Handling control flow in AD

UCSD CSE 291
Tzu-Mao Li
Today: differentiating control flow

we will focus on reverse mode: handling control flow in forward mode is trivial

```python
if ...:
    ...
else:
    ...
```

```python
for i in range(...):
    ...
```

```python
while (...):
    ...
```

```python
f(...):
```

if/else conditions

loops

function calls
Today: differentiating control flow

we will focus on reverse mode: handling control flow in forward mode is trivial

```python
if ...:
  ...
else:
  ...
```

if/else conditions

```python
for i in range(...):
  ...
```

for i in range(...):
  ...

loops

```python
while (...):
  ...
```

while (...):
  ...

function calls
Differentiating if/else statements

```python
z : float = 0
if x > y:
    z = z0
else:
    z = z1
```
Differentiating if/else statements

```python
z : float = 0
if x > y:
    z = z0
else:
    z = z1
```

```python
dz : float = 0
if x > y:
    dz0 += dz
else:
    dz1 += dz
```
Differentiating if/else statements

```
z : float = 0
if x > y:
    t : float = z0 * z0
    z = t * t
else:
    z = z1
```
Differentiating if/else statements

```
z : float = 0
if x > y:
    t : float = z0 * z0
    z = t * t
else:
    z = z1
```

```
z : float = 0
dz : float = 0
if x > y:
    t : float = z0 * z0
    z = t
else:
    z = z1
if x > y:
    t : float = z0 * z0
dt : float = 0
dt += dz * 2 * t
dz0 += dt * 2 * z0
else:
dz1 += dz
```
Differentiating if/else statements

\[
\begin{align*}
    z & : \text{float} = 0 \\
    \text{if } x > y: \\
    \quad t & : \text{float} = z0 \times z0 \\
    \quad z & = t \times t \\
    \text{else:} \\
    \quad z & = z1
\end{align*}
\]

\[
\begin{align*}
    z & : \text{float} = 0 \\
    \text{if } x > y: \\
    \quad t & : \text{float} = z0 \times z0 \\
    \quad z & = t \\
    \text{else:} \\
    \quad z & = z1 \\
    \text{if } x > y: \\
    \quad t & : \text{float} = z0 \times z0 \\
    \quad dt & : \text{float} = 0 \\
    \quad dt & += dz \times 2 \times t \\
    \quad dz0 & += dt \times 2 \times z0 \\
    \text{else:} \\
    \quad dz1 & += dz
\end{align*}
\]

duplicated computation!
Differentiating if/else statements

\[
z : \text{float} = 0 \\
\text{if } x > y: \\
\hspace{1em} t : \text{float} = z_0 \times z_0 \\
\hspace{1em} z = t \times t \\
\text{else:} \\
\hspace{1em} z = z_1
\]
Differentiating if/else statements

```
z : float = 0
if x > y:
    t : float = z0 * z0
    z = t * t
else:
    z = z1
```
Differentiating if/else statements

in loma, we disallow variable declaration inside if/else statements

```
z : float = 0
if x > y:
    t : float = z0 * z0
    z = t * t
else:
    z = z1
```

compile error

```
z : float = 0
if x > y:
    t = z0 * z0
    z = t * t
else:
    z = z1
```

Ok
Differentiating if/else statements

def f(x : In[float]) -> float:
    if x > 0:
        return 2 * x
    y : float = x * x
    return y * y
Differentiating if/else statements

def f(x : In[float]) -> float:
    if x > 0:
        return 2 * x
    y : float = x * x
    return y * y

def DTf(x : In[float], dx : Out[float],
    return : In[float]):
    ret : float
dret : float = 0
    if x > 0:
        ret = 2 * x
        y : float = x * x
        dy : float = 0
        if x <= 0:
            ret = y * y
            dret += dreturn
            if x <= 0:
                dy += dret * 2 * y
                dx += dy * 2 * x
            if x > 0:
                dx += dret * 2
Differentiating if/else statements

in loma, we disallow early return of functions

```python
def f(x : In[float]) -> float:
    if x > 0:
        return 2 * x
    y : float = x * x
    return y * y
```

```python
def f(x : In[float]) -> float:
    ret : float
    if x > 0:
        ret = 2 * x
    y : float = x * x
    if x <= 0:
        ret = y * y
    return ret
```

compile error  Ok
Today: differentiating control flow

we will focus on reverse mode: handling control flow in forward mode is trivial

```python
if ...:
  ...
else:
  while (...):
    ...
for i in range(...):
  ...
```

if/else conditions

loops

function calls
Differentiating loops

```python
for i in range(10):
    x = sin(i * x)  # ?
```
Differentiating loops

for i in range(10):
    x = sin(i * x)

for i in reversed(range(10)):
    x = stack.pop()
    dx_ : float = dx * cos(i * x) * i
    dx = 0
    dx += dx_
Differentiating loops

\[
\text{for } i \text{ in range}(10): \\
x = \sin(i \times x)
\]

\[
\text{for } i \text{ in range}(10): \\
\text{stack.push}(x) \\
x = \sin(i \times x) \\
\text{dynamic heap allocation — slow!}
\]

\[
\text{for } i \text{ in reversed(range(10))}: \\
x = \text{stack.pop}() \\
dx_\_ : \text{float} = dx \times \cos(i \times x) \\
dx = 0 \\
dx += dx_\_
\]
Differentiating loops

in loma, we always ask user to provide an upper bound of the loop, so we can preallocate the stack

```python
for i in range(10):
    x = sin(i * x)
```

```python
stack.reserve(10)
for i in range(10):
    stack.push(x)
    x = sin(i * x)

for i in reversed(range(10)):
    x = stack.pop()
    dx_ : float = dx * cos(i * x)
    dx = 0
    dx += dx_
```
Differentiating loops

while (x > 0, max_iter := 10):
    x = sin(x) → ?
Differentiating loops

```cpp
stack.reserve(10)
ctr : int = 0
while (x > 0, max_iter := 10):
    stack.push(x)
    x = sin(x)
    ctr += 1

while (ctr > 0, max_iter := 10):
    x = stack.pop()
    dx_ : float = dx * cos(x)
    dx = 0
    dx += dx_
    ctr -= 1
```

while (x > 0, max_iter := 10):
    x = sin(x)
Differentiating loops

while (cond0, max_iter := 10):
    while (cond1, max_iter := 10):
        ...

?
Differentiating loops

while (cond0, max_iter := 10):
    while (cond1, max_iter := 10):
        ...
        stack.reserve(100)
        ctr0 : int = 0
        ctr1 : int = 0
        while (cond0, max_iter := 10):
            while (cond1, max_iter := 10):
                ...
                ctr1 += 1
                ctr0 += 1
        while (ctr0 > 0, max_iter := 10):
            while (ctr1 > 0, max_iter := 10):
                ...
                ctr1 -= 1
                ctr0 -= 1

is this correct?
Differentiating loops

```cpp
stack.reserve(100)
ctr0 : int = 0
ctr1 : Array[int, 10]
ctr1_ptr : int = 0

while (cond0, max_iter := 10):
  while (cond1, max_iter := 10):
    ...
    ctr1[ctr1_ptr] += 1
    ctr1_ptr += 1
    ctr0 += 1

while (ctr0 > 0, max_iter := 10):
  ctr1_ptr -= 1
  while (ctr1[ctr1_ptr] > 0, max_iter := 10):
    ...
    ctr1[ctr1_ptr] -= 1
    counter0 -= 1
```

need to construct counter arrays for nested while loops
Differentiating loops

while (cond0, max_iter := 10):
    while (cond1, max_iter := 10):
        while (cond2, max_iter := 10):
            ...

?
Differentiating loops

while (cond0, max_iter := 10):
    while (cond1, max_iter := 10):
        while (cond2, max_iter := 10):
            ...
            ctr2[ctr2_ptr] += 1
            ctr2_ptr += 1
            ctr1[ctr1_ptr] += 1
            ctr1_ptr += 1
            ctr0 += 1
            while (ctr0 > 0, max_iter := 10):
                ctr1_ptr -= 1
                while (ctr1[ctr1_ptr] > 0, max_iter := 10):
                    ctr2_ptr -= 1
                    while (ctr2[ctr2_ptr] > 0, max_iter := 10):
                        ...
                        ctr2[ctr2_ptr] -= 1
                        ctr1[ctr1_ptr] -= 1
                        ctr0 -= 1

stack.reserve(100)
ctr0 : int = 0
ctr1 : Array[int, 10]
ctr1_ptr : int = 0
ctr2 : Array[int, 100]
ctr2_ptr : int = 0

while (cond0, max_iter := 10):
    while (cond1, max_iter := 10):
        while (cond2, max_iter := 10):
Differentiating loops

while (True, max_iter := 10):
    if x > 0:
        break
...

?
Differentiating loops

in loma, we disallow break statements

```python
while (True, max_iter := 10):
    if x > 0:
        break
...
flag : int = True
while (flag, max_iter := 10):
    if x > 0:
        flag = False
    if flag:
        ...
```
Differentiating loops

while (cond, max_iter := 10):
    if x > 0:
        continue
...

?
Differentiating loops

in loma, we disallow continue statements

while (cond, max_iter := 10):
    if x > 0:
        continue
    ...

    while (True, max_iter := 1):
        if x > 0:
            break
    ...

    break
For loops vs while loops

for i in range(n):
    for j in range(i, n):
        v.s.

i : int = 0
while i < n:
    j : int = i
    while j < n:
        ...
        j += 1
    i += 1
For loops vs while loops

for i in range(n):
    for j in range(i, n):

vs.

i : int = 0
while i < n:
    j : int = i
    while j < n:
        ...
        j += 1
    i += 1

for loops provide more structures for us to revert them

while loops can require more memory to store the loop counters
For loops vs while loops

take home message: program abstraction is important for the AD system to figure out the right optimization!!

for i in range(n):
    for j in range(i, n):

    i : int = 0
    while i < n:
        j : int = i
        while j < n:
            ...
            j += 1
        i += 1

for loops provide more structures for us to revert them
while loops can require more memory to store the loop counters
Today: differentiating control flow

we will focus on reverse mode: handling control flow in forward mode is trivial

if ...
...
else:
...

for i in range(...):
...

while (...):
...

if/else conditions
loops
function calls
Differentiating function calls

# def f(x : In[float]) -> float

y = f(x) → ?
Differentiating function calls

# def f(x : In[float]) -> float

y = f(x)

y = f(x)

\[ \text{DTf}(x, dx, dy) \]
Differentiating function calls

# def f(x : In[float], y : Out[float])
f(x, y) → ?
Differentiating function calls

```python
# def f(x : In[float], y : Out[float])
f(x, y)
stack.push(y)
f(x, y)
y = stack.pop()
DTf(x, dx, dy)
```
Differentiating function calls

```python
# def f(x : In[float], y : Out[float])
f(2 + x, y)  →  ?
```
Differentiating function calls

we provide the utility code for doing this transformation in your homework

```python
# def f(x : In[float], y : Out[float])
f(2 + x, y)
```

\[ t = 2 + x \]

\[ f(t, y) \]
def f(x: float, n: int) -> float:
    ret = float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
    return ret
def f(x : In[float], n : In[int]) -> float:
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
    return ret

def DTf(x : In[float],
        dx : Out[float],
        n : In[int],
        dret : Out[float]):
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
        if n <= 0:
            dx += dret
        else:
            DTf(x, dx, n-1, dret * x)
        dx += dret * f(x, n - 1)

is this correct?
Differentiating recursion

naive differentiation of recursion violates cheap gradient principle!!

is this correct?
Differentiating recursion

```python
def f(x : In[float], n : In[int]) -> float:
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
    return ret

def DTf(x : In[float],
        dx : Out[float],
        n : In[int],
        dret : Out[float],
        f_cache = {}):
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f_cache.lookup(x, n - 1) * x
        if n <= 0:
            dx += dret
        else:
            DTf(x, dx, n-1, dret * x)
        dx += dret * f_cache.lookup(x, n - 1)
```

```ruby
```
Differentiating recursion

```python
def f(x : In[float], n : In[int]) -> float:
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
    return ret
```

```python
def DTf(x : In[float], dx : Out[float],
        n : In[int],
        dret : Out[float],
        f_cache = {}):
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f_cache.lookup(x, n - 1) * x
```

for a better way to construct f_cache, see this paper

the trick is to apply “Continuation Passing Style” in functional programming to store variables in a continuation

Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator

FEI WANG, Purdue University, USA
DANIEL ZHENG, Purdue University, USA
JAMES DECKER, Purdue University, USA
XILUN WU, Purdue University, USA
GRÉGORY M. ESSERTÉL, Purdue University, USA
TIARK ROMPF, Purdue University, USA
Differentiating recursion

alternatively, recursions can always be rewritten as iterative loops
(Church-Turing thesis)

```python
def f(x : In[float], n : In[int]) -> float:
    ret : float
    if n <= 0:
        ret = x
    else:
        ret = f(x, n - 1) * x
    return ret
```

```python
ret = 1
for i in range(n):
    ret *= x
```
Food for thoughts

if we can differentiate general function calls (with recursion), do we need to handle anything else?

\[
z = x + y \quad \rightarrow \quad \text{add } x \ y \ z
\]

for \( i \) in range(10):

\[
\rightarrow \quad \text{scan } \ldots \text{ range(10)}
\]

if \( x \):

\[
\rightarrow \quad x \ a \ b
\]

else:

\[
\rightarrow \quad (\text{Church boolean})
\]