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, *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 in describing the *Connection Graph*.

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 $a \in ClassObject$ having attribute x, where a belongs to function f if $\exists a.x = v \in S_f$. 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, primitives, and *return* node(s). 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, primitives, and return statements. 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 where the graph describes the event of a chain of assignments 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.

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

4. Interprocedural Analysis