JIT for Dynamic Programming Languages Considered Easy

Taine Zhao
University of Tsukuba

Lambda Days & TFP, 2021

Outline

- Background
- Call-site Specialisation: IIS and PCS
- Assumptions and Optimisations
- Use for Partial Evaluation
- Pros and Cons

Background

Dynamic programming languages are getting popular...

but... JIT?

Issues:

- non-technical: human resources, money...
- technical: compatibility(C extensions, etc.), complexity, time...

A working approach applied to CPython:

- implemented within 1000 lines
- non-invasive changes to language implementation
- compatibility
- extensibility

Call-site Specialisation

Call-site specialisation is common:

$$f(arg_1 \cdots arg_n)$$

By partly knowing the information of the callable f, and its arguments $arg_1 \cdots arg_n$, we may be able to

1. choose the appropriate implementation of f
2. eliminate runtime type checks
3. infer the representation of the return

```python
import diojit as jit

def append3(xs, x):
    xs.append(x)
    xs.append(x)
    xs.append(x)

jit_append3 = jit.specialise(
    append3, jit.oftype(list), jit.Top)
jit_append3(...) # performance gain: >100%
```

An *immediately invoked specialisation*(IIS) is the specialisation performed on a callable exactly when executing the code. JIT compilation is triggered here, which is time-consuming.

IISs are slow, but chances are there to make programs faster.

```
import diojit as jit

def append3(xs, x):
    xs.append(x)
    xs.append(x)
    xs.append(x)

jit_append3 = jit.specialise(
    append3, jit.oftype(list), jit.Top)
# performance gain: >100%
jit_append3(...)
```

A pre-call specialisation(PCS) happens inside an IIS.
It is performed on a callable ****before**** executing the code.

PCS for list.append


```
def append3(xs, x):
    xs.append(x)

jit_append3 = jit.specialise(
    append3, jit.oftype(list), jit.Top)
jit_append3(...)
```

$$\text{when } \sigma = \{xs : \texttt{list}, x : \top\}$$

$$\boxed{\texttt{xs.append(x)}}$$

PCS for list.append

$\boxed{\texttt{xs.append(x)}}$

Specialisation rules used at analysis time:

$$
\begin{array}{rl}
\textit{When} & \texttt{xs} \text{ is a } \texttt{\textbf{list}} \\
\textit{case} & \texttt{xs}.\mathit{append}(a_1) \Rightarrow \mathit{CListAppend}(xs, a_1) : \texttt{unit} \\
\textit{case} & \texttt{xs}.\mathit{append}(\bar{a}) \;\; \Rightarrow (\texttt{fall back to CPython}) : \top \\
\textit{where} & \mathit{CListAppend} \text{ is a specialised implementation that}
\end{array}
$$

1. avoids type-checking the arguments
2. eliminates dynamic method lookup
3. guarantees the return value is `None`, which holds the type `unit`.

Assumptions and Optimisations


```
class Node:
    def __init__(self, n, val):
        self.next = n
        self.val = val

def sum_chain(n):
    a = 0
    while n is not None:
        a += n.val
        n = n.next
    return a

n_0   = Node(None, 0)
n_{i+1} = Node(n_i, i)
sum_chain(n_100)
```

Running it a million times costs **8.75**s. It's slow. The performance is poor because...

6.23s / million. 40% speed up.

```
class Node:
    def __init__(self, n, va
        self.next = n
        self.val = val

def sum_chain(n):
    a = 0
    while n is not None:
        a += n.val
        n = n.next
    return a

jit_func = jit.specialise(sum_chain, jit.oftype(Node))
jit_func(n_N)
```

PC	Store	Jump
7	$\{n : \texttt{Node}\}$	8
8	$\{a : \texttt{int} = 0, n : \texttt{Node}\}$	9
9	$\{a : \texttt{int} = 0, n : \texttt{Node}\}$	10
10	$\{a : \top, n : \texttt{Node}\}$	8
8	$\{a : \top, n : \top\}$	9 or 11
9	$\{a : \top, n : \top\}$	10
10	$\{a : \top, n : \top\}$	8
8	$\{a : \top, n : \top\}$	terminate

Assumptions and Optimisations


```
1  unit = type(None)
2  @jit.eagerjit
3  class Node:
4      next: Union[Node, unit]
5      val: int
6      def __init__(self, n, val
7          self.next = n
8          self.val = val
9
10 def sum_chain(n):
11     a = 0
12     while n is not None:
13         a += n.val
14         n = n.next
15     return a
```

PC	Store	Jump
11	$\{n : \texttt{Node}\}$	12
12	$\{a : \texttt{int} = 0, n : \texttt{Node}\}$	13
13	$\{a : \texttt{int} = 0, n : \texttt{Node}\}$	14
14	$\{a : \texttt{int}, n : \texttt{Node}\}$	12
12	$\{a : \texttt{int}, n : \texttt{unit} \mid \texttt{Node}\}$	U_1 or U_2
U_1	$\{a : \texttt{int}, n : \texttt{Node}\}$	13
13	$\{a : \texttt{int}, n : \texttt{Node}\}$	14
14	$\{a : \texttt{int}, n : \texttt{Node}\}$	terminate
U_2	$\{a : \texttt{int}, n = \texttt{None}\}$	12
12	$\{a : \texttt{int}, n = \texttt{None}\}$	15

3.57s / million, 140% performance gain.

Partial Evaluation

func is a variant of `func` that tries computations at analysis time.

```
@jit.eagerjit
def fib(x): # performance gain: >600%
    if x <= 2: return 1
    return fib(x - 1) + fib(x - 2)

@jit.eagerjit
def fib(x): # performance gain: >35000%
    if x <= 2: return 1
    return  fib(x - 1) + fib(x - 2)
# def fib[x=10](_): return 55

@jit.eagerjit
def main(y):
    x = 10
    x += fib(x)
    x += fib(y)

jit_main = jit.specialise(main, jit.oftype(int))
jit_main()
```

Specialisation Rule for +


```python
from jit import register, Judge, AbsVal, S, CallSpec
import operator # python operator implementation
@register(+, method="__call__")
def call_const_add(self: Judge, *args: AbsVal):
    if len(args) != 2:
        # no specialisation
        return NotImplemented
    [a, b] = args
    if a.is_static() and b.is_static():
        # .base get the static value if applicable
        const = a.base + b.base
        type_const = type(const)
        ret_types = (S(type_const), )
        return CallSpec(const, const, ret_types)

    # roll back to ordinary 'a + b' specialisation
    static_fn = S(operator.__add__)
    return self.spec(static_fn, "__call__", args)
```

Item	PY38	JIT PY38	PY39	JIT PY39
brainf**k	265.74	134.23	244.50	140.34
append3	23.94	10.70	22.29	11.21
DNA READ	16.96	14.82	15.03	14.38
fib(15)	11.63	1.54	10.41	1.51
hypot	6.19	3.87	6.53	4.29
selectsort	46.95	33.88	38.71	29.49

Pros:

- minimal: core within 1000 lines
- non-invasive: implemented as a 3rd-party library
- compatible: encouraging C extensions
- extensible: defining new specialisation rules
- functional: benefiting from immutability and small functions

Cons:

- issues with non-generics: when `xs` is analysed as a `list`, we cannot analyse the type of `xs[0]`
- issues with dynamic scoping: deciding which object a global variable is referencing is a must
- generated code size is huge