Summary: A Just-In-Time Compiler for Python

September 21, 2020

Architecture

We have a global state **JIT STATE** to maintain JIT compiler-awared code, generated IRs, the information to tell whether a Python object has been or can be JIT-ed, and counters for hot paths, etc.

Given the Python source code, the standard compiler of Python will get us the Python bytecode instructions and associate runtime objects.

We then convert Python instructions into **Core CPY**, which is more compact and easier to handle in the following pipeline.

Afterwards, by specifying a global variable context indicating non-volatile global variables, we can specialize **Core CPY** into a register based IR, hereafter as **Dynjit**.

The process of transforming **Core CPY** into **Dynjit** is sophiscated and considered as the major part of our JIT compiler.

This transformation takes advantage of **JIT STATE** and assumptions of Python builtins, to statically partial evaluate the programs to produce the register based IR **Dynjit**.

The top-level partial evaluation happens when calling the "main" function. This does not mean we're restricted to programs containing a "main" function, but rather indicates that the point in time to perform runtime specialization. Any function whose invocation triggers runtime specialization can be regarded as a "main" function.

By knowing the argument types of a "main" function, we can perform many kinds of specializations in the program. The function calls inside "main" function can be specialized while specializing "main" function, in other words, it is possible to only trigger runtime code generation once to run your program.

However, not every piece of Python code can be efficiently optimized by our JIT compiler.

Changing global variables(**global**), changing free variables in nested functions(nonlocal), and many other bad practice will not only ruin the readability or maintainability, but also affect the performance of our JIT compiler.

Finally, the JIT compiler is guaranteed to work fine for code written under a good specification, which is given in the Appendix(Appendix is WIP).

Overview: Capability of the JIT Compiler

Our JIT compiler mainly does 3 kinds of specializations.

- Method lookup specializations
- Control flow specializations
- Union splitting specializations

Method Lookup Specializations

Given the following code, and use function f as a "main" function.

```
class S:
def f(self):
    ...

def f(x):
    return x.f()

# 'f' is a 'main'
f(S())
```

After partial evaluation for *f(S())* , we can know that the argument x has type S, hence we can load the method object *S.f* as a constant, and call *S.f(x)*.

Note that in standard Python interpreter, every time accessing a method or field will constantly lookup hash tables.

By gathering method lookup specialization and Python-to-C compilation, we can eliminate the overhead of method lookup.

Performance gain of this optimization is more significant when it comes to larger inheritance chains.

Control Flow Specializations

```
def f(x):
    if isinstance(x, int):
        return x + 10

    return 0
f(1)
```

Above code shows a case that the condition expression of if-else is specialize-able.

By giving the correct global variable context, we can recognize Python built-ins from the symbol *int* and *isinstance*.

Under this situation, our compiler can partial evaluate *isinstance(x, int)* to a constant *True*, and then partial evaluate *if True*, statically select the first arm and eliminate the redundant branch.

So the function call *f(1)* becomes in fact something like *f_int(1)*, where

```
def f_int(x: int):
    return x + 10
```

Of course, the specialized code is generated C code. Above Python code about *f_int* is made for understandability.

Union Splitting Specializations

Sometimes an expression can hold union types.

```
def one_of_string(x):
    if x == "int":
        return 1
    elif x == "float":
        return 1.0
    elif x == "str":
        return "_"
    else:
        raise ValueError

def two_of_string(x):
    x = zero_of_string(x)
    return x + x


two_of_string("int")
```

The "main" function here is *two_of_string*, however, the argument type is simply a *str*, and the JIT compiler presented here does not know the argument value.

So the return *two_of_string(x)* will be a union of *int*, *float*, and *str*, which will prevent the specialization of the later expression $x + x$.

To address this, we use the idea "union splitting" from Julia programming language.

After calling a function, the return can have union types.

We will insert code for union splitting specialization to somewhere prior to the first time we use the value of union type as function argument.

In the example of *two_of_string(x: str)*, we will specialize it to

```
def two_of_string(x: str):
    x = zero_of_string(x)
    switch type(x):
    case int:
        return plus_int(x, x)
    case float:
        return plus_float(x, x)
    case str:
        return plus_str(x, x)
```

We contrive a switch statement for Python, to express the type dispatching semantics of generated C code.

*plus_** are the specialized methods for + function. It is worth noting that every operation in Python bytecode is converted to a function call in **Core CPY**.

Core CPY

Core CPY is a simplification of Python bytecode instructions.

Instructions of Core CPY

$$
\begin{array}{rcl}
\langle \text{instr} \rangle & ::= & \mathit{Const}\ \textbf{constant} \\
& | & \mathit{Load}\ \textbf{symbol} \\
& | & \mathit{Store}\ \textbf{symbol} \\
& | & \mathit{Jump}\ \textbf{int} \\
& | & \mathit{Call}\ \textbf{int} \\
& | & \mathit{Rot}\ \textbf{int} \\
& | & \mathit{Peek}\ \textbf{int} \\
& | & \mathit{Pop} \\
& | & \mathit{Return} \\
& | & \mathit{JumpIf}\ \textbf{bool bool int}
\end{array}
$$

Notations and Prerequisites

The semantics here is not complete, but it is unnecessary to be complete.

Many parts, such as Python data models, function closures, default arguments, keyword arguments, variadic arguments, etc., are omitted in Core CPY.

This is a must because any of those omitted parts will involve a book longer than this document, and it is sufficient to understand the semantics by the following alternative approach.

Instead of elaborating how values transform with respect to heaps on each instruction of Core CPY, we gives semantics targeting equivalent Python code. In this regarding, we can use built-ins from Python runtime to avoid massive but quite uninformative explanations.

Specifically, the term "Python object" used here, specially means a regular value in Python runtime, which is compatible to *PyObject** in C level.

Following are about the notations we need for the semantics.

Like Python bytecode instructions, Core CPY also leverages a value stack, which can be denoted with S.

Besides, we need

- $i, i_0, i_1, \cdots$, an integer.
- $n, n_0, n_1, \cdots$, a symbol, a name.
- $a, a_0, a_1, \cdots$, a Python object.
- a sequence of instruction $\mathbf{I}$, can be indexed like $I[i]$.
- A map Δ to map a name n to the index of value stack $\Delta(n)$. This is fixed for every function/instruction sequence.
- The side-effect-free substitution operation $\cdot[\cdot \leftarrow \cdot]$. Given an offset of value stack $i \in \mathbb{N}$, a Python object v, $S[i \leftarrow v]$ returns a new stack S' which equals to S except $S[i]$ is v.
- $\{\cdot\}_{python}$ means evaluating the Python code, and returns a Python object

The value stack S is not empty in the beginning. It contains arguments and free variables.

```
def f(x,y,z):
    a = x + 1
    return (a, x)

f(1.0, 2, "3.0")
```

The function call *f(1.0)*, will initialize a value stack $[1.0, 2, \text{"3.0"}]$.

Semantics

VAR

$$\frac{}{S \vdash_{r-val} n \Rightarrow S[\Delta(n)]}$$

INSTR

$$\frac{\_, \overbrace{\cdots}^{S'} = S}{S \vdash_{instr} \textbf{Pop} \Rightarrow S'} \quad \frac{a_1, \cdots, a_i, a_{peek}, \cdots = S}{S \vdash_{instr} \textbf{Peek}\ i \Rightarrow a_{peek}, \overbrace{\cdots}^{S}}$$

$$\frac{S \vdash_{r-val} n \Rightarrow a}{S \vdash_{instr} \textbf{Load}\ n \Rightarrow a, \overbrace{\cdots}^{S}} \quad \frac{}{S \vdash_{instr} \textbf{Const}\ a \Rightarrow a, \overbrace{\cdots}^{S}}$$

$$\frac{a_1, a_2, \cdots, a_i, \overbrace{\cdots}^{S'} = S}{S \vdash_{instr} \textbf{Rot}\ i \Rightarrow a_i, a_1, a_2, \cdots, a_{i-1}, \overbrace{\cdots}^{S'}}$$

$$\frac{a, \overbrace{\cdots}^{S'} = S \quad i = \Delta(n)}{S \vdash_{instr} \textbf{Store}\ n \Rightarrow S'[i \leftarrow a]}$$

$$\frac{a_1, a_2, \cdots, a_n, f, \overbrace{\cdots}^{S'} = S \quad a_r = \{f(a_1, a_2, \cdots, a_n)\}_{python}}{S \vdash_{instr} \textbf{Call}\ n \Rightarrow a_r, \overbrace{\cdots}^{S'}}$$

JUMP

$$\frac{\textbf{Jump}\ off = I[i] \quad I, S \vdash_{jump} off \Rightarrow a}{I, S \vdash_{jump} i \Rightarrow a}$$

$$\frac{\textbf{JumpIf}\ b\ False\ off = I[i] \quad a, \overbrace{\cdots}^{S'} = S \quad b = \{bool(a)\}_{python} \quad I, S' \vdash_{jump} off \Rightarrow a_r}{I, S \vdash_{jump} i \Rightarrow a_r}$$

$$\frac{\textbf{JumpIf}\ b\ False off = I[i] \quad a, \overbrace{\cdots}^{S'} = S \qquad b \neq \{bool(a)\}_{python} \quad I, S' \vdash_{jump} i+1 \Rightarrow a_r}{I, S \vdash_{jump} i \Rightarrow a_r}$$

$$\frac{\textbf{JumpIf}\ b\ True\ off = I[i] \quad a, \cdots = S \qquad b = \{bool(a)\}_{python} \quad I, S \vdash_{jump} off \Rightarrow a_r}{I, S \vdash_{jump} i \Rightarrow a_r}$$

$$\frac{\textbf{JumpIf}\ b\ True off = I[i] \quad a, \overbrace{\cdots}^{S'} = S \qquad b \neq \{bool(a)\}_{python} \quad I, S' \vdash_{jump} i+1 \Rightarrow a_r}{I, S \vdash_{jump} i \Rightarrow a_r}$$

$$\frac{\textbf{Return} = I[i] \quad a, \cdots = S}{I, S \vdash_{jump} i \Rightarrow a}$$

$$\frac{instr = I[i] \quad S \vdash_{instr} instr \Rightarrow S' \quad I, S' \vdash_{jump} i+1 \Rightarrow a_r}{I, S \vdash_{jump} i \Rightarrow a_r}$$

Dynjit IR

Dynjit IR can be easily translated to C language or some Python-to-C language like Cython, etc.

Notations and Prerequisites

$$\begin{array}{rcl} \langle \text{repr} \rangle & ::= & S\,\textbf{constant} \\ & & \mid\ D\,\textbf{symbol} \end{array}$$

(**var** is defined in the section of **Core CPY**)

$$\begin{array}{rcl} \langle \text{absvalue} \rangle & ::= & (\textbf{repr}\ ,\ \textbf{type}) \\ \langle \text{expr} \rangle & ::= & \textbf{absvalue} \\ & & \mid\ \textbf{expr}\ (\textbf{expr}\ ^*) \\ \langle \text{stmt} \rangle & ::= & \textbf{var} = \textbf{expr} \\ & & \mid\ \textit{goto}\ \textbf{label} \\ & & \mid\ \textit{label}\ \textbf{label} \\ & & \mid\ \textit{return}\ \textbf{expr} \\ & & \mid\ \textit{typecheck}(\textbf{expr}, \textbf{type})\ \textit{then}\ \textbf{stmts}\ \textit{else}\ \textbf{stmts} \\ & & \mid\ \textit{if}\ \textbf{expr}\ \textit{then}\ \textbf{stmts}\ \textit{else}\ \textbf{stmts} \\ \langle \text{stmts} \rangle & ::= & \textbf{stmt}\ ^* \end{array}$$

(**type** can be any Python type, such as *bool*, *int*, but not composite type)

- $i, i_0, i_1, \cdots$, an integer.
- $n, n_0, n_1, \cdots$, a symbol, a name.
- ct, a function that maps a constant a_c to type $ct(a_c)$.
- S, a stack of abstract values. The partial evaluation configuration state.
- P, the offset of Core CPY instructions.
- G, a map from partial evaluation configuration to the generated label in *Dynjit*.
- a sequence of instruction **I**, can be indexed like $I[P]$.
- A map Δ to map a name n to the index of value stack $\Delta(n)$. This is fixed for every function/instruction sequence.
- The side-effect-free substitution operation $\cdot[\cdot \leftarrow \cdot]$. Given an offset $i \in \mathbb{N}$ of the value stack, and an abstract value v, $S[i \leftarrow v]$ returns a new stack S' which equals to S except $S[i] = v$.

- *specialize*, a function to perform partial evaluation for a function according to the argument types

Note that G and I does not appear in the left hand side of $\vdash$, because each function has their unique and constant G and I. They're used for lookup only. G and I can be found from **JIT STATE** when the function is known.

Partial Evaluation: from Core CPY to Dynjit

CACHE

$$\frac{\exists lbl : (S, P) \mapsto lbl \quad \in G}{S \vdash P \Rightarrow \textbf{goto } lbl}$$

LABEL

$$\frac{\begin{array}{c}\forall lbl : (S, P) \mapsto lbl \quad \notin G \\ lbl = gensym() \quad G \overset{mutate}{\leftarrow} (S, P) \mapsto lbl, \quad G \quad S \vdash P \Rightarrow L\end{array}}{S \vdash^* P \Rightarrow \textbf{label } lbl; \overbrace{\cdots}^{L}}$$

$\vdash^*$ is a wrap of $\vdash$ to record generated configurations.

BOOL-SPLIT

$$\frac{(D\ n, bool),\ \overbrace{\cdots}^{S'} = S \qquad S_1 = (S\ True, bool),\ \overbrace{\cdots}^{S'} \quad S_2 = (S\ False, bool),\ \overbrace{\cdots}^{S'} \quad S_1 \vdash P \Rightarrow L_1 \quad S_2 \vdash P \Rightarrow L_2}{S \vdash P \Rightarrow \textbf{if}\ (D\ n, bool)\ \textbf{then}\ L_1\ \textbf{else}\ L_2}$$

UNION-SPLIT

$$\frac{(repr, t_1|t_2),\ \overbrace{\cdots}^{S'} = S \qquad S_1 = (repr, t_1),\ \overbrace{\cdots}^{S'} \quad S_2 = (repr, t_2),\ \overbrace{\cdots}^{S'} \quad S_1 \vdash P \Rightarrow L_1 \quad S_2 \vdash P \Rightarrow L_2}{S \vdash P \Rightarrow \textbf{typecheck}((repr, \top), t_1)\ \textbf{then}\ L_1\ \textbf{else}\ L_2}$$

INSTR

$$\frac{\textbf{Pop} = I[P] \quad a,\ \overbrace{\cdots}^{S'} = S \quad S' \vdash P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Peek}\ i = I[P] \quad a_1, \cdots, a_i, a_{peek}, \cdots = S \quad a_{peek},\ \overbrace{\cdots}^{S} \vdash P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Load}\ n = I[P] \quad i = \Delta(n) \quad a = S[i] \quad a,\ \overbrace{\cdots}^{S'} \vdash I \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Const}\ a = I[P] \quad (S\ a, ct(a)),\ \overbrace{\cdots}^{S} \vdash P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Rot}\ i = I[P] \quad a_1, a_2, \cdots, a_i,\ \overbrace{\cdots}^{S'} = S \quad a_i, a_1, a_2, \cdots, a_{i-1},\ \overbrace{\cdots}^{S'} \vdash P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Store}\ n = I[P] \quad i = \Delta(n) \quad (S\ a, t),\ \overbrace{\cdots}^{S'} = S \qquad S'[i \leftarrow (S\ a, t)] \vdash P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Store}\ n = I[P] \quad i = \Delta(n) \quad (D\ n', t),\ \overbrace{\cdots}^{S'} = S \qquad S'[i \leftarrow (D\ n', t)] \vdash P + 1 \Rightarrow L}{n = (n', t);\ \overbrace{\cdots}^{L}}$$

$$\frac{\begin{array}{c}\textbf{Call}\ n = I[P] \quad n = gensym() \\ (r_1, t_1), (r_2, t_2), \cdots, (r_n, t_n), (r_f, t_f), \overbrace{\cdots}^{S'} = S \\ closure, func, t_r = specialize(t_f, (t_1, t_2, \cdots, t_n)) \\ (Dn, t_r), \overbrace{\cdots}^{S'} \vdash P + 1 \Rightarrow L\end{array}}{S \vdash P \Rightarrow n = func(closure, (r_1, t_1), (r_2, t_2), \cdots, (r_n, t_n)); \overbrace{\cdots}^{L}}$$

JUMP

$$\frac{\textbf{Jump}\ \mathit{off} = I[P] \quad S \vdash^* \mathit{off} \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{JumpIf}\ b_1\ False\ \mathit{off} = I[P] \quad (S\ b_2, bool), \overbrace{\cdots}^{S'} = S \qquad b_1 = b_2 \quad S' \vdash^* \mathit{off} \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{JumpIf}\ b_1\ False\ \mathit{off} = I[P] \quad (S\ b_2, bool), \overbrace{\cdots}^{S'} = S \qquad b_1 \neq b_2 \quad S' \vdash^* P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{JumpIf}\ b_1\ True\ \mathit{off} = I[P] \quad (S\ b_2, bool), \cdots = S \qquad b_1 = b_2 \quad S \vdash^* \mathit{off} \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{JumpIf}\ b_1\ True\ \mathit{off} = I[P] \quad (S\ b_2, bool), \overbrace{\cdots}^{S'} = S \qquad b_1 \neq b_2 \quad S' \vdash^* P + 1 \Rightarrow L}{S \vdash P \Rightarrow L}$$

$$\frac{\textbf{Return} = I[P] \quad a, \cdots = S}{S \vdash P \Rightarrow \textbf{return}\ a}$$