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
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
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.

```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)

# performance gain: >100%

jit_append3(...)  
```

A *pre-call specialisation* (PCS) happens inside an IIS. It is performed on a callable **before** executing the code.
def append3(xs, x):
    xs.append(x)

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

jit_append3(...)

when σ = \{xs : list, x : \top\}

xs.append(x)


Specialisation rules used at analysis time:

When $xs$ is a list

- case $xs$.append($a_1$) ⇒ $C\text{ListAppend}(xs, a_1): \text{unit}$
- case $xs$.append($\overline{a}$) ⇒ (fall back to CPython): $\top$

where $C\text{ListAppend}$ is a specialised implementation that

1. avoids type-checking the arguments
2. eliminates dynamic method lookup
3. guarantees the return value is $\text{None}$, which holds the type $\text{unit}$. 

$xs$.append(x)
Assumptions and Optimisations

```python
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_1 = Node(n_0, 1)
sum_chain(n_100)
```

Running it a million times costs 8.75s. It’s slow. The performance is poor because...
6.23s / million. 40% speed up.

```python
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

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

```
unit = type(None)
@jit.eagerjit
class Node:
    next: Union[Node, unit]
    val: int
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
```

3.57s / million, 140% performance gain.
**Partial Evaluation**

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

```python
@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()
```
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)
<table>
<thead>
<tr>
<th>Item</th>
<th>PY38</th>
<th>JIT PY38</th>
<th>PY39</th>
<th>JIT PY39</th>
</tr>
</thead>
<tbody>
<tr>
<td>brainf**k</td>
<td>265.74</td>
<td>134.23</td>
<td>244.50</td>
<td>140.34</td>
</tr>
<tr>
<td>append3</td>
<td>23.94</td>
<td>10.70</td>
<td>22.29</td>
<td>11.21</td>
</tr>
<tr>
<td>DNA READ</td>
<td>16.96</td>
<td>14.82</td>
<td>15.03</td>
<td>14.38</td>
</tr>
<tr>
<td>fib(15)</td>
<td>11.63</td>
<td>1.54</td>
<td>10.41</td>
<td>1.51</td>
</tr>
<tr>
<td>hypot</td>
<td>6.19</td>
<td>3.87</td>
<td>6.53</td>
<td>4.29</td>
</tr>
<tr>
<td>selectsort</td>
<td>46.95</td>
<td>33.88</td>
<td>38.71</td>
<td>29.49</td>
</tr>
</tbody>
</table>
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