# AN ANALYSIS ON LIST ALLOCATION ON THE STACK THROUGH ESCAPE ANALYSIS

ROSS BLASSINGAME, MICHAEL TANG

## 1. abstract

This analysis presents a simple escape analysis algorithm to determine (i) which variables outlive its scope, and subsequently, (ii) which variables that are lists, will be stack allocable, reducing list allocations on the heap. This algorithm takes inspiration from escape analysis found in Java, which determine objects that are stack allocable in methods. A *Connection Graph* will be utilized to represent connections between lists and references to lists during the analysis to determine escaping variables.

## 2. Introduction

Our Python compiler depends upon the C runtime library for current heap-allocated objects. The responsibility of memory management belongs to the C runtime to release, *free*, heap allocated- resources. In an ideal environment, heap-allocated resources should be released; otherwise, (1) memory leak will occur. (2) The latency of heap usage arise within the generated assembly code, which calls the runtime C function for object creation and subscription, *in terms of lists*. From escape analysis, we will be able to determine which variables escape its scope and which variables remain local to its scope (*non-escaping*). Once determined, variables that are lists and non-escaping can be stack allocatable. Stack allocation removes the risks in (1) & (2).

## 3. Notation

We define some mathematical notation to aid our analysis.

**Definition 3.1.** Let $v$ be a variable defined in a scope $S$. We say $v$ escapes $S$ if the lifetime of $v$ outlives $S$, denoted $Escape(v, S)$.

3.1 describes a scenario where a variable $v$, defined in some scope, is escaping when its usage is existent after the scope no longer exists.

**Definition 3.2.** Let $v$ be a variable that is defined or referenced in a scope $S$. Then $\forall v \in S$, we say that $v$ will either be in one of the three sets: $NoEscape$, $ArgumentEscape$, $GlobalEscape$ i.e. $(v \in NE \oplus v \in AE \oplus v \in GE)$ $!= (v \in NE \wedge v \in AE \wedge v \in GE)$.

3.2 states that a variable $v \in S$ can only be categorized in one of the three escaping sets. The following definitions describe these sets.

**Definition 3.3.** $v \in NE$ if for some scope $S$, $\neg Escape(v, S)$.

**Definition 3.4.** $v \in AE$ for some function's scope $S_f$ if for some function argument $a$ that is an object having attribute $x$, where $a$ belongs to function $f$, $\exists a.x = v \in S_f$. We also say that $Escape(v, S_f)$ and $Escape(a.x, S_f)$.

**Definition 3.5.** $v \in GE$ for some scope $S$ if for some $p \in OuterScopeVariable \cup ClassStaticField$, $\exists p = v \in S$. We say $Escape(v, S)$ and $Escape(p, S)$.

**Definition 3.6.** A *Connection Graph* is a directed graph where node within the graph are composed of variables $v \in NE \cup AE \cup GE$, list objects, and *return* node(s). Each node is marked with escape behavior $\tau$. Edges within the graph show assignments. We denote $v \xrightarrow{\text{P}} l$, where the $P$ notation is named a *point-to path* edge; meaning, variable $v$ is assigned to the list value $l$. We denote $v \xrightarrow{\text{D}} v_1$, where $D$ is named a *deferred edge*; meaning, a variable $v$ is assigned to variable $v_1$.

Clarifying on 3.6, a connection graph is made up of nodes that are variables of a given scope, lists, and return statements. Each node possess a marking $\tau$, which signifies its escape behavior: $NE, AE, GE$. We specify return statements as its own node in the connection graph since it serves a special purpose on figuring out what escapes by what its edge points to in the graph. All return nodes in a connection graph are initially marked to be globally escaping, and we will speak more as to why in section four. Edges are directed in a connection graph, and for our paper, we will condense them to be *point-to path* or *deferred*. A point-to path edge describes the scenario that a variable is assigned directly to a value, and on the graph it takes one-step from said variable node to the value node. A deferred edge describes the case where a variable $v$ is assigned to another variable $v_1$. It may be the case that the graph describes a chain of variable assignments, $v_i$ defined in a scope $S$ i.e. $v \xrightarrow{\text{D}} v_1 \xrightarrow{\text{D}} v_2 \xrightarrow{\text{D}} ... \xrightarrow{\text{D}} v_n \xrightarrow{\text{P}} l$ where $l$ terminates the chain since it's a list object. Or instead, $... \xrightarrow{\text{D}} v_n \xrightarrow{\text{P}} g$, where $g \in AE \cup GE$.

**Definition 3.7.** Let $l$ be a list defined within a scope $S$ and $CG$ be a *connection graph* that describes $S$. By definition 3.6, $NE \subseteq CG$. If $l \in NE$, then $l$ is stack allocable in $S$. If $l \notin NE$, then $l$ must be heap allocated.

3.7 gives a restriction on the condition on when a list may be stack allocated. If $l \in NE$, then it means that $l$ does not outlive the scope $S$. This means $l$ can be located on the stack without fear when the stack frame collapses that no references of $l$ exists outside of $S$. Otherwise, $l$ must be heap allocated if $l \notin NE$ since references may exists beyond the lifetime of the stack frame. The worry here is that accessing a stack-allocated list that escapes after the stack frame collapses leads to undefined-behavior.

## 4. Building the Connection Graph

For our analysis, we will assume that all statements within a scope are flatten. We initially let $CG = \varnothing$; then we append nodes to the graph under the following rules. If a given variable node already exists in $CG$, we replace its outgoing edges with new node's edges.

4.1. **p = [...]** A new list is created, and it's assigned to a variable $p$. We add a new node $p$ to the graph with a point-to path edge $\xrightarrow{\text{P}}$ to a new list node $l_0$.

$$\begin{tikzcd} p \arrow[r] & l_0 \end{tikzcd}$$

*figure*.1 $CG$ after $p$ is assigned to a new list $l_0$.

4.2. **q = p.** We have a variable $q$ that is assigned to the variable $p$. Building along the previous graph, the new graph will have a new node $q$ that has a deferred edge $\xrightarrow{\text{D}}$ to $p$.

$$\begin{tikzcd} q \arrow[r] & p \arrow[r] & l_0 \end{tikzcd}$$

*figure*.2 $CG$ after $q$ is assigned to $p$.

4.3. **o.x = p.** We have an object $o$ with attribute $x$ that is assigned to the variable $p$. The new graph will have two new nodes $o \xrightarrow{\text{D}} o_x$ where $o_x \xrightarrow{\text{D}} p$.

$$\begin{tikzcd} q \arrow[r] & p \arrow[r] & l_0 \\ o \arrow[r] & o_x \arrow[u] & \end{tikzcd}$$

*figure*.3 $CG$ after $o.x$ is assigned to $p$.

4.4. **s = o.x.** We have an object $o$ with attribute $x$ that variable $s$ is assigned. The new graph will have two new nodes, if $o$ does not already exist, $s \xrightarrow{\text{D}} o$ where $o \xrightarrow{\text{D}} o.x$.

$$\begin{tikzcd} q \arrow[rr] & & p \arrow[r] & l_0 \\ & s \arrow[dl] & & \\ o \arrow[rr] & & o_x \arrow[uu] & \end{tikzcd}$$

*figure*.4 $CG$ after $s$ is assigned to $o.x$.

4.5. **return q.** We have a return statement that returns the variable $q$. A return statement is a node that is marked as $GE$. $\forall n \in CG$, if there exists a path from *return* to $n$, then we also mark $n$ as $GE$. *Due to formatting of the graph size, we will abbreviate the node **return** as **r** and GE as* $\Gamma$.

$$\begin{tikzcd}
r:\Gamma \arrow[r] & q:\Gamma \arrow[r] & p:\Gamma \arrow[r] & l_0:\Gamma \\
s \arrow[r] & o \arrow[r] & o_x \arrow[u] &
\end{tikzcd}$$

*figure*.5 *CG* after $q$ is returned.

4.6. **Remarks to Connection Graph Construction.** In *figure*.5, we hypothetically described the Python program for some function assuming $o$ has already be defined elsewhere:

```
p = [1,2,3]
q = p
o.x = p
s = o.x
return q
```

We stated that a return node is marked as $GE$ and that all nodes that can be reached will also be marked as $GE$. Once marking is completed, we have determined which variables can be stack allocated. From *figure*.5, we see that $q$ must be heap allocated by definition 3.7. If we were to modify the code example above to be:

```
p = [1,2,3]
q = p
o.x = p
s = o.x
return 1.618
```

The graph will result in looking like:

$$\begin{tikzcd}
r:\Gamma & q \arrow[r] & p \arrow[r] & l_0 \\
s \arrow[r] & o \arrow[r] & o_x \arrow[u] &
\end{tikzcd}$$

*figure*.6 *CG* stack allocated list.

We notice that $r : \Gamma$ node is disjoint in the $CG$ graph. Any node that is not marked can be implied that it is non-escaping or $NE$. We can deduce that the list $[1, 2, 3]$ can be stack allocated, again by definition 3.7.