Automatic Stack/Heap Inference for Bytecode Programs

Wei-Ngan Chin[1,2], Huu Hai Nguyen[1], Corneliu Popeea[2], and Shengchao Qin[3]
[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Department of Computer Science, University of Durham

{chinwn,nguyenh2,popeeaco}@comp.nus.edu.sg, shengchao.qin@durham.ac.uk

ABSTRACT

Embedded systems are becoming more widely used but these systems are often resource constrained. Programming models for these systems should take into formal consideration resources such as stack and heap. In this paper, we show how memory usage/bounds may be inferred for bytecode programs. Our inference process captures the memory needs of each method in terms of the symbolic values of its parameters. For better precision, we infer path-sensitive information through a novel guarded expression form. Our current proposal relies on a Presburger solver to compute symbolic memory usage/bound and perform fixpoint analysis for loops/recursion. Initial experiments suggest that our prototype can handle a reasonably large class of programs. Furthermore, we provide the safety guarantee that each method never fails from insufficient memory when it is given memory space equal to (or more than) its inferred memory bounds at the start of its execution. Apart from better system reliability, our proposal can reduce memory costs for embedded devices and improve system performance with fewer runtime checks.

1. INTRODUCTION

While formal specification and functional correctness [17, 15, 16, 7, 5] have for a long time been a central focus for the software engineering community, the orthogonal issue of performance concerns on resource utilization and adequacy is gradually gaining more attention. This is especially so with the proliferation of resource-constrained mobile devices and the growing importance of high reliability systems.

In this paper, we consider the determination of adequate memory spaces for data that are dynamically used and recycled during computation. Memory subsystems for data are typically organised into two main components: stack and heap. Stack is an efficient way for using and recovering memory spaces, and is particularly important for method invocations and transient data structures. Each method invocation typically reserves a frame of memory on the stack for holding local variables and the return address to its caller. Heap is used for more complex data structures that may live past the method calls where they are created. It is typically used with a garbage collector for recovering dead spaces.

Tech. Report, September 2005

For applications running in resource-constrained environments, such as smart cards or embedded devices, it is important to be fully aware of the memory space needed by each computational unit. Previous systems have relied on past experience, profiling, or informal estimate to predict the amount of memory space required. To minimise errors due to inadequate memory, developers often over-estimate on the memory spaces that are needed. However, this translates into greater hardware cost and yet gives no guarantee of memory adequacy.

There are few previous systems for predicting the symbolic memory usage of programs, especially portable bytecode programs. Recent works [19, 20] are mostly based on analysing functional programs where the presence of immutable data structures makes such analysis easier to formalise. Even though [3, 2] are targeted at Java-based bytecode programs, their frameworks again assume that bytecode programs are compiled from first-order functional programs. Other works, such as [12, 22], merely provide a framework for checking that the memory usage of object-oriented programs conform to user-supplied memory specifications either through static analysis or runtime checking. However, user-supplied annotations may be onerous to use and are likely to be impractical for bytecode programs.

The focus on bytecode programs is important as these codes, possibly coming from untrusted sources, may cripple the host system if their resource demands are left unchecked. Furthermore, resource usage may be affected by optimising compilers which could render memory analysis, if done at the source level, unsafe to use. In this paper, we make the following major contributions:

- **Memory Bounds**: We infer *memory upper bounds* on stack and heap usage for each computation unit where possible. Our inference is formulated for imperative bytecode programs. To make this task tractable, we organise the inference mechanism as a multi-pass process where analysis from a previous pass may be embedded in code for subsequent use.

- **Path Sensitivity**: Our inference is path-sensitive as it takes into account dynamic tests from conditionals. We achieve this through a novel *guarded expression form* which can capture a more precise symbolic condition for each memory use/bound. We also provide a set of normalisation rules for the simplification of this guarded form.

- **Recursion**: A key technical challenge for memory analysis is the handling of recursion and loops. We show how conventional fixpoint techniques may be deployed. Our innovation towards *simpler fixpoint analysis* is to separate out the inference mechanism into several stages yielding: (i) input/output relation for abstract program states, (ii) net memory usage, and (iii) memory bound for high watermark. We handle loops by mapping to their tail-recursive counterparts.

- **Prototype**: We build a *prototype system* and use it to successfully infer memory bounds for a set of small benchmark programs. The prototype and experiments provide evidence on the viability of our proposal.

- **Soundness**: We formalise *safety theorems* to prove that inferred memory usage and bounds are safe to use. Our theorems proclaim that each method always executes without memory adequacy error when it is given memory space equal to (or more than) its inferred bound.

We believe that our memory inference system is useful for embedded and safety-critical software because: 1) such software often operates on platforms with limited memory, and 2) failing, because of insufficient memory, can have severe real-world consequences. The next section describes technical challenges in precisely analysing stack/heap behaviour where possible, and outlines the overall inference system. This is followed by details of the key phases of our proposed system, including: (i) frame inference, (ii) abstract state inference, (iii) stack inference, and (iv) heap inference.

2. OVERVIEW

Our goal is to develop a formal system that can precisely predict memory requirements prior to execution. Such a system must be provably sound and use only safe approximations. Several issues make this task challenging, which we shall highlight in this section. To keep our discussion at a higher level, we shall confine our examples to programs that are written in a source level imperative language. Their translation to bytecode programs and subsequent analyses are straightforward but tedious for human understanding.

2.1 Memory Usage and Bounds

Due to allocation and recovery, our system is expected to infer two key metrics for each computation unit:

- *Memory Usage*: to represent a net usage at the end of its computation

- *Memory Bound*: to represent the high watermark of memory usage at all points over its computation

While net memory usage may be a negative quantity whenever more memory is released than consumed, memory bound is always a non-negative quantity. To compute the latter, we must keep track of memory usage at every possible computation point so as to choose the maximum one as its high watermark. Consider the hypothetical example below :

```
void f(c x, c y, d z) {        //{(c,0),(d,0)}
  x = new c();                 //{(c,1),(d,0)}
  dispose(z);                  //{(c,1),(d,−1)}
  y = new c();                 //{(c,2),(d,−1)}
  dispose(x);                  //{(c,1),(d,−1)}
  dispose(y);                  //{(c,0),(d,−1)}
}
```

The `new c()` command is to create a new `c`-type object on the heap while the `dispose(x)` command is to explicitly recover the object space at reference `x`. By tracing through heap usage (as shown on the right) at each computation point, we arrive at a net heap usage of $\{(\mathtt{c},0),(\mathtt{d},-1)\}$. However, heap bound may only be inferred by taking the maximal of heap usage at all program points. In the above example, this bound is $\{(\mathtt{c},2),(\mathtt{d},0)\}$, capturing the high watermarks of the `c` and `d` object types independently.

There are several safety and technical issues relating to the use of the `dispose` command. First, we should not dispose of an object more than once. Second, we should avoid trying to dispose of a `null` reference since this does not result in memory recovery. Third, to cater to languages such as Java that are based on implicit memory management, we expect `dispose` commands to be automatically inserted whenever it is safe to do so. These issues can be resolved by the use of alias annotations that identify unique and non-null references as candidates for memory recovery. Such a technique is orthogonal to the focus of the current paper, but the interested reader may refer to [12] for a possible solution.

Stack inference may also be analysed in a similar way, except that net usage at method boundary is always zero due to perfect space recovery. However, commands that push and pop data from the stack (including parameter passing) will affect the stack's usage and must be symbolically traced for their high bounds.

2.2 Path-Sensitive Analysis

Conditional branching is essential for writing interesting programs but it affects memory analysis as precision may be lost by naive solutions. As an example, consider another hypothetical example below:

```
void f1(int n, c x, c y) {
  int v = n+1;                 //{(c,0)}
  if (v<5) {
    x = new c();               //{(c,1)}
    y = new c();               //{(c,2)}
    dispose(x);                //{(c,1)}
    dispose(y);                //{(c,0)}
  } else {
    x = new c();               //{(c,1)}
    dispose(x);                //{(c,0)}
    x =new c();                //{(c,1)}
  }}
```

In the then branch of the above code, heap usage is $\{(\mathtt{c},0)\}$ while heap bound is $\{(\mathtt{c},2)\}$. In the else branch, heap usage is $\{(\mathtt{c},1)\}$ while heap bound is $\{(\mathtt{c},1)\}$. If we combine the maximal effects of the two branches by ignoring the conditional test, we get $\{(\mathtt{c},1)\}$ and $\{(\mathtt{c},2)\}$ for heap usage and heap bound, respectively. However, this assumes the worse case scenario from the two branches. In program analysis, this phenomena is known as path insensitivity.

To provide a path-sensitive analysis, we introduce a new guarded expression of the form $\{g_i \rightarrow \mathcal{B}_i\}_{i=1}^{n}$ where g_i is a boolean expression guard and $\mathcal{B}_i$ denotes heap size in bag notation of the form $\{(c, s)^*\}$. Here, c denotes object type, and s its symbolic count. For example, $\{(c_1, 1), (c_2, 2{+}r)\}$ denotes a heap space of one object of c_1 and $2{+}r$ objects of c_2. Both these expressions shall be expressed in terms of the input parameters of its method. For our example, a guarded expression for heap usage is $\{\mathtt{n}{<}4{\rightarrow}\{(\mathtt{c}, 0)\}, \mathtt{n}{\geq}4{\rightarrow}\{(\mathtt{c}, 1)\}\}$ while that for heap bound is $\{\mathtt{n}{<}4{\rightarrow}\{(\mathtt{c}, 2)\}, \mathtt{n}{\geq}4{\rightarrow}\{(\mathtt{c}, 1)\}\}$. Note our use of the variable `n` instead of `v` as the latter is not a parameter. The restriction to input parameters is crucial to supporting interprocedural analysis.

We provide path-sensitive analysis by tracking the abstract states of boolean variables and their relations to other variables. Path sensitivity not only adds precision to memory analysis but is critical to analysing recursive methods as they are often expressed using conditionals.

2.3 Recursion and Loops

Analysing recursive methods is naturally challenging. In theory, we are to trace through every recursive call for memory usage so as to compute its high watermark as its memory bound. However, recursive calls cannot be finitely enumerated. Instead, we have to apply fixpoint analysis to determine the properties of recursive methods through safe approximations.

While the basics of fixpoint are mostly known [14], our innovation is in the formulation of key pieces of information to facilitate memory analysis in a modular fashion. Two pieces of information are crucial, namely: (i) input/output relation to compute abstract program states, and (ii) memory usage/bound across recursive calls. To derive them for each recursive method (or loop), we first infer a constraint abstraction for each method from a mutual recursive set. A constraint abstraction is essentially an abstract definition in constraint form that may be recursive. This is followed by conventional fixpoint analysis which can maintain precision via disjunctive formulae where needed. Consider:

```
int f2(int n) {
  if (n≤0) {return 1}
  else { c x = new c(); x = new c();
    int v = 2+f2(n−1);
    dispose(x); return v } }
```

We first build a constraint abstraction for the above method, as follows:

$$rec(n, r) = n{\leq}0{\wedge}r{=}1 \vee (\exists r_1{\cdot}n{>}0{\wedge}rec(n{-}1, r_1){\wedge}r{=}2{+}r_1)$$

Here, n and r denote the input parameter and the method's result, respectively. Fixpoint analysis would proceed with the first version $rec_0(n, r)$ assumed to be `false`, and with each subsequent version $rec_{i+1}(n, r)$ computed from the previous version $rec_i(n{-}1, r_1)$, as follows:

$$\begin{array}{ll}
rec_1(n, r) & = n{\leq}0{\wedge}r{=}1 \\
rec_2(n, r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_1(n{-}1, r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}(n{-}1{\leq}0{\wedge}r_1{=}1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(0{<}n{\leq}1{\wedge}r{=}2n{+}1) \\
rec_3(n, r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_2(n{-}1, r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1 \vee (0{<}n{\leq}2{\wedge}r{=}2n{+}1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(0{<}n \quad {\wedge}r{=}2n{+}1) \\
rec_4(n, r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_3(n{-}1, r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(0{<}n{\wedge}r{=}2n{+}1)
\end{array}$$

The process requires safe approximation techniques using *hulling* (which combines related disjunctive formulae into a conjunct) and *widening* (which drops certain sub-formulae) that are standard for fixpoint analysis of relational formulae (see [14]). We reach a fixpoint when $rec_{i+1}(n, r) \Longrightarrow rec_i(n, r)$. For the above example, its fixpoint gives:

$$rec(n, r) = n{\leq}0{\wedge}r{=}1 \vee n{>}0{\wedge}r{=}2n{+}1$$

The second technique is to build two abstractions to relate input parameter(s) with heap usage and heap bound, respectively. For heap usage, we can derive the following recursive formula in guarded expression form:

$$rec_{\mathcal{H}}(n) = \{n{\leq}0{\rightarrow}\{(c, 0)\}\}{\cup}\{n{>}0{\rightarrow}\{(c, 1)\}\}{+}rec_{\mathcal{H}}(n{-}1)$$

Fixpoint analysis on the above abstraction results in the following heap usage in symbolic form:

$$rec_{\mathcal{H}}(n) = \{n{\leq}0{\rightarrow}\{(c, 0)\},\ n{>}0{\rightarrow}\{(c, n)\}\}$$

There is a net allocation of one c-type object per recursive call. Hence, we have a net heap allocation of n c-type objects for input $n{>}0$. For heap bound, we can derive the following recursive formula in guarded expression form:

$$\begin{array}{ll}
rec_{\mathcal{M}}(n) = & \{n{\leq}0{\rightarrow}\{(c, 0)\}\}{\cup}\{n{>}0{\rightarrow}\{(c, 2)\}\} \\
 & \cup\ (\{n{>}0{\rightarrow}\{(c, 2)\}\}{+}rec_{\mathcal{M}}(n{-}1))
\end{array}$$

Fixpoint analysis on the above abstraction results in the following symbolic heap bound:

$$rec_{\mathcal{M}}(n) = \{n{\leq}0{\rightarrow}\{(c, 0)\},\ n{>}0{\rightarrow}\{(c, 2n)\}\}$$

The high watermark of $2n$ c-type objects is pushed up by two allocations of objects prior to each recursive call; whereas the deallocation (via `dispose`) occurs only after the return from recursion. This effect is safely and accurately captured by our analysis.

Loops can be considered a special case of tail recursion and are dealt in a similar way. In case a recursive call appears in a loop of its method, the method and its loop are considered to be mutually recursive.

2.4 A Structured Bytecode Language

We shall formalise our inference system for a small bytecode language. To keep our presentation simple, we provide a conditional statement (with two branches) and a `while` loop structure. In reality, most bytecode programs are organised as blocks of instructions and allow conditional jumps to these blocks through program labels. This block-level view does not cause any serious technical problems but may obscure our exposition. Furthermore, it is helpful to recover higher-level language constructs when directly analysing assembly-level codes, as demonstrated in [4]. For pedagogical reasons, we propose using a more structured bytecode language, and omit features relating to `jsr`-like subroutines. Our language is given in Figure 1.

$$\begin{array}{l}
P ::= M_1, \ldots, M_n \\
M ::= t\ m(t_1, .., t_n)\ l; \phi_{pr}\ \{E\} \\
E ::= I \mid E_1; E_2 \mid \mathtt{if}\ E_1\ E_2 \mid \mathtt{while}\ E \\
I ::= \mathtt{load}\langle t\rangle\ i \mid \mathtt{store}\langle t\rangle\ i \mid \mathtt{invoke}\ m \mid \mathtt{const}\langle t\rangle\ k \\
\qquad \mid \mathtt{new}\ c \mid \mathtt{dispose}\ c \\
t ::= \mathtt{bool} \mid \mathtt{int} \mid \mathtt{float} \mid \mathtt{ref} \mid \mathtt{void} \mid \cdots \\
c \in \textbf{ObjectType} \qquad \textit{(Set of Object Types)} \\
\phi \in \textbf{F} \qquad \textit{(Presburger Constraint)} \\
\quad ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists n \cdot \phi \mid \forall n \cdot \phi \\
b \in \textbf{BExp} \qquad \textit{(Boolean Expression)} \\
\quad ::= \mathtt{true} \mid \mathtt{false} \mid s_1{=}s_2 \mid s_1{<}s_2 \mid s_1{\leq}s_2 \\
s \in \textbf{AExp} \qquad \textit{(Arithmetic Expression)} \\
\quad ::= k^{\mathtt{int}} \mid \pi_i \mid k^{\mathtt{int}} * s \mid s_1{+}s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) \\
\textit{where } k^{\mathtt{int}} \textit{ is an integer constant; } \pi_i \textit{ is a size variable}
\end{array}$$

Figure 1: Syntax for a Bytecode Language

For each method, we expect types of parameters and the result to be declared. Here, ϕ_{pr} denotes a user-supplied precondition for input parameters (denoted by $\pi_1, .., \pi_n$) in Presburger form. This precondition may help us derive a stronger postcondition but it is entirely optional. Furthermore, l denotes the number of local variables (including parameters). The stack frame of each method call is organised as shown in Figure 2. The parameters are assumed to occupy the first n local variables starting at position 1. This is followed by the other $l-n$ local variables before the operand stack of the frame. In addition, each frame also contains a previous frame pointer and its caller's return address.

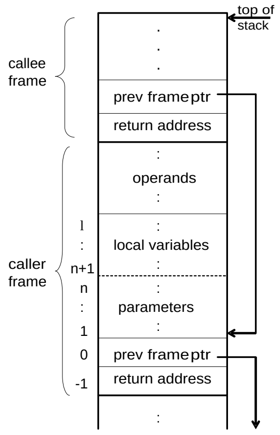

Figure 2: Stack Frames

The `load`$\langle t\rangle$ i instruction is intended to transfer a value of t-type in the local variable at position i to the top of the (operand) stack. The `const`$\langle t\rangle$ k instruction places a constant value on the top of the stack. The `store`$\langle t\rangle$ i moves a value from the top of the stack to a specified local variable. The `invoke` m instruction calls a method m with its arguments on the stack. On return from the callee, the arguments are removed and a result is placed on the stack.

The `new` c command allocates an object of the c-type in the heap with its new reference placed on the stack while the `dispose` c command recovers space for a c-type object from the heap whose reference is on top of the stack. While we support objects in our language, we shall omit details on how fields are declared and accessed as they can be supported through primitive methods. Size properties such as length of linked nodes and values of components may also be tracked in our analysis. However, due to mutability and sharing, an alias annotation scheme is needed to identify unique references and/or read-only fields for more precise state tracking. Such a mechanism was proposed in [11] in the context of a type-checking system for enforcing safety protocols. It can be adapted for use in memory inference. Section 8 describes how this extension may be achieved.

2.5 Multi-Pass Inference

We present our inference as a modular multi-pass system. Breaking a complex inference system into smaller phases has helped to simplify our formalisation considerably. The first phase is to build a call dependency graph to group each set of mutual recursive methods for simultaneous inference. After that, we determine stack/heap bounds through four main stages, namely:

- frame bound inference
- abstract state inference
- stack inference
- heap inference

The target of our inference system is a set of annotations for each method declaration. Given a method:

$$t\ m(t_1, .., t_n)\ l; \phi_{pr}\ \{\ldots\}$$

Our system infers the following extended declaration for each method processed:

$$t\,m(t_1, .., t_n)\ l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}; \mathcal{H}_{po}; \mathcal{M}_{po}\,\{\ldots\}$$

where F is its frame bound, ϕ_{po} its post-state, $\mathcal{S}$ its stack bound, H_{po} its net heap usage, and $\mathcal{M}_{po}$ its heap bound.

3. FRAME BOUND INFERENCE

For our bytecode language, each method call is expected to place parameters, local variables and an operand stack into its own stack frame. This frame has a bounded size that can be inferred. We propose inferring the size of each stack frame using rules of the following form:

$$l, \Gamma \vdash_F E \rightsquigarrow A, \Gamma_1, \mathcal{F}$$

where l indicates size of the local variables area in the frame and Γ (resp. Γ_1) captures the types of elements in the current frame before (resp. after) the execution of E. For example, $\Gamma = [t_1, \ldots, t_n]$ denotes there are n elements on the stack, where the element at the top is of type t_1, and the element at the bottom is of type t_n. $\mathcal{F}$ denotes the (high watermark of) stack frame size inferred so far for E.

While such frame bound validation is common in bytecode compilers/verifiers, we re-cast them in our framework to facilitate subsequent more sophisticated abstract state and stack inference mechanisms. One novelty we introduce is to embed the current *top frame pointer* at each program point into an intermediate code A. For each code fragment E with stack frame Γ, we embed its current *top frame pointer* $p{=}|\Gamma|$ into an intermediate code as (p, E_A), and recursively for E_A. The expression A is defined inductively as follows:

$$\begin{array}{lcl} A & ::= & (p,\ E_A) \\ E_A & ::= & I \mid A; A \mid \texttt{if}\ A\ A \mid \texttt{while}\ A \end{array}$$

A set of syntax-directed rules for frame bound inference are listed in Figure 3. These rules can be easily converted into an inference algorithm. Apart from frame bound inference, we also perform some checks to ensure that there is no underflow of the operand stack and there is simple type safety. Also, each frame of a method call is not affected by the operations of its callees as the latter have their own stack frames. Thus, frame bound inference is intra-procedural in nature, and there is no need to apply fixpoint analysis for recursive methods. To account for the presence of the return address and a pointer to the previous stack frame, we add 2 to the inferred frame bound in [FS−METH]. Furthermore, most types at bytecode level occupy a word per value, except for `void` which takes no space, and `long` and `double` which take two words per value. For ease of presentation, we shall assume that each type (including `void` and return address) takes a word per value on the stack frame. Our implementation computes the actual size for each type.

Notation-wise, we use $E :: t$ to denote that E is of type t. Given $\Gamma = [t_1, .., t_n]$, the notation $t : \Gamma$ inserts type t to the head of Γ, yielding $[t, t_1, .., t_n]$. We use $+$ to concatenate two sequences. For example, $[t_1, t_2]{+}[t_3, .., t_n] = [t_1, .., t_n]$. $|\Gamma|$ represents the number of elements in Γ, i.e., n. $\Gamma[i]$ retrieves its ith element, i.e., t_i. $\Gamma{\oplus}(i{\mapsto}t)$ returns a sequence

$$\boxed{\text{FS-CONST}}\quad \frac{k{::}t \quad \Gamma_1{=}t{:}\Gamma}{l,\Gamma \vdash_F \texttt{const}\langle t\rangle\ k \rightsquigarrow (|\Gamma|, \texttt{const}\langle t\rangle\ k), \Gamma_1, |\Gamma_1|}$$

$$\boxed{\text{FS-LOAD}}\quad \frac{i{\le}l \quad \Gamma[i]{=}t \quad \Gamma_1{=}t{:}\Gamma}{l,\Gamma \vdash_F \texttt{load}\langle t\rangle\ i \rightsquigarrow (|\Gamma|, \texttt{load}\langle t\rangle\ i), \Gamma_1, |\Gamma_1|}$$

$$\boxed{\text{FS-STORE}}\quad \frac{i{\le}l{\le}|\Gamma| \qquad \Gamma_1{=}\Gamma{\oplus}(i \mapsto t) \qquad r{=}|\Gamma|{+}1}{l, t{:}\Gamma \vdash_F \texttt{store}\langle t\rangle\ i \rightsquigarrow (r, \texttt{store}\langle t\rangle\ i), \Gamma_1, r}$$

$$\boxed{\text{FS-DISPOSE}}\quad \frac{l{\le}|\Gamma| \qquad r{=}|\Gamma|{+}1}{l, \texttt{ref}{:}\Gamma \vdash_F \texttt{dispose}\ c \rightsquigarrow (r, \texttt{dispose}\ c), \Gamma, r}$$

$$\boxed{\text{FS-NEW}}\quad \frac{\Gamma_1{=}\texttt{ref}{:}\Gamma}{l,\Gamma \vdash_F \texttt{new}\ c \rightsquigarrow (|\Gamma|, \texttt{new}\ c), \Gamma_1, |\Gamma_1|}$$

$$\boxed{\text{FS-SEQ}}\quad \frac{l,\Gamma \vdash_F E_1 \rightsquigarrow A_1, \Gamma_1, \mathcal{F}_1 \quad l,\Gamma_1 \vdash_F E_2 \rightsquigarrow A_2, \Gamma_2, \mathcal{F}_2}{l,\Gamma \vdash_F E_1; E_2 \rightsquigarrow (|\Gamma|, A_1; A_2), \Gamma_2, \mathit{max}(\mathcal{F}_1, \mathcal{F}_2)}$$

$$\boxed{\text{FS-IF}}\quad \frac{\begin{array}{c} l,\Gamma \vdash_F E_1 \rightsquigarrow A_1, \Gamma_1, \mathcal{F}_1 \quad l{\le}|\Gamma| \quad |\Gamma_1|{=}|\Gamma_2| \\ l,\Gamma \vdash_F E_2 \rightsquigarrow A_2, \Gamma_2, \mathcal{F}_2 \quad \mathcal{F}_3 = \mathit{max}(\mathcal{F}_1, \mathcal{F}_2) \quad \Gamma_3{=}\Gamma_1{\sqcup}\Gamma_2 \end{array}}{l, \texttt{bool}{:}\Gamma \vdash_F \texttt{if}\ E_1\ E_2 \rightsquigarrow (|\Gamma|{+}1, \texttt{if}\ A_1\ A_2), \Gamma_3, \mathcal{F}_3}$$

$$\boxed{\text{FS-INVOKE}}\quad \frac{\begin{array}{c} t\, m(t_1, .., t_n) \cdots \{\cdots\} \in P \\ \Gamma = [t_n, .., t_1]{+}\Gamma_1 \quad l{\le}|\Gamma_1| \quad \Gamma_2{=}t{:}\Gamma_1 \end{array}}{l,\Gamma \vdash_F \texttt{invoke}\ m \rightsquigarrow (|\Gamma|, \texttt{invoke}\ m), \Gamma_2, |\Gamma_2|}$$

$$\boxed{\text{FS-WHILE}}\quad \frac{l{\le}|\Gamma| \quad l,\Gamma \vdash_F E \rightsquigarrow A, \texttt{bool}{:}\Gamma, \mathcal{F}}{l, \texttt{bool}{:}\Gamma \vdash_F \texttt{while}\ E \rightsquigarrow (|\Gamma|{+}1, \texttt{while}\ A), \Gamma, \mathcal{F}}$$

$$\boxed{\text{FS-METH}}\quad \frac{l, [\top]_{i=n+1}^{l}{+}[t_n, \ldots, t_1] \vdash_F E \rightsquigarrow A, t{:}\Gamma, \mathcal{F} \qquad |\Gamma|{=}l}{\vdash_F \ \ t\, m(t_1, .., t_n)\, l; \phi_{pr}\, \{E\} \rightsquigarrow t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}{+}2\, \{A\}}$$

Figure 3: Frame Bound Inference

similar to Γ but with its ith element replaced by t, i.e., $[t_1, .., t_{i-1}, t, t_{i+1}, .., t_n]$. The function $\mathit{max}(n_1, n_2)$ returns the maximum of n_1 and n_2, while function $\Gamma_1{\sqcup}\Gamma_2$ computes the least upper bound of types over sequences of the same length.

4. ABSTRACT STATE INFERENCE

The second stage of our analyser attempts to infer an abstract program state at every program point. Each abstract state Δ is expressed as a Presburger formula over values on the stack $[\pi_p, ..., \pi_1]$. Following the primed notation advocated in [18] to capture state change, we use π_i to denote the original value of the stack at location i and π_i' to denote the latest value at the same location. Our analyser employs syntax-directed rules of the following form:

$$\Delta \vdash_A A \rightsquigarrow B, \Delta_1$$

where Δ (resp. Δ_1) represents the abstract state before (resp. after) the evaluation of A. Note that the input A is an expression previously annotated with top frame pointers. The output expression B is obtained from A by inserting the corresponding abstract state into each program point. It can be inductively defined as follows:

$$\begin{array}{lll} B & ::= & (p,\ \Delta,\ E_B) \\ E_B & ::= & I \ \mid\ B; B \ \mid\ \texttt{if}\ B\ B \ \mid\ \texttt{while}\ B\ \Delta_1 \end{array}$$

We also attach a post-state for the body of each `while` loop as it is needed for various fixpoint analyses. Let us examine how abstract program state is inferred by our rules. The $\texttt{const}\langle t\rangle\ k$ instruction is analysed as follows:

$$\boxed{\text{AS-CONST}}\quad \frac{\Delta_1 = \Delta{\wedge}\mathit{eq}_t(\pi'_{p+1}, k)}{\Delta \vdash_A (p, \texttt{const}\langle t\rangle\ k) \rightsquigarrow (p, \Delta, \texttt{const}\langle t\rangle\ k), \Delta_1}$$

A new k value of type t is placed on top of the stack at location $p{+}1$. Our rule adds $\mathit{eq}_t(\pi'_{p+1}, k)$ to the post-state to mirror this effect. As abstract state is based on integer domain, the eq_t relation converts boolean constants to integers and ignores other non-integer types:

$$\begin{array}{lll} \mathit{eq}_{\texttt{bool}}(v, \texttt{true}) & =_{df} & (v = 1) \\ \mathit{eq}_{\texttt{bool}}(v, \texttt{false}) & =_{df} & (v = 0) \\ \mathit{eq}_{\texttt{int}}(v, k) & =_{df} & (v = k) \\ \mathit{eq}_{\texttt{t}}(v, k) & =_{df} & \mathit{true},\ \texttt{IF}\ t = \texttt{float} \mid \texttt{void} \mid \texttt{ref} \end{array}$$

The rule for $\texttt{store}\langle t\rangle\ i$ instruction is highlighted next:

$$\boxed{\text{AS-STORE}}\quad \frac{\Delta_1 = \Delta \circ_{\{\pi_i\}} \pi_i'{=}\pi_p'}{\Delta \vdash_A (p, \texttt{store}\langle t\rangle\ i) \rightsquigarrow (p, \Delta, \texttt{store}\langle t\rangle\ i), \exists \pi_p' \cdot \Delta_1}$$

The current value on top of stack π_p' is copied into location i. To capture state change at this location, we compose the abstract state Δ with the change $\pi_i'{=}\pi_p'$ as follows: $\Delta \circ_{\{\pi_i\}} \pi_i'{=}\pi_p'$. Given an existing state Δ and a change ϕ whereby $X = \{x_1, \ldots, x_n\}$ denotes the set of variables to be updated, we can define the composition, $\circ_X$, as follows:

$$\begin{array}{l} \Delta \circ_X \phi =_{df} \exists\ r_1..r_n \cdot \rho_2\, \Delta \wedge \rho_1\, \phi \\ \text{where}\ \ r_1, \ldots, r_n \text{ are fresh variables} \\ \qquad\qquad \rho_1 = [x_i \mapsto r_i]_{i=1}^{n}\ ;\ \rho_2 = [x_i' \mapsto r_i]_{i=1}^{n} \end{array}$$

Note that ρ_1 and ρ_2 are substitutions. Later, we may use $\rho_1{\cup}\rho_2$ to combine two substitutions with disjoint domains.

As an example, if the current state is captured using $\pi_1'{=}\pi_1{\wedge}\pi_2'{=}\pi_1{+}2$, then its update by $\pi_1'{=}\pi_2'$ is computed as shown:

$$\begin{array}{l} (\pi_1'{=}\pi_1{\wedge}\pi_2'{=}\pi_1{+}2) \circ_{\{\pi_1\}} \pi_1'{=}\pi_2' \\ \qquad \equiv \exists r{\cdot}r{=}\pi_1{\wedge}\pi_2'{=}\pi_1{+}2{\wedge}\pi_1'{=}\pi_2' \\ \qquad \equiv \pi_2'{=}\pi_1{+}2{\wedge}\pi_1'{=}\pi_2' \end{array}$$

Furthermore, as a value on the stack is being popped out, we shall existentially quantify it using $\exists \pi_p' \cdot \Delta_1$. For the

$$\frac{\boxed{\textbf{AS-LOAD}} \quad \Delta_1 = \Delta \wedge \pi'_{p+1}{=}\pi'_i}{\Delta \vdash_A (p, \texttt{load}\langle t\rangle\ i) \rightsquigarrow (p, \Delta, \texttt{load}\langle t\rangle\ i), \Delta_1}
\qquad
\frac{\boxed{\textbf{AS-NEW-DISPOSE}} \quad I = \texttt{new}\ c \mid \texttt{dispose}\ c}{\Delta \vdash_A (p, I) \rightsquigarrow (p, \Delta, I), \Delta}$$

$$\frac{\boxed{\textbf{AS-SEQ}} \quad \Delta \vdash_A A_1 \rightsquigarrow B_1, \Delta_1 \quad \Delta_1 \vdash_A A_2 \rightsquigarrow B_2, \Delta_2}{\Delta \vdash_A (p, A_1; A_2) \rightsquigarrow (p, \Delta, B_1; B_2), \Delta_2}$$

$$\frac{\begin{array}{c}\boxed{\textbf{AS-INVOKE}} \\ t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}; \phi_{po} \{\cdots\} {\in} P \quad \textit{fresh } r \\ \rho = [\pi_i {\mapsto} \pi'_{p-n+i}]_{i=1}^n \cup [\pi'_{l+1} {\mapsto} r] \quad \Delta \Longrightarrow \rho\phi_{pr} \\ \Delta_1 \ = \ (\exists \pi'_{p-n+1} .. \pi'_p {\cdot} \Delta \wedge \rho\phi_{po}) \wedge (\pi'_{p-n+1}{=}r)\end{array}}{\Delta \vdash_A (p, \texttt{invoke}\ m) \rightsquigarrow (p, \Delta, \texttt{invoke}\ m), \exists r {\cdot} \Delta_1}$$

$$\frac{\begin{array}{c}\boxed{\textbf{AS-METH}} \\ \Delta \ = \ \phi_{pr} \wedge \bigwedge_{i=1}^n \pi'_i{=}\pi_i \quad \Delta \vdash_A A \rightsquigarrow B, \Delta_1 \\ \phi_{rec} = \{m(\pi_1, .., \pi_n, \pi'_{l+1}){=}\Delta_1\} \quad \phi_{po} = \textit{fixpt}(\phi_{rec})\end{array}}{\vdash_A \quad t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}\, \{A\} \rightsquigarrow t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}; \phi_{po}\, \{B\}}$$

Figure 4: Abstract State Inference

above example, this leads to:

$\exists\ \pi'_2 {\cdot} \pi'_2 {=} \pi_1 {+} 2 \wedge \pi'_1 {=} \pi'_2$
$\equiv \pi'_1 {=} \pi_1 {+} 2$

For the conditional construct, we build path sensitivity into our rules by adding $\pi'_p{=}1$ and $\pi'_p{=}0$ to the abstract states of the true and false branches, respectively:

$$\frac{\begin{array}{c}\boxed{\textbf{AS-IF}} \\ \exists \pi'_p {\cdot} (\Delta \wedge \pi'_p{=}1) \vdash_A A_1 \rightsquigarrow B_1, \Delta_1 \\ \exists \pi'_p {\cdot} (\Delta \wedge \pi'_p{=}0) \vdash_A A_2 \rightsquigarrow B_2, \Delta_2\end{array}}{\Delta \vdash_A (p, \texttt{if}\ A_1\ A_2) \rightsquigarrow (p, \Delta, \texttt{if}\ B_1\ B_2), \Delta_1 \vee \Delta_2}$$

A new post-state $\Delta_1 \vee \Delta_2$ is obtained via a disjunction from outcomes of the two branches.

The remaining rules for abstract state inference are listed in Figure 4. For the `invoke` m instruction, the postcondition of the callee is added to the current abstract state. We also check to ensure that the precondition of the callee is met using $\Delta \implies \rho\phi_{pr}$.

For both `while` loop and method declaration, we first build a constraint abstraction before applying fixpoint analysis, if needed, to approximate the effect of recursion. The rule for the loop construct [AS-WHILE] is more complex than that for method declaration [AS-METH] despite the fact that it can be viewed as a special case of tail recursion. Two features make a loop special: (i) all variables in scope may be regarded as parameters to the loop body, and (ii) these variables must be considered to be passed by reference since their effects are visible outside of the loop.

Given a loop $(p, \texttt{while}\ A)$, the variables $\pi_1, .., \pi_{p-1}$ are in scope and π_p is the boolean test. A corresponding tail-recursive counterpart to this loop may be written as:

$\alpha(\pi_1, .., \pi_{p-1}){=}A\ ;\ \texttt{if}\ \alpha(\pi_1, .., \pi_{p-1})\ \texttt{nop}$

where `nop` denotes a skip command. As we have to model the parameters through a pass by reference mechanism, we shall use a constraint abstraction $\alpha(\pi_1, .., \pi_{p-1}, r_1, .., r_{p-1})$ with $r_1, .., r_{p-1}$ to denote the outputs for the input parameters $\pi_1, .., \pi_{p-1}$. This is captured by the abstraction ϕ_{rec} that is built from Δ_1 (which captures the poststate of A) and Δ_a (which captures the effect of conditional prior to termination or loop) in the rule below:

$$\frac{\begin{array}{c}\boxed{\textbf{AS-WHILE}} \\ \bigwedge_{i=1}^{p-1} \pi'_i{=}\pi_i \vdash_A A \rightsquigarrow B, \Delta_1 \quad \rho{=}[r_i {\mapsto} \pi'_i]_{i=1}^{p-1} \quad \textit{fresh}\ r_1, .., r_{p-1} \\ \Delta_a{=}\pi'_p{=}0 \wedge \bigwedge_{i=1}^{p-1} r_i{=}\pi'_i \vee \pi'_p{=}1 \wedge \alpha(\pi'_1, .., \pi'_{p-1}, r_1, .., r_{p-1}) \\ \phi_{rec}{=}\{\alpha(\pi_1, .., \pi_{p-1}, r_1, .., r_{p-1}){=}\Delta_1 \wedge \Delta_a\} \quad \Delta_{post}{=}\textit{fixpt}(\phi_{rec}) \\ \Delta_2{=}(\exists \pi'_p \cdot \Delta \wedge \pi'_p{=}0) \vee ((\exists \pi'_p \cdot \Delta \wedge \pi'_p{=}1) \circ_{\{\pi_1 .. \pi_{p-1}\}} \rho\Delta_{post})\end{array}}{\Delta \vdash_A (p, \texttt{while}\ A) \rightsquigarrow (p, \Delta, \texttt{while}\ B\ \Delta_1), \Delta_2}$$

Applying fixpoint analysis to the recursive abstraction gives a postcondition for executing this loop. The abstract state $(\exists \pi'_p \cdot \Delta \wedge \pi'_p{=}0)$ is to account for the scenario in which the loop is never executed.

5. STACK INFERENCE

The main rationale behind a separate frame inference stage is to limit the effects of primitive operations, such as `load`$\langle t\rangle$ and `store`$\langle t\rangle$, to the caller's local frame area. To support interprocedural analysis, we must also analyse how method invocations affect the global stack. One special feature of the stack is that it has perfect recovery of space at the method call boundary. This means that there is always zero net stack usage for each method call. Consequently, we only need to infer stack bound (and not stack usage) for each method declaration. We achieve this through a set of inference rules of the form:

$$a \vdash_S B \rightsquigarrow \mathcal{S}$$

where a is the arity of the current method, and B is the expression with top frame pointers and abstract states inserted by prior analyses. The inferred result $\mathcal{S}$ denotes the high watermark of stack usage encountered during (i.e. from start to end of) the execution of B. $\mathcal{S}$ contains path-sensitive information for stack space. It is captured by the guarded form $\{g{\rightarrow}s\}^*$ where g is a predicate and $s{\in}\textbf{AExp}$ (from Figure 1) denotes the stack space when g is true.

The most interesting rule for stack bound inference is that for method invocation, as shown below:

$$\frac{\begin{array}{c}\boxed{\textbf{SS-INVOKE}} \\ t\, m_1(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}\, \{B\} \in P \qquad r{=}p{-}n{+}2 \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^n \qquad \mathcal{S}_1 \ = \ \textit{enrich}(a, \Delta, \rho\mathcal{S}){+}r\end{array}}{a \vdash_S (p, \Delta, \texttt{invoke}\ m_1) \rightsquigarrow \mathcal{S}_1}$$

Note that ρ captures the argument substitution process. We use a special function $\textit{enrich}(a, \Delta, \mathcal{S})$ to incorporate path sensitivity through the current abstract state. Specifically, we strengthen the guarded expression from $\mathcal{S}$ as follows:

$$\textit{enrich}(a, \Delta, \mathcal{S}) =_{df} \{\exists \pi'_{a+1} \ldots {\cdot} \Delta \wedge g {\rightarrow} s \ \mid \ (g{\rightarrow}s) \in \mathcal{S}\}$$

Here, a is the arity of the current method. The existential quantification $\exists \pi'_{a+1} \ldots$ removes all variables other than $\pi_1, .., \pi_a$ from the guarded formulae. For each method invocation, we have a choice of either building the next frame on top of the current frame or immediately above a frame pointer at $p{-}n{+}2$, after the removal of n arguments. In the above rule, we assume that our abstract machine uses

$$\boxed{\textbf{SS-INSTR}}\quad \frac{I = \texttt{const}\langle t\rangle\ k \mid \texttt{load}\langle t\rangle\ i \mid \texttt{store}\langle t\rangle\ i \mid \cdots}{a \vdash_S (p, \Delta, I) \leadsto \{\}}$$

$$\boxed{\textbf{SS-SEQ}}\quad \frac{a \vdash_S B_1 \leadsto \mathcal{S}_1 \quad a \vdash_S B_2 \leadsto \mathcal{S}_2}{a \vdash_S (p, \Delta, B_1; B_2) \leadsto \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$\boxed{\textbf{SS-IF}}\quad \frac{a \vdash_S B_1 \leadsto \mathcal{S}_1 \quad a \vdash_S B_2 \leadsto \mathcal{S}_2}{a \vdash_S (p, \Delta, \texttt{if}\ B_1\ B_2) \leadsto \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$\boxed{\textbf{SS-WHILE}}\quad \frac{a \vdash_S B \leadsto \mathcal{S} \quad \mathcal{S}_{rec} = \{\alpha(\pi_1, .., \pi_{p-1}) = \mathcal{S} \cup \mathit{enrich}(p-1, \Delta_1 \wedge \pi'_p = 1, \alpha(\pi'_1, .., \pi'_{p-1}))\}}{a \vdash_S (p, \Delta, \texttt{while}\ B\ \Delta_1) \leadsto \mathit{enrich}(a, \Delta \wedge \pi'_p = 1, \mathit{fixpt}(\mathcal{S}_{rec}))}$$

$$\boxed{\textbf{SS-METH}}\quad \frac{n \vdash_S B \leadsto \mathcal{S} \quad \mathcal{S}_{rec} = \{m(\pi_1, .., \pi_n) = \mathcal{S} \cup \{\phi_{pr} \rightarrow \mathcal{F}\}\}}{\vdash_S\ \ t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}; \phi_{po}\, \{B\} \leadsto t\, m(t_1, .., t_n)\, l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathit{fixpt}(\mathcal{S}_{rec})\, \{B\}}$$

Figure 5: Stack Bound Inference

the second convention as this can give a lower stack bound. Also, an expression s without its guard is an abbreviation for $\{\texttt{true} \rightarrow s\}$. For example, $\mathit{enrich}(a, \Delta, \rho\mathcal{S}) + r$ is a shorthand for $\mathit{enrich}(a, \Delta, \rho\mathcal{S}) + \{\texttt{true} \rightarrow r\}$.

Furthermore, we have the option of mirroring the effect of tail-call optimisation. Obviously, this depends on whether the particular abstract machine supports it. Assuming it does, we can mark each tail call identified with a special $\texttt{invoke}_{Tail}$ instruction. For each such invocation, we can build the next stack frame by overwriting the current one. Its effect on the stack can be captured by the rule below:

$$\boxed{\textbf{SS-INVOKE-TAIL}}\quad \frac{\begin{array}{c} t\, m_1(t_1, .., t_n)\, \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}\, \{\cdots\} \in P \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \qquad \mathcal{S}_1 = \mathit{enrich}(a, \Delta, \rho\mathcal{S}) \end{array}}{a \vdash_S (p, \Delta, \texttt{invoke}_{Tail}\ m_1) \leadsto \mathcal{S}_1}$$

The rest of the stack inference rules are listed in Figure 5. The guarded formulae used in our rules are built from two operators, namely $\cup$ (for upper bound) and $+$ (for summation). Both these operators are associative and commutative with $+$ distributing over $\cup$. The guarded formulae can be simplified by the following set of normalisation rules:

$$\begin{array}{ll}
\{\texttt{false} \rightarrow s\} & \Rrightarrow \{\} \\
\{p_1 \rightarrow s\} \cup \{p_2 \rightarrow s\} & \Rrightarrow \{p_1 \vee p_2 \rightarrow s\} \\
\{p_1 \rightarrow s_1\} \cup \{p_2 \rightarrow s_2\} & \Rrightarrow \{p_1 \wedge p_2 \rightarrow max(s_1, s_2)\} \\
 & \quad \cup \{p_1 \wedge \neg p_2 \rightarrow s_1\} \cup \{\neg p_1 \wedge p_2 \rightarrow s_2\} \\
\{p_1 \rightarrow s_1\} + \{p_2 \rightarrow s_2\} & \Rrightarrow \{p_1 \wedge p_2 \rightarrow s_1 + s_2\} \\
(G_1 \cup G_2) + G_3 & \Rrightarrow (G_1 + G_3) \cup (G_2 + G_3)
\end{array}$$

The first three $\cup$ rules are applied to each set of guarded formulae until all guards are disjoint from each other. The last two rules show how $+$ can be simplified, and how $+$ distributes over $\cup$. The third rule may lead to an explosion in the number of cases but these cases may be reduced with the help of the first two rules. Furthermore, we may heuristically apply the approximation rule below where desired.

$$\{p_1 \rightarrow s_1\} \cup \{p_2 \rightarrow s_2\} \Rrightarrow \{p_1 \vee p_2 \rightarrow max(s_1, s_2)\}$$

As an example, consider the guarded formula below:
$\{0 \leq n \leq 5 \rightarrow 10\} \cup \{3 \leq n \leq 9 \rightarrow 20\} \cup \{n < 3 \vee n > 9 \rightarrow 5\}$
The first two guards overlap. Applying the third rule, followed by the second rule gives:

$$\begin{array}{ll}
\Rrightarrow & \{0 \leq n < 3 \rightarrow 10\} \cup \{3 \leq n \leq 5 \rightarrow max(10, 20)\} \\
 & \cup \{5 < n \leq 9 \rightarrow 20\} \cup \{n < 3 \vee n > 9 \rightarrow 5\} \\
\Rrightarrow & \{0 \leq n < 3 \rightarrow 10\} \cup \{3 \leq n \leq 9 \rightarrow 20\} \cup \{n < 3 \vee n > 9 \rightarrow 5\}
\end{array}$$

The first and third guard now overlaps. Applying the third rule, followed by the second rule gives:

$$\Rrightarrow \quad \{0 \leq n < 3 \rightarrow 10\} \cup \{3 \leq n \leq 9 \rightarrow 20\} \cup \{n < 0 \vee n > 9 \rightarrow 5\}$$

The final normalised form is a guarded expression with disjoint predicates and captures memory usage/bound after safe approximation.

6. HEAP INFERENCE

We organise heap inference as a set of syntax-directed rules of the form:

$$a, \mathcal{H} \vdash_H B \leadsto \mathcal{H}_1, \mathcal{M}$$

As before, a is the arity of the current method. $\mathcal{H}$ (resp. $\mathcal{H}_1$) denotes the heap effect before (resp. after) the execution of the (annotated) expression B while $\mathcal{M}$ indicates the high watermark of heap usage during the execution of B. For heap space specification, the guarded formulae is of the form $\{g_1 \rightarrow \mathcal{B}_1,\ g_2 \rightarrow \mathcal{B}_2, \ldots\}$, where each g_i is a predicate, and each $\mathcal{B}_i$ denotes heap size in bag notation.

While we have formulated heap inference as a single set of rules, it is really computing two pieces of information, namely: (i) heap usage, and (ii) heap bound. Furthermore, the latter depends on the former. Our implementation therefore organises this inference stage as two separate tasks whereby heap usage is computed before heap bound. This is mandatory when handling recursive methods as fixpoint analysis for heap usage must be computed before the analysis of heap bound. Furthermore, we have to track heap usage (but not heap bound) in a flow-sensitive manner by passing the heap usage from a prior computation to the next one. This is best illustrated in the following rule:

$$\boxed{\textbf{HS-SEQ}}\quad \frac{a, \mathcal{H} \vdash_H B_1 \leadsto \mathcal{H}_1, \mathcal{M}_1 \quad a, \mathcal{H}_1 \vdash_H B_2 \leadsto \mathcal{H}_2, \mathcal{M}_2}{a, \mathcal{H} \vdash_H (p, \Delta, B_1; B_2) \leadsto \mathcal{H}_2, \mathcal{M}_1 \cup \mathcal{M}_2}$$

Ultimately, heap usage is directly caused by two primitive heap instructions, namely `new` and `dispose`. Their effects are captured by the following rules:

$$\boxed{\textbf{HS-NEW}}\quad \frac{\mathcal{H}_1 = \mathcal{H} + \mathit{enrich}(a, \Delta, \{(c, 1)\})}{a, \mathcal{H} \vdash_H (p, \Delta, \texttt{new}\ c) \leadsto \mathcal{H}_1, \mathcal{H}_1}$$

$$\boxed{\textbf{HS-DISPOSE}}\quad \frac{\mathcal{H}_1 = \mathcal{H} + \mathit{enrich}(a, \Delta, \{(c, -1)\})}{a, \mathcal{H} \vdash_H (p, \Delta, \texttt{dispose}\ c) \leadsto \mathcal{H}_1, \mathcal{H}}$$

$$\boxed{\textbf{HS-PRIM}}\quad \frac{I = \texttt{const}\langle t\rangle\ k \mid \texttt{load}\langle t\rangle\ i \mid \texttt{store}\langle t\rangle\ i}{a, \mathcal{H} \vdash_H (p, \Delta, I) \rightsquigarrow \mathcal{H}, \mathcal{H}}$$

$$\boxed{\textbf{HS-IF}}\quad \frac{a, \mathcal{H} \vdash_H B_1 \rightsquigarrow \mathcal{H}_1, \mathcal{M}_1 \quad a, \mathcal{H} \vdash_H B_2 \rightsquigarrow \mathcal{H}_2, \mathcal{M}_2}{a, \mathcal{H} \vdash_H (p, \Delta, \texttt{if}\ B_1\ B_2) \rightsquigarrow \mathcal{H}_1 \cup \mathcal{H}_2, \mathcal{M}_1 \cup \mathcal{M}_2}$$

$$\boxed{\textbf{HS-WHILE}}\quad \frac{\begin{array}{c} p{-}1, \mathcal{H} \vdash_H B \rightsquigarrow \mathcal{H}_1, \mathcal{M}_1 \qquad \mathcal{M}_{rec} {=} \{\alpha(\pi_1, .., \pi_{p-1}) {=} \mathcal{M}_1 \cup enrich(p{-}1, \Delta_1 \wedge \pi'_p{=}1, \mathcal{H}_1 {+} \alpha(\pi'_1, .., \pi'_{p-1}))\} \\ \Delta_0 {=} \Delta \wedge \pi'_p {=} 1 \quad \mathcal{H}_{rec} {=} \{\alpha(\pi_1, .., \pi_{p-1}) {=} enrich(p{-}1, \Delta_1 \wedge \pi'_p {=} 0, \mathcal{H}_1) \cup enrich(p{-}1, \Delta_1 \wedge \pi'_p {=} 1, \mathcal{H}_1 {+} \alpha(\pi'_1, .., \pi'_{p-1}))\} \end{array}}{a, \mathcal{H} \vdash_H (p, \Delta, \texttt{while}\ B\ \Delta_1) \rightsquigarrow enrich(a, \Delta_0, fixpt(\mathcal{H}_{rec})), enrich(a, \Delta_0, fixpt(\mathcal{M}_{rec}))}$$

$$\boxed{\textbf{HS-INVOKE}}\quad \frac{\begin{array}{c} t\, m_1(t_1, .., t_n)\ l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}; \mathcal{H}_u; \mathcal{M}_h\ \{B\} \in P \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \\ \mathcal{H}_1 {=} \mathcal{H} {+} enrich(a, \Delta, \rho\mathcal{H}_u) \quad \mathcal{M}_1 {=} \mathcal{H} {+} enrich(a, \Delta, \rho\mathcal{M}_h) \end{array}}{a, \mathcal{H} \vdash_H (p, \Delta, \texttt{invoke}\ m_1) \rightsquigarrow \mathcal{H}_1, \mathcal{M}_1}$$

$$\boxed{\textbf{HS-METH}}\quad \frac{\begin{array}{c} n, \{\mathbf{0}\} \vdash_H B \rightsquigarrow \mathcal{H}, \mathcal{M} \\ \mathcal{H}_{rec} {=} \{m(\pi_1, .., \pi_n) {=} \mathcal{H}\} \quad \mathcal{H}_{po} {=} fixpt(\mathcal{H}_{rec}) \\ \mathcal{M}_{rec} {=} \{m(\pi_1, .., \pi_n) {=} \mathcal{M}\} \quad \mathcal{M}_{po} {=} fixpt(\mathcal{M}_{rec}) \end{array}}{\begin{array}{l} \vdash_H \ \ t\, m(t_1, .., t_n)\ l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}\ \{B\} \\ \qquad \rightsquigarrow t\, m(t_1, .., t_n)\ l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}; \mathcal{H}_{po}; \mathcal{M}_{po}\ \{B\} \end{array}}$$

Figure 6: Heap Usage and Bound Inference

As before, the *enrich* function incorporates path sensitivity by adding the current abstract state into the guards of a heap specification. For heap notation, we define the guard enhancement function *enrich* as follows:

$$enrich(a, \Delta, \mathcal{H}) =_{df} \{\exists \pi'_{a+1} \cdots \cdot \Delta \wedge g \rightarrow \mathcal{B} \ \mid \ (g \rightarrow \mathcal{B}) \in \mathcal{H}\}$$

The rest of the heap inference rules are listed in Figure 6. The initial heap usage $\{\mathbf{0}\}$ (used in [HS-METH]) and *max* function (used in the definition of $\cup$) are defined as:

$$\begin{array}{lll} \{\mathbf{0}\} & =_{df} & \{(c, 0) \mid c \in \textbf{ObjectType}\} \\ max(\mathcal{B}_1, \mathcal{B}_2) & =_{df} & \{(c, max(s_1, s_2)) \mid c \in \textbf{ObjectType} \\ & & \quad \wedge (c, s_1) \in \mathcal{B}_1 \wedge (c, s_2) \in \mathcal{B}_2\} \end{array}$$

where **ObjectType** denotes the set of object types used in the current program.

7. SOUNDNESS

In this section we first present a dynamic semantics for our Bytecode language. We then formalise two safety theorems which confirm that the inferred memory bounds are safe.

7.1 The Operational Semantics

We formalise the operational semantics in this subsection. The operational rules are given in small steps of the following form:

$$\langle f : \Pi, \omega, P, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle \Pi_1, f_1, \omega_1, P_1, \texttt{s}_1, \texttt{h}_1\rangle$$

Note that $\langle f : \Pi, \omega, P, \texttt{s}, \texttt{h}\rangle$ represents a runtime configuration, where $f : \Pi$ denotes the current stack of frames. $f = (\pi, \mathcal{F}, p)$ represents the current frame with the local array π, the frame size $\mathcal{F}$, and the frame pointer p. ω (resp.ω_1) denotes the current heap before (resp. after) the evaluation. ω is a mapping from references to object values. P is the program to be executed. $\texttt{s}$ ($\texttt{s}_1$) is the available stack space before (resp. after) the execution. Similarly, $\texttt{h}$ (resp. $\texttt{h}_1$) is the available heap space before (resp. after) the execution. In our model, we assume local variables and operands stay in the same array π, where local variables occupy the bottom part and operands occupy the top part.

An intermediate expression $\texttt{ret}(E)$ is used in the operational semantics to facilitate to return to the parent frame when the currently invoked method returns.

$$\boxed{\textbf{OP-CONST}}\quad \frac{p < \mathcal{F} \qquad f_1 = (\pi \oplus \{p{+}1 \mapsto k\}, \mathcal{F}, p{+}1)}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{const}\langle t\rangle\ k, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : \Pi, \omega, \epsilon, \texttt{s}, \texttt{h}\rangle}$$

$$\boxed{\textbf{OP-LOAD}}\quad \frac{p < \mathcal{F} \qquad f_1 = (\pi \oplus \{p{+}1 \mapsto n\}, \mathcal{F}, p{+}1)}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{load}\langle t\rangle\ n, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : \Pi, \omega, \epsilon, \texttt{s}, \texttt{h}\rangle}$$

$$\boxed{\textbf{OP-STORE}}\quad \frac{f_1 = (\pi \oplus \{i \mapsto \pi_p\}, \mathcal{F}, p{-}1)}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{store}\langle t\rangle\ i, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : \Pi, \omega, \epsilon, \texttt{s}, \texttt{h}\rangle}$$

$$\boxed{\textbf{OP-INVOKE}}\quad \frac{\begin{array}{c} t\ m(t_1, .., t_n)..; \mathcal{F}_1; ..\{E\} \in P \quad f_1 = (\pi_1, \mathcal{F}_1, n{+}2) \\ f = (\pi, \mathcal{F}, p) \qquad \texttt{s}_1 = \texttt{s} {+} (\mathcal{F} {-} p) {-} \mathcal{F}_1 \end{array}}{\langle f : \Pi, \omega, \texttt{invoke}\ m, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : f : \Pi, \omega, \texttt{ret}(E), \texttt{s}_1, \texttt{h}\rangle}$$

$$\boxed{\textbf{OP-RET1}}\quad \frac{\langle \Pi, \omega, E, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle \Pi_1, \omega_1, E_1, \texttt{s}_1, \texttt{h}_1\rangle}{\langle \Pi, \omega, \texttt{ret}(E), \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle \Pi_1, \omega_1, \texttt{ret}(E_1), \texttt{s}_1, \texttt{h}_1\rangle}$$

$$\boxed{\textbf{OP-RET2}}\quad \frac{\begin{array}{c} \langle \Pi, \omega, E, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f : f_1 : \Pi_1, \omega_1, \epsilon, \texttt{s}_1, \texttt{h}_1\rangle \\ f = (\pi, \mathcal{F}, p) \qquad f_1 = (\pi_1, \mathcal{F}_1, p_1) \qquad \texttt{s}_2 = \texttt{s}_1 {+} \mathcal{F} {-} (\mathcal{F}_1 {-} p_1) \end{array}}{\langle \Pi, \omega, \texttt{ret}(E), \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : \Pi_1, \omega_1, \epsilon, \texttt{s}_2, \texttt{h}_1\rangle}$$

$$\boxed{\textbf{OP-NEW}}\quad \frac{\begin{array}{c} fresh\ \iota \quad \omega_1 = \omega \cup \{\iota \mapsto o_c\} \quad \texttt{h}_1 = \texttt{h} {-} \{(c, 1)\} \geq \{\} \\ p < \mathcal{F} \quad f_1 = (\pi \oplus \{p{+}1 \mapsto l\}, \mathcal{F}, p{+}1) \end{array}}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{new}\ c, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle f_1 : \Pi, \omega_1, \epsilon, \texttt{s}, \texttt{h}_1\rangle}$$

$$\boxed{\textbf{OP-DISPOSE}}\quad \frac{\omega_1 = \omega \setminus \pi_p \qquad \texttt{h}_1 = \texttt{h} {+} \{(c, 1)\}}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{dispose}\ c, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle (\pi, \mathcal{F}, p{-}1) : \Pi, \omega_1, \epsilon, \texttt{s}, \texttt{h}_1\rangle}$$

$$\boxed{\textbf{OP-IF1}}\quad \frac{\pi_p = \texttt{true}}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{if}\ E_1\ E_2, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle (\pi, \mathcal{F}, p{-}1) : \Pi, \omega, E_1, \texttt{s}, \texttt{h}\rangle}$$

$$\boxed{\textbf{OP-IF2}}\quad \frac{\pi_p = \texttt{false}}{\langle (\pi, \mathcal{F}, p) : \Pi, \omega, \texttt{if}\ E_1\ E_2, \texttt{s}, \texttt{h}\rangle \hookrightarrow \langle (\pi, \mathcal{F}, p{-}1) : \Pi, \omega, E_2, \texttt{s}, \texttt{h}\rangle}$$

$$\frac{\pi_p = \texttt{true}}{\langle(\pi,\mathcal{F},p):\Pi,\omega,\texttt{while}\ E,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle(\pi,\mathcal{F},p{-}1):\Pi,\omega,E;\texttt{while}\ E,\texttt{s},\texttt{h}\rangle}\ [\textbf{OP-WHILE1}]$$

$$\frac{\pi_p = \texttt{false}}{\langle(\pi,\mathcal{F},p):\Pi,\omega,\texttt{while}\ E,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle(\pi,\mathcal{F},p{-}1):\Pi,\omega,\epsilon,\texttt{s},\texttt{h}\rangle}\ [\textbf{OP-WHILE2}]$$

$$\frac{\langle\Pi,\omega,E_1,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle\Pi_1,\omega_1,\epsilon,\texttt{s}_1,\texttt{h}_1\rangle}{\langle\Pi,\omega,E_1;E_2,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle\Pi_1,\omega_1,E_2,\texttt{s}_1,\texttt{h}_1\rangle}\ [\textbf{OP-SEQ1}]$$

$$\frac{\langle\Pi,\omega,E_1,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle\Pi_1,\omega_1,E_1',\texttt{s}_1,\texttt{h}_1\rangle}{\langle\Pi,\omega,E_1;E_2,\texttt{s},\texttt{h}\rangle \hookrightarrow \langle\Pi_1,\omega_1,E_1';E_2,\texttt{s}_1,\texttt{h}_1\rangle}\ [\textbf{OP-SEQ2}]$$

7.2 Memory Adequacy

In what follows, we formulate two safety theorems on inferred memory bounds. The first theorem shows that the frame bound inferred for each method is correct. The second theorem states that the execution of a bytecode program never fails due to insufficient memory, provided it has been supplied with an amount of memory equal to (or greater than) the inferred stack and heap bounds.

THEOREM 1. *Given $t\,m(t_1,..,t_n)\ldots;\mathcal{F};\ldots\{E\}$ with $\mathcal{F}$ denoting the inferred frame bound. Whenever m is invoked with a frame f of size $\mathcal{F}$, the frame pointer p in f will always be in range $[0..\mathcal{F}]$. Formally, let $f=(\pi,\mathcal{F},n{+}2)$, and $C_0 =_{df} \langle f:\Pi,\omega,\mathit{cxt}[\texttt{ret}(E)],s,h\rangle$ be the configuration for the method body to start, where $\mathit{cxt}[\_]$ denotes the program context in which m is invoked: $\mathit{cxt}[\_] =_{df} \_;E_2 \mid \texttt{while}\ \_$. For any $C=\langle f_1:\Pi,\omega_1,\mathit{cxt}[\texttt{ret}(E_1)],s_1,h_1\rangle$ where*

$$\langle f:\Pi,\omega,\mathit{cxt}[\texttt{ret}(E)],s,h\rangle \hookrightarrow^* C \quad\text{and}\quad f_1=(\pi_1,\mathcal{F},p_1)$$

we have $p_1\le\mathcal{F}$. The relation $\hookrightarrow^*$ is defined as $\hookrightarrow^* =_{df} \bigcup_{n\ge0}\hookrightarrow^n$ where $\hookrightarrow^0$ is the identity relation and $C\hookrightarrow^{n+1} C_1 =_{df} \exists C_0\cdot C\hookrightarrow C_0 \wedge C_0\hookrightarrow^n C_1$.

Proof The proof can be easily derived based on the following fact:

For any expression E, suppose we have the frame bound inference result: $l,\Gamma\vdash_F E \rightsquigarrow (p,E_A),\Gamma_1,\mathcal{F}$. Then, we have $\mathcal{F}\ge p$. This fact is proved by structural induction on E.

Suppose there exists $C=\langle f_1:\Pi,\omega_1,\texttt{cxt}[\mathit{ret}(E_1)],\texttt{s}_1,\texttt{h}_1\rangle$ such that $C_0\hookrightarrow^n C$ for some n, but $p_1>\mathcal{F}$. Applying frame bound inference to E_1, we have: $l,\Gamma_1\vdash_F E_1 \rightsquigarrow (p_1,E_A'),\Gamma_1',\mathcal{F}_1$. From the above fact, we have $\mathcal{F}_1>p_1$. From inference rule [**FS-METH**], we have $\mathcal{F}>\mathcal{F}_1$. This yields $\mathcal{F}>\mathcal{F}_1>p_1>\mathcal{F}$, a contradiction.

THEOREM 2. *Given program P with M as its main method. For brevity, we assume M has body E but does not have any parameter. Suppose a frame bound $\mathcal{F}$, a stack bound $\mathcal{S}$, and a heap bound $\mathcal{M}$ have been inferred for method M. If the initial configuration $C_0=\langle(\pi,\mathcal{F},2):\Pi,\omega,\texttt{ret}(E),s{-}\mathcal{F},h\rangle$ satisfies the following conditions:*

$$(1)\ \pi\models h\ge\mathcal{M} \quad (2)\ \pi\models s\ge\mathcal{S}$$

then the stack space s and the heap space h are adequate for the execution of the program. That is, for any $C=\langle\Pi_1,\omega_1,E_1,s_1,h_1\rangle$ where $C_0\hookrightarrow^* C$, we have $s_1\ge0$, and $h_1\ge\{\mathbf{0}\}$.

Note that conditions (1) and (2) ensure the available stack and heap spaces in the initial state are not less than what the method requires (the inferred bounds).

We now provide formal definitions for (1) and (2). For brevity, we assume normalisation has been applied to $\mathcal{M}$ and $\mathcal{S}$: all guards are disjoint. $\pi\models\texttt{h}\ge\mathcal{M}$ is $\texttt{true}$ if and only if the following holds: $\forall g{\rightarrow}\mathcal{B}\in\mathcal{M}$, if guard g is satisfied, i.e., $\pi\models g$, then $\texttt{h}\ge\mathcal{B}$. The definition for $\pi\models\texttt{s}\ge\mathcal{S}$ is similar.

In what follows we give some auxiliary definitions and then state and prove two lemma from which Theorem 2 can be readily derived.

DEFINITION 1 (MEMORY REQUIREMENT). *Given an abstract state Δ and an expression E. If there exist l,Γ,a such that*

$$\begin{array}{l} l,\Gamma\vdash_F E\rightsquigarrow A,\Gamma_1,\mathcal{F} \\ \Delta\vdash_A A\rightsquigarrow B,\Delta_1 \\ a\vdash_S B\rightsquigarrow\mathcal{S} \\ a,\{\mathbf{0}\}\vdash_H B\rightsquigarrow\mathcal{H},\mathcal{M} \end{array}$$

Then we say E requires stack space $\mathcal{S}$ and heap space $\mathcal{M}$ at Δ, denoted as $\Delta,E\propto\mathcal{S},\mathcal{M}$.

DEFINITION 2 (CONSISTENCY RELATION). *Suppose $\Delta,E\propto\mathcal{S},\mathcal{M}$. Given runtime configuration $\langle(\pi,\mathcal{F},p):\Pi,\omega,E,s,h\rangle$. We say the runtime memory configuration is consistent with the compile-time requirement, denoted as $\langle\pi,s,h\rangle\models\langle\Delta,\mathcal{S},\mathcal{M}\rangle$, if the following conditions hold:*

$$(1)\ \pi\models\Delta \quad (2)\ \pi\models s\ge\mathcal{S} \quad (3)\ \pi\models h\ge\mathcal{M}$$

Note the satisfaction relation $\pi\models\Delta$ indicates the concrete state π is subsumed in the abstract state Δ. It is defined as follows:

$$\pi\models\Delta \ =_{df}\ \mathit{size}(\pi)\implies\exists\pi_1..\pi_p\cdot\Delta$$

The function $\mathit{size}(\pi)$ returns a predicate of the form $\bigwedge_i\pi_i'=k_i$, by taking into account boolean and integer values while ignoring values of other types. For example, suppose the concrete state is $\{\pi_1{=}2,\pi_2{=}1.3,\pi_3{=}\texttt{true}\}$ where $\pi_1::\texttt{int},\pi_2::\texttt{float},\pi_3::\texttt{bool}$, then $\mathit{size}(\pi)=(\pi_1'=2\wedge\pi_3'=1)$

The following *consistency preservation* lemma indicates that the consistency relation between compile-time and runtime is preserved by each derivation step.

LEMMA 3. *Suppose $C=\langle(\pi,\mathcal{F},p):\Pi,\omega,E,s,h\rangle$ is the configuration for E to start, and $\langle\pi,s,h\rangle\models\langle\Delta,\mathcal{S},\mathcal{M}\rangle$. For any $C_1=\langle(\pi_1,\mathcal{F}_1,p_1):\Pi_1,\omega_1,E_1,s_1,h_1\rangle$, where $C\hookrightarrow C_1$, there exist $\Delta_1,\mathcal{S}_1,\mathcal{M}_1$, such that $\Delta_1,E_1\propto\mathcal{S}_1,\mathcal{M}_1$, and $\langle\pi_1,s_1,h_1\rangle\models\langle\Delta_1,\mathcal{S}_1,\mathcal{M}_1\rangle$.*

Proof By structural induction on E. Details are omitted here.

LEMMA 4. *Suppose $C=\langle(\pi,\mathcal{F},p):\Pi,\omega,E,s,h\rangle$ is the configuration for E to start, and $\langle\pi,s,h\rangle\models\langle\Delta,\mathcal{S},\mathcal{M}\rangle$. For any $C_1=\langle(\pi_1,\mathcal{F}_1,p_1):\Pi_1,\omega_1,E_1,s_1,h_1\rangle$, where $C\hookrightarrow^* C_1$, we have $s_1\ge0$, and $h_1\ge\{\mathbf{0}\}$.*

Proof By induction on the length n of derivation from C to C_1. For the base case $n=1$, it is obvious due to Lemma 3. For the case $n{+}1$, we consider the first derivation step, which yields $C_0=\langle(\pi_0,\mathcal{F}_0,p_0):\Pi_0,\omega_0,E_0,\texttt{s}_0,\texttt{h}_0\rangle$. From Lemma 3, we have $\langle\pi_0,\texttt{s}_0,\texttt{h}_0\rangle\models\langle\Delta_0,\mathcal{S}_0,\mathcal{M}_0\rangle$ for some $\Delta_0,\mathcal{S}_0,\mathcal{M}_0$. The derivation length from C_0 to C_1 is now n. By induction hypothesis, we have $\texttt{s}_1\ge0$, and $\texttt{h}_1\ge\{\mathbf{0}\}$. It then concludes.

8. DISCUSSION

In this section, we proceed with a high level discussion on abstract states for objects.

To keep our presentation simple, we have omitted the inference of abstract states for heap allocated objects. A technical challenge is to deal with objects that are both *mutable* and *shareable*. Such objects are more difficult to track accurately. To deal with them, we propose providing an alias inference system to identify two main groups of trackable objects (or references), namely: (i) references that are unique whose abstract states may change, and (ii) references whose abstract states (fields) are immutable but may be freely shared.

The target for such alias inference is also useful for supporting the verification of protocol safety, as advocated in [11]. For the current work, the properties that we are interested in analysing are mainly size-related properties. For example, consider a binary tree object of the following type declaration:

```
object BNode {int val; BNode left; BNode right}
```

Two properties we may track for this BNode object type are size (number of nodes) of the tree and height of the tree. We can define these two properties by introducing two abstract fields s and h, as shown in the type declaration below:

object BNode$\langle s, h\rangle$ where $s{=}1{+}\texttt{left}.s{+}\texttt{right}.s$
$h{=}1{+}max(\texttt{left}.h, \texttt{right}.h)$

Given a method to compute the height of a tree:

```
int height(BNode t) 2, true {
  if t==null {return 0}
  else {int v=1+max(height(t.left), height(t.right));
        return v}
```

Our inference system would derive:

$\mathcal{F}{\equiv}k_1$; $\phi_{po}{\equiv}\pi'_3{=}\pi_1.h$; $\mathcal{S}{\equiv}max(k_2{\times}\pi_1.h, k_1)$;
$\mathcal{H}{\equiv}\{\mathbf{0}\}$; $\mathcal{M}{\equiv}\{\mathbf{0}\}$

where k_1 and k_2 are some integer constants. Given another method to sum values of a tree, followed by the disposal of its nodes:

```
int sum(BNode t) 2, true {
  if t==null {return 0}
  else {int v=t.val+sum(t.left)+sum(t.right);
        dispose(t) ; return v}
```

Our inference system would derive:

$\mathcal{F}{\equiv}k_3$; $\phi_{po}{\equiv}\texttt{true}$; $\mathcal{S}{\equiv}max(k_4{\times}\pi_1.h, k_3)$;
$\mathcal{H}{\equiv}\{(\texttt{BNode}, -\pi_1.s)\}$; $\mathcal{M}{\equiv}\{\mathbf{0}\}$

where k_3 and k_4 are some integer constants.

9. RELATED WORKS

Past research on memory usage prediction [20, 19] mainly focused on functional programs where data structures are mostly immutable and thus easier to handle. Hughes and Pareto [20] proposed a type-checking system on space usage estimation for a first-order functional language, extended with regions. The use of a region model facilitates recovery of heap space. However, no inference mechanism is proposed and its recovery mechanism is limited as each region is only deleted when all its objects are dead. Hofmann and Jost [19] proposed a solution to obtain linear bounds on the heap space usage of first-order functional programs. A key feature of their solution is the use of linear typing which allows the space of each last-use data constructor/object to be directly recycled by matching allocation. With this approach, memory recovery can be supported within each function but not across functions unless the dead objects are explicitly passed. While their model incorporates an inference system, it does not cover stack usage and is limited to a linear form without disjunction. As a result, path sensitivity is not fully exploited.

Aspinall et al. [3] applied ideas from proof-carrying code to the problem of resource certification for mobile code. In their system, memory adequacy proofs are checked at the level of a linearly typed assembly language with the help of theorem proving techniques. However, the system presumes that source programs come from a first-order functional language with a resource-aware type system [19]. Similarly, Amadio et al. [2] defined a simple stack machine for a first-order functional language and showed how to perform type, size and termination verifications at the bytecode level. Their main result is a proof that each program with the quasi-interpretation property[1] that terminates has a polynomial stack bound. No algorithm was proposed for precisely inferring this stack bound. Furthermore, heap space was not considered. More recently, Cachera et al. [8] proposed constraint-based memory analysis for a Java-based bytecode language. For a given program, their loop-detecting algorithm can find methods and instructions that execute an unbounded number of times. This fact can be used to check whether memory usage is bounded or not. However, their analysis does not infer such a bound nor check whether a given amount of memory is adequate or not.

There were also several works on analysing the stack space requirement of interrupt-driven programs. Brylow et al. [6] proposed stack size analysis using a context-free reachability algorithm based on model checking. Chatterjee et al. [10] investigated complexity of the stack boundedness problem and the exact maximum stack size problem. These techniques apply to only interrupt stacks (but not the more general runtime stacks), and are for programs without recursive (interrupt) invocations. Stack Analyzer [1] is a commercial product that can determine worst-case stack usage. However, it assumes a user-specified limit on recursion depth.

Our previous work [12] proposed a modular heap usage verification system for object-oriented programs. The system can check whether or not a certain amount of memory is adequate for safe execution of a given program. However, programmers must provide the pre/post conditions on memory usage for each method. This approach (without inference) may be impractical for bytecode programs.

Some other recent works [23, 21] addressed memory safety and configuration concerns that are orthogonal to the more fundamental issue of memory adequacy. Xu et al. [23] proposed techniques to detect both spatial and temporal memory errors in C programs. Poladian et. al. [21] formulated the dynamic configuration of resource-aware applications as an optimization problem: seeking a capability point such that quality of service (QoS) is maximised while imposed resource constraints are still met. They considered various resources including computing cycles, network bandwidth, power and memory. It may be interesting to see if their empirical approach could benefit from automatic static analysis techniques such as those proposed here.

10. CONCLUDING REMARKS

[1]This essentially implies that each definition has some non-increasing measure.

We have proposed a sound inference system for a structured bytecode language to predict the amount of memory space needed to execute given programs. Our system can infer both net usage and upper bound of stack/heap spaces required for a reasonably large class of programs. We use a special guarded expression form to track both memory usage and memory bound in a path-sensitive manner. Our approach can handle both recursion and loops. We build recursive constraint abstraction to model program state, net memory usage, and memory bound. Their corresponding abstractions are subjected to a set of normalisation rules, prior to conventional fixpoint analysis.

A prototype for our inference system has been built to confirm the viability and practicality of our approach. We have carried out experiments to infer stack/heap bounds for both the Java Olden benchmark[9] and also a suite of small programs from the RegJava collection[13]. These codes are converted to bytecode form prior to inference. Almost all stack usage bounds can be captured, except for the Ackermann function which requires a stack space that is exponential to its parameters' sizes. This stack bound is beyond the Presburger arithmetic form used in our current system. We can safely infer heap usage/bound for all our benchmark programs, except for a function of Voronoi program in Olden benchmark. That function has a complex loop with heap use that is linearly bounded but requires a sophisticated proof. While the current prototype is based on a decidable Presburger solver, the inference framework that we have proposed can be directly plugged to more sophisticated and/or efficient constraint solvers. Our proposal may therefore ride on future advances in underlying constraint solving technology.

Our initial experiments are preliminary in nature but have confirmed the viability of our approach. We envision our framework to be most useful whenever memory resources are to be carefully quantified. Possible application domains include embedded devices, safety critical systems, and high-reliability server systems where memory footprints are to be tightly accounted for. We also intend the current system to be a tool for software engineers to assist in static prediction of memory space metrics.

Acknowledgement We thank Florin Craciun for insightful technical comments and Alexia Leong for careful proofreading of this paper. This work is supported by research grant R-252-000-151-112, SMA and a gift from Microsoft.

11. REFERENCES

[1] AbsInt. StackAnalyzer - Stack Usage Analysis. http://www.absint.com/stackanalyzer/.

[2] R. M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. In *18th Int'l Conf. on Computer Science Logic (CSL04)*. Springer-Verlag, LNCS, September 2004.

[3] D. Aspinall, S. Gilmore, M. Hofmann, D.Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer-Verlag, LNCS, 2004.

[4] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86-A Platform for Analyzing x86 Executables. In *Intl Symp. on Compiler Construction*, pages 250–254, 2005.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer-Verlag, LNCS, 2004.

[6] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *Proceedings of the International Conference on Software Engineering (ICSE'01)*, Toronto, Canada, May 2001.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[8] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *13th International Symposium of Formal Methods Europe (FM'05)*, July 2005.

[9] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *4th Principles and Practice of Parallel Programming*, Santa Barbara, California, May 1993.

[10] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, and J. Palsberg. Stack Size Analysis for Interrupt-Driven Programs. In *10th Annual International Static Analysis Symposium (SAS '03)*, San Diego, California, June 2003. Springer-Verlag.

[11] W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *27th International Conference on Software Engineering (ICSE'05)*, St. Louis, Missouri, May 2005.

[12] W.N. Chin, H.H. Nguyen, S.C. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Proceedings of the The 12th International Static Analysis Symposium (SAS'05)*, Springer LNCS, London, UK, September 2005.

[13] M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, 1977.

[15] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.

[16] C. Flanagan and K.R.M. Leino. Houdini, an Annotation Assistant for ESC/Java . In J. N. Oliveira and P. Zave, editors, *Intl Symp of Formal Methods Europe (FM'01)*, March 2001.

[17] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[18] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[19] M. Hofmann and S. Jost. Static prediction of heap space usage for first order functional programs. In *ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 2003.

[20] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the International Conference on Functional Programming (ICFP '99)*, September 1999.

[21] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the International Conference on Software Engineering (ICSE'04)*, May 2004.

[22] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context*, Toronto, Ontario, Canada, May 2001. ACM.

[23] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and

Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *12th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT04/FSE12)*, October 2004.