Analysing Memory Resource Bounds for Low-Level Programs

Wei-Ngan Chin[1] Huu Hai Nguyen[1] Corneliu Popeea[1] Shengchao Qin[2]

[1] Department of Computer Science, National University of Singapore, Singapore
[2] Department of Computer Science, Durham University, UK
{chinwn,nguyenh2,corneliu}@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 resource bounds can be inferred for assembly-level 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 format. Our current proposal relies on a Presburger solver to capture memory requirements symbolically, and to perform fixpoint analysis for loops and recursion. Apart from safety in memory adequacy, our proposal can provide estimate on memory costs for embedded devices and improve performance via fewer runtime checks against memory bound.

1. Introduction

While formal specification and functional correctness [17, 7] have for a long time been a central focus of the software engineering community, an orthogonal consideration on resource adequacy and utilization is gradually gaining importance. This trend is being driven by the proliferation of resource-constrained mobile devices, coupled with the high expectations on reliability and usability from consumers. Previous work in this area (amongst the real-time and embedded systems community) have mostly focused on real-time aspects, with major inroads made in WCET (worst-case execution time) domain. In this paper, we focus on memory as a constrained resource and approach the problem from a program analysis perspective. To make it relevant to a wider community, we shall formulate our analysis for low-level assembly-like programs.

We consider the determination of memory resource requirements for data structures that can be dynamically allocated and recycled during program computation. Such memory subsystems 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 beyond the method calls where they are created. It is typically used with a garbage collector for recovering dead spaces.

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 language restriction (e.g. no heap allocated objects), profiling, or informal estimate to predict the amount of memory space required. To minimise errors due to inadequate memory, developers often over-estimate the memory space that is needed. However, this translates into greater hardware cost without giving any guarantee on memory adequacy.

There are few previous systems for predicting the symbolic memory usage of programs, especially of low-level programs. Recent works [19, 21] are mostly based on analysing functional programs where the presence of immutable data structures makes such analysis easier to formalise. Even though [5, 4] 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 [13, 28], 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 hard to provide and are likely to be impractical for assembly-level programs.

The focus on low-level programs is important for two reasons. Firstly, they can represent the intermediate form for a variety of higher-level languages. Secondly, resource usage may be affected by optimising compilers which can render memory analyses done at the source level possibly 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 low-level assembly-like 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 passes.
- **Guarded Expression**: 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.
- **Fixpoint Analysis**: 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 is towards a *fixpoint analysis* that separates 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.

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

We believe that our memory inference system is useful for embedded software systems because: 1) such software often operates on platforms with limited memory, and 2) failing, because of insufficient memory, can have costly real-world consequences. The next section describes technical challenges in accurately 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. Our Approach

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 high level, we shall confine our examples to programs that are written in a source level imperative language. Their translation to assembly 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 explore net usage at the end of its computation
- *Memory Bound*: to represent high watermark of memory usage at all points over its computation

Both metrics are computed in a conservative matter. 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 :

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

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`. The right side of the figure traces the current heap usage and bound in terms of number of instances of object types `c` and `d`. By tracing through heap usage 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. We propose to capture memory estimation by counting each object type rather than counting number of bytes used, since this is more abstract and be used to model memory fragmentation. If required, the conversion to byte count is straightforward and has been carried out for our experiments.

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 for each method call. 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 Dealing with Explicit Disposal for Heap Memory

To support compile-time analysis of heap recovery, there are at least two possible solutions, namely: (i) use explicit disposal (ii) use region-based memory. The first solution [13] uses a special `dispose` command to help recover each heap object that is no longer required, where possible. The second solution [21] is to use region-based memory to group objects with different lifetimes under separate regions, that can then be recovered whenever their lifetimes expire. Both these solutions can be considered as an over-approximation for recovery that can be achieved by garbage collection, provided that dangling references are either avoided during disposal or suitably marked for the garbage collector.

We have opted for the first solution in our paper, as it can help achieve fine-grain heap recovery. However, automatically inserting `dispose` commands is challenging since we have to minimise on memory leaks and must also ensure that they are safe for memory accounting purpose. For example, we should not dispose of an object more than once, and we should not apply `dispose` for a `null` reference since this does not result in any memory recovery. However, it is safe to forget (or delay) the invocation of a `dispose` command for a dead object, but this could result in a higher estimated heap bound.

A prior work of ours in [13] handles the automatic insertion of `dispose` commands with the help of an alias/uniqueness type system which works well for unshared data structures. A current limitation is that it cannot handle data structures with shared nodes, such as cyclic or graph-like structures. Nevertheless, the technique for automatically inserting dispose commands can be considered to be orthogonal to the technique for memory estimation. Each improvement to automatically inserting dispose more precisely will indirectly result in better memory estimation, without the need to modify the underlying estimation technique. At the present moment, our system supports the automatic insertion of `dispose` commands, after supplying annotations for alias system type with uniqueness. This aspect can be further improved in the future but is outside of the scope of the present paper.

2.3 Guarded Expression Form

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 the program code 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 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 both branches. In program analysis, this phenomenon 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. Guarded 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 important for 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.

2.4 Fixpoint Analysis

Analysing recursive methods is naturally challenging. In theory, we are to trace through every recursive call for memory usage so as to compute suitable 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 analysis are mostly known (see [14] for conjunctive polyhedra analysis or [27] for disjunctive analysis), 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:

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

Here, n and r denote the input parameter and the method's result, respectively. Fixpoint analysis would proceed with the first version $\mathit{rec}_0(n, r)$ assumed to be `false`, and with each subsequent version $\mathit{rec}_{i+1}(n, r)$ computed from the previous version $\mathit{rec}_i(n{-}1, r_1)$, as follows:

$$\begin{array}{ll}
\mathit{rec}_1(n,r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_0(n{-}1,r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1 \\
\mathit{rec}_2(n,r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_1(n{-}1,r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}(n{-}1{\leq}0{\wedge}r_1{=}1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1{\vee}(0{<}n{\leq}1{\wedge}r{=}2{+}1) \\
\mathit{rec}_3(n,r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_2(n{-}1,r_1){\wedge}r{=}2{+}r_1) \\
 & = n{\leq}0{\wedge}r{=}1 \vee (n{=}1{\wedge}r{=}3) \vee (n{=}2{\wedge}r{=}5) \\
 & =_h n{\leq}0{\wedge}r{=}1 \vee (0{<}n{\leq}2{\wedge}r{=}2n{+}1) \\
 & =_w n{\leq}0{\wedge}r{=}1 \vee (0{<}n \quad {\wedge}r{=}2n{+}1) \\
\mathit{rec}_4(n,r) & = n{\leq}0{\wedge}r{=}1{\vee}(\exists r_1{\cdot}n{>}0{\wedge}rec_3(n{-}1,r_1){\wedge}r{=}2{+}r_1) \\
 & =_h n{\leq}0{\wedge}r{=}1{\vee}(0{<}n{\wedge}r{=}2n{+}1)
\end{array}$$

The process requires safe approximation techniques using *hulling* and *widening* that are standard for fixpoint analysis of relational formulae (see [14, 27, 25]). The hulling operation (denoted above by $=_h$) combines related disjunctive formulae into a conjunct. The widening operation (denoted above by $=_w$) drops the sub-formulae that are changed compared to the previous version. We reach a fixpoint when $\mathit{rec}_{i+1}(n, r) \Longrightarrow \mathit{rec}_i(n, r)$. For the above example, its fixpoint gives:

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

After that, we 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:

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

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

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

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:

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

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

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

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. Note that our analysis tolerates memory leakage (from missing `dispose` commands) by reporting a higher bound than strictly required. For this example, a missing `dispose` for the first object constructed by `new` has resulted in n extra c-type heap objects for both the estimated heap usage and heap bound. Though conservative, our estimation remains sound and would benefit automatically from improvements in insertion of `dispose` commands.

Each loop can be considered a special case of tail recursion and is similarly handled. In case a method definition has a loop containing a self (or mutual) recursive call, the method and its loop are considered to be in mutual recursion.

2.5 A Structured Assembly Language

We shall formalise our inference system for a small assembly language. To keep our presentation simple, we provide a conditional statement (with two branches) and a `while` loop structure. In reality, most low-level 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 major technical difficulty but may obscure our exposition. Furthermore, it is helpful to recover higher-level language constructs when directly analysing assembly-level codes, as advocated in [6]. For pedagogical reasons, we propose using a more structured assembly language, and omit features relating to `jsr`-like subroutines. Our low-level language is given in Figure 1.

For each method, we expect types of parameters and the result to be declared. Furthermore, a symbol l (denoting the number of local variables including parameters) is easily precomputed for each method. The stack frame of each method call is organised as shown in Figure 2. The parameters are assumed to occupy the first n slots of the local variables section starting at position 1. This is

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

Figure 1. Syntax for a Assembly-like Language

followed by another $l{-}n$ slots for other 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.

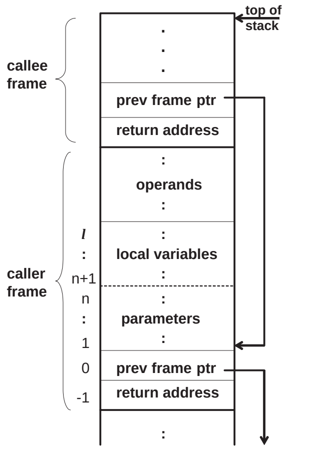

Figure 2. Stack Frames

The $\texttt{load}\langle t \rangle\ i$ 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 $\texttt{const}\langle t \rangle\ k$ instruction places a constant value on the top of the stack. The $\texttt{store}\langle t \rangle\ i$ moves a value from the top of the stack to a specified local variable. The $\texttt{invoke}\ m$ instruction calls a method m after its arguments are placed on the stack. On return from the callee, the arguments are removed and a result is placed on the stack. Both the conditional `if` and `while` loop commands expect a boolean test to be already in the data stack which can help direct the control-flow of the program.

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. Note that c denotes the static type of the object in question. Since allocation via `new` always use the exact type, while that by `dispose` may actually recover a larger subtype (from an OO source language), our estimation of heap usage via these two commands are conservative, and hence sound. While we support objects in our language, we shall omit details on how fields are declared and accessed as they can be indirectly supported through primitive methods. Furthermore, in the present work, we currently only tracked the values of program variables and immutable fields, but not mutable fields. A short consideration on how mutable fields can be tracked via strong update techniques is discussed later in Sec 7.

Lastly, though our symbolic constraint is currently limited to Presburger arithmetic, our inference framework allows the use of more sophisticated constraint domains. Presburger arithmetic represents a good middle ground that is expressive yet practical. Despite the possibility of exponential-time complexity, a well-built solver, like Omega [26] that is used by our prototype, can give mostly fast execution times when handling medium-sized formulae. We believe that the key to good performance is to exploit modularity that our approach offers. For example, we break a large fixpoint analysis to several smaller ones, and also limit the size of inferred constraints via safe approximation (using convex hulling), where possible.

2.6 Multi-Pass Inference

We present our inference as a modular multi-pass system. Breaking a complex inference system into smaller phases can simplify our formalisation considerably. The first phase is to build a call dependency graph that will 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 final target of our inference system is a set of annotations for each method declaration. Given a method:

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

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_{pr} its precondition, ϕ_{po} its postcondition, $\mathcal{S}$ its stack bound, H_{po} its net heap usage, and $\mathcal{M}_{po}$ its heap bound.

3. Frame Bound Inference

For our low-level 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:

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

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 a computation (on frame bound) 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* (denoted by an integer offset) 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:

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

A set of rules for frame bound inference are listed in Figure 3. Being syntax-directed, these rules constitute an inference algorithm. Apart from frame bound inference, we also perform some

$$[\textsc{fs-const}]\quad \frac{k{::}t \quad \Gamma_1{=}t{:}\Gamma}{l,\Gamma \vdash_F \texttt{const}\langle t\rangle\ k \rightsquigarrow (|\Gamma|,\texttt{const}\langle t\rangle\ k),\Gamma_1,|\Gamma_1|}$$

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

$$[\textsc{fs-store}]\quad \frac{i{\le}l{\le}|\Gamma| \qquad \Gamma_1{=}\Gamma{\oplus}(i\mapsto t) \qquad r{=}|\Gamma|{+}1}{l,t{:}\Gamma \vdash_F \texttt{store}\langle t\rangle\ i \rightsquigarrow (r,\texttt{store}\langle t\rangle\ i),\Gamma_1,r}$$

$$[\textsc{fs-dispose}]\quad \frac{l{\le}|\Gamma| \qquad r{=}|\Gamma|{+}1}{l,\texttt{ref}{:}\Gamma \vdash_F \texttt{dispose}\ c \rightsquigarrow (r,\texttt{dispose}\ c),\Gamma,r}$$

$$[\textsc{fs-new}]\quad \frac{\Gamma_1{=}\texttt{ref}{:}\Gamma}{l,\Gamma \vdash_F \texttt{new}\ c \rightsquigarrow (|\Gamma|,\texttt{new}\ c),\Gamma_1,|\Gamma_1|}$$

$$[\textsc{fs-seq}]\quad \frac{l,\Gamma \vdash_F E_1 \rightsquigarrow A_1,\Gamma_1,\mathcal{F}_1 \quad l,\Gamma_1 \vdash_F E_2 \rightsquigarrow A_2,\Gamma_2,\mathcal{F}_2}{l,\Gamma \vdash_F E_1;E_2 \rightsquigarrow (|\Gamma|,A_1;A_2),\Gamma_2,\mathit{max}(\mathcal{F}_1,\mathcal{F}_2)}$$

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

$$[\textsc{fs-invoke}]\quad \frac{\begin{array}{c} t\,m(t_1,..,t_n)\ \cdots\ \{\cdots\} \in P \quad \mathcal{F}{=}\mathit{max}(|\Gamma|,|\Gamma_2|) \\ \Gamma = [t_n,..,t_1]{+}\Gamma_1 \quad l{\le}|\Gamma_1| \quad \Gamma_2{=}t{:}\Gamma_1 \end{array}}{l,\Gamma \vdash_F \texttt{invoke}\ m \rightsquigarrow (|\Gamma|,\texttt{invoke}\ m),\Gamma_2,\mathcal{F}}$$

$$[\textsc{fs-while}]\quad \frac{l{\le}|\Gamma| \quad l,\Gamma \vdash_F E \rightsquigarrow A,\texttt{bool}{:}\Gamma,\mathcal{F}}{l,\texttt{bool}{:}\Gamma \vdash_F \texttt{while}\ E \rightsquigarrow (|\Gamma|{+}1,\texttt{while}\ A),\Gamma,\mathcal{F}}$$

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

Figure 3. Frame Bound Inference

checks to ensure that there is no underflow of the operand stack and that simple type safety exists. 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 here. 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 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. Note from [FS−WHILE] that execution of while loops do not cause any increase in the frame size. Furthermore, the test $l{\le}|\Gamma|$ is to ensure that current stack frame does not underflow into the area that has been reserved for local variables.

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 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 (via strongest postcondition reasoning within the chosen abstract domain) 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:

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

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:

$$[\textsc{as-const}]\quad \frac{\Delta_1 = \Delta{\wedge}\mathit{eq}_t(\pi_{p+1}',k)}{\Delta \vdash_A (p,\texttt{const}\langle t\rangle\ k) \rightsquigarrow (p,\Delta,\texttt{const}\langle t\rangle\ k),\Delta_1}$$

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:

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

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

$$[\textsc{as-store}]\quad \frac{\Delta_1 = \Delta \circ_{\{\pi_i\}} \pi_i'{=}\pi_p'}{\Delta \vdash_A (p,\texttt{store}\langle t\rangle\ i) \rightsquigarrow (p,\Delta,\texttt{store}\langle t\rangle\ i),\exists\pi_p'\cdot\Delta_1}$$

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:

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

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:

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

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 above example, this leads to:

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

$$\frac{\underset{}{[\underline{\textbf{AS-LOAD}}]} \quad \Delta_1 = \Delta \wedge \pi'_{p+1} {=} \pi'_i}{\Delta \vdash_A (p, \texttt{load}\langle t \rangle\ i) \rightsquigarrow (p, \Delta, \texttt{load}\langle t \rangle\ i), \Delta_1}
\qquad
\frac{[\underline{\textbf{AS-NEW-DISPOSE}}] \quad I = \texttt{new}\ c \mid \texttt{dispose}\ c}{\Delta \vdash_A (p, I) \rightsquigarrow (p, \Delta, I), \Delta}
\qquad
\frac{[\underline{\textbf{AS-SEQ}}] \quad \Delta \vdash_A A_1 \rightsquigarrow B_1, \Delta_1 \quad \Delta_1 \vdash_A A_2 \rightsquigarrow B_2, \Delta_2}{\Delta \vdash_A (p, A_1; A_2) \rightsquigarrow (p, \Delta, B_1; B_2), \Delta_2}$$

$$\frac{\begin{array}{c}[\underline{\textbf{AS-INVOKE}}] \\ t\, m(t_{1..n})\, l; \phi_{pr}; \mathcal{F}; \phi_{po}\, \{\cdots\} {\in} P \quad \rho = [\pi_i {\mapsto} \pi'_{p-n+i}]_{i=1}^{n} {\cup} [\pi'_{l+1} {\mapsto} r] \\ \textit{fresh}\ r \quad \Delta \Longrightarrow \rho\phi_{pr} \quad \Delta_1 = (\exists \pi'_{p-n+1} .. \pi'_p {\cdot} \Delta {\wedge} \rho\phi_{po}) {\wedge} (\pi'_{p-n+1} {=} r)\end{array}}{\Delta \vdash_A (p, \texttt{invoke}\ m) \rightsquigarrow (p, \Delta, \texttt{invoke}\ m), \exists r {\cdot} \Delta_1}
\qquad
\frac{\begin{array}{c}[\underline{\textbf{AS-METH}}] \\ \Delta = \bigwedge_{i=1}^{n} \pi'_i {=} \pi_i \quad \Delta \vdash_A A \rightsquigarrow B, \Delta_1 \quad \phi_{pr} = \textit{prefixpt}(\phi_{rec}) \\ \phi_{rec} = \{m(\pi_1, .., \pi_n, \pi'_{l+1}) {=} \Delta_1\} \quad \phi_{po} = \textit{fixpt}(\phi_{rec})\end{array}}{\vdash_A \quad t\, m(t_{1..n})\, l; \mathcal{F}\, \{A\} \rightsquigarrow t\, m(t_{1..n})\, l; \phi_{pr}; \mathcal{F}; \phi_{po}\, \{B\}}$$

Figure 4. Abstract State Inference

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:

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

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 $\Delta \Longrightarrow \rho\phi_{pr}$.

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:

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

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

Applying fixpoint analysis to the recursive abstraction gives a postcondition for executing this loop. Note that 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 rationale behind a separate frame inference stage is to limit the effects of primitive operations, such as $\texttt{load}\langle t \rangle$ and $\texttt{store}\langle t \rangle$, 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 at the end of 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 $\{g {\to} s\}^*$ where g is a predicate and $s{\in}\mathbf{AExp}$ (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:

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

Note that ρ captures the argument substitution process. We use a special function $\textit{enrich}(a, \Delta, \mathcal{S})$ to incorporate path-sensitive guarded formula $\mathcal{S}$ into the current abstract state Δ, as follows:

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

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

Furthermore, we have the option of mirroring 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:

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

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

$$[\underline{\mathbf{SS{-}INSTR}}]\quad \frac{I = \mathtt{const}\langle t\rangle\ k \mid \mathtt{load}\langle t\rangle\ i \mid \mathtt{store}\langle t\rangle\ i \mid \cdots}{a \vdash_S (p, \Delta, I) \rightsquigarrow \{\}}$$

$$[\underline{\mathbf{SS{-}SEQ}}]\quad \frac{a \vdash_S B_1 \rightsquigarrow \mathcal{S}_1 \quad a \vdash_S B_2 \rightsquigarrow \mathcal{S}_2}{a \vdash_S (p, \Delta, B_1; B_2) \rightsquigarrow \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$[\underline{\mathbf{SS{-}IF}}]\quad \frac{a \vdash_S B_1 \rightsquigarrow \mathcal{S}_1 \quad a \vdash_S B_2 \rightsquigarrow \mathcal{S}_2}{a \vdash_S (p, \Delta, \mathtt{if}\ B_1\ B_2) \rightsquigarrow \mathcal{S}_1 \cup \mathcal{S}_2}$$

$$[\underline{\mathbf{SS{-}WHILE}}]\quad \frac{\begin{array}{c} a \vdash_S B \rightsquigarrow \mathcal{S} \\ \mathcal{S}_{rec} = \{\alpha(\pi_1..\pi_{p-1}) = \mathcal{S} \cup \mathit{enrich}(p-1, \Delta_1 \wedge \pi'_p = 1, \alpha(\pi'_1..\pi'_{p-1}))\} \end{array}}{a \vdash_S (p, \Delta, \mathtt{while}\ B\ \Delta_1) \rightsquigarrow \mathit{enrich}(a, \Delta \wedge \pi'_p = 1, \mathit{fixpt}(\mathcal{S}_{rec}))}$$

$$[\underline{\mathbf{SS{-}METH}}]\quad \frac{n \vdash_S B \rightsquigarrow \mathcal{S} \quad \mathcal{S}_{rec} = \{m(\pi_1..\pi_n) = \mathcal{S} \cup \{\phi_{pr} \rightarrow \mathcal{F}\}\} \quad \mathcal{S}_\mu = \mathit{fixpt}(\mathcal{S}_{rec})}{\vdash_S \quad t\, m(t_{1..n})\, l; \phi_{pr}; \mathcal{F}; \phi_{po}\ \{B\} \rightsquigarrow t\, m(t_{1..n})\, l; \phi_{pr}; \mathcal{F}; \phi_{po}; \mathcal{S}_\mu\ \{B\}}$$

Figure 5. Stack Bound Inference

normalisation rules:

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

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. In particular, the second rule can be viewed as a special case of the third rule. Furthermore, we may heuristically apply, where desired, the following approximation rule:

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

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:

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

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 stack bound after safe approximation.

6. Heap Inference

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

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

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:

$$[\underline{\mathbf{HS{-}SEQ}}]\quad \frac{a, \mathcal{H} \vdash_H B_1 \rightsquigarrow \mathcal{H}_1, \mathcal{M}_1 \quad a, \mathcal{H}_1 \vdash_H B_2 \rightsquigarrow \mathcal{H}_2, \mathcal{M}_2}{a, \mathcal{H} \vdash_H (p, \Delta, B_1; B_2) \rightsquigarrow \mathcal{H}_2, \mathcal{M}_1 \cup \mathcal{M}_2}$$

Ultimately, heap usage is affected by two primitive heap instructions, namely `new` and `dispose`. Their heap effects can be captured by the following rules:

$$[\underline{\mathbf{HS{-}NEW}}]\quad \frac{\mathcal{H}_1 = \mathcal{H} + \mathit{enrich}(a, \Delta, \{(c, 1)\})}{a, \mathcal{H} \vdash_H (p, \Delta, \mathtt{new}\ c) \rightsquigarrow \mathcal{H}_1, \mathcal{H}_1}$$

$$[\underline{\mathbf{HS{-}DISPOSE}}]\quad \frac{\mathcal{H}_1 = \mathcal{H} + \mathit{enrich}(a, \Delta, \{(c, -1)\})}{a, \mathcal{H} \vdash_H (p, \Delta, \mathtt{dispose}\ c) \rightsquigarrow \mathcal{H}_1, \mathcal{H}}$$

As before, the *enrich* function incorporates heap usage effects 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:

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

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

$$\begin{array}{lll}
\{\mathbf{0}\} & =_{df} & \{(c, 0) \mid c \in \mathbf{ObjType}\} \\
\mathit{max}(\mathcal{B}_1, \mathcal{B}_2) & =_{df} & \{(c, \mathit{max}(\mathcal{B}_1(c), \mathcal{B}_2(c))) \mid c \in \mathbf{ObjType}\} \\
\mathcal{B}(c) & =_{df} & \mathit{if}\ (c, s) \in \mathcal{B}\ \mathit{then}\ s\ \mathit{else}\ 0
\end{array}$$

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

7. Discussion

In this section, we proceed with a brief discussion on two remaining important issues: (i) soundness of the inference system, and (ii) abstract states for objects.

We can formulate a safety theorem which proclaims that each method always executes without error from insufficient memory, when it is given memory resource equal to (or more than) its inferred bound.

THEOREM 1. *Consider program P with M as its main method with body E but without parameters. 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 $C_0 = \langle(\pi, \mathcal{F}, 2) : \Pi, \omega, \mathtt{ret}(E), s{-}\mathcal{F}, h\rangle$ satisfies the following conditions: (1) $\pi \models h \geq \mathcal{M}$ (2) $\pi \models s \geq \mathcal{S}$, 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 $s_1 \geq 0$, and $h_1 \geq \{\mathbf{0}\}$.*

Proof Sketch: Consider an operational semantics in small steps of the following form:

$$\langle f : \Pi, \omega, E, \mathtt{s}, \mathtt{h}\rangle \hookrightarrow \langle \Pi_1, \omega_1, E_1, \mathtt{s}_1, \mathtt{h}_1\rangle$$

$$\frac{I = \texttt{const}\langle t\rangle\ k \mid \texttt{load}\langle t\rangle\ i \mid \texttt{store}\langle t\rangle\ i}{a,\mathcal{H} \vdash_H (p,\Delta,I) \rightsquigarrow \mathcal{H},\mathcal{H}}\ [\mathbf{HS{-}PRIM}] \qquad \frac{a,\mathcal{H} \vdash_H B_1 \rightsquigarrow \mathcal{H}_1,\mathcal{M}_1 \quad a,\mathcal{H} \vdash_H B_2 \rightsquigarrow \mathcal{H}_2,\mathcal{M}_2}{a,\mathcal{H} \vdash_H (p,\Delta,\texttt{if}\ B_1\ B_2) \rightsquigarrow \mathcal{H}_1{\cup}\mathcal{H}_2,\mathcal{M}_1{\cup}\mathcal{M}_2}\ [\mathbf{HS{-}IF}]$$

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

$$\frac{\begin{array}{c} t\,m_1(t_1,..,t_n)\ l;\phi_{pr};\mathcal{F};\phi_{po};\mathcal{S};\mathcal{H}_u;\mathcal{M}_h\ \{B\} \in P \\ \rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^{n} \\ \mathcal{H}_1{=}\mathcal{H}{+}\mathit{enrich}(a,\Delta,\rho\mathcal{H}_u) \quad \mathcal{M}_1{=}\mathcal{H}{+}\mathit{enrich}(a,\Delta,\rho\mathcal{M}_h)\end{array}}{a,\mathcal{H} \vdash_H (p,\Delta,\texttt{invoke}\ m_1) \rightsquigarrow \mathcal{H}_1,\mathcal{M}_1}\ [\mathbf{HS{-}INVOKE}] \qquad \frac{\begin{array}{c} n,\{\mathbf{0}\} \vdash_H B \rightsquigarrow \mathcal{H},\mathcal{M} \\ \mathcal{H}_{rec}{=}\{m(\pi_1,..,\pi_n){=}\mathcal{H}\} \quad \mathcal{H}_{po}{=}\mathit{fixpt}(\mathcal{H}_{rec}) \\ \mathcal{M}_{rec}{=}\{m(\pi_1,..,\pi_n){=}\mathcal{M}\} \quad \mathcal{M}_{po}{=}\mathit{fixpt}(\mathcal{M}_{rec})\end{array}}{\begin{array}{l}\vdash_H \quad t\,m(t_1,..,t_n)\ l;\phi_{pr};\mathcal{F};\phi_{po};\mathcal{S}\ \{B\} \\ \qquad \rightsquigarrow t\,m(t_1,..,t_n)\ l;\phi_{pr};\mathcal{F};\phi_{po};\mathcal{S};\mathcal{H}_{po};\mathcal{M}_{po}\ \{B\}\end{array}}\ [\mathbf{HS{-}METH}]$$

Figure 6. Heap Usage and Bound Inference

where *f* : Π is the frame of stacks with *f* as its current frame. Each frame $f = (\pi, \mathcal{F}, p)$ contains an array π, the frame size $\mathcal{F}$, and a top frame pointer p. ω represents the heap, and *E* is the expression to be evaluated. This model contains two runtime instrumentations, namely: available stack (s) and available heap space (h). Our semantics will flag memory adequacy errors whenever s or h becomes negative. We can formalise a notion of well-formed annotations, and then show that our inference algorithm derives such annotations. With this, we can prove by co-induction over the operational semantics that each program with well-formed annotation never fail due to memory adequacy error, whenever sufficient heap/stack space are given. Note that conditions (1) and (2) state that the available stack and heap spaces in the initial state are not less than the method's inferred bounds. □

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 have provided an alias type system (similar to that used in [13]) 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. 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 number of nodes in 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⟨s, h⟩ where  s=1+left.s+right.s
                          h=1+max(left.h, right.h)
```

Given a method to compute the height of a tree:

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

Our inference system would derive:

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

where k_1 and k_2 are some integer constants and $\pi_1.h$ denotes the height of tree at π_1. Given another method to sum values of a tree, followed by the disposal of its nodes:

```
int sum(BNode t) l, 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:

$$\mathcal{F}{\equiv}k_3;\ \phi_{po}{\equiv}\pi'_3{=}\pi_1.s;\ \mathcal{S}{\equiv}\mathit{max}(k_4{\times}\pi_1.h, k_3);$$
$$\mathcal{H}{\equiv}\{(\texttt{BNode}, -\pi_1.s)\};\ \mathcal{M}{\equiv}\{\mathbf{0}\}$$

where k_3 and k_4 are some integer constants and the bag expression $\{(\texttt{BNode}, -\pi_1.s)\}$ denotes the recovery of BNode objects equal to the size of tree at π_1.

8. Experiments

A prototype for our inference system has been built to confirm the viability and practicality of our approach. We have carried our experiments to infer stack/heap bounds for a set of small programs with challenging recursion and for programs from the Scimark benchmark suite: Fast Fourier Transform, LU decomposition, Monte Carlo, Sparse Matrix Multiplication, Successive Over Relaxation [23]. The system uses the CIL infrastructure [22] to translate the C programs to an intermediate language. An additional preprocessing phase is needed for obtaining code in our assembly-level form (e.g. translating away some intraprocedural control-flow). Our prototype is built using the Glasgow Haskell Compiler [24] and makes use of the Omega constraint solving library [26] augmented with the disjunctive fixpoint analyzer from [25]. Our test platform was a Pentium 2.8 Ghz system with 2GB main memory, running Fedora Linux 4.0.

Figure 7 shows the statistics obtained for each program that we inferred. The program size is indicated in terms of number of lines of C code (Column 2). For each program, we present four timing measurements. Column 3 represents the time taken by the compilation of the original C program, while Column 4 represents the time taken by state inference (frame bound inference plus abstract state inference). Column 5 represents the time taken by frame bound inference, abstract state inference and stack inference. Lastly, Column 8 represents the time taken by frame bound inference, abstract state inference and heap inference. The time for abstract state inference roughly correlates with the program size and with the complexity of the relations between program variables. The additional time taken for stack inference was significant due to the intensive use of the stack by all of the programs. The time taken for heap

Benchmark Programs	Source (lines)	Compilation (secs)	State Inf. (sec)	Stack Analysis			Heap Analysis		
				Inf.(secs)	Estimation	Execution	Inf.(secs)	Estimation	Execution
ackermann	16	0.047	0.98	1.52	*	248	0.98	0	0
binary search	31	0.052	0.62	1.14	88	88	0.67	40	40
bubble sort	39	0.052	0.68	1.36	104	104	0.72	40	40
init array	5	0.047	0.23	0.42	64	64	0.26	40	40
queens	39	0.054	1.05	2.26	84	84	1.06	32	32
quick sort	43	0.054	1.66	4.26	624	624	1.68	400	400
FFT	336	0.108	17.52	35.81	856	846	17.74	128	128
LU Decomp.	191	0.080	9.20	27.89	580	580	11.64	480	480
Monte Carlo	36	0.038	0.27	0.33	24	24	0.27	0	0
SOR	84	0.062	2.54	5.09	540	540	2.61	400	400
Sparse Mult.	79	0.062	2.96	5.22	40292	40292	3.92	160800	160800

Figure 7. Experimental Results on Memory Bounds Inference

inference was less substantial, due to the nature of our programs. Most of the benchmarks used few heap objects, with the exception of the Sparse Multiplication benchmark.

We also provide results on the precision of our memory bounds analysis. For stack analysis (Columns 6,7) and heap analysis (Columns 9,10), we report the number of bytes corresponding to the statically computed memory watermark and the actual watermark obtained from the execution of the programs. All stack usage bounds were successfully 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. Nevertheless, the patterns of recursion that we have tested made full use of the power of our fixpoint analysis with support for disjunctions and relational analysis.

To gain more confidence in our system, we performed one more experiment with the Susan program described in [16]. Susan is an image processing package that uses more heap-allocated arrays to represent patterns for image recognition. To keep our results in linear form, we employed an annotation technique which can fix upper bounds for two constant parameters representing the width and the height of the image to be manipulated. With this manual annotation, we were able to capture bounds for both stack and heap. Overall, we can conclude that our initial experiments, though preliminary in nature, have confirmed the viability of our approach.

9. Related Work

Past research on memory usage prediction [21, 19] mainly focused on functional programs where data structures are mostly immutable and thus easier to handle. Hughes and Pareto [21] 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 the recovery mechanism used has coarser granularity since each region is only deleted when all objects in its region 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 a 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. [5] 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 assumes that source programs come from a first-order functional language with a resource-aware type system [19]. Similarly, Amadio et al. [4] 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. However, their focus is on termination verification, rather than actual inference on stack bounds. Furthermore, heap space was not considered. More recently, Cachera et al. [11] 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 for an unbounded number of times. This fact is only used to check if memory usage is theoretically bounded or not. However, this result is not sufficient for highly constrained system which requires precise upper bounds on memory usage.

There were also several works on analysing the stack space requirement of interrupt-driven programs. Brylow et al. [10] proposed stack size analysis using a context-free reachability algorithm based on model checking. Chatterjee et al. [12] 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.

Chin et. al. [13] proposed a modular memory usage verification system for object-oriented programs. The system can check whether 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. Without inference, this approach is impractical for low-level programs. Hofmann and Jost [20] proposed a type-based heap space analysis for Java-like source language with explicit deallocation. Their analysis is based on an amortised complexity analysis where a potential is assigned to each datum according to its size and layout. Heap space usage can then be calculated during the type inference based on the annotated potential for each input. More recently, Albert et. al. [3] reported a heap space analysis for Java Bytecode. Similar to our work, their analysis also makes use of a linear size analysis to help recover size relations for data structures under manipulation though other domains may be also used if appropriate constraint solvers are available. Different from type-based approaches, their analysis is based on a cost model reported in [2]. In their work, they propose to use escape analysis to refine the heap space inference, while our system supports the more fine-grain explicit deallocation. Their focus is mainly on heap space inference, while our system handles the inference for both stack and heap space bounds.

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

Independent of our work, Braberman et al. [8] deal with the memory consumption inference problem (for a Java-like imperative language) in the same proceedings by a different approach, where memory bound is modeled using a more expressive polynomial expression that is solved by resorting to Bernstein basis. As an extension to their earlier work [9], their approach now takes into account object deallocation via region-based memory. Their system handles non-recursive programs very well with the help of Daikon system [15] to generate loop invariants. For recursive programs, user annotations may be required. Different from their work, our system infers both stack and heap usage bounds for recursive methods (and loops) using fixpoint analyses.

10. Concluding Remarks

We have proposed a sound inference system for a structured assembly language useful for predicting the amount of memory space needed during program execution. Our system can infer both net usage and upper bound of stack/heap spaces required, for a reasonably wide range of programs. We use a special guarded expression form to track both memory usage and high watermark in a path-sensitive manner. Our approach can handle both recursion and loops. We use 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.

We envision our framework to be most useful whenever memory resources must 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.

Acknowledgement Alexandru Stefan helped with the experiments and provided insightful feedbacks on the paper. The reviewers of ISMM08 provided professional and helpful comments that led to improvements to the paper's presentation. This work is supported by A*STAR grant R-252-000-233-305 and a gift from Microsoft. Shengchao Qin is supported in part by the EPSRC project EP/E021948/1.

References

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

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *European Symposium on Programming (ESOP)*, March 2007.

[3] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proceedings of the International Symposium on Memory Management (ISMM '07)*, June 2007.

[4] 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, LNCS, September 2004.

[5] 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 LNCS, 2004.

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

[7] 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 LNCS, 2004.

[8] V. Braberman, F. Fernandez, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory Management (ISMM '08)*, Tucson, Arizona, June 2008.

[9] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology. Special Issue: ECOOP 2005 Workshop FTfJP*, 5(5):31–58, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] W.N. Chin, H.H. Nguyen, S.C. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Static Analysis Symposium*, Springer LNCS, London, UK, September 2005.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL*, pages 84–96, 1978.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th IEEE International Workshop on Workload Characteristics*, 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 POPL*, New Orleans, Louisiana, January 2003.

[20] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming (ESOP)*, Vienna, Austria, March 2006.

[21] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *ICFP*, September 1999.

[22] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. 2002.

[23] National Institute of Standards and Technology. Java SciMark benchmark for scientific computing. `http://math.nist.gov/scimark2/`.

[24] S. Peyton-Jones and et al. Glasgow Haskell Compiler. Available at http://www.haskell.org/ ghc.

[25] C. Popeea and W.N. Chin. Inferring disjunctive postconditions. In *Asian Computing Science Conference (ASIAN)*, volume 4435 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2006.

[26] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[27] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*, Springer LNCS, August 2006.

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