プログラミング言語処理系論 (9)
Design and Implementation of Programming Language Processors

佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

今日の予定

- VM生成
  - Vmgen
- Internals of VM
  - Cpython VM
- JIT Compiler
  - PyPy→RPython→MetaInterpreter

stack machine vs. register machine

- 微妙なpros/consがある
- 少し古い論文だが

Y. Shi, K. Casey, M. A. Ertl, D. Gregg:

Virtual Machine Showdown: Stack Versus Registers, ACM Transactions on Architecture and Code Optimization, r(4), 2008.

（このグループはvmgenの開発者です）

オチとして

- 命令dispatchの方法の改善が重要
  - vmgenはいろいろ実験ができる。
  - switch文を使ったdispatch
    - ノーマルな方法
  - direct-threaded dispatch
    - Next PCの指す命令に対応するラベルに直接goto
    - 言語のサポート必要(GCCならできる)

VMの実装

- Stack/register machine ISA
- Code/data/stack segment
- Symbol table
- Exception, Synchronization, …

- これらが決まったら、後は各命令の解釈を書く
  - vmgenなど、便利そうに見えるツールがある
  - vmgenは20年前のプロジェクトです

VMGENの強調すること

- VMGENは、Gforthの実装言語として世に出た(らしい)
- http://www.complang.tuwien.ac.at/anton/vmgen/
  - VMのエンジン部分を生成してくれる
- 命令dispatchに凝っている
  - Switch (naïve, simple, but slow)
  - Direct threaded dispatchのサポート

Interpreter Loop

```
vmexec(int pc)
{
  for (;;) {
    switch (codeseg[pc].op)
{
    case OP_NOP:
      NEXTPC;
    case OP_POP:
      stacktop -=
         codeseg[pc].operand
;        NEXTPC;
case OP_U_NOT:
      stackseg[stacktop] =
!stackseg[stacktop];
         NEXTPC;
    case OP_B_ADD:
      binop(+); NEXTPC;
    case OP_B_SUB:
      binop(-); NEXTPC;
```

Direct Threaded Dispatch

```
vmexec(int pc)
{
  for (;;) {
    switch (codeseg[pc].op) {
    case OP_NOP:
      goto codeseg[NEXTPC];
    case OP_POP:
      stacktop -=
          codeseg[pc].operand;
       goto codeseg[NEXTPC];
```

```
case OP_U_NOT:
      stackseg[stacktop] =
!stackseg[stacktop];
      goto codeseg[NEXTPC];
case OP_B_ADD:
      binop(+);
      goto codeseg[NEXTPC];
    case OP_B_SUB:
      binop(-);
      goto codeseg[NEXTPC];
```

VMの設計

- VMを設計する

- Engine部分なら、ごく簡単
- load/link、データ領域の確保を忘れずに
- 関数コール
- 例外処理、スレッド等、並列サポートもあればなおよい
- 教材のVMは、何の工夫もないgenericな実装方法を採用しています

ターゲット言語の重要な要素

- オブジェクト指向言語
  - オブジェクト
  - Method invocation
    - フレームは普通にスタックに積むのだが、Java VMではそこまでは強制されていない
- スクリプト言語
  - 動的型付け
  - 文字列の操作が…

Internals of Python VM

- 以下、Python 3.10.4で解説
- CPython ではどうなっているか？
- 関係する部分はceval.c
  - コード生成の詳細を確認するためにcompile.cの内部を確認する必要がある
    - disモジュールの出力だけでは…
  - Objects/ でのクラスの実装、Lib/ でのサポート関数、Modules/ での基本モジュールを参照する必要がある

データ型

- Javaのようなstrictな型チェックをコンパイル時にしない
  - Binary addはNumberを対象に (int/float/complexは実行時にチェック)
- データ型
  - Number
  - Iterator
  - Sequence
    - list, tuple, range
  - Text Sequence (=string)
  - Binary Sequence
  - Set
  - Mapping (Dictionary)
  - Context manager

データと制御

- オブジェクトに使うデータ構造の確認
  - Objects/
- 実行時型チェック
  - ＝ オブジェクトに適用できるメソッドの管理
- 制御構造の確認
  - 関数コール
  - Try and except

PythonのISA ()

- Coroutine関係
  - get_awaitable, get_aiter, yield_from, …
- Build_(データ型)
  - build_{tuple, list, set, map, iter, …}
- Iterationの実行
  - get_iter, get_aiter
- 特殊なデータ型操作のサポート
  - map_add, set_add, list_append, …
- Binary_matrix_matrix (!)

実行エンジン

- Import
  - PyImport_AddModuleObject()
  - PyModule_GetDict()
- run_mod (pythonrun.c)
  - PyAST_CompileObject() ← コンパイル
  - run_eval_code_obj() ← Go

run_eval_code_obj() in pythonrun.c

- pythronrun.cからrun_eval_code_obj()をコール
- ceval.c 中のPyEval_EvalCode() → _PyEval_Vector→_PyEval_EvalFrame →PyEval_EvalFrameDefault()
  - Frameの生成
  - 必要に応じて辞書の調整 (keyword, positional引数の処理)
  - Cell vars/free vars用のメモリの初期化
  - Generator, coroutine, asyncの初期化

- Interpreterとして_PyEval_EvalFrameDefault()をコール
  - この関数はPyInterpreterstate_New() (pystate.c) で設定 (pylifecycle.c)
- EvalFrameDefault()
  - Javaの記述を思い出す

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

メインループの構造

```
_PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
{
…
main_loop:
   for(;;)
            例外のチェック
            GILの獲得
            switch (opcode) {
             …      ← この部分がメインのエンジン
            }
            error:…
            exception_unwind:…
（errorとexception_unwindでtry and exceptを実現）
            exit_returning:…
            exit_yielding:…
}
exit_eval_frame:…
```

議論する点

- Computed goto (threaded code)の利用（vmgenを思い出す）

```
#if USE_COMPUTED_GOTOS
#define TARGET(op) op: TARGET_##op
#define DISPATCH() ¥
    { ¥
        if (trace_info.cframe.use_tracing OR_DTRACE_LINE OR_LLTRACE) { ¥
            goto tracing_dispatch; ¥
        } ¥
        f->f_lasti = INSTR_OFFSET(); ¥
        NEXTOPARG(); ¥  ← opcode, opargのセット
        goto *opcode_targets[opcode]; ¥ ← main_loopの該当部分に直接ジャンプ
    }
#else
#define TARGET(op) op
#define DISPATCH() goto predispatch;
#endif
```

- 命令のprediction

```
 #define PREDICT_ID(op)          PRED_##op

#if defined(DYNAMIC_EXECUTION_PROFILE) || USE_COMPUTED_GOTOS
#define PREDICT(op)             if (0) goto PREDICT_ID(op)
#else
#define PREDICT(op) ¥
    do { ¥
        _Py_CODEUNIT word = *next_instr; ¥
        opcode = _Py_OPCODE(word); ¥
        if (opcode == op) { ¥
            oparg = _Py_OPARG(word); ¥
            next_instr++; ¥
            goto PREDICT_ID(op); ¥
        } ¥
    } while(0)
#endif
```

- 例えば以下のように使う

```
case TARGET(GET_ITER): {
        /* before: [obj]; after [getiter(obj)] */
        PyObject *iterable = TOP();
        PyObject *iter = PyObject_GetIter(iterable);
        Py_DECREF(iterable);
        SET_TOP(iter);
        if (iter == NULL)
            goto error;
        PREDICT(FOR_ITER);
        PREDICT(CALL_FUNCTION);
        DISPATCH();
    }
```

Predict

- Speculationのつもりで書いた？
  - あまり似ていない？
- もちろん、これはthreaded codeと相性が悪い
  - どちらかがdisableされる
- 性能的にどうなのか疑問である
  - COMPUTED GOTOがサポートされればもはや不要
  - 報告が（ネット上にすら）見当たらない

ceval.c中のBINARY_ADDの部分

```
case TARGET(BINARY_ADD): {
        PyObject *right = POP();
        PyObject *left = TOP();
        PyObject *sum;
        if (PyUnicode_CheckExact(left) &&
                PyUnicode_CheckExact(right)) {
            sum = unicode_concatenate(tstate, left, right, f, next_instr);
            /* unicode_concatenate consumed the ref to left */
        }
        else {
            sum = PyNumber_Add(left, right);
            Py_DECREF(left);
        }
        Py_DECREF(right);
        SET_TOP(sum);
        if (sum == NULL)
            goto error;
        DISPATCH();
    }
```

各Instructionの実行

- スタックマシン
  - TOP(), POP(), PUSH()で、スタック上のデータのハンドリング
  - 実行部分を提供
    - unicode_concatenate(tstate, left, right, f, next_instr);
    - PyNumber_Add(left, right);

Data Areaへのアクセス

- load_const
- load_name
  - Nameにassociateされた値のロード
- load_attr
  - Fieldへのアクセス
- load_global
  - reference by name
- load_fast
  - Frame上のvariableへのアクセス
- load_closure
- load_deref
- load_classderef

- Objects/abstract.c中のPyNumber_Addがこれがまた…
  - Objectの持つ演算の取り出し(メソッド)
  - binary_op1(v, w, NB_SLOT(nb_add))
  - NB_SLOTとは…

Numberの実装

PyNumberMethods (Includes/cpython/object.h)

```
typedef struct {
    /* Number implementations must check *both*
       arguments for proper type and implement the necessary conversions
       in the slot functions themselves. */

    binaryfunc nb_add;
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
   …
```

- 普通、nb_addには何が入っているか
  - __add__
  - float_add
  - long_add

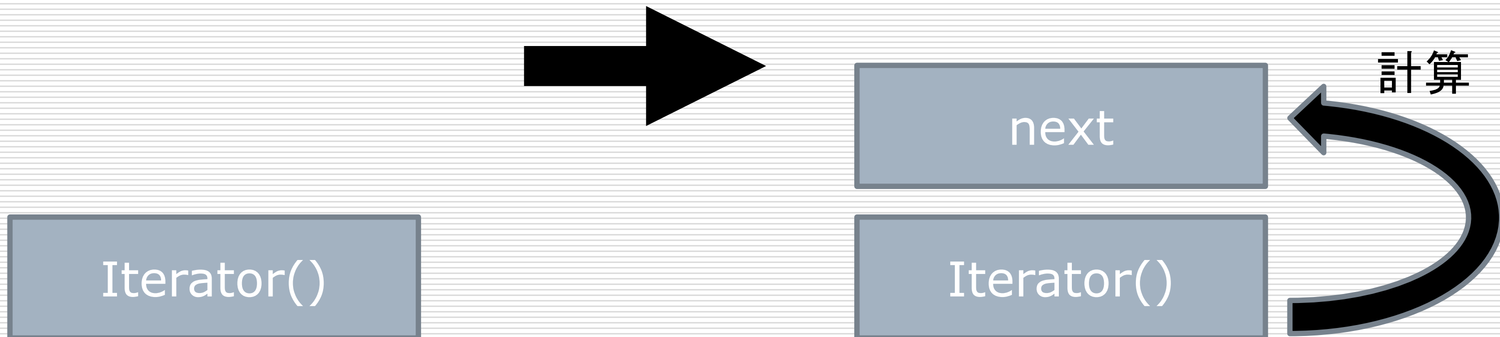
- __add__ in Lib/_pydecimal.py

Iterator

```
case TARGET(FOR_ITER): {
        PREDICTED(FOR_ITER);
        /* before: [iter]; after: [iter, iter()] *or* [] */
        PyObject *iter = TOP();
        PyObject *next = (*iter->ob_type->tp_iternext)(iter);
        if (next != NULL) {
            PUSH(next);
            PREDICT(STORE_FAST);
            PREDICT(UNPACK_SEQUENCE);
            DISPATCH();
        }
}
        if (_PyErr_Occurred(tstate)) {
            if (!_PyErr_ExceptionMatches(tstate, PyExc_StopIteration)) {
                goto error;
            }
            else if (tstate->c_tracefunc != NULL) {
                call_exc_trace(tstate->c_tracefunc, tstate->c_traceobj, tstate, f, &trace_info);
            }
            _PyErr_Clear(tstate);
        }
        /* iterator ended normally */
        STACK_SHRINK(1);
        Py_DECREF(iter);
        JUMPBY(oparg);
        DISPATCH();
    }
```

FOR_ITER

- tp_iternext メソッドを各クラスに用意



callの実装: Calling Convention

- Caller側:
  - stack上 0: function reference (push)
  - stack上 1—n: 引数 (push)
  - Call_function (n)
- Callee側
  - load_fast 0—(n-1): 引数へのアクセス (frame)

callの実装

```
case TARGET(CALL_FUNCTION): {
        PREDICTED(CALL_FUNCTION);
        PyObject **sp, *res;
        sp = stack_pointer;
        res = call_function(tstate, &sp, oparg, NULL);
        stack_pointer = sp;
        PUSH(res);
        if (res == NULL) {
            goto error;
        }
        DISPATCH();
    }
```

- call_function()で、フレームの調整をしてからコール
- 例えば、native コードでのcallは、return addressをスタックに積んでjumpするだけだが…

- function_call() → PyObject_VectorCall(これ自体はinline手介されて_PyObject_VectorCallTstateをコール)→
-

```
static inline PyObject *
_PyObject_VectorcallTstate(PyThreadState *tstate, PyObject *callable,
                           PyObject *const *args, size_t nargsf,
                           PyObject *kwnames)
{
    vectorcallfunc func;
    PyObject *res;

    assert(kwnames == NULL || PyTuple_Check(kwnames));
    assert(args != NULL || PyVectorcall_NARGS(nargsf) == 0);

    func = PyVectorcall_Function(callable);
    if (func == NULL) {
        Py_ssize_t nargs = PyVectorcall_NARGS(nargsf);
        return _PyObject_MakeTpCall(tstate, callable, args, nargs, kwnames);
    }
    res = func(callable, args, nargsf, kwnames);
    return _Py_CheckFunctionResult(tstate, callable, res, NULL);
}
```

Cの配列として与える

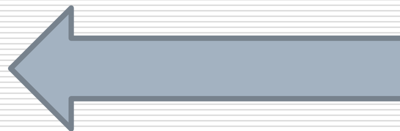
Objects/call.c(これは3.8の古いコードで少しわかりやすい)

```
static PyObject* _Py_HOT_FUNCTION
function_code_fastcall(PyCodeObject *co, PyObject *const *args, Py_ssize_t nargs,
                       PyObject *globals)
{
    PyFrameObject *f;
    PyThreadState *tstate = _PyThreadState_GET();
    PyObject **fastlocals;
    Py_ssize_t i;
    PyObject *result;

    …
    f = _PyFrame_New_NoTrack(tstate, co, globals, NULL);
    if (f == NULL) {
        return NULL;
    }

    fastlocals = f->f_localsplus;

    for (i = 0; i < nargs; i++) {
        Py_INCREF(*args);
        fastlocals[i] = *args++;
    }
    result = PyEval_EvalFrameEx(f,0);
```



Stack → Frame内の局所変数へ

Try and catch (except)の処理

- JVMの解説（前回）で、この処理が腕の見せ所だと言いました
  - CPythonの場合、GIL（global interpreter lock）の解放等の処理を含む
  - Interpreter loopの先頭ごとにチェックすると効率が落ちる
    - Computed gotoで、interpreter loopの実行は、for(;;)を通らないことも多い
    - シグナルが発生するようなところを把握しておけば、効率が落ちないかも

Try and Except

- compile.cの中のコードを確認

```
compiler_try_except(struct compiler *c, stmt_ty s)
{
    basicblock *body, *orelse, *except, *end;
    Py_ssize_t i, n;

    body = compiler_new_block(c);
    except = compiler_new_block(c);
    orelse = compiler_new_block(c);
    end = compiler_new_block(c);
    if (body == NULL || except == NULL || orelse == NULL || end == NULL)
        return 0;
    ADDOP_JUMP(c, SETUP_FINALLY, except);
    compiler_use_next_block(c, body);
    if (!compiler_push_fblock(c, EXCEPT, body, NULL))
        return 0;
    VISIT_SEQ(c, stmt, s->v.Try.body);
    ADDOP(c, POP_BLOCK);
    …
```

Try: S except E1 as V1:...

```
    SETUP_FINALLY  L1
    <code for S>
    POP_BLOCK
    JUMP_FORWARD L0 (finallyに)
L1: DUP
<evaluate E1>
```


…

ceval.cのmain_loop （冒頭部分）

```
if (_Py_atomic_load_relaxed(eval_breaker)) {
    opcode = _Py_OPCODE(*next_instr);
    if (opcode != SETUP_FINALLY &&
        opcode != SETUP_WITH &&
        opcode != BEFORE_ASYNC_WITH &&
        opcode != YIELD_FROM) {
        /* Few cases where we skip running signal handlers and other
           pending calls:
           - If we're about to enter the 'with:'. It will prevent
             emitting a resource warning in the common idiom
             'with open(path) as file:'.
           - If we're about to enter the 'async with:'.
           - If we're about to enter the 'try:' of a try/finally (not
             *very* useful, but might help in some cases and it's
             traditional)
           - If we're resuming a chain of nested 'yield from' or
             'await' calls, then each frame is parked with YIELD_FROM
             as its next opcode. If the user hit control-C we want to
             wait until we've reached the innermost frame before
             running the signal handler and raising KeyboardInterrupt
             (see bpo-30039).
        */
        if (eval_frame_handle_pending(tstate) != 0) {
            goto error;
        }
    }
}
```

ループのお尻の部分

```
error:
   …
exception_unwind:
        f->f_state = FRAME_UNWINDING;
        /* Unwind stacks if an exception occurred */
        while (f->f_iblock > 0) {
            /* Pop the current block. */
            PyTryBlock *b = &f->f_blockstack[--f->f_iblock];

            if (b->b_type == EXCEPT_HANDLER) {
                UNWIND_EXCEPT_HANDLER(b);
                continue;
            }
            UNWIND_BLOCK(b);
            if (b->b_type == SETUP_FINALLY) {
                PyObject *exc, *val, *tb;
                int handler = b->b_handler;
                _PyErr_StackItem *exc_info = tstate->exc_info;
                /* Beware, this invalidates all b->b_* fields */
                PyFrame_BlockSetup(f, EXCEPT_HANDLER, f->f_lasti, STACK_LEVEL());
                PUSH(exc_info->exc_traceback);
                PUSH(exc_info->exc_value);
                if (exc_info->exc_type != NULL) {
                    PUSH(exc_info->exc_type);
                }
                else {
                    Py_INCREF(Py_None);
                    PUSH(Py_None);
                }
```

Try and exceptionの実行？

- Computed gotoがあるために、エラーが起きない限り、interpreter loopの内側でgotoが連続する
- 命令の実行でエラーが観察されるときにgoto errorで、main_loopの終わり→先頭にくる
  - Try and exceptionの処理が始まる

```
case TARGET(SETUP_FINALLY): {
            PyFrame_BlockSetup(f, SETUP_FINALLY, INSTR_OFFSET() + oparg,
                               STACK_LEVEL());
            DISPATCH();
        }
```

- PyFrame_BlockSetupで、ジャンプ先を指定。メインループでシグナルをチェック（ループの先頭）、exception_unwindでジャンプするところを指定する

Concurrency関係

- Thread based concurrency
- 実際は、ループの非同期実行 (coroutineで間に合うという強弁が…)
- Critical sectionの管理をgilで行う
  - threadが並列に動かない
- VM(プロセス)を複数持って管理

関係する命令

- GET_AITER, GET_ANEXT
- BEFORE_ASYNC_WITH, BEGIN_FINALLY, END_ASYNC_FOR, SETUP_ASYNC_WITH
- GET_AWAITABLE
- YIELD_FROM, YIELD_VALUE

- Compileでは、decorator (@asyncio.coroutine) またはqualifier (async) で指定

問題9

- Stack machineのVMを一つ設計せよ
  - 命令セットはODC.zip中のopcode.hでよい。
    - 例外処理しない。スレッドの管理しない。
    - ここまで簡単化すればvmgenは（古いツールだが）役に立つかもしれない
  - stack segmentとdata segmentを定義すること
  - 各命令を実行したときのstack/data segmentの状態変化を記述すること
    - OP_CALL とOP_RETURNについては詳述すること
    - この記述に従ってvmgenでVMを作成するとなおよい
  - コンパイラを作る必要はない

動的言語では

- 動的な型変換が性能にどれだけのマイナス要因になっているかを理解する
  - TypeScriptでは、「静的な型システム」がソフトウェア工学的に重要であることを説明しました
  - 実行性能の面でも、動的な型チェックが負担になっていることが分かったと思います
- JVM上でスクリプト言語を実行する場合
  - 引数の動的型チェックを行う
  - 引数の型チェックをできるものは静的にすませてしまう

動的型変換においては

- ルーチン内で、型が伝播する可能性がある
  - ローカルな解析をもとにして
  - データフロー解析をもとにして
- ルーチンをまたいで、型が分かる場合がある
  - 手続き間解析をもとにして

- ___add___のようなgenericなルーチンを用意するのではなく、float_add, long_addを直接適用できるようにならないか？

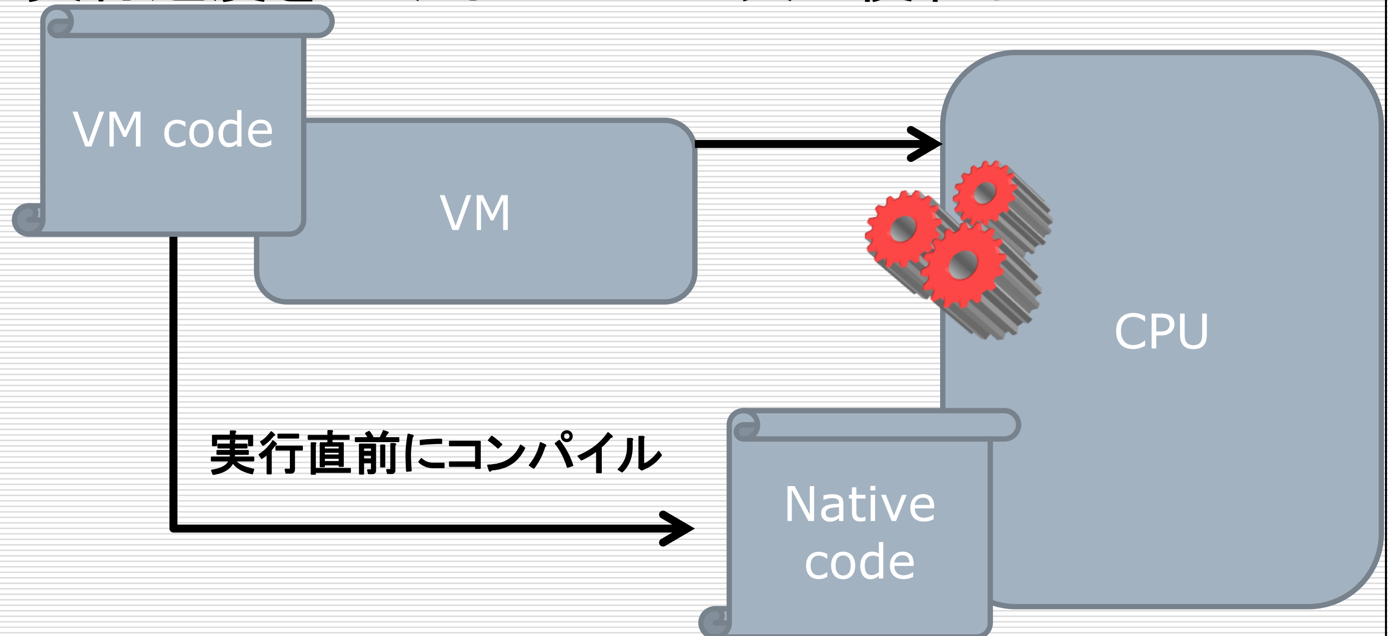
PyPyではどうだろうか

- pypy/interpreter/pyopcode.py
  - 典型的なinterpreter loopになっている
    - while True
      ...
      if ...
  - 最後にあるこれはなんだ…

```
if jit.we_are_jitted():
        return next_instr
```

Just in Time Compiler

- 昔、そういう技術がありまして… Native Codeに変換
- Javaの実行速度を上げるための工夫に使われた



- 実用的には、このアイデアはいろいろなところに残っている
  - LTOなど

JIT Compilerに採用されている技術

- Tracing JIT
  - 実行ログ(トレース)を取っておき、実行頻度の高いものをコンパイルする
    - Java HotSpot (1999--)
    - Javascript Mozilla TraceMonkey (2008--)
- MetaInterpreter (2009--)
  - VMのinterpreterのトレースを取っておき、実行頻度の高いものをコンパイルする
  - ここで、対象になるのはdispatcher loop
  - PyPy (RPython)

JIT Compilerの評価

- どの部分を動的にコンパイルするかによって性能が決まる
  - 動的に動くルーチンを含むようなものならばあまり効果はないだろう
    - たとえばJavaScriptのDOM関係
    - JavaScriptは、この手のコードの実行が大きな割合を占める→役に立つのか？
- 単純な計算のループならば、速くなるだろうが…

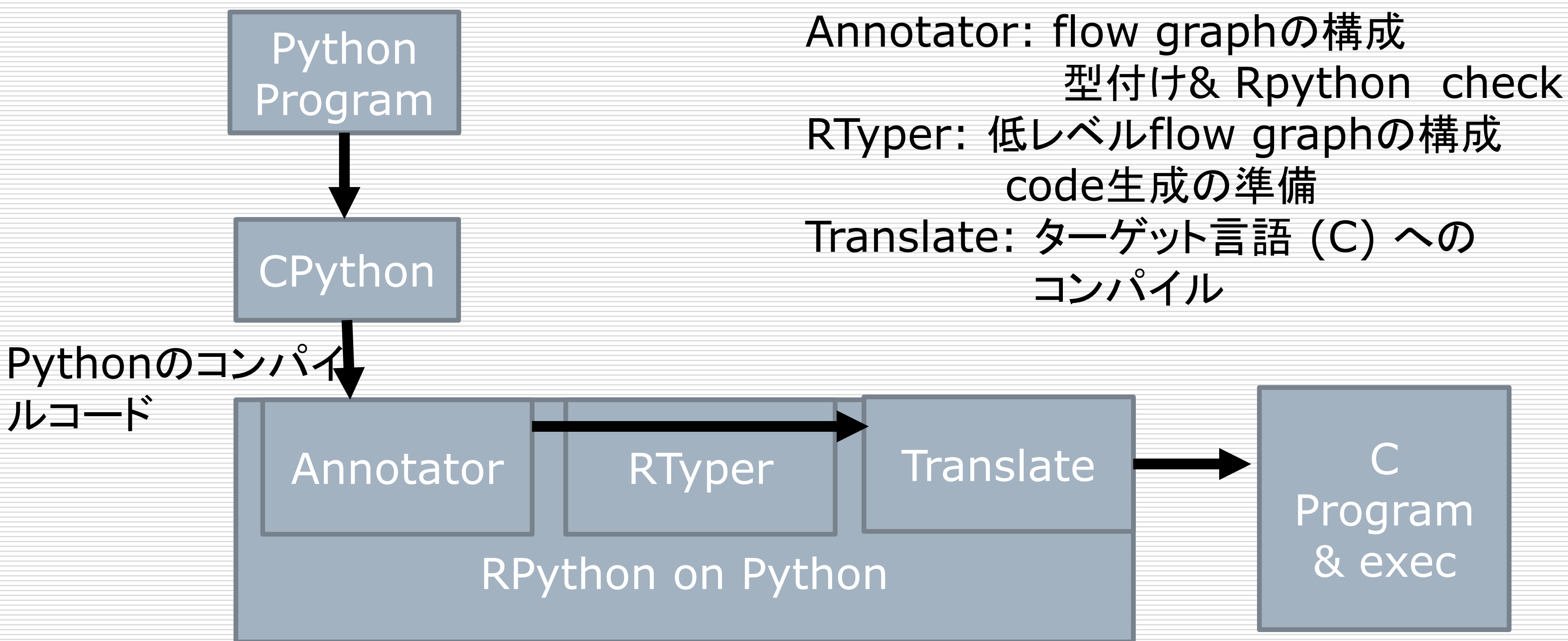
PyPy

- PyPyのJIT
- MetaInterpreter
  - C.F. Bolz “Meta-Tracing Just-in-Time Compilation for Rpython,” PhD thesis, Heinrich-Heine-Universitat Dusseldorf, 2012.
  - 10年前のD論です

RPython

- PythonをCに変換するために制限を加える
  - 型をつける（基本型と静的にサイズがわかるリストに限る）
  - データの大きさが変更されないと、Cへの変更が楽になる
    - 少なくともGCが呼ばれるタイミングがわかる
- Pythonのバイトコードをそのまま受け入れ、サポートしていない命令を検知すると「RPythonではない」とレポート
  - この部分は、annotatorのフェーズで行われる

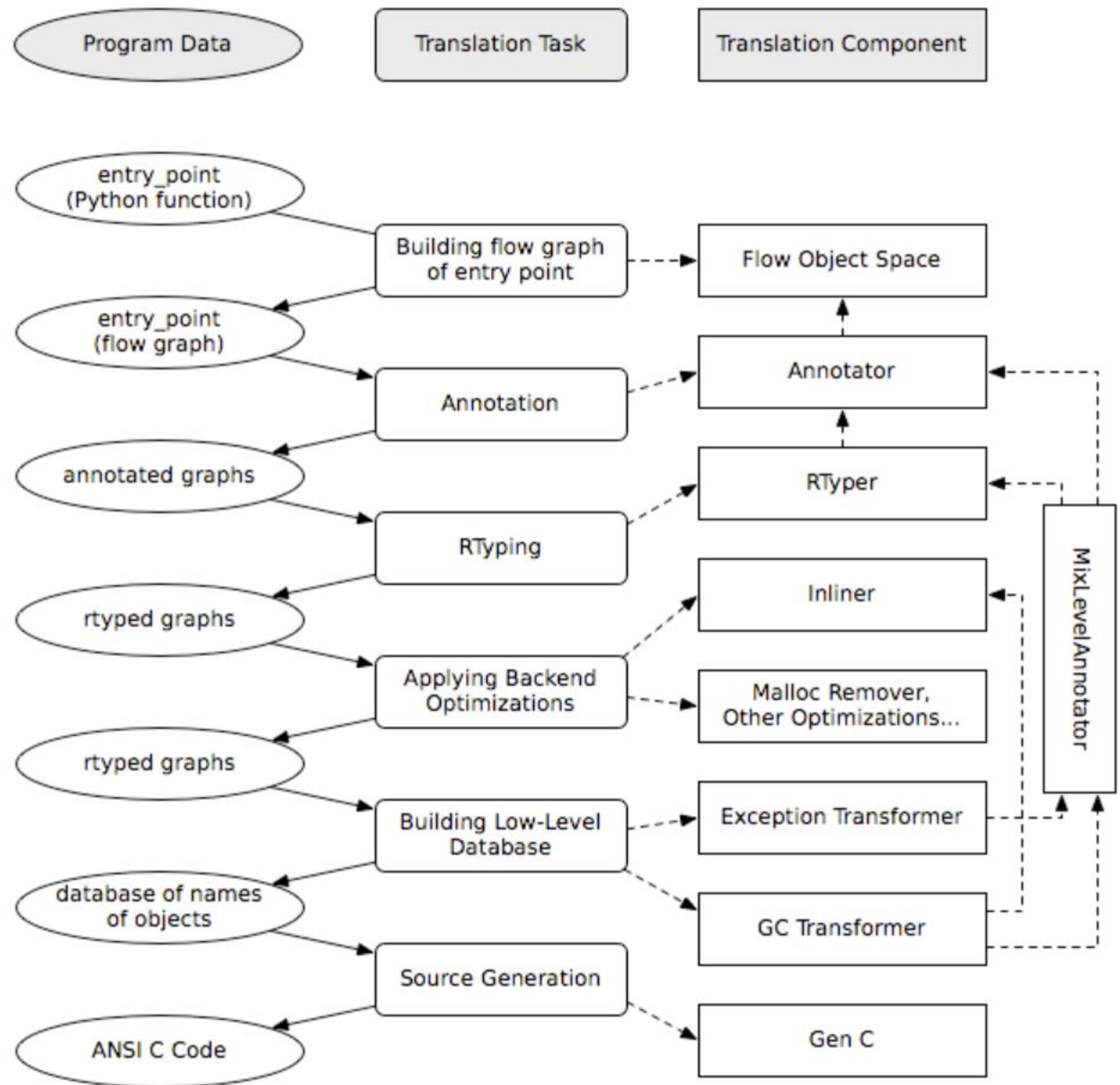
PyPyの全体像の図式



Annotator: flow graphの構成
型付け& Rpython check
RTyper: 低レベルflow graphの構成
code生成の準備
Translate: ターゲット言語 (C) へのコンパイル

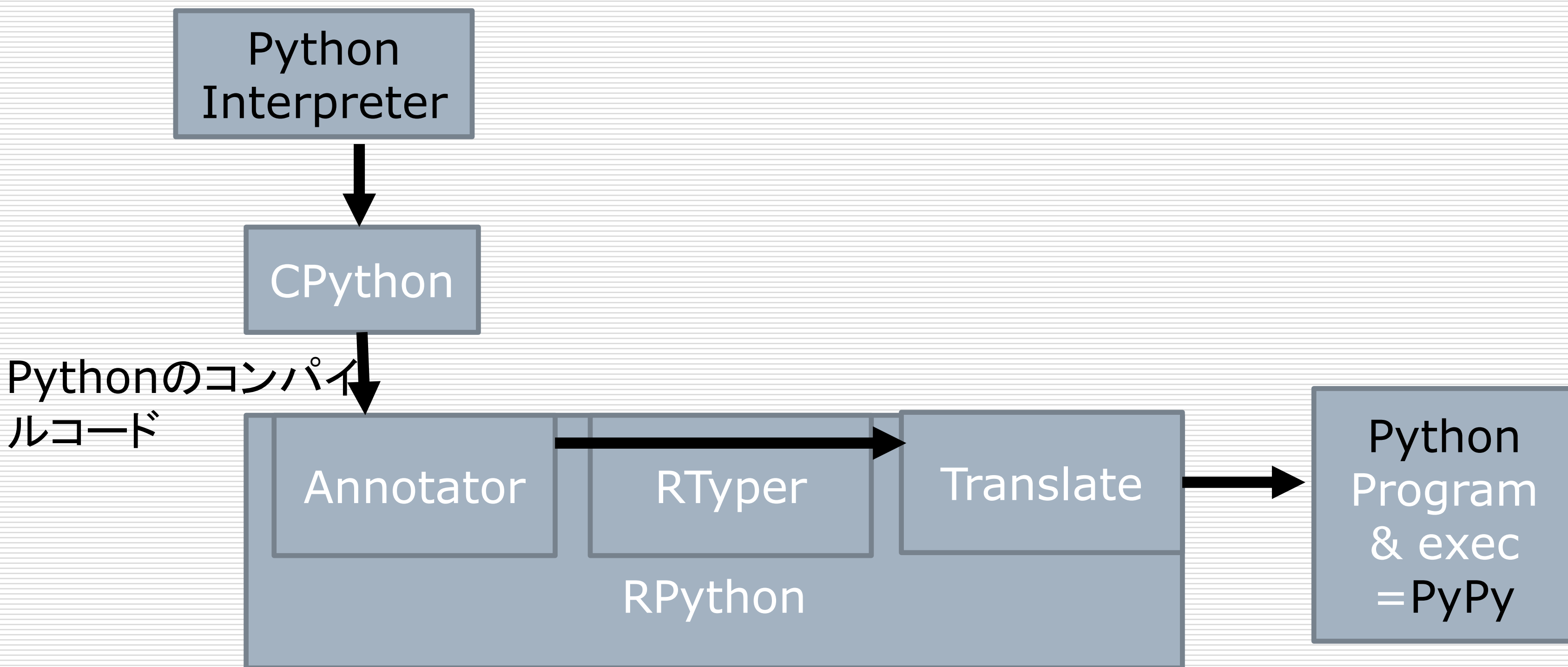
Rpytho

- RPython Documentationからの Copy&Paste



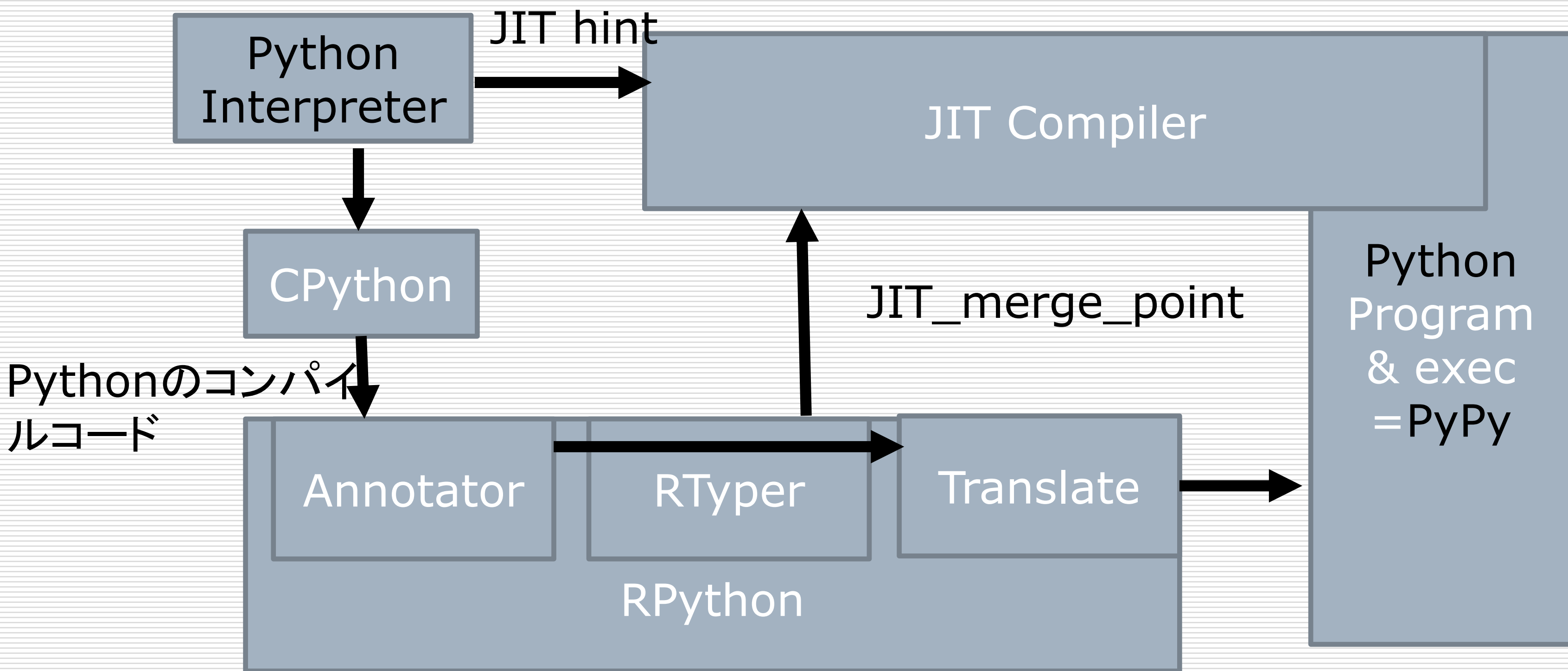
PyPy

- PyPyの全体像の図式



PyPyとJIT

- PyPyの全体像の図式



RPythonの具体的な動作

- RTyperのフェーズ
  - jit_merge_pointを生成
- apply_jit()で、このpointをmaybe_compile_and_run()に置き換え
- Traceを収集し続け、hotspotと判断した時点でnative codeにコンパイル

Low Level Code
…
…
jit_merge_point
…
…

Jit enabled
…
…
maybe_compile_and_run()
…
…

MetaInterpreter（最適化の対象）

- ループ
  - Pythonで書いたループ
    - RPythonのinterpreterの中でも、dispatcher loopの中でのループになっている
    - ということで、Pythonで書いたループではなく、RPythonのループを対象にした
      - PyPy用に最適化ルーチンを書くのではなく、RPython用に最適化ルーチンを書くことで、概念を一般化＝MetaInterpreter
  - （問題10）以下の論文を要約せよ。3—7章だけで構わない
    C.F. Bolz "Meta-Tracing Just-in-Time Compilation for Rpython," PhD thesis, Heinrich-Heine-Universitat Dusseldorf, 2012.

ループアンローリング

- ループを、VM上で実行するのではなく、CPU上で実行する
- ！VMの実行は、高速化の余地が少ない（後でpeephole最適化をやるけど、それくらいしか候補がない）
- CPUは、複雑な実行エンジンを持っているので、高速化の余地がたくさんある
- ループの最適化の中で、CPU orientedなものとして有名なのがloop unrolling

Loop Unrolling Really Effective?

- なぜ、ループアンローリングは性能が出るのか？
  - Unrolling and jam
  - 一つのループ中の命令数が増えれば、最適化の余地がその分大きくなるという発想（VLIWの時に提案され、superscalarでも有効であることが確認された）
- 本当に性能が出るのか？
  - 現状のCPUアーキテクチャでどの程度有効か？
    - 深いパイプライン
    - Speculative execution

Loop Unrolling Challenge

- (問題 11)以下のプログラムが、今使われている代表的なCPU上で、unrollingを使って速くなるかどうかを検証せよ
  - 行列積(double/int)
  - キャッシュ最適化の効果と分別できるように、十分小さなループで検証すること(サイズを次第に大きくして言って、キャッシュの効果が現れ始める直前でやめることはできるか？)
  - 小さいループでは、実行時間の計測が不安定になるが、それを安定させるためにはどうしたらよいか考えてみよ
  - CPUとコンパイラを明記すること
  - コンパイラによっては、勝手にunrollする最適化を行うものがあるので、アセンブリ出力、又はコンパイラのメッセージ出力を確認して、unrolling段数を正しく把握することが必要