Java 仮想機械手続きのための新しいデータフロー解析について

萩谷昌己
東京大学・大学院理学系研究科
情報科学専攻

Java バイトコードの安全性を検査する Java 仮想機械のバイトコード検証系は、信頼できないリモートホストからのモーバイルコードの安全性を保証するための Java のセキュリティーモデルの根幹をなしている。しかし、Java バイトコードのための型体系はいくつかの技術的な問題を有している。その一つは手続きの扱いに関するものである。

Stata と Abadi の仕事、および、Qian の仕事をベースにして、本研究では、Java 仮想機械の手続きのための型体系を新たに提案する。我々の型体系は `last`(x) という形の型を含んでいる。この型を持つ値は、手続きの呼び手の x 番目の変数の値と同じになる。また、戻りアドレスの型は `return`(n) という形であり、これは n 段階前の呼び手に戻ることを意味する。これらの型により、純粋に型のみを用いて命令を解析することができる。さらに、プログラムによっては、既存の方法よりも強力な解析を行なうことができる。

On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines

Masami Hagiya
Department of Information Science,
Graduate School of Science,
University of Tokyo
hagiya@is.s.u-tokyo.ac.jp

The bytecode verifier of Java Virtual Machine, which checks type safety of Java bytecode, is a basis of the security model of Java for guaranteeing safety of mobile code sent from an untrusted remote host. However, the type system for Java bytecode has some technical problems, one of which is in the handling of subroutines.

Based on the work by Stata and Abadi and that by Qian, this paper presents yet another type system for subroutines of Java Virtual Machine. Our type system includes types of the form `last`(x). A value whose type is `last`(x) is the same as that of the x-th variable of the caller of the subroutine. In addition, we represent the type of a return address by the form `return`(n), which means returning to the n-th upward caller. Thanks to these types, we can analyze instructions purely in terms of types. Moreover, for some programs our method turns out to be more powerful than existing ones.

1 Introduction

One of the contributions of Java is in its bytecode verifier, which checks type safety of bytecode for JVM (Java Virtual Machine) prior to execution. Thanks to the bytecode verifier, bytecode sent from an untrusted remote host can be executed without the danger of causing type errors and destroying the entire security model of JVM, even when source code is not available. Verifying type safety of bytecode (or native code) seems to be a new research area that is not only technically interesting but also practically important due to availability of remote binary code in web browsers or other applications.

Bytecode verification of JVM has some technical challenges. One is that of handling object initialization, where objects created but not initialized yet may open security holes. Another is that of handling polymorphism of subroutines. This paper is on the latter issue.

Stata and Abadi defined a type system for a small subset of JVM and proved its correctness with respect to the operational semantics of the subset [4]. Qian also defined a similar type system [3]. Both the systems make use of the information on which variables are accessed or modified in a subroutine. Those variables that are not accessed or modified are simply ignored during analysis of the subroutine.

This paper takes a different approach. We introduce types of the form $\texttt{last}(x)$. A value whose type is $\texttt{last}(x)$ is the same as that of the x-th variable of the caller of the subroutine. In addition, we represent the type of a return address by the form $\textbf{return}(n)$, which means returning to the n-th upward caller. Thanks to these types, we can analyze instructions purely in terms of types.

For some programs (unfortunately not those produced by the Java compiler) our method is more powerful than the existing ones. We hope that our method can be modified and applied to analysis of other kinds of bytecode or native code [1, 2].

In this short paper, we present our type system and discuss its correctness. We also briefly describe implementation.

2 Virtual Machine

2.1 Values

A value is a return address or an integer or an object pointer. We can easily add other kinds of value, such as that of floating point number. In the following formal treatment of the operational semantics of the virtual machine, a return address has the constructor **retaddr**, an integer the constructor **intval**, and an object pointer the constructor **objval**. They all take an integer as an argument.

2.2 Instructions

A bytecode program is a list of instructions. An instruction takes one of the following formats.

$\textbf{jsr}(L)$	(L: subroutine address)
$\textbf{ret}(x)$	(x: variable index)
$\textbf{load}(x)$	(x: variable index)
$\textbf{store}(x)$	(x: variable index)
const0	
constNULL	
$\textbf{inc}(x)$	(x: variable index)
$\textbf{if0}(L)$	(L: branch address)
$\textbf{ifNULL}(L)$	(L: branch address)
halt	

Each mnemonic is considered as a constructor of instructions. Some of the mnemonics takes a nonnegative integer x or L as an operand.

2.3 Operational Semantics

The virtual machine consists of

- the program, which is a list of instructions and denoted by P,
- the program counter, which is an index to P,
- the local variables, where the list of values of the local variables is denoted by f, and
- the operand stack, denoted by s.

Let us use the notation $l[i]$ for extracting the i-th element of list l, where the first element of l has the index 0. The i-th instruction of the program P is denoted by $P[i]$. The value of the x-th local variable is denoted by $f[x]$. The p-th element of the operand stack s is denoted by $s[p]$, where $s[0]$ denotes the top element of s.

As in the work by Stata and Abadi, the operational semantics of the virtual machine is defined as a transition relation between triples of the form $\langle i, f, s\rangle$, where i is the program counter, i.e., the index to the program P, f the value list of the local variables, and s the operand stack. While the length of s may change during execution of the virtual machine, the length of f, i.e., the number of local variables is unchanged. The program P, of course, never changes during execution.

The transition relation is defined as follows.

- If $P[i] = \textbf{jsr}(L)$, then
  $$\langle i, f, s\rangle \rightarrow \langle L, f, \textbf{retaddr}(i+1)::s\rangle.$$
  The return address $\textbf{retaddr}(i+1)$ is pushed onto the operand stack. The operator :: is the *cons*

operator for lists.

- If $P[i] = \mathtt{ret}(x)$ and $f[x] = \mathtt{retaddr}(j+1)$, then
  $\langle i, f, s\rangle \rightarrow \langle j+1, f, s\rangle$.
- If $P[i] = \mathtt{load}(x)$, then
  $\langle i, f, s\rangle \rightarrow \langle i+1, f, f[x]{::}s\rangle$.
- If $P[i] = \mathtt{store}(x)$, then
  $\langle i, f, v{::}s\rangle \rightarrow \langle i+1, f[x \mapsto v], s\rangle$.
  The notation $f[x \mapsto v]$ means a list whose element is the same as that of f except for the x-th element, which is set to v.
- If $P[i] = \mathtt{const0}$, then
  $\langle i, f, s\rangle \rightarrow \langle i+1, \mathtt{intval}(0), s\rangle$.
- If $P[i] = \mathtt{constNULL}$, then
  $\langle i, f, s\rangle \rightarrow \langle i+1, \mathtt{objval}(0), s\rangle$.
- If $P[i] = \mathtt{inc}(x)$ and $f[x] = \mathtt{intval}(k)$, then
  $\langle i, f, s\rangle \rightarrow \langle i+1, f[x \mapsto \mathtt{intval}(k+1)], s\rangle$.
- If $P[i] = \mathtt{if0}(L)$, then
  $\langle i, f, \mathtt{intval}(0){::}s\rangle \rightarrow \langle L, f, s\rangle$.
  If $P[i] = \mathtt{if0}(L)$ and $k \neq 0$, then
  $\langle i, f, \mathtt{intval}(k){::}s\rangle \rightarrow \langle i+1, f, s\rangle$.
- If $P[i] = \mathtt{ifNULL}(L)$, then
  $\langle i, f, \mathtt{objval}(0){::}s\rangle \rightarrow \langle L, f, s\rangle$.
  If $P[i] = \mathtt{ifNULL}(L)$ and $k \neq 0$, then
  $\langle i, f, \mathtt{objval}(k){::}s\rangle \rightarrow \langle i+1, f, s\rangle$.

The transition relation $\rightarrow$ is considered as the least relation satisfying the above conditions.

The relation is defined so that when a type error occurs, no transition is defined. This means that to show type safety is to show that a transition sequence stops only at the **halt** instruction.

For proving the correctness of our bytecode analysis, we also need another version of the operational semantics that maintains invocation histories of subroutines. This semantics corresponds to the *structured dynamic semantics* of Stata and Abadi. The transition relation is now defined for quadruples of the form $\langle i, f, s, h\rangle$, where the last component h is an invocation history of subroutines. It is a list of addresses of callers of subroutines. This component is only changed by the `jsr` and `ret` instructions.

- If $P[i] = \mathtt{jsr}(L)$, then
  $\langle i, f, s, h\rangle \rightarrow \langle L, f, \mathtt{retaddr}(i+1){::}s, i{::}h\rangle$.
  Note that the address i of the caller of the subroutine is pushed onto the invocation history.
- If $P[i] = \mathtt{ret}(x)$, $f[x] = \mathtt{retaddr}(j+1)$ and $h = h'@[j]@h''$, where j does not appear in h', then
  $\langle i, f, s, h\rangle \rightarrow \langle j+1, f, s, h''\rangle$.
  The operator @ is the *append* operator for lists.

For other instructions, the invocation histories before and after transition are the same.

As for the two transition relations, we immediately have the following proposition.

Proposition 1: If $\langle i, f, s, h\rangle \rightarrow \langle i', f', s', h'\rangle$, then $\langle i, f, s\rangle \rightarrow \langle i', f', s'\rangle$.

3 Analysis

3.1 Types

Types in our analysis are among the following syntactic entities:

$\top, \bot$	(top and bottom)
$\mathtt{INT}, \mathtt{OBJ}, \cdots$	(basic types)
$\mathtt{return}(n)$	(n: caller level)
$\mathtt{last}(x)$	(x: variable index)

A type is $\top$, $\bot$, a basic type, a **return** type, or a **last** type. In this paper, we assume as basic types `INT`, the type of integers, and `OBJ`, the type of object pointers. It is easy to add other basic types, such as that of floating point numbers.

return types and **last** types are only meaningful inside a subroutine. A **return** type is the type of a return address. For positive integer n, $\mathtt{return}(n)$ denotes the type of the address for returning to the n-th upward caller. For example, $\mathtt{return}(1)$ denotes the type of the address for returning to the direct caller of the subroutine, and $\mathtt{return}(2)$ the type of the address for returning to the caller of the caller.

A **last** type means that a value is passed from the caller of the subroutine. For nonnegative integer x, $\mathtt{last}(x)$ denotes the type of a value that was stored in the x-th local variable of the caller. A value can have this type only when it is exactly the same as the value of the x-th local variable when the subroutine was called.

3.2 Order among Types

We define the order among types as follows.

$$\top > \mathtt{INT} > \bot$$
$$\top > \mathtt{OBJ} > \bot$$
$$\top > \mathtt{return}(n) > \bot$$
$$\top > \mathtt{last}(x) > \bot$$

Since we do not distinguish object pointers by their classes in this paper, the order is *flat*, with $\top$ and $\bot$ as the top and bottom elements.

This order is extended to lists of types. For type lists $\vec{t_1}$ and $\vec{t_2}$, $\vec{t_1} > \vec{t_2}$ holds if and only if $\vec{t_1}$ and $\vec{t_2}$ are of the same length and $\vec{t_1}[i] > \vec{t_2}[i]$ holds for any i ranging over the indices for the lists.

3.3 Target of Analysis

The target of our bytecode analysis is to obtain the following pieces of information for the i-th instruction of the given program P.

$F_i \quad S_i \quad H_i$

F_i is a type list. $F_i[x]$ describes the type of $f[x]$, i.e., the value of the x-th local variable of the virtual machine. S_i is a also type list. Each element of S_i describes the type of the corresponding element of the operand stack of the virtual machine. Both F_i and S_i describe the types of the components of the virtual machine just before the i-th instruction is executed. H_i is a set of invocation histories for the i-th instruction.

F, S and H should follow a rule that is defined for each kind of $P[i]$. The rule says that certain conditions must be satisfied before and after the execution of $P[i]$.

Rule for `jsr`) If $h \in H_i$ and $P[i] = \mathtt{jsr}(L)$, then the following conditions must be satisfied.

- $0 \leq L < |P|$.
- For each variable index y, either
  $F_L[y] \geq \mathtt{return}(n+1)$
  (if $F_i[y] = \mathtt{return}(n)$), and
  $F_L[y] \geq F_i[y]$
  (if $F_i[y]$ is neither `return` nor `last`)
  or
  $F_L[y] \geq \mathtt{last}(y)$
  (even if $F_i[y]$ is `last`).
- $|S_L| = |S_i| + 1$, where $|l|$ denotes the length of list l.
- $S_L[0] \geq \mathtt{return}(1)$.
- For each index p, where $0 \leq p < |S_i|$,
  $S_i[p]$ is not `last`,
  $S_L[p+1] \geq S_i[p]$
  (if $S_i[p]$ is not `return`), and
  $S_L[p+1] \geq \mathtt{return}(n+1)$
  (if $S_i[p] = \mathtt{return}(n)$).
- i does not appear in h. (Recursion is not allowed.)
- $i{::}h \in H_L$.

Note that when $F_i[y]$ is not `last`, $F_L[y]$ cannot be determined uniquely. We must use some kind of backtracking for implementing our analysis.

Rule for `ret`) If $h \in H_i$ and $P[i] = \mathtt{ret}(x)$, then the following conditions must be satisfied.

- $F_i[x] = \mathtt{return}(n)$.
- $h = h'@[j]@h''$, where $|h'| = n-1$.
- $0 \leq j+1 < |P|$.
- For each variable index y,
  $F_{j+1}[y] \geq \mathtt{follow\_last}(n, h, F_i[y])$.
- $S_{j+1} \geq \mathtt{follow\_last}(n, h, S_i)$.
- $h'' \in H_{j+1}$.

`follow_last` is a function for extracting the type of a variable in a caller of a subroutine according to an invocation history. For nonnegative integer n, invocation history h and type t, $\mathtt{follow\_last}(n, h, t)$ is defined as follows.

$$\mathtt{follow\_last}(0, h, t) = t$$
$$\mathtt{follow\_last}(n+1, i{::}h, \mathtt{return}(m)) =$$
$$\textbf{if } m > n+1 \textbf{ then } \mathtt{return}(m-n-1)$$
$$\textbf{else } \top$$
$$\mathtt{follow\_last}(n+1, i{::}h, \mathtt{last}(x)) =$$
$$\mathtt{follow\_last}(n, h, F_i[x])$$
$$\mathtt{follow\_last}(n+1, i{::}h, t) = t \quad \textbf{(otherwise)}$$

`follow_last` is extended to type lists, i.e., $\mathtt{follow\_last}(n, h, \vec{t})$ is also defined when $\vec{t}$ is a type list.

Rule for `load`) If $h \in H_i$ and $P[i] = \mathtt{load}(x)$, then the following conditions must be satisfied.

- $0 \leq i+1 < |P|$.
- $F_{i+1} \geq F_i$.
- $S_{i+1} \geq F_i[x]{::}S_i$.
- $h \in H_{i+1}$.

Rule for `store`) If $h \in H_i$ and $P[i] = \mathtt{store}(x)$, then the following conditions must be satisfied.

- $0 \leq i+1 < |P|$.
- $S_i = t{::}\vec{t}$.
- $F_{i+1} \geq F_i[x \mapsto t]$.
- $S_{i+1} \geq \vec{t}$.
- $h \in H_{i+1}$.

Rule for `const0`) If $h \in H_i$ and $P[i] = \mathtt{const0}$, then the following conditions must be satisfied.

- $0 \leq i+1 < |P|$.
- $F_{i+1} \geq F_i$.
- $S_{i+1} \geq \mathtt{INT}{::}S_i$.
- $h \in H_{i+1}$.

The rule for `constNULL` is similar.

Rule for `inc`) If $h \in H_i$ and $P[i] = \mathtt{inc}(x)$, then the following conditions must be satisfied.

- $0 \leq i+1 < |P|$.
- $F_i[x] = \mathtt{INT}$.
- $F_{i+1} \geq F_i$.
- $S_{i+1} \geq S_i$.
- $h \in H_{i+1}$.

Rule for `if0`) If $h \in H_i$ and $P[i] = \mathtt{if0}(L)$, then the following conditions must be satisfied.

- $0 \leq L < |P|$. $0 \leq i+1 < |P|$.
- $S_i = \mathtt{INT}{::}\vec{t}$.
- $F_L \geq F_i$. $F_{i+1} \geq F_i$.
- $S_L \geq \vec{t}$. $S_{i+1} \geq \vec{t}$.
- $h \in H_L$. $h \in H_{i+1}$.

The rule for `ifNULL` is similar.

Rule for `halt`)

There is no rule for `halt`.

3.4 Correctness of Analysis

In order to state the correctness of our analysis, we first introduce the following relation.

$$\langle v, h \rangle : t$$

v is a value and h is an invocation history. t is a type. By $\langle v, h \rangle : t$, we mean that the value v belongs to the type t provided that v appears with the invocation history h. Following is the definition of this relation.

- $\langle v, h \rangle : \top$.
- $\langle \mathtt{intval}(k), h \rangle : \mathtt{INT}$.
- $\langle \mathtt{objval}(k), h \rangle : \mathtt{OBJ}$.
- If $h[n-1] = j$, then $\langle \mathtt{retaddr}(j+1), h \rangle : \mathtt{return}(n)$.
- If $\langle v, h \rangle : F_i[x]$, then $\langle v, i{::}h \rangle : \mathtt{last}(x)$

This definition is also inductive, i.e., $\langle v, h \rangle : t$ holds if and only if it can be derived only by the above rules.

We have two lemmata.

Lemma 1: If $\langle v, h \rangle : t$ and $t' \geq t$, then $\langle v, h \rangle : t'$.

Lemma 2: Let h' be a prefix of h of length n and h'' be its corresponding suffix, i.e., $h = h'@h''$ and $|h'| = n$. If $\langle v, h \rangle : t$, then $\langle v, h'' \rangle : \mathtt{follow\_last}(n, h, t)$.

We say that the quadruple $\langle i, f, s, h \rangle$ is sound with respect to $\langle F, S, H \rangle$ and write $\langle i, f, s, h \rangle : \langle F, S, H \rangle$, if the following conditions are satisfied.

- $0 \leq i < |P|$.
- For each variable index y, $\langle f[y], h \rangle : F_i[y]$.
- For each index p for s, $\langle s[p], h \rangle : S_i[p]$.
- $h \in H_i$.
- h does not have duplication, i.e., no element of h occurs more than once in h.

We have the following correctness theorem. It says that if F, S and H follow the rule for each instruction of P, then the soundness is preserved under the transition of quadruples. This means that if the initial quadruple is sound, then quadruples that appear during execution of the virtual machine are always sound.

Theorem (correctness of analysis): Assume that F, S and H follow the rule for each instruction of P. If $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$, then $\langle i', f', s', h' \rangle : \langle F, S, H \rangle$.

The theorem is proved by the case analysis on the kind of $P[i]$. In this short paper, we only examine the case when $P[i] = \mathtt{ret}(n)$.

Assume that $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$. Since F, S and H follow the rule for ret, the following facts hold.

(i) $F_i[x] = \mathtt{return}(n)$.
(ii) $h = h_1@[j]@h_2$, where $|h_1| = n - 1$.
(iii) $0 \leq j + 1 < |P|$.
(iv) For each variable index y,
$$F_{j+1}[y] \geq \mathtt{follow\_last}(n, h, F_i[y]).$$
(v) $S_{j+1} \geq \mathtt{follow\_last}(n, h, S_i)$.
(vi) $h_2 \in H_{j+1}$.

By (i) and the soundness of $\langle i, f, s, h \rangle$, $\langle f[x], h \rangle : \mathtt{return}(n)$. Therefore, by (ii), $f[x] = \mathtt{retaddr}(j+1)$ and $i' = j + 1$. Moreover, since h does not have duplication, h_1 does not contain j. This implies that $h' = h_2$. We also have that $f' = f$ and $s' = s$.

Let us check the conditions for the soundness of $\langle i', f', s', h' \rangle = \langle j+1, f, s, h_2 \rangle$.

- By (iii), $0 \leq i' < |P|$.
- By (iv), $F_{i'}[y] \geq \mathtt{follow\_last}(n, h, F_i[y])$. By the soundness of $\langle i, f, s, h \rangle$, $\langle f[y], h \rangle : F_i[y]$. By Lemma 2, $\langle f[y], h' \rangle : \mathtt{follow\_last}(n, h, F_i[y])$. Therefore, by Lemma 1, $\langle f[y], h' \rangle : F_{i'}[y]$.
- Similarly, by (v), we have that $\langle s[p], h' \rangle : S_{i'}[p]$.
- By (vi) and since $h' = h_2$, $h' \in H_{i'}$
- Finally, since h does not have duplication, h' does not have duplication, either.

Proposition 2: If $\langle i, f, s, h \rangle : \langle F, S, H \rangle$ and $\langle i, f, s \rangle \rightarrow \langle i', f', s' \rangle$, then there exists some h' such that $\langle i, f, s, h \rangle \rightarrow \langle i', f', s', h' \rangle$.

The only case that must be examined is that of ret. Note that h' is uniquely determined.

The above proposition guarantees that if F, S and H follow the rule for each instruction and the initial quadruple $\langle i, f, s, h \rangle$ is sound, then the transition sequence starting from the triple $\langle i, f, s \rangle$ can always be lifted to a sequence starting from $\langle i, f, s, h \rangle$. This means that the semantics for triples and that for quadruples coincide when F, S and H follow the rule for each instruction.

The following final theorem guarantees type safety.

Theorem (type safety): If F, S and H follow the rule for each instruction and the initial quadruple $\langle i, f, s, h \rangle$ is sound, then a transition sequence stops only at the halt instruction.

3.5 Example

Let us abbreviate last(x) and return(n) by l(x) and r(n), respectively.

i	instruction	F_i	S_i	H_i
0	const0	$[\bot, \bot, \bot]$	[]	$\emptyset$
1	store(1)	$[\bot, \bot, \bot]$	[INT]	$\emptyset$
2	jsr(7)	$[\bot, \mathtt{INT}, \bot]$	[]	$\emptyset$
3	constNULL	$[\top, \mathtt{INT}, \mathtt{INT}]$	[]	$\emptyset$
4	store(1)	$[\top, \mathtt{INT}, \mathtt{INT}]$	[OBJ]	$\emptyset$
5	jsr(7)	$[\top, \mathtt{OBJ}, \mathtt{INT}]$	[]	$\emptyset$
6	halt	$[\top, \mathtt{OBJ}, \mathtt{OBJ}]$	[]	$\emptyset$
7	store(0)	[l(0), l(1), l(2)]	[r(1)]	{[2], [5]}
8	load(1)	[r(1), l(1), l(2)]	[]	{[2], [5]}
9	store(2)	[r(1), l(1), l(2)]	[l(1)]	{[2], [5]}
10	ret(0)	[r(1), l(1), l(1)]	[]	{[2], [5]}

4 Implementation

A dataflow analysis is usually implemented by an iterative algorithm. For each instruction, we check if the rule for the instruction is satisfied by F, S and H. If not, we update F, S and H accordingly, and check the next instruction that is affected by the update.

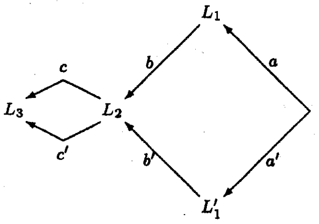
There are two problems for implementing our analysis by such an iterative algorithm. Firstly, the rule for `jsr` does not uniquely determine $F_L[y]$ when $F_i[y]$ is not `last`. We have two choices: one is to set $F_L[y] = \texttt{last}(y)$, and the other is to set $F_L[y] = F_i[y]$ (or $F_L[y] = \texttt{return}(n+1)$ if $F_i[y] = \texttt{return}(n)$). In our current implementation, we first set $F_L[y] = \texttt{last}(y)$ and proceed the analysis. If the analysis fails at some point because $F_L[y] = \texttt{last}(y)$, we take the alternative and redo the analysis from L. (We need not completely abandon the work after we set $F_L[y] = \texttt{last}(y)$.)

The second problem is that by a naïve iterative algorithm, a subroutine is analyzed each time it is called. In the worst case this may require an exponential number of steps with respect to the length of the program. This problem can be avoided by representing invocation histories by a node in the call graph of the program, which is a graph whose nodes are addresses of subroutines and whose (directed) edges are labeled with addresses of `jsr` instructions. Since JVM does not allow recursion, it is a connected acyclic graph with a unique root node representing the initial address.

A path from the root to a node in the graph corresponds to an invocation history by concatenating the labels of edges in the path. Each node in the graph then represents the set of all the invocation histories from the root to the node. Now, instead of keeping a set of invocation histories (i.e., H_i), we can keep a set of nodes in the graph.

From a program in the following (left), the call graph in the right is constructed. The node L_3 represents the set $[[c, b, a], [c, b', a'], [c', b, a], [c', b', a']]$ of invocation histories.

```
 a :  jsr(L1)
      ...
a' :  jsr(L1')
      ...
L1 :  ...
 b :  jsr(L2)
      ...
L1':  ...
b' :  jsr(L2)
      ...
L2 :  ...
 c :  jsr(L3)
      ...
c' :  jsr(L3)
      ...
L3 :  ...
```



5 Discussion

Since we introduced types of the form $\texttt{last}(x)$, we can assign types to polymorphic subroutines that move a value from a variable to another. By not simply ignoring unaccessed variables, our analysis is towards real polymorphism of subroutines in binary code.

A dataflow analysis, in general, assigns an abstract value x_i to the i-th instruction so that a certain predicate $P(x_i, \sigma)$ always holds for any state σ that reaches the i-th instruction. To this end, for any transition $\sigma \rightarrow \sigma'$, where σ' is at i', one must show that $P(x_i, \sigma)$ implies $P(x_{i'}, \sigma')$.

Since σ corresponds to $\langle i, f, s, h\rangle$ in our analysis, x seems to correspond to $\langle F_i, S_i, H_i\rangle$. However, the predicate $\langle i, f, s, h\rangle : \langle F, S, H\rangle$, which should correspond to $P(x_i, \sigma)$, does not only refer to $\langle F_i, S_i, H_i\rangle$. When $F_i[y]$ is **last**, it also refers to $F_{h[0]}$. This means that in terms of `last` types, our analysis relates values assigned to different instructions. This makes the analysis powerful while keeping the overall data structure for the analysis compact.

The representation of invocation histories by a node in the call graph is also for making the data structure small and efficient.

References

[1] George C. Necula: Proof-Carrying Code, *the Proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 1997.

[2] George C. Necula, Peter Lee: The Design and Implementation of a Certifying Compiler, submitted to *PLDI'98.*

[3] Zhenyu Qian: A Formal Specification of Java$^{\text{TM}}$ Virtual Machine Instructions, `http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html`, 1997.

[4] Raymie Stata and Martín Abadi: A Type System for Java Bytecode Subroutines, to appear in *the Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 1998.