Regularizing Python using Structured Control Flow

Valentin Haenel

Senior Software Engineer - Open Source / Numba
https://anaconda.com

29 Aug 2024 – EuroScipy – Maritime University of Szczecin

Outline

whoami

- Val Haenel ("val" like `let val =`)
- https://github.com/esc
- Compiler Engineer at Anaconda
- Working full-time on Numba
    - (the function compiler for numerical Python)
- Doing this for over 5 years

Introduction

- Using a Structured Control Flow Graph (SCFG)
- Regularize Python, identify branches and loops
- Branch and loop regions are closed and clearly identified
- Algorithm based on Bahmann2015
- Implemented in package numba-rvsdg
- Two main contributions:
  - $\rightarrow$ Application of an academic paper to Python source
  - $\rightarrow$ Solution to desugaring Python for-loops

Motivation

- Make code more ammeanable for Python compilers
- First step towards an source frontend for Numba
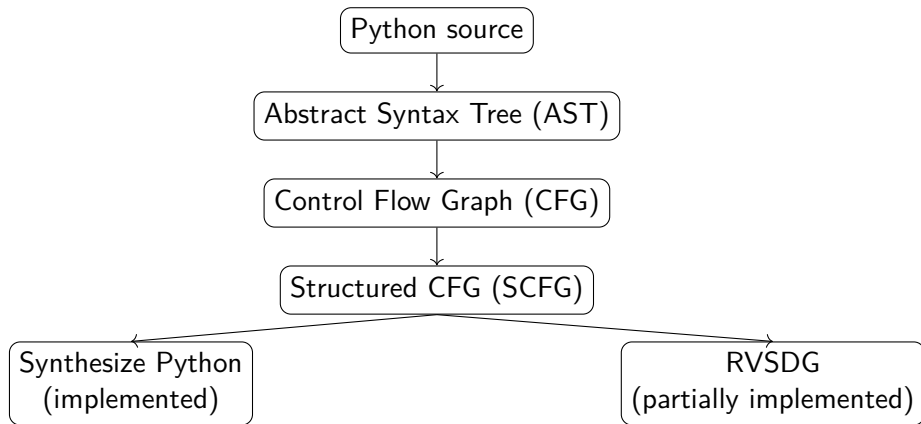- (Numba currently uses bytecode...)

Outline

Pipeline

```
                    ┌─────────────────┐
                    │  Python source  │
                    └─────────────────┘
                             │
                             ▼
              ┌──────────────────────────────┐
              │ Abstract Syntax Tree (AST)    │
              └──────────────────────────────┘
                             │
                             ▼
              ┌──────────────────────────────┐
              │  Control Flow Graph (CFG)     │
              └──────────────────────────────┘
                             │
                             ▼
              ┌──────────────────────────────┐
              │  Structured CFG (SCFG)        │
              └──────────────────────────────┘
               ↙                          ↘
┌─────────────────────┐         ┌─────────────────────────┐
│ Synthesize Python   │         │        RVSDG            │
│ (implemented)       │         │ (partially implemented) │
└─────────────────────┘         └─────────────────────────┘
```

CFG → SCFG

- Identifies loop and branch regions
- Loops become closed and tail controlled (do-while) with a single backedge
- Branch regions are identified as a structure of
  - one head region
  - two or more branch regions
  - one tail region

Outline

Outline

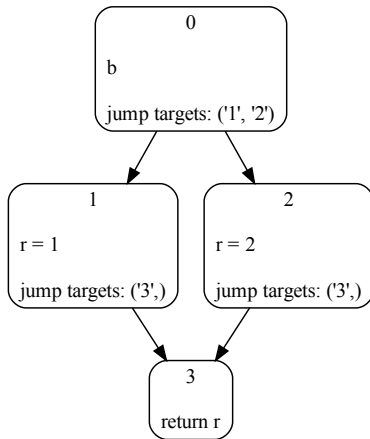# Branch

- A simple example

```python
def branch(b: int) -> int:
    if b:
        r = 1
    else:
        r = 2
    return r
```

AST

```
FunctionDef(
    name='branch',
    args=arguments(
        posonlyargs=[],
        args=[
            arg(
                arg='b',
                annotation=Name(id='int', ctx=Load()))],
        kwonlyargs=[],
        kw_defaults=[],
        defaults=[]),
    body=[
        If(
            test=Name(id='b', ctx=Load()),
            body=[
                Assign(
                    targets=[
                        Name(id='r', ctx=Store())],
                    value=Constant(value=1))],
            orelse=[
                Assign(
                    targets=[
                        Name(id='r', ctx=Store())],
                    value=Constant(value=2))]),
        Return(
            value=Name(id='r', ctx=Load()))],
    decorator_list=[],
    returns=Name(id='int', ctx=Load()),
    type_params=[]),
```

CFG

- CFG is already closed


```
          ┌─────────────────────┐
          │          0          │
          │                     │
          │ b                   │
          │ jump targets:       │
          │ ('1', '2')          │
          └─────────────────────┘
           ↙                  ↘
┌─────────────────┐   ┌─────────────────┐
│       1         │   │       2         │
│                 │   │                 │
│ r = 1           │   │ r = 2           │
│ jump targets:   │   │ jump targets:   │
│ ('3',)          │   │ ('3',)          │
└─────────────────┘   └─────────────────┘
           ↘                  ↙
            ┌───────────┐
            │     3     │
            │           │
            │ return r  │
            └───────────┘
```

SCFG

- Branch regions are identified, no "restructuring required"
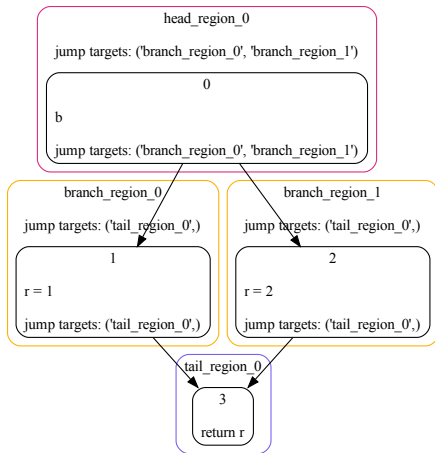
Outline

Multi Return
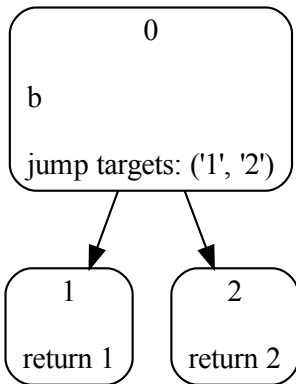
- This example has two `return` statements

```python
def multi_return(b: int):
    if b:
        return 1
    else:
        return 2
```
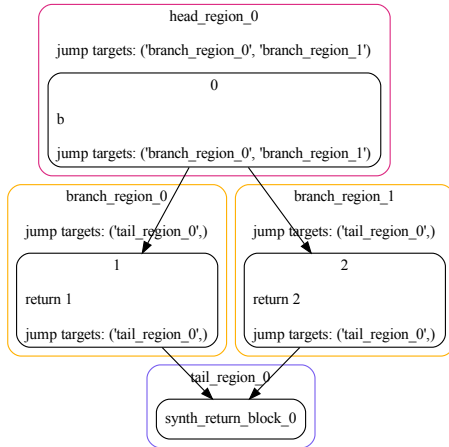
CFG

- The CFG is not closed, two exit nodes

SCFG

- The CLOSE CFG algorithm will restructure to insert an exit node

Transformed

- Finally, we can synthesize Python
- The transformed function only has a single `return`
- It is now closed

```python
def transformed_multi_return(b: int):
    if b:
        __scfg_return_value__ = 1
    else:
        __scfg_return_value__ = 2
    return __scfg_return_value__
```

Outline

While Loop

- Next up: a simple loop
- Remember: Loops are closed and tail controlled (do-while) with a single backedge

```python
def while_loop() -> int:
    c = 0
    while c < 10:
        c += 3
    return c
```

CFG
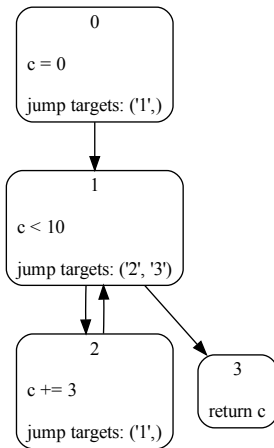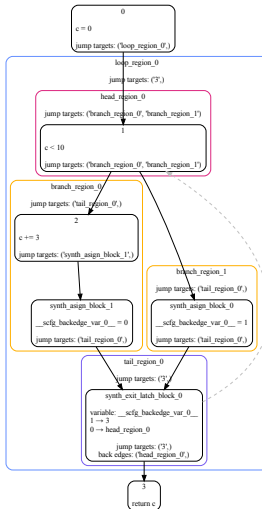
SCFG

Transformed

- The transformed variant is as close to a do-while loop as we can get in Python

```python
def transformed_while_loop() -> int:
    c = 0
    __scfg_loop_cont__ = True
    while __scfg_loop_cont__:
        if c < 10:
            c += 3
            __scfg_backedge_var_0__ = 0
        else:
            __scfg_backedge_var_0__ = 1
        __scfg_loop_cont__ = not __scfg_backedge_var_0__
    return c
```

Outline

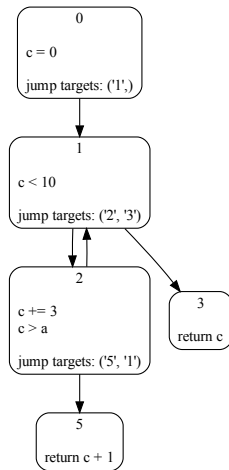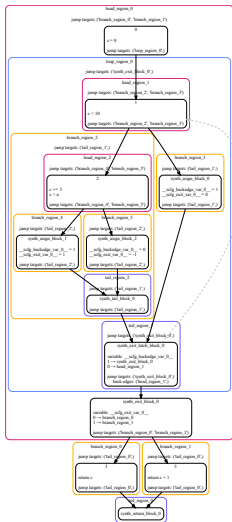# Early Exit

- Let's combine all three stages in this example
- The loop has an "early exit"
- This may be a problem for compilers, e.g, loop-unroll

```python
def early_exit(a: int) -> int:
    c = 0
    while c < 10:
        c += 3
        if c > a:
            return c + 1
    return c
```

CFG

SCFG

Transformed

```python
def transformed_early_exit(a: int) -> int:
    c = 0
    __scfg_loop_cont__ = True
    while __scfg_loop_cont__:
        if c < 10:
            c += 3
            if c > a:
                __scfg_exit_var_0__ = 1
                __scfg_backedge_var_0__ = 1
            else:
                __scfg_backedge_var_0__ = 0
                __scfg_exit_var_0__ = -1
        else:
            __scfg_exit_var_0__ = 0
            __scfg_backedge_var_0__ = 1
        __scfg_loop_cont__ = not __scfg_backedge_var_0__
    if __scfg_exit_var_0__ in (0,):
        __scfg_return_value__ = c
    else:
        __scfg_return_value__ = c + 1
    return __scfg_return_value__
```
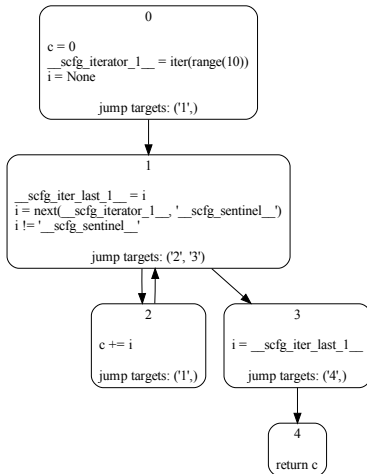
Outline

For Loop

- Python for-loops need to be "de-sugared"
    - $\rightarrow$ such that they can be represented using the "blocks and edges" semantics of the CFG formalism

- Setup the induction variable
- Setup the iterator
- Use `next` to determine when to stop instead of catching the `StopIteration` exception
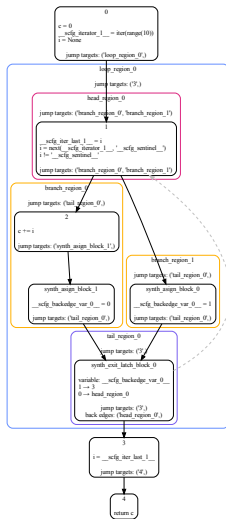- The induction variable must escape the scope

```python
def for_loop() -> int:
    c = 0
    for i in range(10):
        c += i
    return c
```

CFG

- De-sugaring happens on-the-fly during conversion to CFG

0
c = 0
__scfg_iterator_1__ = iter(range(10))
i = None
jump targets: ('1',)

1
__scfg_iter_last_1__ = i
i = next(__scfg_iterator_1__, '__scfg_sentinel__')
i != '__scfg_sentinel__'
jump targets: ('2', '3')

2
c += i
jump targets: ('1',)

3
i = __scfg_iter_last_1__
jump targets: ('4',)

4
return c

SCFG

Transformed

```python
def transformed_for_loop() -> int:
    c = 0
    __scfg_iterator_1__ = iter(range(10))
    i = None
    __scfg_loop_cont__ = True
    while __scfg_loop_cont__:
        __scfg_iter_last_1__ = i
        i = next(__scfg_iterator_1__, '__scfg_sentinel__')
        if i != '__scfg_sentinel__':
            c += i
            __scfg_backedge_var_0__ = 0
        else:
            __scfg_backedge_var_0__ = 1
        __scfg_loop_cont__ = not __scfg_backedge_var_0__
    i = __scfg_iter_last_1__
    return c
```

Outline

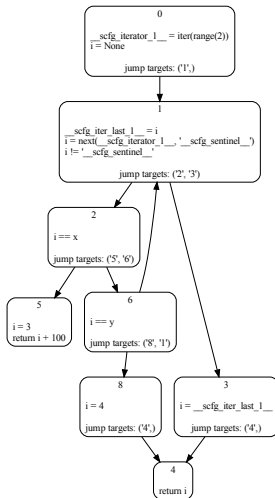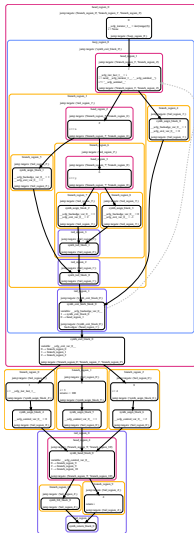- Both `break` and `continue` will be removed
  - (They are just syntactic sugar)

```python
def break_and_continue(x: int, y: int) -> int:
    for i in range(2):
        if i == x:
            i = 3
            return i + 100
        elif i == y:
            i = 4
            break
        else:
            continue
    return i
```

CFG

SCFG

Transformed

```python
def transformed_break_and_continue(x: int, y: int) -> int:
    __scfg_iterator_1__ = iter(range(2))
    i = None
    __scfg_loop_cont__ = True
    while __scfg_loop_cont__:
        __scfg_iter_last_1__ = i
        i = next(__scfg_iterator_1__, '__scfg_sentinel__')
        if i != '__scfg_sentinel__':
            if i == x:
                __scfg_exit_var_0__ = 1
                __scfg_backedge_var_0__ = 1
            elif i == y:
                __scfg_exit_var_0__ = 2
                __scfg_backedge_var_0__ = 1
            else:
                __scfg_backedge_var_0__ = 0
                __scfg_exit_var_0__ = -1
        else:
            __scfg_exit_var_0__ = 0
            __scfg_backedge_var_0__ = 1
        __scfg_loop_cont__ = not __scfg_backedge_var_0__
```

Transformed

```
if __scfg_exit_var_0__ in (0,):
    i = __scfg_iter_last_1__
    __scfg_control_var_0__ = 0
elif __scfg_exit_var_0__ in (1,):
    i = 3
    __scfg_return_value__ = i + 100
    __scfg_control_var_0__ = 1
else:
    i = 4
    __scfg_control_var_0__ = 2
if __scfg_control_var_0__ in (0, 2):
    __scfg_return_value__ = i
else:
    pass
return __scfg_return_value__
```

Outline

Future Work

- Explore full transformation to RVSDG
- Implement Source/AST frontend for Numba
- Find other potential uses...

Conclusion

- Open source tools used to make this presentation:
    - Wiki2beamer
    - LATEXbeamer
    - Dia
    - Pygments
    - Minted
    - Solarized theme for pygments

Questions?

- Questions?