Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling

MINSEOK JEON, Korea University, Republic of Korea
SEHUN JEONG, Korea University, Republic of Korea
HAKJOO OH∗ , Korea University, Republic of Korea

OOPSLA 2018

动机

上下文敏感是开发精确和可扩展的指向分析的关键。目前已经提出的上下文敏感性有：

- 调用点敏感性：以调用位置作为上下文元素
- 对象敏感性：以调用方法所属的抽象对象作为上下文元素，在Java中尤为重要
- 类型敏感性：以调用方法所属的抽象对象的类型作为上下文元素
- 混合上下文敏感性：混合使用调用点敏感性和对象敏感性

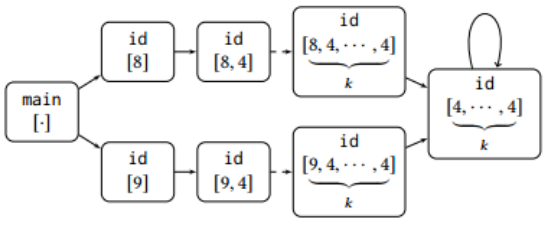
在实现中，对于上述敏感性都会采用k-limited的方式，只记录最近的k个上下文元素，因为记录所有的上下文元素是不现实也不可能的。

然而，这些方法会给指向分析的空间使用和耗时带来巨大的开销，因为它需要记录每个调用点的上下文，并且对每个不同上下文的同一个调用点都需要进行分析。因此，选择上下文敏感性成为了一种有效提升上下文敏感可扩展性的方式。选择上下文敏感性会有选择地对精度相关的调用点进行上下文敏感。

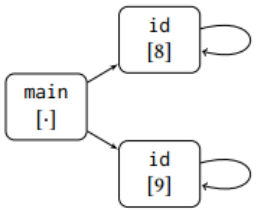
本文提出了一种基于上下文隧道的选择上下文敏感指向分析算法，它通过机器学习的算法确定哪些调用不需要更新上下文，从而能够一定程度上降低开销并且提高精度。

```
1 class A {} class B {}
2 class C {
3   static Object id (Object v, int i){
4       return i >= 0 ? id(v, i-1) : v;
5   }
6   public static void main (){
7     int i = input();
8     A a = (A) id(new A(), i); //Query 1
9     B b = (B) id(new B(), i); //Query 2
10  }
11 }
```

(a) Example code



(b) Call-graph by k-CFA



(c) Call-graph by 1-CFA with tunneling

上图给出了一个能够显示上下文隧道作用的示例。(a)中定义了三个类A、B、C，C中定义了两个方法id和main。其中id方法是一个递归函数，不过无论i的输入如何，它都会返回第一个参数v；main方法中会在第8、9行创建两个类A、B的对象，并传递给id函数。假设指向分析的目标是确定第8、9行的强制类型转换是否合法。在k-调用点敏感中，以调用点的位置作为上下文元素，例如[8, 4]表示第4行的调用点，它的前两层调用者分别是在第4、8行被调用的。但是由于k-limited会在每个函数调用点处不加选择地更新上下文元素，所以对于这样的递归调用，算法有可能会由于k的限制，将能够区分参数v类型的上下文元素

[8]、[9]更新掉，最后上下文只剩下了k个[4]，导致算法认为形参v指向的对象是{a, b}，从而会认为8、9两行的强制类型转换可能发生错误。

而本文提出的上下文隧道想法则认为，对于一些函数调用，我们不需要更新上下文元素，例如在这个例子中，对于第4行的调用，我们如果不更新上下文元素，让它直接复用之前的上下文元素，我们就会得到如图(c)所示的CFA，从而就可以得到在上下文为8时v的指向集合为a、在上下文为9时指向集合为b，从而能够实现更高精度的指向分析。

实际上，上下文隧道的概念可以应用到任何k-limited上下文敏感分析上，包括上述提到的四种。

算法概述

上下文隧道的指向分析算法主要是基于k-limited的上下文敏感指向分析算法的改进，它通过机器学习得到的启发式函数来决定对于哪些函数调用是需要创建"隧道"的。我们将一个函数对(m_1, m_2)称为一个隧道，它表示在分析m_1调用方法m_2时不更新上下文元素，让m_2直接复用m_1的上下文元素进行分析。

因此，相对于原来的上下文敏感分析，上下文隧道的分析只需要略微修改针对调用语句的处理规则。

上下文隧道启发式的学习

本文通过机器学习的方式来洗的一个生成上下文隧道的启发式。

1. 将程序中的方法/函数使用以下23个特征来表示，每个方法可以表示成特征的合取式如 $A1 \wedge A2 \wedge B12 \wedge \neg B7$。

Class A (Signature features)									
A1	"java"	A2	"lang"	A3	"sun"	A4	"()"	A5	"void"
A6	"security"	A7	"int"	A8	"util"	A9	"String"	A10	"init"

Class B (Additional features)			
B1	Methods contained in nested class	B7	Methods containing static method invocation
B2	Methods taking multiple arguments	B8	Methods containing virtual method invocation
B3	Methods containing array load	B9	Static method
B4	Methods containing local assignments	B10	Methods containing a single heap allocation
B5	Methods containing local variables	B11	Methods taking an argument of Object type
B6	Methods containing field store	B12	Methods containing multiple heap allocations
		B13	Methods contained in a large class

2. 学习的目标为一个启发式函数$\Pi = \langle f_1, f_2 \rangle$。对于一个函数对$(m_1, m_2)$，若$m_1$满足$f_1$或$m_2$满足$f_2$, 它是一个上下文隧道。
    - 函数m满足f_1，意味着m作为调用者时直接将它的上下文元素赋给它的被调用者会有更好的表现
    - 函数m满足f_2，意味着m作为被调用者时直接复用调用者的上下文元素会有更好的表现

本质上是要解决下列优化问题：找到一个最优的$\Pi = \langle f_1, f_2 \rangle$，使得根据其所能证明的安全类型转换最多，并且分析的开销低于不采用上下文隧道时的开销。

学习算法会先尝试学习f_2，再在此基础上学习f_1.

Algorithm 1 Overall Algorithm

Input: Static analyzer F, codebase $\mathbf{P}$, atomic features $\mathbb{A}$
Output: Model parameters f_1 and f_2

procedure LEARN(F, $\mathbf{P}$, $\mathbb{A}$)
  $f_2 \leftarrow$ LEARNPARAMETER(2, *false*, F, $\mathbf{P}$, $\mathbb{A}$) ▷ learn methods for child contexts
  $f_1 \leftarrow$ LEARNPARAMETER(1, f_2, F, $\mathbf{P}$, $\mathbb{A}$) ▷ learn methods for parent contexts
  return $\langle f_1, f_2 \rangle$
end procedure

学习的具体方式为：先收集种子特征。随后对于最有潜力的种子，先尝试对它进行调优，随后检查根据这个种子生成的规则是否优于之前的规则，是的化则更新规则。最终迭代到所有种子都遍历完毕为止。

Algorithm 2 Learning a Single Parameter

Input: Index i of parameter to learn, parameter f_2, static analyzer F, codebase $\mathbf{P}$, atomic features $\mathbb{A}$
Output: i^{th} parameter f_i

procedure LEARNPARAMETER(i, f_2, F, $\mathbf{P}$, $\mathbb{A}$)
  $f_1 \leftarrow$ *false*
  $\Pi \leftarrow \langle f_1, f_2 \rangle$
  $W \leftarrow \{a \in (\mathbb{A} \cup \neg\mathbb{A}) \mid \text{SeedFeature}(a, i, \Pi, F, \mathbf{P})\}$ ▷ collect seed features
  while $W \neq \emptyset$ **do**
    $s \leftarrow \text{ChooseSeed}(i, \Pi, F, \mathbf{P}, W)$ ▷ pick a seed feature from W with highest potential
    $W \leftarrow W \setminus \{s\}$
    $c \leftarrow$ REFINESEED(s, i, Π, $\mathbb{A}$, F, $\mathbf{P}$) ▷ refine seed feature s
    $\Pi' \leftarrow \Pi[f_i \mapsto f_i \vee c]$ ▷ new parameter to be evaluated
    if BetterHeuristicFound(Π, Π', $\mathbf{P}$) **then** ▷ check whether new parameter improves
      $\Pi \leftarrow \Pi'$ ▷ update parameter
    end if
  end while
  return $\Pi(i)$ ▷ return f_i
end procedure

种子调优的方式为：在种子的基础上尝试和别的特征组合，若组合能够提高精度或者在保持精度不变的情况下降低开销，那么就是一个有效的调优。

Algorithm 3 Refining a Seed Feature

Input: Seed feature s, parameter index i, parameters Π, atomic features $\mathbb{A}$, static analyzer F, codebase $\mathbf{P}$
Output: refined conjunction c

procedure REFINESEED(s, i, Π, $\mathbb{A}$, F, $\mathbf{P}$)
  $c \leftarrow s$ ▷ initial conjunction
  Failed $\leftarrow \emptyset$ ▷ *Failed* will maintain features that fail to refine c
  while $(a \leftarrow \text{ChooseRefiner}(\mathbb{A}, f_i, c, \mathbf{P}, \textit{Failed})) \neq \textit{false}$ **do** ▷ iteratively refine conjunction c
    $c' \leftarrow c \wedge a$ ▷ refine c with a
    $\Pi' \leftarrow \Pi[f_i \mapsto f_i \vee c]$ ▷ old parameter
    $\Pi'' \leftarrow \Pi[f_i \mapsto f_i \vee c']$ ▷ new (refined) parameter
    if $\text{Prec}^{+}(\Pi', \Pi'', \mathbf{P}) \wedge \text{HasPotential}(\Pi'', \Pi, \mathbf{P})$ **then** ▷ precision improved
      $c \leftarrow c'$
    else if $\text{Prec}^{=}(\Pi', \Pi'', \mathbf{P}) \wedge \text{Cost}^{-}(\Pi', \Pi'', \mathbf{P})$ **then** ▷ cost reduced without precision loss
      $c \leftarrow c'$
    else
      Failed $\leftarrow$ *Failed* $\cup \{a\}$ ▷ record failed attempt
    end if
  end while
  return c
end procedure

评估

评估分为三个部分：

1. 上下文隧道的有效性

2. 学习方法的必要性和有效性

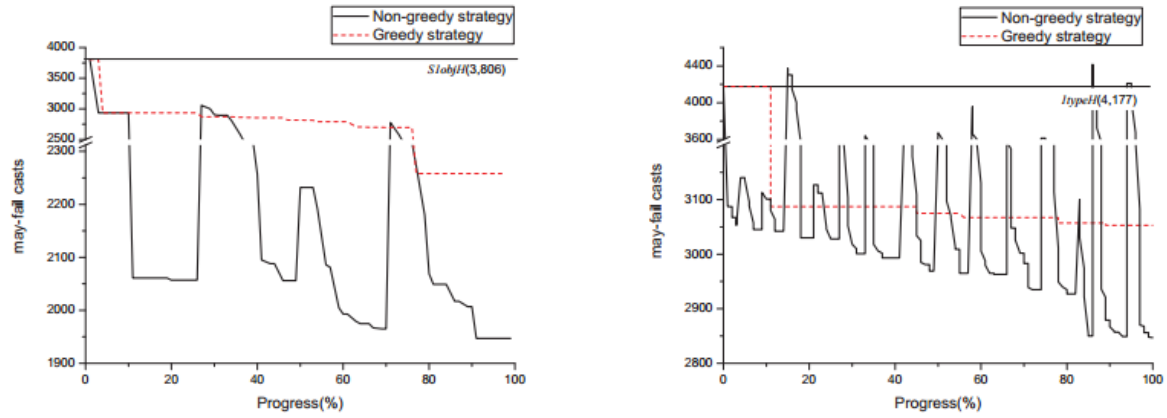
3. 学到启发式

对于第一个问题，通过将上下文隧道应用到不同的上下文敏感算法中，并进行对比来说明：

			Hybrid Context Sensitivity			Object Sensitivity		
			S1objH+T	*S1objH*	*S2objH*	*1objH+T*	*1objH*	*2objH*
Training programs	luindex	**may-fail casts**	**371**	783	415	462	796	496
		analysis time(s)	**34**	66	36	37	51	37
		reachable mthds	7,700	7,907	7,702	7,702	7,876	7,702
		call-graph-edges	0.2M	0.5M	0.9M	0.3M	0.5M	1M
	lusearch	**may-fail casts**	**380**	850	420	469	812	508
		analysis time(s)	**37**	79	63	41	56	64
		reachable mthds	8,342	8,580	8,344	8,344	8,526	8,344
		call-graph-edges	0.2M	0.5M	2M	0.3M	0.5M	2.1M
	antlr	**may-fail casts**	**483**	956	530	570	985	611
		analysis time(s)	**47**	85	50	50	67	45
		reachable mthds	8,712	8,917	8 ,714	8,714	8,886	8,714
		call-graph-edges	0.2M	0.6M	0.9M	0.3M	0.6M	1M
	pmd	**may-fail casts**	**713**	1,217	761	812	1,210	846
		analysis time(s)	**53**	129	56	57	77	57
		reachable mthds	9,086	9,322	9,090	9,090	9,277	9,090
		call-graph-edges	0.3M	0.7M	1.3M	0.3M	0.6M	1.3M
Testing programs	eclipse	**may-fail casts**	**586**	1,061	625	698	1,092	729
		analysis time(s)	**41**	129	49	47	94	51
		poly v-calls	1,180	1,404	1,179	1,181	1,395	1,179
		reachable mthds	9,195	9,461	9,188	9,197	9,408	9,188
		call-graph-edges	0.3M	0.8M	1.4M	0.4M	0.8M	1.5M
	xalan	**may-fail casts**	**572**	1,129	623	680	1,055	720
		analysis time(s)	**64**	187	465	400	179	2,047
		poly v-calls	1,628	1,916	1,624	1,633	1,861	1,628
		reachable mthds	10,325	10,560	10,327	10,336	10,511	10,336
		call-graph-edges	0.4M	1M	9M	1.9M	1M	35M
	fop	**may-fail casts**	**1,080**	1,975	1,107	1,253	1,968	1,270
		analysis time(s)	**121**	916	513	176	1,797	428
		poly v-calls	2,081	2,733	2,041	2,063	2,650	2,047
		reachable mthds	14,374	15,741	14,373	14,376	15,733	14,373
		call-graph-edges	1M	3.2M	12.1M	1.6M	3.9M	11.4M
	chart	**may-fail casts**	**876**	2,290	915	1,011	2,226	1,055
		analysis time(s)	**73**	1,299	488	107	2248	316
		poly v-calls	1,614	2,792	1,614	1,616	2,670	1,614
		reachable mthds	12,503	16,037	12,510	12,510	15,977	12,510
		call-graph-edges	0.5M	2.6M	11M	0.7M	3.2M	11.3M
	bloat	**may-fail casts**	**1,251**	1,931	1,326	1,374	1,911	1,407
		analysis time(s)	**464**	707	2,211	463	557	2,314
		poly v-calls	1,668	2,075	1,650	1,652	2,071	1,650
		reachable mthds	9,928	10,159	9,914	9,914	10,116	9,914
		call-graph-edges	1.4M	2.1M	35M	1.4M	1.9M	35.3M
	jython	**may-fail casts**	**837**	1,308	-	-	-	-
		analysis time(s)	**425**	730	>5,400	>5,400	>5,400	>5,400
		poly v-calls	1,394	1,619	-	-	-	-
		reachable mthds	10,626	11,012	-	-	-	-
		call-graph-edges	1.1M	2.1M	-	-	-	-

实验说明，上下文隧道能够应用到所有主流的上下文敏感算法中，并且大部分都能得到更高的精度和更低的开销

对于第二个问题，本文给出了多个维度的评估，包括学习是否采用贪婪算法、采用不同的特征描述会给算法精度和开销带来哪些影响，当前算法的开销如何。分别如下图所示。



(a) Hybrid context-sensitivity　　(b) Type-sensitivity

上图反映了是否采用贪婪算法对算法精度的影响，可以发现非贪婪算法的精度更高（纵轴越低精度越高），但是贪婪算法的收敛更快。

Benchmarks	Baseline(*S1objH*)		With A only		With B only		With A and B	
	alarms	time(s)	alarms	time(s)	alarms	time(s)	alarms	time(s)
eclipse	1,061	129	807	69	583	47	586	41
xalan	1,129	187	866	179	585	137	572	64
fop	1,975	916	1,250	179	1,102	163	1,080	121
chart	2,290	1,299	1,200	120	887	98	876	73
bloat	1,931	707	1,634	1,156	1,250	548	1,251	464
jython	1,308	730	1,039	379	844	3,747	837	425
TOTAL	9,694	3,968	6,796	2,082	5,251	4,740	5,202	1,188

上表则反映了不同特征使用下算法的开销和精度如何。A类特征对算法的开销影响较大，B类特征对算法的精度影响较大，前者是降低开销，后者是提升精度。

	Learning Cost		eclipse	xalan	fop	chart	bloat	jython
Full learning ($\langle f_1, f_2 \rangle$)	54 hours	alarms	586	572	1,080	876	1,251	837
		time(s)	41	64	121	73	464	425
Approximate ($\langle false, f_2 \rangle$)	29 hours	alarms	605	588	1,099	897	1,317	855
		time(s)	33	54	90	57	335	367

上表显示了在分析f_1和不分析f_1的情况下算法的精度和开销，f_1函数可以提高精度，但是开销的提升更明显。