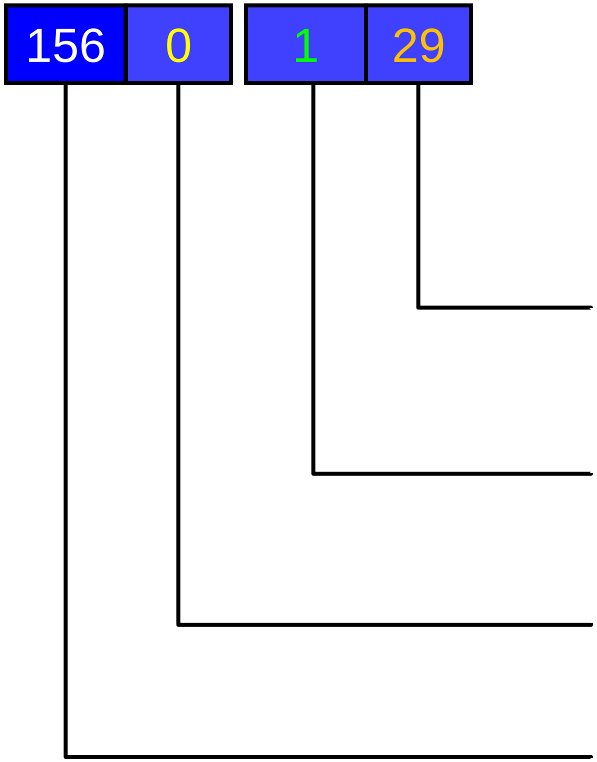
```
def fib(n):
  if n <= 1:
    return 1
  else:
    return fib(n - 2) + fib(n - 1)
```

```
2      0 LOAD_FAST          0 (n)
       3 LOAD_CONST         1 (1)
       6 COMPARE_OP         1 (<=)
       9 JUMP_IF_FALSE      5 (to 17)
      12 POP_TOP
3     13 LOAD_CONST         1 (1)
      16 RETURN_VALUE
>>    17 POP_TOP
5     18 LOAD_GLOBAL        0 (fib)
      21 LOAD_FAST          0 (n)
      24 LOAD_CONST         2 (2)
      27 BINARY_SUBTRACT
      28 CALL_FUNCTION      1
      31 LOAD_GLOBAL        0 (fib)
      34 LOAD_FAST          0 (n)
      37 LOAD_CONST         1 (1)
      40 BINARY_SUBTRACT
      41 CALL_FUNCTION      1
      44 BINARY_ADD
      45 RETURN_VALUE
      46 LOAD_CONST         0 (None)
      49 RETURN_VALUE
```

```
2       0 FAST_BINOP_INT        0 (n) 1 29 (<=)
        2 JUMP_IF_FALSE         2 (to 5)
3       3 RETURN_CONST          1 (1)
        4 JUMP_FORWARD          10 (to 15)
5  >>   5 LOAD_GLOBAL           0 (fib)
        6 FAST_BINOP_INT        0 (n) 2 6 (-)
        8 QUICK_CALL_FUNCTION   1 (1 0)
        9 LOAD_GLOBAL           0 (fib)
       10 FAST_BINOP_INT        0 (n) 1 6 (-)
       12 QUICK_CALL_FUNCTION   1 (1 0)
       13 BINARY_ADD
       14 RETURN_VALUE
   >>  15 RETURN_CONST          0 (None)
```

An INTEGER opcode dissected

FAST_BINOP_INT `0 (n) 1 29 (<=)`



156	0	1	29

Binary operator “sub-opcode”. 29 -> CMP_LE -> “<=“

Short integer value. No “index” on constants table

Index on local variables table. 0 -> “n”

FAST_BINOP_INT “main” opcode

A look inside

```
#define _Py_Int_FromByteNoRef(byte) \
  ((PyObject *) _Py_Int_small_ints[ \
  _Py_Int_NSMALLNEGINTS + (byte)])
```

```
case FAST_BINOP_INT:
    x = GETLOCAL(oparg);
    if (x != NULL) {
        NEXTARG16(oparg);
        if (PyInt_CheckExact(x))
            x = INT_BINARY_OPS_Table[EXTRACTARG(oparg)](
                    PyInt_AS_LONG(x), EXTRACTOP(oparg));
        else
            x = BINARY_OPS_Table[EXTRACTARG(oparg)](
                    x, _Py_Int_FromByteNoRef(EXTRACTOP(oparg)));
        if (x != NULL) {
            PUSH(x);
            continue;
        }
        break;
    }
    PyRaise_UnboundLocalError(co, oparg);
    break;
```

Extracts operator

Extracts integer

Converts PyInt to integer

Converts integer to PyInt

Optimized integer code

```
static PyObject *
Py_INT_BINARY_OR(register long a, register long b)
{
    return PyInt_FromLong(a | b);
}

static PyObject *
Py_INT_BINARY_ADD2(register long a, register long b)
{
    register long x = a + b;
    if ((x^a) >= 0 || (x^b) >= 0)
        return PyInt_FromLong(x);
    _Py_Int_FallBackOperation(PyLong_FromLong,
        PyLong_Type.tp_as_number->nb_add(v, w));
}
```

```
#define _Py_Int_FallBackOperation(
  newtype, operation) { \
    PyObject *v, *w, *u; \
    v = newtype(a); \
    if (v == NULL) \
        return NULL; \
    w = newtype(b); \
    if (w == NULL) { \
        Py_DECREF(v); \
        return NULL; \
    } \
    u = operation; \
    Py_DECREF(v); \
    Py_DECREF(w); \
    return u; \
}
```

Direct integer functions call

```
static PyObject *
(*INT_BINARY_OPS_Table[])(register long, register
long) = {
    Py_INT_BINARY_POWER,  /* BINARY_POWER */
    _Py_int_mul,  /* BINARY_MULTIPLY */
    Py_INT_BINARY_DIVIDE,  /* BINARY_DIVIDE */
```

Defined in intobject.h

```
PyObject *
_Py_int_mul(register long a, register long b)
{
    long longprod;            /* a*b in native long arithmetic */
    double doubled_longprod;  /* (double)longprod */
    double doubleprod;        /* (double)a * (double)b */

    longprod = a * b;
    doubleprod = (double)a * (double)b;
   doubled_longprod = (double)longprod;
```

Implemented in intobject.c

WPython future

- Stopped as wordcodes “proof-of-concept”: no more releases!
- No Python 2.7 porting (2.x is at a dead end)
- Python 3.2+ reimplementation, if community asks
- Reintroducing PyObject “hack”, and extending to other cases
- CPython “limits” proposal (e.g. max 255 local variables)
- Define a “protected” interface to identifiers for VM usage
- Rethinking something (code blocks, stack usage, jumps)
- Tons optimizations waiting…