软件分析与架构设计

控制流分析

何冬杰

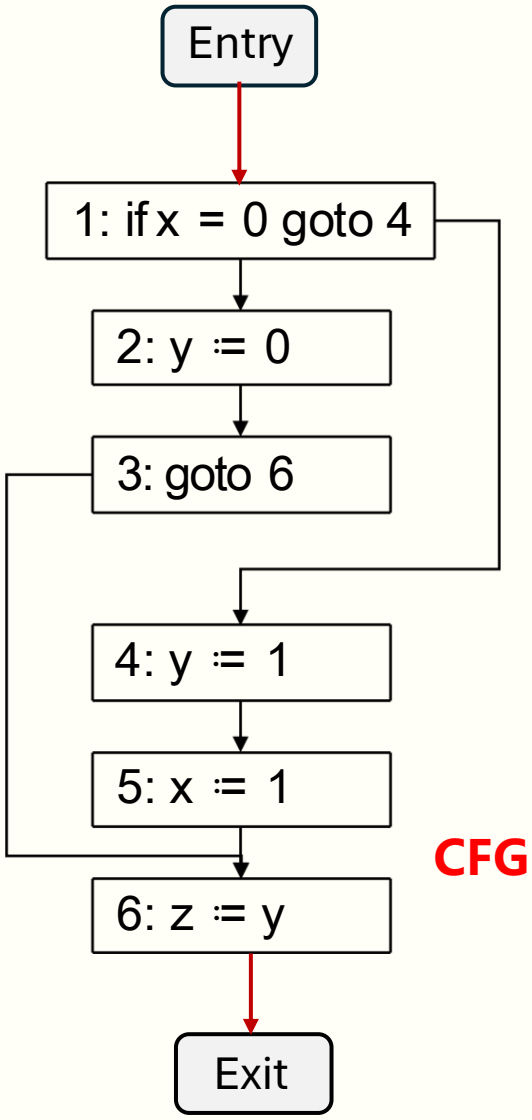
重庆大学

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

$$
\begin{aligned}
1: &\quad \text{if } x = 0 \text{ goto } 4 \\
2: &\quad y := 0 \\
3: &\quad \text{goto } 6 \\
4: &\quad y := 1 \\
5: &\quad x := 1 \\
6: &\quad z := y
\end{aligned}
$$

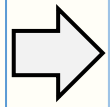
三地址码



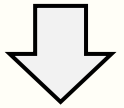
CFG

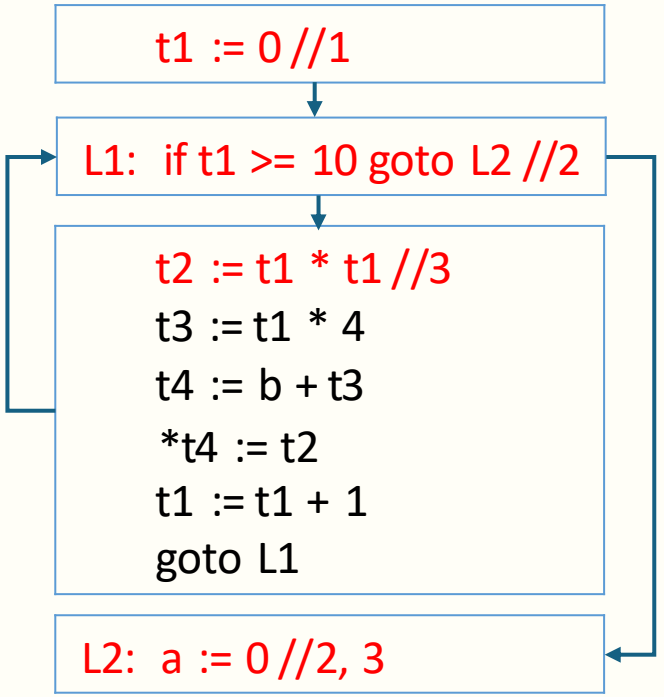
控制流图构建算法

```
      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
```





- **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

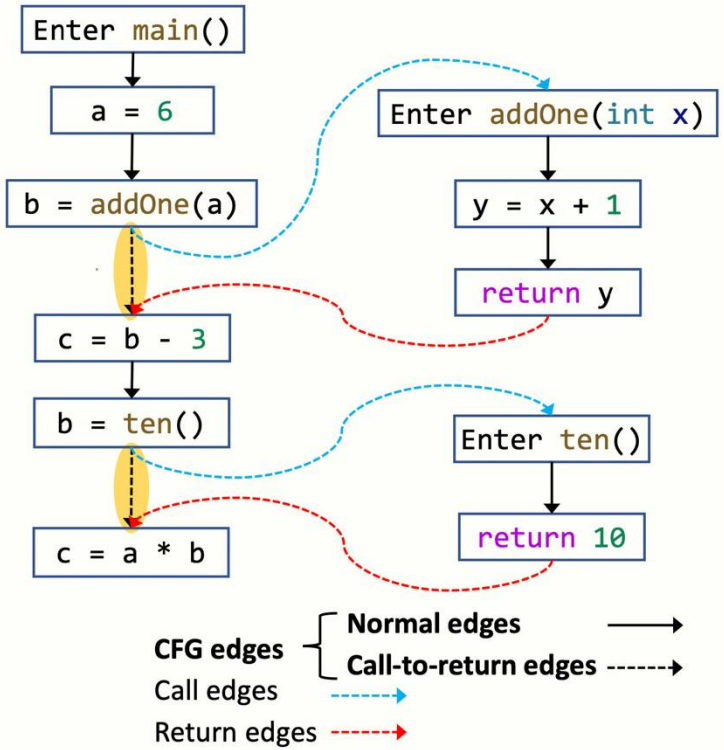
过程间控制流图 (Interprocedural CFG)

❑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）

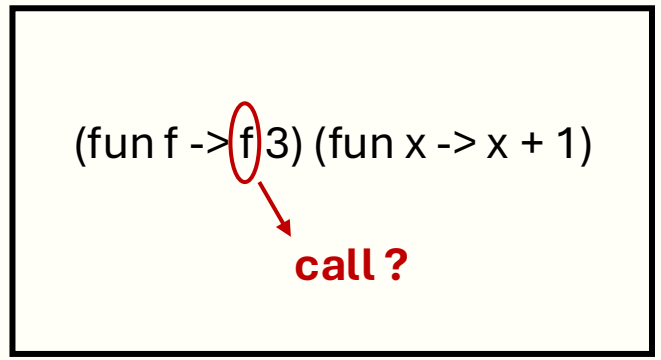
❑确定函数调用目标的分析叫做控制流分析

```
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



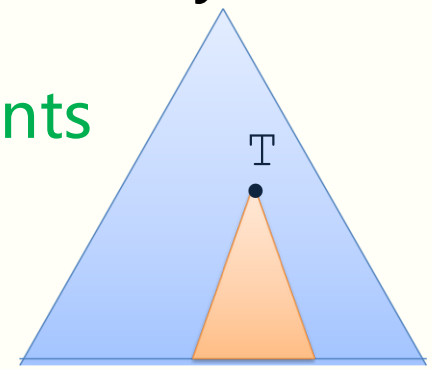
Function application in Ocaml/Haskell

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

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

CFA for OO

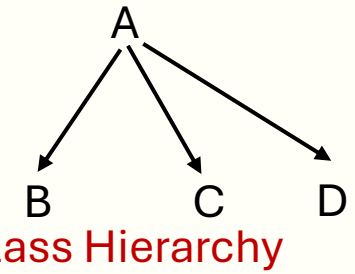
- **CFA in an Object-Oriented language:**
  - Resolve call targets for x.m(a1,…,an)
- **Simplest solution:**
  - Select all methods named **m** with the same number of arguments
- **Class Hierarchy Analysis (CHA):**
  - For a polymorphic call site **m()** on declared type **T**: Call edge to **T.m** 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 ?
```



Both calls A:foo, B:foo, C:foo, and D:foo

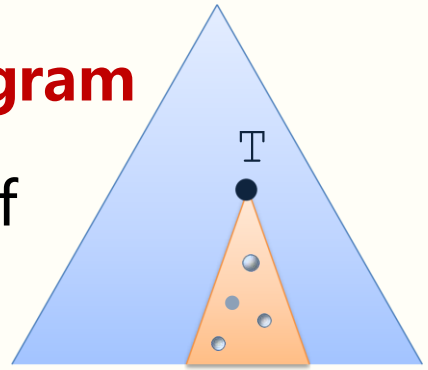
In CHA

In RTA

Both calls A:foo, B:foo, and C:foo

❑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



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

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

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

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

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

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: $[\![\mathrm{x}]\!]$: collected types of x**

$$\frac{l: x = new\ T \quad m = methodOf(l) \quad m \in Methods}{T \in [\![\mathrm{x}]\!]} \qquad \frac{l: x = \mathrm{y} \quad m = methodOf(l) \quad m \in Methods}{[\![\mathrm{y}]\!] \subseteq [\![\mathrm{x}]\!]}$$

Field-based Analysis

$$\frac{l: x = \mathrm{y.f} \quad m = methodOf(l) \quad m \in Methods \quad \mathrm{T} = \mathrm{declareType(y)}}{[\![\mathrm{T.f}]\!] \subseteq [\![\mathrm{x}]\!]}$$

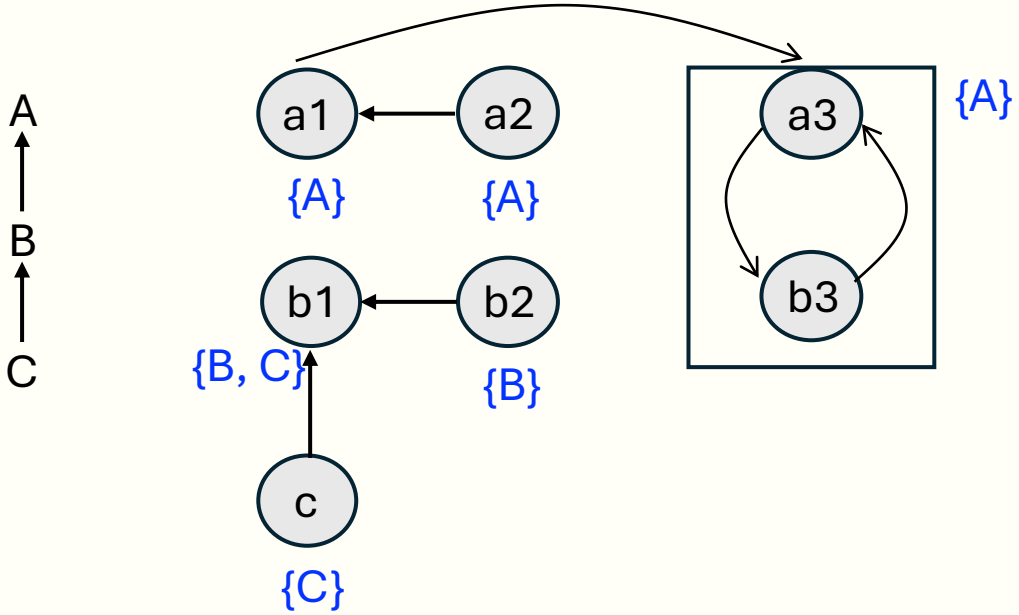
$$\frac{l: x.\mathrm{f} = \mathrm{y} \quad m = methodOf(l) \quad m \in Methods \quad \mathrm{T} = \mathrm{declareType(x)}}{[\![\mathrm{y}]\!] \subseteq [\![\mathrm{T.f}]\!]}$$

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

$$\frac{x=\textbf{new } T//O}{O \in PTS(x)}\ [\text{New}] \qquad \frac{x=y}{PTS(y) \subseteq PTS(x)}\ [\text{Assign}]$$

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

$$\frac{\text{l}: x = \text{r.m}(a_1, \cdots, a_n) \quad \text{O} \in PTS(\text{r}) \quad \boldsymbol{m' = dispatch(l, O)}}{\text{O} \in PTS(this^{m'}) \quad \boldsymbol{m' \in targets(l)}}\ [\text{Call}]$$

$$\forall \text{i} \in [1, \text{n}]: \text{PTS}(a_i) \subseteq \text{PTS}(p_i) \qquad \text{PTS}(ret^{m'}) \subseteq \text{PTS}(\text{x})$$

Spark Analysis

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

```
class A {
  A foo (A x1 ) { return x1 ; } }
class B extends A {
  A foo (A x2) { return new A(); // O1 }}
class C extends A {
  A foo (A x3) { return this; }}
class D extends A {
  A foo (A x4) { return new D(); // O2 }}
A x = new A(); // O3
while (...)
  x = x.foo(new B() // O4); // call ?
A y = new C(); // O5
y.foo(x); // call ?
```

Calls A:foo, B:foo

Calls C:foo

❑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:** $f(a_1, \ldots, a_n)$
  - 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 $func \in \mathrm{Pts(f)}$, there is a call from the callsite to function $func$

$$\boldsymbol{Inst} ::= x := \&p \mid x :=* p \mid *p := x \mid x := y.f \mid x.f := y \mid p := q \mid x := malloc() \mid x = \&f(p_i, \ldots, p_n) \mid x := f(a_i, \ldots, a_n)$$

Rules of CFA for SIMP

$$\frac{}{[\![p := \&f(p_i, \ldots, p_n)]\!] \hookrightarrow \&f(p_i, \ldots, p_n) \in p}\ func$$

$$\frac{l: r = v(a\_i, \ldots, a\_n) \quad \&f(p\_i, \ldots, p\_n) \in v}{\forall\, i \in [1, n]:\ p_i \supseteq a_i \quad r \supseteq ret^f}\ call$$

New Rules for Handling Function Pointers

$$\frac{}{[\![n: p := malloc()]\!] \hookrightarrow l_n \in p}\ malloc$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ address\text{-}of$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ copy$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ assign$$

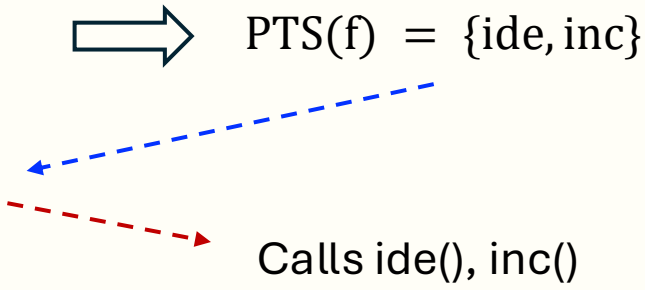
$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ dereference$$

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);
```



❑Expensive to compute but quite precise!

Grammar of λ-calculus

$$
\begin{array}{rcl}
e & ::= & \boxed{\lambda x.e} \\
 & | & x \\
 & | & \boxed{e_1\ e_2} \\
 & | & \boxed{\text{let } x = e_1 \text{ in } e_2} \\
 & | & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \\
 & | & n \mid e_1 + e_2 \mid \ldots
\end{array}
$$

x is the function argument, e is the function body

e_1 is the function to be invoked;
e_2 is passed as an argument

Local binding, bind the value of e_1 to x, which is used in e_2

- 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**: (fun f -> f 3) (fun x -> x + 1)
  - **Higher-order functions**: let twice f x = f (f x)
    - "**twice**" is a higher-order function
  - **Closures**: lambda expression + environment
    - Let f = let y = 3 in fun x -> x + y, closure captures y
    - (λy. (λx. x + y)) 3, the closure is < λx. x + y , { y ↦ 3 } >
  - **Recursion** as a primary control structure: recursive factorial
    - let rec fact n = if n = 0 then 1 else n * fact (n-1)
  - **Pure functions** (no side effects), immutable data
    - Lambda calculus has no assignment; everything is immutable
    - let x = 5
    - let y = x + 1 (* no mutation *)

- Code examples are given in OCaml

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 $\{\lambda x_1.\, e, \ldots, \lambda x_2.\, e\}$
- **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)

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

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

CFA Rules for the λ-calculus

$$\begin{array}{lcl} e & ::= & \lambda x.e \\ & | & x \\ & | & e_1\ e_2 \\ & | & \text{let } x = e_1 \text{ in } e_2 \\ & | & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \\ & | & n \mid e_1 + e_2 \mid \ldots \end{array}$$

$$\frac{}{[\![n]\!]^l \hookrightarrow \varnothing}\ const \qquad \frac{}{[\![x]\!]^l \hookrightarrow \sigma(x) \sqsubseteq \sigma(l)}\ var$$

$$\frac{[\![e_1^{l_1}]\!] \hookrightarrow C_1 \quad [\![e_2^{l_2}]\!] \hookrightarrow C_2}{[\![e_1^{l_1} + e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2}\ add$$

注意：$\sigma(l) = \emptyset$

Other arithmetic expressions are similar

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

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

$$\frac{[\![e]\!]^{l_0} \hookrightarrow C}{[\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{\lambda x.e\} \sqsubseteq \sigma(l) \cup C}\ lambda$$

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

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

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

- $\sigma(x)$**表示**x**的**值可能是哪些lambdas
- $\sigma(l)$**表示**l处会调用哪些 lambdas
- 规则用于生成表达式及其子表达式蕴**含的所有**约束条件
- 约束条件刻画$\sigma(x)$**和**$\sigma(y)$**之**间的关
  - x和y均可以是变量或标签

CFA Rules for the λ-calculus

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

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

⇩ 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

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

Extensions

- **The CFA introduced for λ-calculus is actually 0-CFA**
  - 0-CFA is less precise, 0 indicates context insensitivity

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

- **k-CFA: k-limited call-string-based CFA**
  - δ称为context

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

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

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

(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

$$
\begin{aligned}
\textbf{let}\ adde &= \lambda x. \\
&\quad \textbf{let}\ h = \lambda y.\ \lambda z.\ x + y + z \\
&\quad \textbf{let}\ r = (h\ 8)^r \\
&\quad \textbf{in}\ r \\
\textbf{let}\ t &= (adde\ 2)^t \\
\textbf{let}\ f &= (adde\ 4)^f \\
\textbf{let}\ e &= (t\ 1)^e
\end{aligned}
$$

main

function	Context δ	Environment η
main		
adde		
h		
r		

- 阅读Qilin代码，谈一谈Qilin如何支持CFA?