Parametric Heap Usage Analysis for Functional Programs *

Leena Unnikrishnan Scott D. Stoller

Computer Science Department
Stony Brook University
{leena,stoller}@cs.stonybrook.edu

Abstract

This paper presents an analysis that derives a formula describing the worst-case live heap space usage of programs in a functional language with automated memory management (garbage collection). First, the given program is automatically transformed into *bound functions* that describe upper bounds on the live heap space usage and other related space metrics in terms of the sizes of function arguments. The bound functions are simplified and rewritten to obtain recurrences, which are then solved to obtain the desired formulas characterizing the worst-case space usage. These recurrences may be difficult to solve due to uses of the *maximum* operator. We give methods to automatically solve categories of such recurrences. Our analysis determines and exploits monotonicity and monotonicity-like properties of bound functions to derive upper bounds on heap usage, without considering behaviors of the program that cannot lead to maximal space usage.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Performance, Verification

Keywords Live Heap Space Analysis, Functional Languages, Garbage Collection, Recurrence Relations

1. Introduction

Analysis of the time and space requirements of computer programs is important for virtually all computer applications, especially in embedded systems, real-time systems, and interactive systems. The importance of time analysis for real-time and embedded systems is reflected in the long tradition of research on worst-case execution time (WCET) analysis. Space usage is also critical in many real-time and embedded systems, due to the limited amount of memory and the potential for severe consequences if the system fails due to insufficient memory. Due in part to the difficulty of predicting the space usage of programs that use dynamic memory allocation, real-time and embedded software typically use only statically allocated data structures. However, this approach has disadvantages. First, it can lead to poor utilization of memory, when one data structure is full and another has space to spare. Second, it requires explicit memory management: the programmer must keep track of which array entries are in use and when it is safe to mark an occupied entry as available; this can be difficult for entries that may be used by multiple parts of the program and referenced from multiple data structures. Of course, these are some of the reasons for the growing popularity of languages with automatic memory management (Java, C#, Python, Ruby, etc.) in non-embedded systems. Versions of these languages suitable for embedded systems exist and are being promoted, e.g., Java Platform, Micro Edition (JavaME) and Microsoft .NET Micro Framework. While advances in real-time garbage collection (e.g., [5]) play an essential role in making such languages practical, analyses that can accurately predict the worst-case space usage of programs written in garbage-collected languages are also important for their adoption in embedded systems [13].

Space analysis is important for determining space requirements and for accurate execution-time analysis. For example, analysis of worst-case execution time in real-time systems often uses loop bounds or recursion depths, both of which are commonly determined by the size of the data being processed. Space analysis can also help determine the timing effects of memory-related events such as memory allocation, garbage collection, cache misses, and page faults.

This paper describes a general approach for automatic accurate analysis of heap space usage, specifically, the maximum size of live data on the heap during execution. In other words, the analysis determines heap usage of programs in the presence of perfect garbage collection where garbage is collected as soon as it is created. This result is the minimum amount of heap space needed to run a program, no matter which garbage collection scheme is used. Limiting a program to this "minimum amount of heap" in the presence of imperfect garbage collection schemes, simply means that garbage collection needs to be performed intermittently to free up space for allocation. The analysis can easily be modified to determine related metrics, such as space usage when garbage collection is performed only at fixed points in the program. It can also be adapted to analyze the space usage of continuously running processes with cyclic behavior. Our analysis is designed for a functional language. This is a simplification that allows us to focus on the fundamental issues first, deferring the complications needed to handle imperative updates.

Our approach starts with a program P written in a functional language with garbage collection. We construct, for every function f in P, bound functions that describe worst-case live heap space usage of f and other measures—such as the size of the result of f—necessary to determine the space usage of the program. The inputs to bound functions for f are the sizes of inputs to f. Bound functions are simplified and rewritten to recurrence relations, which are solved into closed form expressions that describe live heap

* This work was supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grants CCF-0613913, CNS-0627447, CNS-0831298, and CNS-0509230.

ISMM'09, June 19–20, 2009, Dublin, Ireland.

space usage and other measures in terms of sizes of inputs to the corresponding original functions. The basic version of our analysis is limited to programs that manipulate non-nested lists, i.e., lists in which every element is a value of a primitive data type. Section 12 describes how to extend the analysis to accommodate other data types.

Solving recurrence relations can be difficult. The availability of increasingly sophisticated recurrence solvers [21, 6], is an asset to our analysis. We use methods such as elimination of redundant arguments from recursive bound functions and recurrences to improve solvability. This may even eliminate dependence on, and hence the need to solve, certain recurrences. Since we are interested in worst-case heap space usage, bound functions and their recurrences may contain the max operator, which is typically not handled (or not handled exactly) by existing recurrence solvers such as [21, 6]. We use templates based on pattern matching and inductive reasoning to solve such recurrences. Results of some bound functions are used as arguments to others. Maximizing the results of the former bound functions leads to maximal live heap space usage for the program only if the latter bound functions are monotonically increasing with respect to their arguments. Monotonicity properties of bound functions are ascertained from their closed form solutions, if known, and using templates for bound functions still expressed as recurrences.

2. Language

We formalize the analysis for a first-order, call-by-value functional language that has literal values of primitive types (e.g., Boolean and integer constants), operations on primitive types (e.g., addition and subtraction), data constructors (e.g., *null* for the empty list, and *cons* to construct lists), testers (which test whether a value is built using a particular constructor), selectors (which extract parts of data structures), conditionals, bindings (i.e., local variable declarations), and function calls. A program is a set of function definitions of the form $f(v_1, ..., v_n) = e$, where an expression e is given by the grammar

$e ::=$	v	variable
$\mid$	l	literal
$\mid$	$cons(e_1, e_2)$	constructor application
$\mid$	$prim(e_1, \ldots, e_n)$	primitive operation
$\mid$	$null?(e)$	tester
$\mid$	$car(e)$	first element of list
$\mid$	$cdr(e)$	tail of list
$\mid$	$\mathbf{if}\ p\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	conditional expression
$\mid$	$\mathbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \mathbf{in}\ e$	binding expression
$\mid$	$f(e_1, \ldots, e_n)$	function application

Predicates in conditional expressions are given by the grammar

$$p ::= v \mid l \mid prim(p_1, \ldots, p_n) \mid null?(p) \mid car(p) \mid cdr(p)$$

The basic version of our analysis supports recursive functions but not mutual recursion. Section 12 describes how to handle mutual recursion.

Our analysis makes use of type information, which may be obtained from type declarations or from automated type inference. $type(v)$ denotes the type of variable v. $rtype(f)$ denotes the return type of function f. The allowed types are *Bool*, *Int* and *List*.

3. Definitions

Data Sizes. The size of an empty list is 0. The sizes of all other atomic objects (booleans and numbers) are their values. The size of a list is its length.

Argument-expression (arg-exp). An arg-exp is an expression whose result is an argument to a function call. In a function application $f(e_1, \ldots, e_n)$ of the function $f(v_1, \ldots, v_n)$, expressions $e_1, \ldots, e_n$ are arg-exps of the call to f. For $i \in [1, n]$, e_i *corresponds* to parameter v_i of f.

Argument-uses (arg-uses) of functions. An arg-use of a function f is an occurrence of an application of f in an arg-exp. For example, $g(1 + f(x), 2)$ contains an arg-use of f which *corresponds* to the first parameter of g. Arg-uses of expressions, e.g., $g(\mathbf{if}\ p\ \mathbf{then}\ e1\ \mathbf{else}\ e2)$, are defined similarly.

Argument-users (arg-users) of functions. In a function application $f(e_1, \ldots, e_n)$, if e_i contains an arg-use of a function g, then f is called an arg-user of g. For example, in $f(1 + g(x), 2)$, f is an arg-user of g. Arg-users of expressions are defined similarly.

Dependence. A function f is dependent on a function g if there is a path from f to g in the call graph of f.

Monotonicity. Consider function $f : X_1 \times X_2 \ldots \times X_m \rightarrow Y$, where X_1, ..., X_m, and Y are sets of whole numbers, integers or real numbers. f is monotonically increasing with respect to its i^{th} argument, if

$$\forall\, a, b \in X_i, c_1 \in X_1, \ldots, c_m \in X_m.\ a \leq b \Rightarrow$$
$$f(c_1, c_2, \ldots, a, \ldots, c_m) \leq f(c_1, c_2, \ldots, b, \ldots, c_m)$$

where a and b are the i'th argument of f. We use the terms *monotonic* and *monotonically increasing* interchangeably.

4. Bound Functions

Consider function $f(v_1, ..., v_n) = e_f$ in input program P. We define three bound functions for f. Bound functions of f take as arguments, the sizes of arguments $v_1, ..., v_n$ of f. We call these *size arguments* and denote them by $v_{1_s}, ..., v_{n_s}$. The *live heap space bound function* of f, S_f, called simply the S function of f, describes the worst-case heap usage of f over all possible combinations and values of arguments $v_1, ..., v_n$ having sizes $v_{1_s}, ..., v_{n_s}$, respectively.

$$S_f(v_{1_s}, \ldots, v_{n_s}) = \underset{\substack{\text{all args } a_1,\ldots,a_n \text{ to } f \\ s.t.\ \forall i \in [1,n].\ |a_i| = v_{i_s}}}{\text{upper bound}} \text{(min heap to evaluate } f(a_1, ..., a_n))$$

The worst-case heap usage corresponds to the maximum size of the live heap seen during the evaluation of f. This is also the minimum heap space required to evaluate f on arguments of sizes $v_{1_s}, ..., v_{n_s}$, no matter which garbage collection scheme is used. A heap object is *live* as long as it can be reached from the program through function arguments or the reference to the result of the most recently evaluated expression. The live heap space usage of program P is described by S_g, where g is the entry function of P.

The *new result-space bound function* of f, also called the N function of f, denoted N_f, describes the newly-allocated space in the result of f in terms of the sizes of f's arguments, and the amounts of newly-allocated space in these arguments. For function call $f(e_1, ..., e_n)$, N_f returns the number of new heap cells in the result r of the call. A heap cell in r is new if it is created after the start of evaluation of $f(e_1, ..., e_n)$, i.e., created in the body of f or in one of the arg-exps $e_1, ..., e_n$. So, in addition to its *size* arguments, N_f also has *newsize* arguments, denoted $v_{1_w}, ..., v_{n_w}$, representing the number of new heap cells in each argument. For $f(e_1, ..., e_n)$, v_{i_w} is the number of heap cells in v_i created during evaluation of expression e_i. v_{i_w} is meaningful only if v_i is a list; if not, then there are no heap cells in v_i and v_{i_w} is not required or

used.

$$N_f(v_{1_s},\ldots,v_{n_s},v_{1_w},\ldots,v_{n_w}) = \underset{\substack{\text{all args } a_1,\ldots,a_n \text{ to } f \\ s.t.\ \forall i\in[1,n].\ |a_i|=v_{i_s} \\ |\text{new heap in } a_i|\le v_{i_w}}}{\text{upper bound}} \begin{bmatrix}\text{amount of new heap in the result of} \\ f(a_1,...,a_n)\end{bmatrix}$$

The *size bound function* of f, also called the R function of f, denoted R_f, describes the size of the result of f in terms of the sizes of f's arguments.

$$R_f(v_{1_s},...,v_{n_s}) = \underset{\substack{\text{all args } a_1,\ldots,a_n \text{ to } f \\ s.t.\ \forall i\in[1,n].\ |a_i|=v_{i_s}}}{\text{upper bound}} |f(a_1,\ldots,a_n)|$$

The bodies of S_f, N_f, and R_f are obtained by applying transformations $\mathcal{S}$, $\mathcal{N}$, and $\mathcal{R}$, respectively, to the body e_f of f. The definitions of transformations $\mathcal{S}$, $\mathcal{N}$and $\mathcal{R}$. are provided in Sections 5, 6, and 7.

$\mathcal{S}[\![e]\!]$**:** Computes an upper bound on the live heap required for evaluation of e.

$\mathcal{N}[\![e]\!]$**:** Computes an upper bound on the number of new heap cells in result r of e. If e occurs in the body of function f, then a heap cell in r is new if it is created in e, or if it is new in $v_1,...,v_n$, which can easily be determined using $v_{1_w},...,v_{n_w}$. Since there is no imperative update in our target language, any node that points to another node must be newer than the latter. So, for a list argument v_i, only the first v_{i_w} nodes are new.

$\mathcal{R}[\![e]\!]$**:** Computes an upper bound on the size of the result of e. $\mathcal{R}[\![e]\!]$ is boolean if e returns a boolean and is numeric if e returns a number or a list.

5. Live Heap Space Bound Functions

Transformation $\mathcal{S}$, defined in Figure 1, is used to derive live heap space bound functions. In the $\mathcal{S}$ transformation of conditionals, if the predicate contains car then its truth value is unknown, because the analysis does not track the values of list elements. The heap space required to evaluate the conditional is the maximum of that required for the branches (the grammar in Section 2 implies that predicates in conditional expressions do not perform heap allocation). If the predicate does not contain car, then we maintain the conditional structure, and transform the predicate and the branches. Such predicates are often tests of recursive functions that control recursion, e.g., base case tests or tests that determine the next recursion step. The transformed conditionals in bound functions eventually (after simplification) become multiple cases in the definition of a recurrence. The original predicate p in the conditional expression is transformed into a predicate over size arguments. $\mathcal{N}$ and $\mathcal{R}$ transformations of conditionals are similar.

In function application $f(e_1,...,e_n)$, arg-exps $e_1,...,e_n$ are evaluated in order, followed by the call to f. The heap usage of $f(e_1,...,e_n)$ is the maximum of the heap usages of the arg-exps $e_1,...,e_n$ and the heap usage of f when called with arguments whose sizes are those of the results of $e_1,...,e_n$. When $e_i,\ i\in[2,n]$, is evaluated, the results of the previous arg-exps are live. So, the maximum size of the live heap during evaluation of e_i is the sum of the newly-allocated space in the results of previous arg-exps and $\mathcal{S}[\![e_i]\!]$. $\mathcal{N}[\![e_i]\!][0/v_w]$ is the amount of new heap in v_i that is allocated in e_i. Newsize arguments are neither required nor applicable and they are substituted with 0s. The function call is evaluated after all arg-exps, and this takes space $S_f(\mathcal{R}[\![e_1]\!],...,\mathcal{R}[\![e_n]\!])$.

See Figure 4 for examples of functions in an input program and corresponding S functions.

For $f(v_1,...,v_n)=e,\ S_f(v_{1_s},...,v_{n_s})=\mathcal{S}[\![e]\!]$.

$$\mathcal{S}[\![l]\!]=\mathcal{S}[\![nil]\!]=\mathcal{S}[\![v]\!]=0$$

$$\mathcal{S}[\![cons(e_1,e_2)]\!]=1+max(\mathcal{S}[\![e_1]\!],\mathcal{S}[\![e_2]\!])$$

$$\mathcal{S}[\![prim(e_1,...,e_n)]\!]=max(\mathcal{S}[\![e_1]\!],...,\mathcal{S}[\![e_n]\!])$$

$$\mathcal{S}[\![null?(e)]\!]=\mathcal{S}[\![car(e)]\!]=\mathcal{S}[\![cdr(e)]\!]=\mathcal{S}[\![e]\!]$$

$$\mathcal{S}[\![\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2]\!] = \begin{cases} max(\mathcal{S}[\![e_1]\!],\mathcal{S}[\![e_2]\!]), & \text{if } p \text{ contains } car \\ \textbf{if } \mathcal{R}[\![p]\!] \textbf{ then } \mathcal{S}[\![e_1]\!] \textbf{ else } \mathcal{S}[\![e_2]\!], & \text{otherwise}\end{cases}$$

$$\begin{aligned}&\mathcal{S}[\![\textbf{let } v_1=e_1,\ \ldots,\ v_n=e_n \textbf{ in } e]\!] \\ &= max(\mathcal{S}[\![e_1]\!], \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]+\mathcal{S}[\![e_2]\!], \\ &\qquad \vdots \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]+\ldots+\mathcal{N}[\![e_{n-1}]\!][0/v_w]+\mathcal{S}[\![e_n]\!], \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]+\ldots+\mathcal{N}[\![e_n]\!][0/v_w]+ \\ &\qquad \mathcal{S}[\![e]\!][\mathcal{R}[\![e_1]\!]/v_{1_s},...,\mathcal{R}[\![e_n]\!]/v_{n_s}])\end{aligned}$$

$$\begin{aligned}&\mathcal{S}[\![f(e_1,...,e_n)]\!] \\ &= max(\mathcal{S}[\![e_1]\!], \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]+\mathcal{S}[\![e_2]\!], \\ &\qquad \vdots \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]...+\mathcal{N}[\![e_{(n-1)}]\!][0/v_w]+\mathcal{S}[\![e_n]\!], \\ &\qquad \mathcal{N}[\![e_1]\!][0/v_w]...+\mathcal{N}[\![e_n]\!][0/v_w]+ \\ &\qquad S_f(\mathcal{R}[\![e_1]\!],...,\mathcal{R}[\![e_n]\!]))\end{aligned}$$

Figure 1. Transformation $\mathcal{S}$. $e[0/v_w]$ substitutes every newsize argument v_w in e with 0.

6. New Result-Space Bound Functions

Transformation $\mathcal{N}$, defined in Figure 2, produces new result space-bound functions. If function f in the input program returns primitive data, then its result uses no heap space and N_f returns 0. N functions exploit the fact that, in the absence of imperative update, newer heap cells reference older heap cells. So, if a list v contains v_w new heap cells, it is the first v_w cells of v that are new. Newsize arguments are decremented in recursive calls to N functions if corresponding size arguments are, e.g., $\mathcal{N}[\![f(cdr(ls))]\!]$ is $N_f(ls_s-1,ls_w-1)$. Recursion often proceeds until $ls_s=0$, which implies decrementing ls_s from ls_w. Only the first ls_w of ls_s cells are new. So, all decrements to newsize arguments are enclosed by applications of $max(0,...)$. See Figure 4 for examples of functions in an input program and corresponding N functions.

N functions may sometimes yield recurrences that are difficult to solve, because of the large number of arguments (size and newsize arguments) and the presence of $max(0,...)$ expressions. If N_f cannot be solved, we redefine it as $N_f(v_{1_s},...,v_{n_s}) = R_f(v_{1_s},...,v_{n_s})$. In other words, instead of determining how much of the list returned by f is new, we over-approximate, saying that the whole list is new. In all our examples, these simpler N functions, wherever used, do not introduce overestimations in the analysis result. The redefinition does not lose accuracy in practice, because our analysis determines *worst-case* space usage, and in the worst case, most functions either build completely new lists or the arguments given to functions are completely new; in both cases, all cells in the result lists are new. This is in keeping with the following observation:

$$N_f(v_{1_s},...,v_{n_s},v_{1_s},...,v_{n_s}) = R_f(v_{1_s},...,v_{n_s})$$

where the values of the newsize arguments of N_f equal those of the corresponding size arguments.

For $f(v_1, ..., v_n) = e$,

$$N_f(v_{1_s}, ..., v_{n_s}, v_{1_w}, ..., v_{n_w}) = \begin{cases} 0, \text{ if } rtype(f) \neq List \\ \mathcal{N}[\![e]\!], \text{ otherwise} \end{cases}$$

$$\mathcal{N}[\![l]\!] = \mathcal{N}[\![nil]\!] = 0$$

$$\mathcal{N}[\![v]\!] = \begin{cases} 0, \text{ if } type(v) \neq List \\ v_w, \text{ otherwise} \end{cases}$$

$$\mathcal{N}[\![cons(e_1, e_2)]\!] = 1 + \mathcal{N}[\![e_2]\!]$$

$$\mathcal{N}[\![prim(e_1, ..., e_n)]\!] = 0$$

$$\mathcal{N}[\![null?(e)]\!] = \mathcal{N}[\![car(e)]\!] = 0$$

$$\mathcal{N}[\![cdr(e)]\!] = max(0, \mathcal{N}[\![e]\!] - 1)$$

$$\mathcal{N}[\![\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2]\!] = \begin{cases} max(\mathcal{N}[\![e_1]\!], \mathcal{N}[\![e_2]\!]), \text{ if } p \text{ contains } car \\ \textbf{if } \mathcal{R}[\![p]\!] \textbf{ then } \mathcal{N}[\![e_1]\!] \textbf{ else } \mathcal{N}[\![e_2]\!], \text{ otherwise} \end{cases}$$

$$\mathcal{N}[\![\textbf{let } v_1 = e_1, \ \ldots, \ v_n = e_n \textbf{ in } e]\!] = (\mathcal{N}[\![e]\!])\ [\mathcal{R}[\![e_1]\!]/v_{1_s}, ..., \mathcal{R}[\![e_n]\!]/v_{n_s}, \mathcal{N}[\![e_1]\!]/v_{1_w}, ..., \mathcal{N}[\![e_n]\!]/v_{n_w}]$$

$$\mathcal{N}[\![f(e_1, \ldots, e_n)]\!] = \begin{cases} 0, \text{ if } rtype(f) \neq List \\ N_f(\mathcal{R}[\![e_1]\!], ..., \mathcal{R}[\![e_n]\!], \mathcal{N}[\![e_1]\!], ..., \mathcal{N}[\![e_n]\!]), \text{ otherwise} \end{cases}$$

Figure 2. Transformation $\mathcal{N}$.

For $f(v1, ..., vn) = e$, $R_f(v_{1_s}, ..., v_{n_s}) = \mathcal{R}[\![e]\!]$

$$\mathcal{R}[\![l]\!] = l, \ \mathcal{R}[\![nil]\!] = 0, \ \mathcal{R}[\![v]\!] = v_s$$

$$\mathcal{R}[\![cons(e_1, e_2)]\!] = 1 + \mathcal{R}[\![e_2]\!]$$

$$\mathcal{R}[\![prim(e_1, ..., e_n)]\!] = prim(\mathcal{R}[\![e_1]\!], ..., \mathcal{R}[\![e_n]\!])$$

$$\mathcal{R}[\![null?(e)]\!] = eq?(\mathcal{R}[\![e]\!], 0)$$

$$\mathcal{R}[\![car(e)]\!] = uk, \ \mathcal{R}[\![cdr(e)]\!] = \mathcal{R}[\![e]\!] - 1$$

$$\mathcal{R}[\![\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2]\!] = \begin{cases} max(\mathcal{R}[\![e_1]\!], \mathcal{R}[\![e_2]\!]), \text{ if } p \text{ contains } car \\ \textbf{if } \mathcal{R}[\![p]\!] \textbf{ then } \mathcal{R}[\![e_1]\!] \textbf{ else } \mathcal{R}[\![e_2]\!], \text{ otherwise} \end{cases}$$

$$\mathcal{R}[\![\textbf{let } v_1 = e_1, \ \ldots, \ v_n = e_n \textbf{ in } e]\!] = (\mathcal{R}[\![e]\!])\ [\mathcal{R}[\![e_1]\!]/v_{1_s}, \ldots, \mathcal{R}[\![e_n]\!]/v_{n_s}]$$

$$\mathcal{R}[\![f(e_1, \ldots, e_n)]\!] = \mathcal{R}_f(\mathcal{R}[\![e_1]\!], \ldots, \mathcal{R}[\![e_n]\!])$$

Figure 3. Transformation $\mathcal{R}$.

7. Size Bound Functions

Transformation $\mathcal{R}$ in Figure 3 produces size bound functions; recall that R_f bounds the size of the result (return value) of f. See figure 4 for examples of functions in an input program and corresponding R functions. From the definitions of transformations $\mathcal{S}$, $\mathcal{N}$, and $\mathcal{R}$, it can be seen that results of R functions are used by S and N functions only as the values of size arguments. Also, recall that we are interested only in the maximal value of the input program's S function. Consider the $\mathcal{R}$ transformation of conditionals. Predicates containing car are concluded as being undeterminable and in such cases, the size of the result of the conditional is the maximum of the sizes of e_1 and e_2, $max(\mathcal{R}[\![e_1]\!], \mathcal{R}[\![e_2]\!])$. Ideally, we should return that size which helps S functions determine the worst-case (maximal) space usage of the input program. But it is impossible to know which size to return, in the general case. Further, it is not tractable to return all sizes. Thus, the use of max in the definition of bound functions is a necessary approximation, and based on the somewhat reasonable expectation that the maximum size leads to maximum space-usage results of dependent S and N functions. We perform checks, namely MC1, MC2, and MC3 described later in this section, to verify if this is indeed the case, i.e., if the use of max in R functions results in dependent S and N functions determining true upper bounds on space measures. Transformation $\mathcal{R}$, along with checks MC1, MC2, and MC3, ensure that all R functions, including ones whose definitions do not contain max but are dependent on R functions that do, uniformly return upper bounds (maximums) on size measures.

Composite recurrences. A composite recurrence is a recurrence (or a definition of a bound function) which contains an application of max in which at least one arg-exp contains a recursive call to the function defined by the recurrence. A regular recurrence is one that does not contain such occurrences of max. The R function R_{append} of *append*, which appends one list to another, is regular, while R_{lls}, the R function of *lesserorequal-list* is composite. *lesserorequal-list*, an auxiliary function of *quick-sort*, takes a list ls and a cutoff x as arguments and returns a new list containing the elements of ls that are less than or equal to x.

$$R_{append}(n, m) = \begin{cases} m \text{ if } n = 0 \\ 1 + R_{append}(n-1, m) \text{ otherwise} \end{cases}$$

$$R_{lls}(n) = \begin{cases} 0 \text{ if } n = 0 \\ max(1 + R_{lls}(n-1), \ R_{lls}(n-1)) \text{ otherwise} \end{cases}$$

Given recurrence T, we define recurrence T^{all} that has the same arguments as T and returns the set of results obtained by using the value of each argument of max in place of the max expression in T. Note that this is done recursively, not only at the top-level call. T, on the other hand, uses only the largest of the arguments to max at every recursion level.

Intuitively, T captures only the maximum value of the size or space measure for a function in the input program, while T^{all} captures all possible values of the size or space measure for that function. We can generate the definition of T^{all} from the definition of T, by replacing each occurrence of max with $union$ and replacing every function call $F(\ldots)$ with $F^{all}(\ldots)$; note that upper case letters (F, G, H) are used to denote bound functions. These replacements in the definition of T need to be accompanied by modifications that ensure type-safety, for instance, the replacement of primitive operators and functions with ones that are overloaded to handle regular argument values as well as sets of argument values. For example, for sets X and Y of argument values to f, $f(X, Y) = \{f(x, y) \mid x \in X, y \in Y\}$, and $f(x, Y) = \{f(x, y) \mid y \in Y\}$. Recurrences R_{lls} and R^{all}_{lls} and their solutions are shown below.

$$\begin{aligned} R_{lls}(n) &= \begin{cases} 0 \text{ if } n = 0 \\ max(1 + R_{lls}(n-1), \ R_{lls}(n-1)) \text{ otherwise} \end{cases} \\ &= n \end{aligned}$$

$$\begin{aligned} R^{all}_{lls}(n) &= \begin{cases} \{0\} \text{ if } n = 0 \\ \cup(1 + R^{all}_{lls}(n-1), \ R^{all}_{lls}(n-1)) \text{ otherwise} \end{cases} \\ &= \{n, n-1, \ldots, 0\} \end{aligned}$$

A composite recurrence T is *simple* if the solution of T^{all} is a singleton set for all evaluations of its arguments; otherwise, we say that T is *complex*. If R_f is a simple composite recurrence, then the size of the result of f is uniquely determined by the sizes of its arguments. If R_f is a complex composite recurrence, then the size of the result of f depends on the actual elements in list arguments of f. The solution of R^{all}_{lls} is not a singleton set and therefore, R_{lls} is complex. The R function R_{ins} of *insert* which inserts a number

insertion-sort	*insert*
insertion-sort$(ls) = \textbf{if } null?(ls) \textbf{ then } nil$ $\textbf{else } insert(car(ls),$ $\quad insertion\text{-}sort(cdr(ls)))$	*insert*$(x, ls) = \textbf{if } null?(ls) \textbf{ then } cons(x, nil)$ $\textbf{else if } lessoreq?(x, car(ls))$ $\quad \textbf{then } cons(x, ls)$ $\quad \textbf{else } cons(car(ls),\ insert(x, cdr(ls)))$
$S_{is}(ls_s) = \textbf{if } ls_s = 0 \textbf{ then } 0$ $\textbf{else } max(S_{is}(ls_s - 1),$ $\quad N_{is}(ls_s - 1) + S_{ins}(R_{is}(ls_s - 1)))$ $= \begin{cases} 0 \text{ if } ls_s = 0 \\ max(S_{is}(ls_s - 1),\ 2ls_s - 1) \text{ otherwise} \end{cases}$ $= \begin{cases} 0 \text{ if } ls_s = 0 \\ 2ls_s - 1 \text{ otherwise} \end{cases}$	$S_{ins}(ls_s) = \textbf{if } ls_s = 0 \textbf{ then } 1$ $\textbf{else } max(1,\ 1 + S_{ins}(ls_s - 1))$ $= \begin{cases} 1 \text{ if } ls_s = 0 \\ 1 + S_{ins}(ls_s - 1) \text{ otherwise} \end{cases}$ $= ls_s + 1$
$N_{is}(ls_s) = \textbf{if } ls_s = 0 \textbf{ then } 0 \textbf{ else } N_{ins}(R_{is}(ls_s - 1))$ $= ls_s$	$N_{ins}(ls_s) = R_{ins}(ls_s) = 1 + ls_s$
$R_{is}(ls_s) = \textbf{if } ls_s = 0 \textbf{ then } 0 \textbf{ else } R_{ins}(R_{is}(ls_s - 1))$ $= \begin{cases} 0 \text{ if } ls_s = 0 \\ 1 + R_{is}(ls_s - 1) \text{ otherwise} \end{cases}$ $= ls_s$	$R_{ins}(ls_s) = \textbf{if } ls_s = 0 \textbf{ then } 1 + 0$ $\textbf{else } max(1 + ls_s,\ 1 + R_{ins}(ls_s - 1))$ $= \begin{cases} 1 \text{ if } ls_s = 0 \\ max(1 + ls_s,\ 1 + R_{ins}(ls_s - 1)) \text{ otherwise} \end{cases}$ $= 1 + ls_s$

Figure 4. Entry function *insertion-sort* and auxiliary function *insert* of program *insertion-sort*, followed by their bound functions, recurrences (if any) obtained from bound functions, and closed form solutions of recurrences.

into a sorted list is a simple composite recurrence, as shown below.

$$R_{ins}(n) = \begin{cases} 1 \text{ if } n = 0 \\ max(1 + n,\ 1 + R_{ins}(n - 1)) \text{ otherwise} \end{cases}$$

$$R^{all}_{ins}(n) = \begin{cases} \{1\}, \text{ if } n = 0 \\ \cup(\{1 + n\},\ 1 + R^{all}_{ins}(n - 1)) \text{ otherwise} \end{cases}$$
$$= \{1 + n\}$$

We determine if composite recurrences are simple or complex using templates. This is described in Section 9.

Complex* R *functions. An R function is *complex* if it is defined by a complex composite recurrence or it depends on another complex R function or its solution contains an application of max over closed forms; otherwise, we say that the R function is *simple*. R functions R_{append} and R_{ins} are simple, while R_{lls} is complex. R_{qs}, shown below, is a complex R function even though the recurrence defining it is regular because of the dependence on complex R function R_{lls}.

$$R_{qs}(ls_s)$$
$$= \begin{cases} ls_s \text{ if } ls_s = 0 \text{ or } ls_s = 1 \\ 1 + R_{qs}(R_{lls}(ls_s - 1)) + R_{qs}(R_{gls}(ls_s - 1)) \text{ otherwise} \end{cases}$$

8. Conditions on Uses of Complex R Functions

We observe that for an R function F that does not depend on any complex R functions (in other words, does not call any R function or depends only on simple R functions), $F = max\ o\ F^{all}$. Consider the following definition of a bound function G that depends on a complex R function F, for which $F = max\ o\ F^{all}$.

$$G(\ldots) = \ldots\ F(e_1, \ldots, e_n)\ \ldots$$

From the discussion in Section 7, the desired result of G is that of $max\ o\ G^{all}$, whether G is an S or N or R function. For this to be true, in place of $F(e_1, \ldots, e_n)$ in the body of G, we should use the specific size $s \in F^{all}(e_1, \ldots, e_n)$ that maximizes the result of evaluation of the body of G. This desired size s is not necessarily the maximum element in $F^{all}(e_1, \ldots, e_n)$, i.e., it is not always the same as the result of $F(e_1, \ldots, e_n)$ (since $F = max\ o\ F^{all}$). Consider function $G_1(v) = v - F_1(v)$ as an example. The expression '$v - F_1(v)$' is the result of multiple rounds of simplification, and this explains the occurrence of the result of an R function in a non-recursive expression that defines G_1 which could be an S or N function. The result of $max\ o\ G^{all}_1$, which is the desired result of G_1, is obtained by replacing '$F_1(v)$' in the body of G with the minimum element of $F^{all}_1(v)$. Thus, the derived definitions of bound functions such as G_1 do not necessarily yield the desired upper bounds because of the use of max in called R functions. So, for every complex R function F for which $F = max\ o\ F^{all}$, we perform the following checks to ensure that the definitions of bound functions dependent on F yield upper bounds on relevant space and size measures.

MC1 All expressions containing calls to F are monotonically increasing with respect to the calls to F (e.g., $1 + F(\ldots)$ is monotonic with respect to the call to F, but $n - (1 + F(\ldots))$ is not. This check ensures that values of size arguments of S and N functions, determined using expressions containing applications of complex R functions, are maximized.

MC2 For every arg-use of F by a function H, occurring in the body of G, the solution of H should be monotonically increasing with respect to the argument corresponding to the arg-use of F. An example from *quick-sort*:

$$N_{qs}(ls_s,\ ls_w)$$
$$= \ldots\ N_{app}(\ldots R_{lls}(ls_s - 1),\ \ldots R_{gls}(ls_s - 1))$$

qs, *app*, lls, and gls are abbreviations for *quick-sort*, *append*, *lesserorequal-list*, and *greater-list*, respectively. R_{lls} and R_{gls} are complex R functions. So, the above definition of N_{qs} is correct only if N_{app} is monotonically increasing with respect to both its arguments (which it is, because $N_{app}(ls_{1_s}, ls_{2_w}) = ls_{1_s} + ls_{2_w}$).

MC3 For every arg-use of F by function G, occurring in the body of G, we need to ascertain from the definition of G, that the use of the result of F (in other words, $max\ o\ F^{all}$) yields the desired

maximum value, namely the result of $max\ o\ G^{all}$. An example from *quick-sort*:

$$S_{qs}(ls_s) = \begin{cases} ls_s \text{ if } ls_s = 0 \text{ or } ls_s = 1 \\ max(2^{ls_s-1} + ls_s - 1 + S_{qs}(R_{gls}(ls_s - 1)), \\ \quad 3.2^{ls_s-1} - 2), \text{ otherwise} \end{cases}$$

The definition of S_{qs} contains the recursive call $S_{qs}(R_{gls}(ls_s - 1))$. R_{gls} is a complex R function. The result of R_{gls} (equal to that of $max\ o\ R^{all}_{gls}$) can be used only if we can ascertain that this will result in the body of S_{qs} evaluating to $max\ o\ S^{all}_{qs}$. Templates are used to determine such properties (see . Templates described in Section 10) are used to determine if MC3 holds for a given recurrence G. They do so by checking if certain requirements, such as the following, are met: G contains only safe patterns of recursion, base cases of G are increasing, and closed form expressions added to results of recursive calls to G are monotonically increasing (this pertains to "$2^{ls_s-1} + ls_s - 1$" in the definition of S_{qs}). These requirements ensure that recurring on a large argument value (in place of the result of the complex R function) leads to a greater result of evaluation of the body of G, than recurring on a small argument value.

Monotonicity properties in MC1 and MC2 follow directly from monotonicity properties of primitive operations, e.g., $+$, $\times$ are monotonic with respect to their arguments, $-$, $\div$ are monotonic with respect to their first arguments but not with respect to their second arguments, and $modulo$ is not monotonic with respect to either of its two arguments.

Consider a bound function G such that for every R function F that G depends on, $F = max\ o\ F^{all}$. If checks MC1, MC2, and MC3 are satisfied by all complex R functions that G is dependent on, then $G = max\ o\ G^{all}$. These checks are performed in the analysis for all bound functions. This includes the case when G is an R function. So, by induction, the previous argument holds for all bound functions because for any called R function F, $F = max\ o\ F^{all}$.

An example of a recurrence T whose body contains an arg-use of a complex R function R_f by T, and which does not satisfy MC3 is:

$$R^{all}_f(m) = \{m-1, \ldots, 0\},\ R_f(m) = m - 1$$
$$T(n, m) = \begin{cases} n, \text{ if } m = 0 \\ T(n-1,\ R_f(m)), \text{ otherwise} \end{cases} = n - m$$

diff(ls_1, ls_2) is an example of an input function from which a bound function similar to T might be derived. *diff*(ls_1, ls_2) returns the tail list of ls_1 of size $|ls_1| - |ls_2|$. A variation of *diff*, with similar bound functions, returns a copy of the tail list.

$$\textit{diff}(ls_1,\ ls_2) = \textbf{if } null?(ls_2) \textbf{ then } ls_1 \textbf{ else } \textit{diff}(cdr(ls_1),\ cdr(ls_2))$$

The need for checks MC2 and MC3 stems from the use of R function results as arguments to other bound functions. Check MC1 simply ensures that size arguments to bound functions are, uniformly, maximums of sizes seen during program execution, rather than an arbitrary size from amongst the possible sizes. These checks are not relevant to applications of S and N functions. Results of S functions are never used as arguments to bound functions. Results of N functions may sometimes be values of newsize arguments (of N functions). We argue that all N functions are, by construction, monotonically increasing with respect to newsize arguments. These arguments are used only to determine the number of new heap cells in an argument that, for example, may be part of the result list. For example, for $f(ls) = cons(1, cdr(ls))$, the N function is $N_f(ls_s, ls_w) = 1 + max(0, (ls_w - 1))$. In particular, newsize arguments to N functions only get added to the result; non-monotonic operators are never applied to them. Further, newsize arguments have no counterparts in the original program, unlike size arguments of numeric variables where operators applied to the latter may translate into the same operators applied to the former in the definitions of bound functions. Predicates of conditionals could result in recurrences that are non-monotonic with respect to the arguments in the predicates; however, newsize arguments do not occur in the predicates of conditional expressions after transformation by $\mathcal{N}$ (note from the definition of $\mathcal{N}$ that predicates are transformed by $\mathcal{R}$). Arg-exps of functions are transformed by $\mathcal{R}$ to obtain size arguments to corresponding bound functions. So, newsize arguments are never used as size arguments with respect to which bound functions can be non-monotonic.

9. Solving Composite Recurrences

We developed a set of templates to solve composite recurrences. The templates are designed to match composite recurrences of the forms that commonly arise in bound functions, such as from the $\mathcal{S}$ transformation of function applications (see Figure 1). The applicability of a template is defined by a pattern that the recurrence must match, together with some requirements that must be satisfied by the constants and expressions in the recurrence. Several templates solve composite recurrences by reducing the given composite recurrence into a regular recurrence containing a single argument e of the max application, such that the evaluation of e is the maximum at the relevant base cases (different from the base cases of the given recurrence), as well as the inductive case. Other templates work by reducing composite recurrences with multiple max applications into component composite recurrences which are recursively solved. A complete list of templates is available in [24].

We now describe a template for solving composite recurrences of the form

$$T(n) = \begin{cases} c_1, \ldots, c_{j+1}, \text{ if } n = i, \ldots, i+j, \text{ respectively} \\ max(e_1(n),\ e_2(n) + aT(n-b)), \text{ otherwise} \end{cases}$$

where $i, j, c_1, ..., c_{j+1}, a, b$ are constants, and $e_1(n)$ and $e_2(n)$ are closed forms in n. T has $j+1$ base cases and one recursive case. The requirements for applicability of the template include a few conditions that ensure the recurrence is well-defined, e.g., $0 < b \leq (j+1)$. The remaining requirements define three separate cases in which a solution can be ascertained. In case (a), we also conclude that the recurrence is simple, and therefore this case is checked first. Let E_1 be $e_1(n)$, and let E_2 be $e_2(n) + aT(n-b)$.

Case (a): If the following conditions hold,

$A1$: $e_1(n) = e_2(n) + ac_{(n-b-i+1)},\ \forall n \in [i+j+1, i+j+b]$
$A2$: $e_1(n) = e_2(n) + ae_1(n-b),\ \forall n > (i+j+b)$

then T is simple, and the solution of T is:

$$T(n) = \begin{cases} c_1, \ldots, c_{j+1}, \text{ if } n = i, \ldots, i+j, \text{ respectively} \\ e_1(n), \text{ otherwise} \end{cases}$$

Informally, this case applies when all unfoldings of T, using all combinations of arg-exps $E1$ and $E2$ of max, yield the same value. $A1$ ensures this for the smallest non-base cases of n (in the range $[i+j+1, i+j+b]$) for which $T(n)$ uses a base value of T; this corresponds to the base case of the proof of case (a). $A2$ performs the necessary check for the inductive case: for $n > (i+j+b)$, the value of $E1$ equals that of $E2$, unfolded once using $E1$. For example, consider the R function R_{re} of *remove-element* in *selection-sort*.

$$R_{re}(ls_s) = \begin{cases} 0, \text{ if } ls_s = 1 \\ max(ls_s - 1,\ 1 + R_{re}(ls_s - 1)), \text{ otherwise} \end{cases}$$

Case (a) applies and gives the solution $R_{re}(ls_s) = ls_s - 1$.

Case (b): If the following conditions hold,

$B1$: $e_1(n) \geq e_2(n) + ac_{(n-b-i+1)},\ \forall n \in [i+j+1, i+j+b]$
$B2$: $e_1(n) \geq e_2(n) + ae_1(n-b),\ \forall n > (i+j+b)$

then T is complex, and the solution of T is:

$$T(n) = \begin{cases} c_1, \ldots, c_{j+1},\ n = i, \ldots, i+j, \text{ respectively} \\ e_1(n), \text{ otherwise} \end{cases}$$

The solution is based on the observation that, under these conditions, $E1$ always yields a larger value than $E2$.

Case (c): Define T' as

$$T'(n) = \begin{cases} c_1, \ldots, c_{j+1}, \text{ if } n = i, \ldots, i+j, \text{ respectively} \\ e_2(n) + aT'(n-b), \text{ otherwise} \end{cases}$$

Suppose the solution of T' is

$$T'(n) = \begin{cases} c_1, \ldots, c_{j+1},\ n = i, \ldots, i+j, \text{ respectively} \\ e_3(n), \text{ otherwise} \end{cases}$$

If $e_1(n) \leq e_3(n)$ for $n > (i+j)$, then T is complex and has the same solution as T'. The solution is based on the observation that, under these conditions, arg-exp $E2$ of max always yields the maximal value.

10. Performing Test MC3

Test MC3 is performed on bound function recurrences that contain arg-uses of complex R functions, where the arg-users are the defining bound functions. Consider such a bound function G:

$$G(v) = \ldots\ G(F(v))\ \ldots$$

where F is a complex R function. We consider a single-argument function for simplicity. The discussion is easily extrapolated for functions with multiple arguments. Test MC3 checks if using the result of $F(v)$ in the definition of G yields $max(G^{all}(v))$, which is the desired result of G.

An example of such a bound function is R_{qs} in Section 7. If R_{qs} does not satisfy MC3, then the analysis cannot proceed without an alternate strategy, e.g., if G is an N function, then it can be redefined without newsize arguments, as discussed in Section 6. This simplification may allow test MC3 to succeed or even make it unnecessary.

Test MC3 is difficult to check in the general case. We developed templates that perform the test by matching recurrences against safe patterns and ensuring certain properties such as the following: base values of the recurrence are monotonically increasing, base values are less than values of the recurrence at the smallest non-base cases, and closed form expressions in the recurrence definition contain only monotonically increasing operators. Templates are constructed such that, when checking MC3 for aforementioned G, it is sufficient to provide its definition using just the solution of F, instead of the set of elements of F^{all}. A complete list of templates is available in [24].

One of the templates that test MC3 matches recurrences of the form

$$T(n) = \begin{cases} c_1, ..., c_{j+1},\ n = i, ..., i+j, \text{ respectively} \\ e(n) + aT(f(n)), \text{ otherwise} \end{cases}$$

where $i, j, c_1, ..., c_{j+1}, a$ are constants, and $e(n)$ is a closed form in n. The given recurrence is required to meet the following conditions (we elide general well-formedness requirements that are orthogonal to MC3).

$R1$: $c_1 \leq ... \leq c_{j+1}$, i.e., base cases are monotonically increasing.
$R2$: $c_{j+1} \leq e(i+j+1) + aT(f(i+j+1))$, i.e., the largest base value is not greater than the value of T at the smallest non-base case.
$R3$: $\forall n > (i+j).\ f(n) < n$. For example, this holds for $f(n) = n - b$ and $f(n) = n/2$ (integer division).
$R4$: f is monotonically increasing w.r.t. n. For example, this holds for $f(n) = n - b$ and $f(n) = n/2$.
$R5$: $\forall n > (i+j),\ e(n) > 0$ and $e(n)$ is monotonically increasing w.r.t. n.

It is easy to see that if R1-R5 are satisfied by a recurrence G, then MC3 is satisfied for G. The above template matches R_{qs} which after evaluation of the calls to R_{lls} and R_{gls} is

$$R_{qs}(ls_s) = \begin{cases} 0, \text{ if } ls_s = 0 \\ 1, \text{ if } ls_s = 1 \\ 1 + 2R_{qs}(ls_s - 1), \text{ otherwise} \end{cases}$$

11. Examples

We applied the analysis to several list-processing programs, namely list reversal, insertion sort, selection sort, merge sort, quicksort, and programs that compute longest common subsequence, string edit distance, and binomial coefficients efficiently using dynamic programming. In the latter set of programs, lists (instead of arrays) are used to store results of subproblems. Figure 5 shows the results of our analysis. The closed forms derived by the analysis include linear and quadratic polynomials, and exponential formulae. A complete listing of example programs, their bound functions, recurrence relations, resulting closed forms, and how these were derived, appears in [24].

The accuracy (tightness) of the analysis results was confirmed by comparing with results from the analysis in [25] for the same programs. The closed form solutions derived by our analysis are exact maximums of live heap space usage of all example programs except *quicksort*. Our analysis gives a loose bound for quicksort because it does not recognize that the sizes of the results of functions *lesserorequal-list* (this is the function abbreviated as lls in Section 7) and *greater-list* (which is similar but returns the sublist containing elements greater than the cutoff) are correlated, specifically, that when both functions are applied to the same list of length n with the same cutoff, the sizes of their results sums to n. We verified that this is the only source of inaccuracy in the analysis of quicksort by manually modifying the relevant bound functions of *quicksort* to take this invariant into account; the analysis then produces a quadratic polynomial that accurately describes the heap usage of quicksort.

Figure 4 shows how live heap space analysis works for an example program, namely *insertion-sort*. It lists the functions of *insertion-sort*, corresponding S, N, and R functions, recurrences obtained by simplifying the bound functions, and closed form solutions of the recurrences.

12. Extensions to the Analysis

Our analysis can be extended to data types other than lists by defining an appropriate R function for each data type and introducing corresponding size arguments to bound functions. Some data types, such as arrays, are similar to lists and can be handled easily. Binary trees are more interesting. Their size can be measured by height and number of nodes, so we define two R functions for binary trees, R_{height} and R_{nodes}, and for each binary-tree argument v of each function f in the original program, we introduce arguments v_h and

Examples	Entry Function	S Function of Entry Function	Derived Heap Usage Complexity	Tight?
reverse	$rev(ls)$	$S_{rev}(ls_s) = \begin{cases} 0 \text{ if } ls_s = 0 \\ 2ls_s - 1, \text{ otherwise} \end{cases}$	Linear	Yes
insertion sort	$is(ls)$	$S_{is}(ls_s) = \begin{cases} 0 \text{ if } ls_s = 0 \\ 2ls_s - 1, \text{ otherwise} \end{cases}$	Linear	Yes
selection sort	$ss(ls)$	$S_{ss}(ls_s) = \frac{ls_s(ls_s+1)}{2}$	Quadratic	Yes
merge sort	$ms(ls)$	$S_{ms}(ls_s) = \begin{cases} 0 \text{ if } ls_s = 0 \\ 2ls_s - 1, \text{ otherwise} \end{cases}$	Linear	Yes
quick sort	$qs(ls)$	$S_{qs}(ls_s) = \begin{cases} 0 \text{ if } ls_s = 0, 1 \\ 3.2^{ls_s - 1} - 2, \text{ otherwise} \end{cases}$	Exponential	No
longest common subsequence	$lcs(ls_1, ls_2)$	$S_{lcs}(ls_{1_s}, ls_{2_s}) = \begin{cases} 1 \text{ if } ls_{1_s} = 0 \text{ or } ls_{2_s} = 0 \\ ls_{2_s} + 2 \text{ if } ls_{1_s} = 1 \\ 2ls_{2_s} + 2, \text{ otherwise} \end{cases}$	Linear	Yes
string edit distance	$se(str_1, str_2)$	$S_{se}(str_{1_s}, str_{2_s}) = \begin{cases} str_{1_s} + str_{2_s} + 1, \\ \quad \text{if } str_{1_s} = 0 \text{ or } str_{2_s} = 0 \\ 2str_{1_s} + 2str_{2_s} + 1, \text{ otherwise} \end{cases}$	Linear	Yes
binomial coefficient	$bc(n, m)$	$S_{bc}(n_s, m_s) = \begin{cases} 1 \text{ if } m_s = n_s \\ m_s + 2 \text{ if } m_s = n_s - 1 \\ 2m_s + 2, \text{ otherwise} \end{cases}$	Linear	Yes

Figure 5. Parametric heap space usage bounds derived for some example programs.

v_n to the bound functions of f, representing the height and number of nodes, respectively, of the tree passed to v.

In the presence of these new data types, bound functions may need to make approximations, e.g., that each branch of a binary tree v has at most $min(2^{v_h} - 1, v_n - 1)$ nodes. We can mitigate, if not eliminate, the effects of such approximations by using type-specific invariants in the analysis, e.g., $v_n \leq 2^{v_h+1} - 1$ and $\mathcal{R}_{nodes}[\![left(e)]\!] + \mathcal{R}_{nodes}[\![right(e)]\!] = R_{nodes}[\![e]\!]$, where $left(e)$ and $right(e)$ return the subtrees of e, respectively. Note that the metric R_{nodes} is useful for all data types, including nested lists. The current R functions are, in fact, R_{length} functions; for a nested list ls, R_{length} describes the length of the top-level list of ls.

S and N functions, and the $\mathcal{S}$ and $\mathcal{N}$ transformations, also need to be specialized to new data types. For some data types, S and N functions can take any one of several size measures as arguments, depending on which size measure yields the most compact and accurate formulas representing live heap usage and new result-space usage, respectively. The analysis may try different size measures and then determine which yields the best results.

The current version of our analysis does not allow mutual recursion in the input programs. Such mutual recursion typically leads to mutual recursion in the bound functions. Mutual recursion can sometimes be handled by converting it into self-recursion (i.e., ordinary recursion): if f and g are mutually recursive, and unfolding f a few times yields a definition of f that recursively calls only f and not g, then f is in fact self-recursive. Our analysis can be extended to first use such unfolding to try to convert mutual recursion into self-recursion. If this does not succeed, then standard mathematical techniques for solving mutual recurrences are used, if applicable.

13. Related Work

There is a large amount of work on analyzing program cost or resource complexities, but the majority of it is on time analysis, e.g., [26, 19, 22, 23, 20, 15]. Analysis of stack space and heap allocation are similar to time analysis [20]. Analysis of live heap space is different because live heap space increases and decreases during program execution, while time always increases. Further, live heap space analysis needs to determine when allocations become garbage.

Wegbreit [26] proposed deriving closed form expressions describing running times of Lisp programs in terms of sizes of inputs, much like we do for live heap space, by deriving recursive functions describing running times, converting them into difference equations and solving the same. Analyses that produce such formulas are sometimes called *parametric* analyses. Probabilities attached to ambiguous predicates (that require information other than sizes of inputs) enable the derivation of mean running times, in addition to best-case and worst-case running times. The prototype implementation is limited to simple Lisp programs. [19] is similar to [26] but derives asymptotic, rather than exact, upper bounds on running times; also, the user is required to aid the analysis in converting the recursive function into a closed form expression. These two papers describe only limited methods to handle their respective equivalents of composite recurrences, and neither of them use the notion of monotonicity requirements.

Most of the work related to analysis of space is on analysis of cache behavior, e.g., [27, 14], much of which is at a lower language level, for compiler generated code, while our analyses are at source level and can serve many purposes. Live heap analysis is a first step towards analyzing cache behavior in the presence of garbage collection.

Several type systems [18, 17, 12, 11] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [18, 12]. They require programmers to annotate their programs with cost functions as types. Furthermore, some programs must be rewritten to have feasible types [18, 17]. [9] proposes a loop-detecting algorithm to identify methods and instructions that execute an unbounded number of times, thereby detecting possibly unbounded memory usage. [4, 7] ensure acceptable memory use in bytecode, by verifying specified resource annotations or memory consumption policies.

[16] proposes a method to obtain linear bounds on heap space consumption of functional programs using type derivations and linear programming. While programs are not required to be linearly

typed, restrictions are made on sharing of heap cells. Our analysis is not restricted to linear bounds and does not restrict sharing of heap cells.

Several methods have been developed to automatically infer heap space bounds for Java-like imperative languages and bytecode. Imperative update poses special challenges for determining liveness. Chin *et al.* [10] propose a method to infer stack and heap usage bounds for an assembly-like language. "dispose" commands that reclaim unused heap space, are explicitly inserted into input programs [11] at points where objects are no longer live. Recursive constraint abstractions derived from these modified input programs describe stack and heap usages of the programs. Fixpoint analysis of constraint abstractions using a Presburger solver yields closed form expressions for stack and heap usages. Our analysis applies to a high-level language and is not limited to Presburger formulae, so it can derive both linear and non-linear bounds. The alias/uniqueness analysis used to determine liveness and insert "dispose" commands appropriately, is an over-approximation of the behaviour of an ideal garbage collector. In the absence of imperative update, live heap analysis is able to determine liveness exactly. As a result, our bounds on heap usage are tighter than those of [10]. While their analysis has been applied to several benchmarks, all except one of the benchmarks use few heap objects.

The heap analysis in [8] is also targeted to an imperative language but is able to derive polynomial, not just linear, heap bounds. Our analysis is not limited to a complexity category, although it is limited by the solvability of the encountered recurrences. Region-based memory management is used to model garbage collection; objects are allocated in regions associated with methods and regions are garbage collected at the end of methods. As in the above discussion, this is an over-approximation of heap usage compared to our analysis, which is, in the absence of imperative update, able to model "ideal" garbage collection (objects are garbage collected as soon as they become dead). The maximum memory occupied by a region configuration is modelled as a parametric polynomial optimization problem. This is then solved using Bernstein basis. The analysis does not handle recursions.

Albert *et al.* propose a two-part analysis of heap space usage for Java bytecode [2]. The first part constructs cost equations characterizing the amount of heap space allocated by each method and then uses the technique in [1] to obtain closed-form upper bounds on the total amount of allocation. The second part takes garbage collection into account in a simple way by using escape analysis to identify allocations that do not escape from the static scope containing the allocation statement, and then ignoring these allocations when computing the total amount of heap allocation. Solving the resulting cost equations provides an upper bound on what they call the *active heap space* for each method. On the one hand, their analysis handles various language features that ours does not, most notably imperative update. On the other hand, their analysis cannot easily be extended to compute what our analysis computes—the maximum live heap space, which they call the *memory high-watermark*. The "active heap space" computed by their analysis only increases (like running time); live heap space is different, because it increases and decreases. They write: "Analysis for finding upper bounds on the memory high-watermark cannot be directly done using cost relations as introducing decrements in the equations requires computing lower bounds" [2, Section 6]. To illustrate the difference between live space and "active heap space", consider the program

$$f(n) = len(g(n)) + len(g(n))$$
$$g(n) = \mathbf{if}\ n = 0\ \mathbf{then}\ nil\ \mathbf{else}\ cons(n,\ g(n-1))$$
$$len(l) = \mathbf{if}\ null?(l)\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + len(cdr(l))$$

Our analysis yields $S_f(n_s) = n_s$, i.e., the maximum live heap space used by f is n cons cells, reflecting that the cells allocated by the first call to $g(n)$ become garbage before the second call to $g(n)$. The method of [1, 2] does not recognize this (since cells allocated in g escape it) and yields a bound of $2n$ on the space usage of f. Albert *et al.*'s paper in the same proceedings analyzes live heap space (as opposed to total heap allocation and active heap space) in the presence of garbage collection [3]. The methods used in our analysis to solve composite recurrences can be enhanced by the techniques in [1] to solve non-deterministic cost relations and those in [15] to handle disjunctive invariants.

14. Conclusions and Future Work

In summary, this paper describes a method that aims to determine worst-case live heap space usage of functional programs automatically and accurately using source-level program analysis and transformations. Specific technical contributions include a framework for deriving bound (S, N, and R) functions that describe the live heap space usage of functional programs, the identification and formulation of conditions (tests MC1, MC2, and MC3) that are necessary to ensure soundness of the analysis, and the templates used to solve recurrences and check part of the monotonicity condition (namely, MC3). The results reported in Section 11 suggest that the analysis usually produces tight bounds in practice. Although these example programs are small, they have non-trivial patterns of heap allocation and garbage collection. In part, this is because they are functional programs and must allocate new data structures instead of updating existing data structures.

Future work on live heap analysis for functional programs includes modifying the analysis to reflect the effect of optimization of tail recursion, incorporating techniques for analysis of higher-order functions [23, 20], and evaluating the accuracy and scalability of the analysis more extensively. Another important direction for future work is to modify the analysis to handle imperative update.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proc. 15th International Static Analysis Symposium (SAS 2008)*, pages 221–237, 2008.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap space analysis for java bytecode. In *Proc. 6th International Symposium on Memory Management (ISMM 2007)*, pages 105–116, 2007.

[3] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Live heap space analysis for languages with garbage collection. In *Proc. 7th International Symposium on Memory Management (ISMM '09)*, 2009.

[4] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS), volume 3362 of LNCS*, pages 1–26. Springer, 2005.

[5] D. F. Bacon. Realtime garbage collection. *Queue*, 5(1):40–49, 2007.

[6] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. PURRS: The Parma University's recurrence relation solver. `http://www.cs.unipr.it/purrs`.

[7] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 86–95, Washington, DC, USA, 2005. IEEE Computer Society.

[8] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, NY, USA, 2008. ACM.

[9] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods 05, volume 3582 of LNCS*, pages 91–106. Springer-Verlag, 2005.

[10] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin. Analysing memory resource bounds for low-level programs. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, NY, USA, 2008. ACM.

[11] W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, pages 70–86. Springer, 2005.

[12] K. Crary and S. Weirich. Resource bound certification. In *Conference Record of the 27th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 2000.

[13] W. Fu and C. Hauser. A real-time garbage collection framework for embedded systems. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 20–26, New York, NY, USA, 2005. ACM.

[14] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, July 1999.

[15] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Proc. 36th Annual Symposium on Principles of Programming Languages (POPL 2009)*, 2009.

[16] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL03 Symposium on Principles of Programming Languages*, pages 185–197. ACM Press, 2003.

[17] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 70–81. ACM Press, Sept. 1999.

[18] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 410–423. ACM Press, 1996.

[19] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.

[20] Y. A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer-Verlag, 1998.

[21] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.

[22] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM Press, Sept. 1989.

[23] D. Sands. Complexity analysis for a lazy higher-order language. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer-Verlag, Berlin, May 1990.

[24] L. Unnikrishnan. *Automatic Live Memory Bound Analysis for High-Level Languages*. PhD thesis, Stony Brook University, 2008. Available at `http://www.cs.sunysb.edu/~leena/thesis.pdf`.

[25] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation*, Jan. 2003.

[26] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–538, Sept. 1975.

[27] R. Wilhelm and C. Ferdinand. On predicting data cache behaviour for real-time systems. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1998.