



软件分析与架构设计

控制流分析

何冬杰
重庆大学

控制流图 (Control Flow Graph)

□ 另一种中间表示 (IR)

□ 控制流图：表示语句执行顺序的图

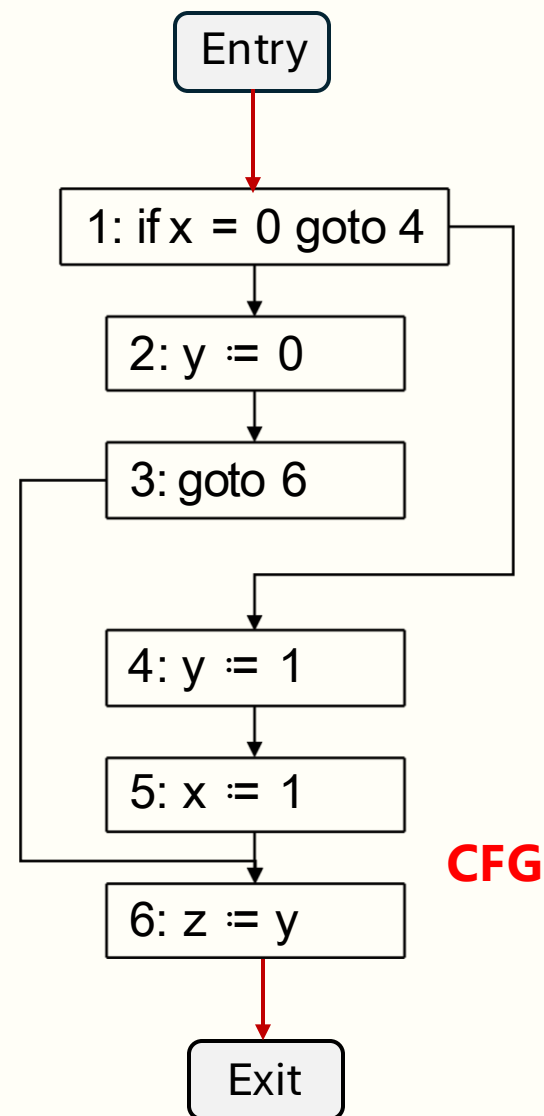
- **Nodes**: statements
- **Edges**: $(s1, s2)$ is an edge iff $s1$ and $s2$ can be executed consecutively
aka "control flow"

□ **Properties**:

- Only one **entry** node
- Only one **exit** (terminal) node
- Weakly connected
 - 边表示控制流的可能转移关系

```
1 : if x = 0 goto 4
2 : y := 0
3 : goto 6
4 : y := 1
5 : x := 1
6 : z := y
```

三地址码



控制流图构建算法

□识别开始位置：

- 程序开始位置；
- （条件或无条件）跳转语句目标
- （条件或无条件）跳转语句目标的下一条语句

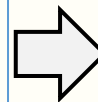
□识别基本块（Basic Blocks, BB）：

- 从开始位置到下一个开始位置的前一条语句

□根据基本块连边：

- 基本块内部顺序执行
- 根据基本块最后一条语句在基本块间加边

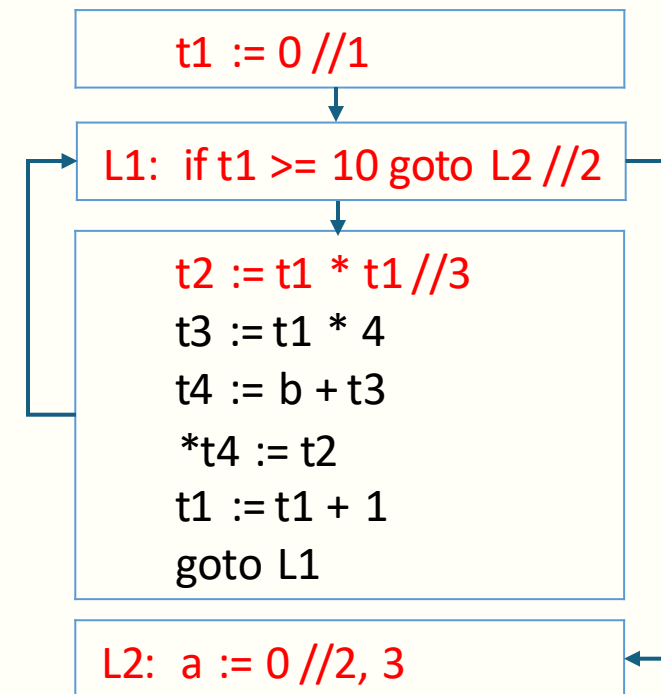
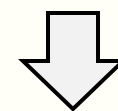
```
t1 := 0 //1
L1: if t1 >= 10 goto L2 //2
t2 := t1 * t1 //3
t3 := t1 * 4
t4 := b + t3
*t4 := t2
t1 := t1 + 1
goto L1
L2: a := 0 //2, 3
```



```
t1 := 0 //1
L1: if t1 >= 10 goto L2 //2

t2 := t1 * t1 //3
t3 := t1 * 4
t4 := b + t3
*t4 := t2
t1 := t1 + 1
goto L1

L2: a := 0 //2, 3
```



过程间控制流图 (Interprocedural CFG)

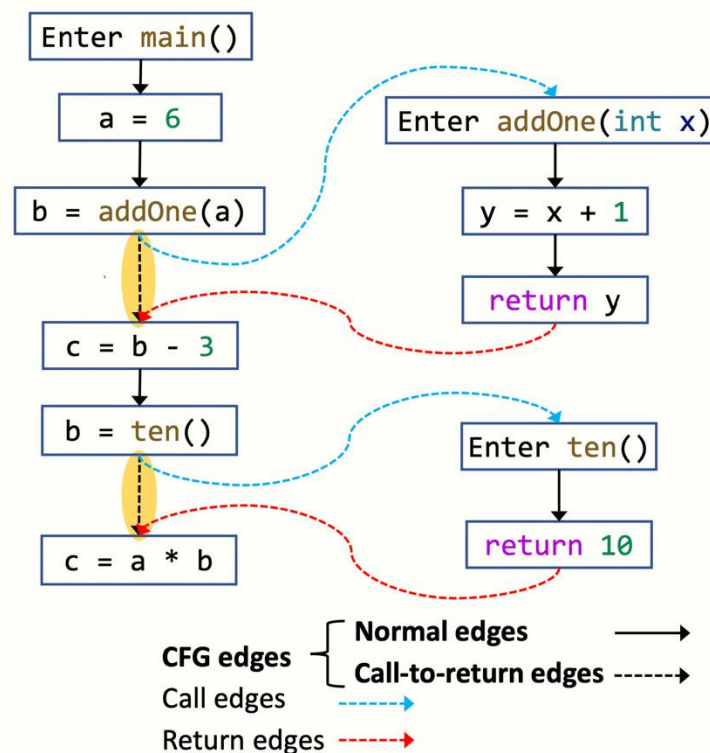
□ICFG是对控制流图 (也称Intraprocedural CFG) 的扩展

➤ For every call to a function f ,

- Add edge $\langle l, \text{entry}_f \rangle$: an edge from the call site l to the entry node of f 's CFG
- Add edge $\langle \text{exit}_f, l' \rangle$: an edge from the exit node of f 's CFG to the following instruction of the call site l

```
void main() {  
    int a, b, c;  
    a = 6;  
    b = addOne(a);  
    c = b - 3;  
    b = ten();  
    c = a * b;  
}  
int addOne() {  
    int y = x + 1;  
    return y;  
}  
int ten() { return 10; }
```

ICFG



控制流分析 (Control Flow Analysis)

□ 确定函数调用目标的分析叫做控制流分析

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }
foo(n, f) {
    if (n=0) {
        f=ide;
    }
    r = f(n); // call ?
    return r;
}
input x;
y = foo(x, inc);
```

Function pointers in C/C++

```
class A {
    A foo (A x ) { return x ; }
}
class B extends A {
    A foo (A x) { return new A(); }
}
class C extends A {
    A foo (A x) { return this; }}
A x = new A();
while (...)
    x = x.foo(new B()); // call ?
A y = new C();
y.foo(x); // call ?
```

Virtual Calls in Java

(fun f -> f 3) (fun x -> x + 1)

call ?

Function application in
Ocaml/Haskell

□ 调用图 (callgraph) : Callsites $\mapsto \mathcal{P}(\text{Functions})$

➤ For each callsite, tells us which functions it may call

CFA for OO

❑ CFA in an Object-Oriented language:

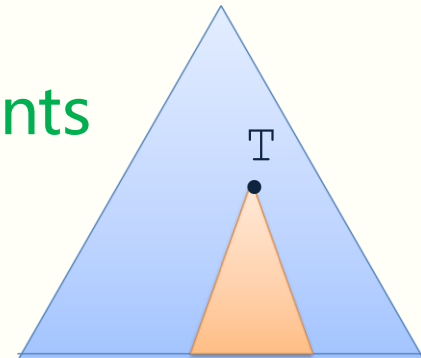
- Resolve call targets for $x.m(a_1, \dots, a_n)$

❑ Simplest solution:

- Select all methods named m with the same number of arguments

❑ Class Hierarchy Analysis (CHA):

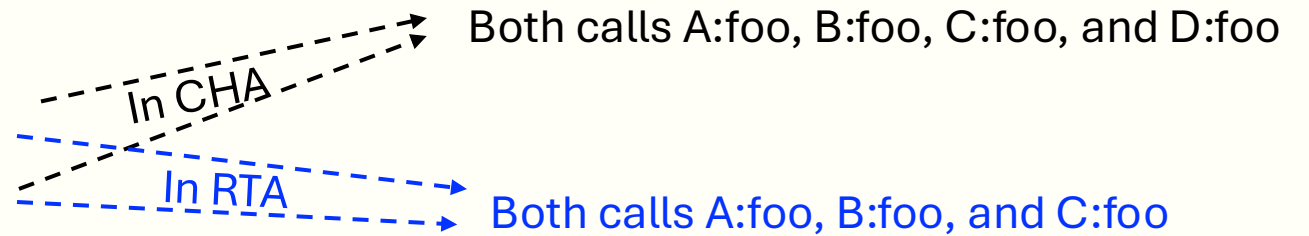
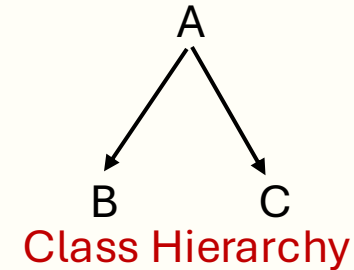
- For a polymorphic call site $m()$ on declared type T : Call edge to $T.m$ and any subclass of T that implements $m()$
- consider only the part of the class hierarchy rooted by the declared type of T
- Very fast and very imprecise but few requirements



CFA for OO

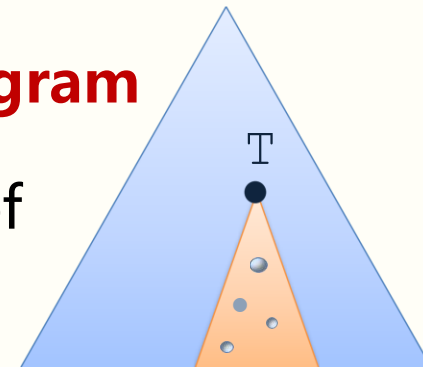
□ Class Hierarchy Analysis (CHA):

```
class A {  
  A foo (A x ) { return x ; }  
class B extends A {  
  A foo (A x) { return new A(); }}  
class C extends A {  
  A foo (A x) { return this; }}  
class D extends A {  
  A foo (A x) { return new D(); }}  
A x = new A();  
while (...)  
  x = x.foo(new B()); // call ?  
A y = new C();  
y.foo(x); // call ?
```



□ Rapid Type Analysis (RTA):

- Like CHA, but take into account only **those types that the program actually instantiates**
- Pretty **fast** and much more **precise** than CHA, eliminating lots of unused classes defined in the library



(Optional) Rapid Type Analysis

□数据结构:

- **Methods**: set of possibly called methods
- **Calls**: set of calls invoked in Methods
- **Targets**: a map from a call to its possible targets

$$\frac{}{main() \in Methods}$$

$$\frac{m \in Methods \quad l: x = new T \quad methodOf(l) = m \quad m' \in declareMethod(T)}{m' \in Methods}$$

$$\frac{m \in Methods \quad l: x = r.foo() \quad methodOf(l) = m}{l \in Calls}$$

$$\frac{m \in Methods \quad l: x = r.foo() \in Calls \quad m \text{ matches the call at } l}{m \in Targets(l)}$$

Variable Type Analysis (VTA)

- Reason about assignments
- Infer what types the objects involved in a call may have
- Prune calls that are infeasible based on the inferred types
 - Require an initial conservative call graph, e.g., using CHA or RTA
- Type Inferences: $\llbracket x \rrbracket$: collected types of x

$$\frac{l: x = \text{new } T \quad m = \text{methodOf}(l) \quad m \in \text{Methods}}{T \in \llbracket x \rrbracket} \quad \frac{l: x = y \quad m = \text{methodOf}(l) \quad m \in \text{Methods}}{\llbracket y \rrbracket \subseteq \llbracket x \rrbracket}$$

Field-based Analysis

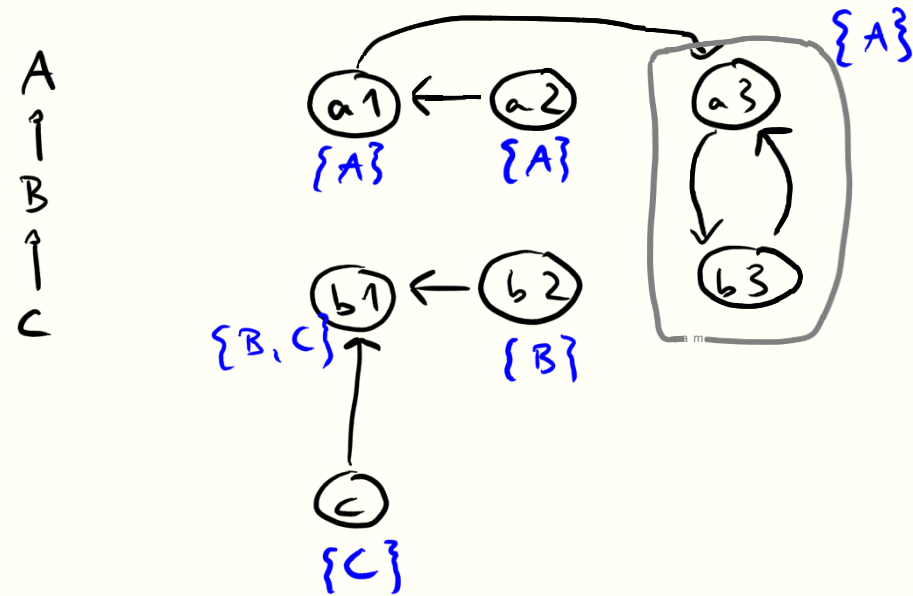
$$\left\{ \begin{array}{l} \frac{l: x = y.f \quad m = \text{methodOf}(l) \quad m \in \text{Methods} \quad T = \text{declareType}(y)}{\llbracket T.f \rrbracket \subseteq \llbracket x \rrbracket} \\ \frac{l: x.f = y \quad m = \text{methodOf}(l) \quad m \in \text{Methods} \quad T = \text{declareType}(x)}{\llbracket y \rrbracket \subseteq \llbracket x.f \rrbracket} \end{array} \right.$$

Function call: arguments to parameters and returns are modeled as assignments

Variable Type Analysis (VTA)

□ Example:

```
A a1, a2, a3;  
B b1, b2, b3;  
C c;  
a1 = new A();  
a2 = new A();  
b1 = new B();  
b2 = new B();  
c = new C();  
a1 = a2;  
a3 = a1;  
a3 = b3;  
b3 = (B) a3;  
b1 = b2;  
b1 = c;  
b1.foo(); // Call?
```



□ More precise than RTA, some imprecision remains

- Due to field-based analysis

Spark Analysis

- Combine call graph construction with points-to analysis
 - Andersen's analysis + on-the-fly callgraph construction

$$\frac{x = \mathbf{new} \ T // O}{O \in PTS(x)} \text{ [New]}$$

$$\frac{x = y}{PTS(y) \subseteq PTS(x)} \text{ [Assign]}$$

$$\frac{x = y.f \quad O \in PTS(y)}{PTS(O.f) \subseteq PTS(x)} \text{ [Load]}$$

$$\frac{x.f = y \quad O \in PTS(x)}{PTS(y) \subseteq PTS(O.f)} \text{ [Store]}$$

$\frac{l: x = r.m(a_1, \dots, a_n) \quad O \in PTS(r) \quad \mathbf{m' = dispatch(c, O)}}{PTS(O) \subseteq PTS(this^{m'}) \quad \mathbf{m' \in targets(l)}} \text{ [Call]}$ $\forall i \in [1, n]: PTS(a_i) \subseteq PTS(p_i) \quad PTS(ret^{m'}) \subseteq PTS(x)$
--

Spark Analysis

Var	PTS	Var	PTS
x1	{O4}	x2	{O4}
x3	{O3, O4, O1}	x4	{}
x	{O3, O4, O1}	y	{O5}

Calls A:foo, B:foo

Calls C:foo

```
class A {  
    A foo (A x1 ) { return x1 ; }  
}  
class B extends A {  
    A foo (A x2) { return new A(); // 01 }}  
class C extends A {  
    A foo (A x3) { return this; }}  
class D extends A {  
    A foo (A x4) { return new D(); // 02 }}  
A x = new A(); // 03  
while (...)  
    x = x.foo(new B() // 04); // call ?  
A y = new C(); // 05  
y.foo(x); // call ?
```

❑ Pros:

- Jointly computing call graph and points-to sets increases precision for both

❑ Cons:

- Can be quite **expensive** to compute
- Context-insensitive, further improve precision via context-sensitive or flow-sensitive techniques
 - Refer to Qilin (<https://github.com/QilinPTA/Qilin>)

CFA for SIMP with first-class functions

- ❑ **For a computed function call:** $f(a_1, \dots, a_n)$
 - We cannot immediately see which function is called
- ❑ **A coarse but sound approximation:**
 - Assume any function with right number of arguments
- ❑ **Use CFA to get a much better result!**
 - Model **functions as special objects**
 - If f is not a direct function call, for every function $func \in \text{Pts}(f)$, there is a call from the callsite to function $func$

$$\mathbf{Inst} ::= x := \&p \mid x := *p \mid *p := x \mid x := y.f \mid x.f := y \mid p := q \mid x := \text{malloc}() \\ \mid x = \&f(p_i, \dots, p_n)$$

Rules of CFA for SIMP

$$\frac{}{\llbracket p := \&f(p_i, \dots, p_n) \rrbracket \hookrightarrow \&f(p_i, \dots, p_n) \in p} \text{ func}$$

$$\frac{l:r = v(a_i, \dots, a_n) \quad \&f(p_i, \dots, p_n) \in v}{\forall i \in [1, n]: p_i \supseteq a_i \quad r \supseteq \text{ret}^f} \text{ call}$$

New Rules for Handling Function Pointers

$$\frac{}{\llbracket n: p := \text{malloc}() \rrbracket \hookrightarrow l_n \in p} \text{ malloc}$$

$$\frac{}{\llbracket p := \&x \rrbracket \hookrightarrow l_x \in p} \text{ address-of}$$

$$\frac{}{\llbracket p := q \rrbracket \hookrightarrow p \supseteq q} \text{ copy}$$

$$\frac{}{\llbracket *p := q \rrbracket \hookrightarrow *p \supseteq q} \text{ assign}$$

$$\frac{}{\llbracket p := *q \rrbracket \hookrightarrow p \supseteq *q} \text{ dereference}$$

Old Rules

Rules of CFA for SIMP

□ Example

```
inc(i) { return i+1; }  
dec(j) { return j-1; }  
ide(k) { return k; }  
foo(n, f) {  
    if (n=0) {  
        f=ide;  
    }  
    r = f(n); // call ?  
    return r;  
}  
input x;  
y = foo(x, inc);
```

⇒ PTS(f) = {ide, inc}

↙
↘ Calls ide(), inc()

□ Expensive to compute but quite precise!

Grammar of λ –calculus

e	$::=$	$\lambda x.e$	$\longleftarrow x$ is the function argument, e is the function body
		x	
		$e_1 e_2$	e_1 is the function to be invoked; e_2 is passed as an argument
		$\text{let } x = e_1 \text{ in } e_2$	Local binding, bind the value of e_1 to x , which is used in e_2
		$\text{if } e_0 \text{ then } e_1 \text{ else } e_2$	
		$n \mid e_1 + e_2 \mid \dots$	

❑ Assume all variable names are distinct (i.e., in SSA form)

❑ Example:

➤ $(\lambda f.f \ 3)(\lambda x.x + 1) \rightarrow (\lambda x.x + 1) \ 3 \rightarrow 4$

CFA for the λ –calculus

□ λ –calculus: theoretical foundation for functional PLs

- E.g., Haskell, Ocaml, Scheme, Erlang, F#, Coq, ...
- Functions as **first-class citizens**: $(\text{fun } f \rightarrow f\ 3)\ (\text{fun } x \rightarrow x + 1)$
- **Higher-order functions**: $\text{let twice } f\ x = f\ (f\ x)$
 - "**twice**" is a higher-order function
- **Closures**: lambda expression + environment
 - Let $f = \text{let } y = 3 \text{ in } \text{fun } x \rightarrow x + y$, closure captures y
 - $(\lambda y. (\lambda x. x + y))\ 3$, the closure is $\langle \lambda x. x + y, \{y \mapsto 3\} \rangle$
- **Recursion** as a primary control structure: recursive factorial
 - $\text{let rec fact } n = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact } (n-1)$
- **Pure functions** (no side effects), immutable data
 - Lambda calculus has no assignment; everything is immutable
 - $\text{let } x = 5$
 - $\text{let } y = x + 1$ (* no mutation *)

CFA for the λ –calculus

□ Goal:

- for each call site " $e_1 \ e_2$ " determine the possible functions for e_1 from the set of definitions $\{\lambda x_1. e, \dots, \lambda x_2. e\}$

□ Notations:

- \mathcal{L} : set of labels associated with expression, i.e., program points
- Var : set of variables in the program
- σ : a mapping from each variable or label to a lattice value (i.e., the set of possible functions)

$$\sigma \in Var \cup \mathcal{L} \rightarrow \mathcal{P}(\lambda x. e)$$

- $\llbracket e \rrbracket^l \hookrightarrow C$: the analysis of expression e with label l generates constraints C over the collecting state σ

CFA Rules for the λ –calculus

e	$::=$	$\lambda x.e$
		x
		$e_1 e_2$
		let $x = e_1$ in e_2
		if e_0 then e_1 else e_2
		$n \mid e_1 + e_2 \mid \dots$

$$\frac{}{\llbracket n \rrbracket^l \hookrightarrow \emptyset} \text{const}$$

$$\frac{}{\llbracket x \rrbracket^l \hookrightarrow \sigma(x) \sqsubseteq \sigma(l)} \text{var}$$

$$\frac{\llbracket e_1^{l_1} \rrbracket \hookrightarrow C_1 \quad \llbracket e_2^{l_2} \rrbracket \hookrightarrow C_2}{\llbracket e_1^{l_1} + e_2^{l_2} \rrbracket^l \hookrightarrow C_1 \cup C_2} \text{add}$$

注意 : $\sigma(l) = \emptyset$

$$\frac{\llbracket e_0^{l_0} \rrbracket \hookrightarrow C_0 \quad \llbracket e_1^{l_1} \rrbracket \hookrightarrow C_1 \quad \llbracket e_2^{l_2} \rrbracket \hookrightarrow C_2}{\llbracket \text{if } e_0^{l_0} \text{ then } e_1^{l_1} \text{ else } e_2^{l_2} \rrbracket^l \hookrightarrow C_0 \cup C_1 \cup C_2 \cup (\sigma(l_1) \cup \sigma(l_2)) \sqsubseteq \sigma(l)} \text{ifelse}$$

Other arithmetic expressions are similar

$$\frac{\llbracket e_1^{l_1} \rrbracket \hookrightarrow C_1 \quad \llbracket e_2^{l_2} \rrbracket \hookrightarrow C_2}{\llbracket \text{let } x = e_1^{l_1} \text{ in } e_2^{l_2} \rrbracket^l \hookrightarrow C_1 \cup C_2 \cup \sigma(l_1) \sqsubseteq \sigma(x) \wedge \sigma(l_2) \sqsubseteq \sigma(l)} \text{lbindexp}$$

$$\frac{\llbracket e \rrbracket^{l_0} \hookrightarrow C}{\llbracket \lambda x. e^{l_0} \rrbracket^l \hookrightarrow \{\lambda x. e\} \sqsubseteq \sigma(l) \cup C} \text{lambda}$$

- $\llbracket e \rrbracket^l \hookrightarrow C$: the analysis of expression e with label l **generates constraints** C over the collecting state σ

$$\sigma \in \text{Var} \cup \mathcal{L} \rightarrow \mathcal{P}(\lambda x. e)$$

$$\frac{\llbracket e_1 \rrbracket^{l_1} \hookrightarrow C_1 \quad \llbracket e_2 \rrbracket^{l_2} \hookrightarrow C_2}{\llbracket e_1^{l_1} e_2^{l_2} \rrbracket^l \hookrightarrow C_1 \cup C_2 \cup \forall \lambda x. e_0^{l_0} \in \sigma(l_1) : \sigma(l_2) \sqsubseteq \sigma(x) \wedge \sigma(l_0) \sqsubseteq \sigma(l)} \text{apply}$$

CFA Rules for the λ –calculus

□Example: $\left(\left(\lambda x. (x^a + 1^b)^c \right)^d (3)^e \right)^g$

$$\frac{\frac{\frac{\llbracket x \rrbracket^a \hookrightarrow \sigma(x) \sqsubseteq \sigma(a) \quad \llbracket 1 \rrbracket^b \hookrightarrow \emptyset}{\llbracket x^a + 1^b \rrbracket^c \hookrightarrow \sigma(x) \sqsubseteq \sigma(a)}}{\llbracket \lambda x. (x^a + 1^b)^c \rrbracket^d \hookrightarrow \{\sigma(x) \sqsubseteq \sigma(a), \{\lambda x. x + 1\} \sqsubseteq \sigma(d)\}} \quad \llbracket 3 \rrbracket^e \hookrightarrow \emptyset}{\llbracket (\lambda x. (x^a + 1^b)^c)^d (3)^e \rrbracket^g \hookrightarrow \boxed{\{\sigma(x) \sqsubseteq \sigma(a), \{\lambda x. x + 1\} \sqsubseteq \sigma(d)\} \cup \forall \lambda t. e_0^{l_0} \in \sigma(d): \sigma(e) \sqsubseteq \sigma(t) \wedge \sigma(l_0) \sqsubseteq \sigma(g)}}$$

↓ 简化

$$\{\sigma(x) \sqsubseteq \sigma(a), \{\lambda x. x + 1\} \sqsubseteq \sigma(d), \sigma(e) \sqsubseteq \sigma(x) \wedge \sigma(c) \sqsubseteq \sigma(g)\}$$

↓ 约束求解出lfp

$$\{\lambda x. x + 1\} \sqsubseteq \sigma(d), \forall e \neq d: \sigma(e) = \emptyset$$

(Optional) Extensions

□ The CFA introduced for λ – calculus is actually 0-CFA

- 0-CFA is less precise, 0 indicates context insensitivity

$$\sigma \in \text{Var} \cup \text{Lab} \rightarrow \mathcal{P}(\lambda x. e)$$

□ k -CFA: k -limited call-string-based CFA

- δ 称为 context

$$\sigma \in (\text{Var} \cup \text{Lab}) \times \Delta \rightarrow \mathcal{P}((\lambda x. e, \delta)) \quad \delta \in \Delta = \text{Lab}^{n \leq k}$$

□ Uniform k -CFA

- Environment η : track the calling context for each variable captured in a closure
- Precise than k – CFA but less efficient

$$\sigma \in (\text{Var} \cup \text{Lab}) \times \Delta \rightarrow \mathcal{P}((\lambda x. e, \eta)) \quad \Delta = \text{Lab}^{n \leq k} \quad \eta \in \text{Var} \rightarrow \Delta$$

(Optional) 作业: CFA for OO and λ

□ 计算下面程序在uniform 1 –CFA分析时, 各函数的上下文和其对应捕获的环境

```
let adde =  $\lambda x.$   
    let h =  $\lambda y. \lambda z. x + y + z$   
    let r = (h 8)r  
    in r  
let t = (adde 2)t  
let f = (adde 4)f  
let e = (t 1)e  
main
```

function	Context δ	Environment η
main		
adde		
h		
r		

□ 阅读Qilin代码, 谈一谈Qilin如何支持CFA?