Optimized Live Heap Bound Analysis

Leena Unnikrishnan*   Scott D. Stoller*   Yanhong A. Liu⋆

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400 USA

Abstract. This paper describes a general approach for optimized live heap space and live heap space-bound analyses for garbage-collected languages. The approach is based on program analysis and transformations and is fully automatic. In our experience, the space-bound analysis generally produces accurate (tight) upper bounds in the presence of partially known input structures. The optimization drastically improves the analysis efficiency. The analyses have been implemented and experimental results confirm their accuracy and efficiency.

1   Introduction

Time and space analysis of computer programs is important for virtually all computer applications, especially in embedded systems, real-time systems, and interactive systems. Space analysis is becoming important due to the increasing use of high-level languages with garbage collection, such as Java, the importance of cache memories in performance [28], and the stringent space requirements of embedded applications [25]. For example, space analysis can determine exact memory needs of embedded applications; it can help determine patterns of space usage and thus help analyze cache misses or page faults; and it can determine memory allocation and garbage collection behavior.

Space analysis is also important for accurate prediction of running time [11]. For example, analysis of worst-case execution time in real-time systems often uses loop bounds or recursion depths [21, 2] both of which are commonly determined by the size of the data being processed. Also, memory allocation and garbage collection, as well as cache misses and page faults, contribute directly to the running time. This is increasingly significant as the processor speed increases, leaving memory access as the performance bottleneck.

Much work on space analysis has been done in algorithm complexity analysis and systems. The former is in terms of asymptotic space complexity in closed forms [5]. The latter is mostly in the form of tracing memory behavior or analyzing cache effects at the machine level [20, 28]. What has been lacking is analysis

of space usage for high-level languages, in particular, automatic and accurate techniques for live heap space analysis for languages with garbage collection, such as Java, ML or Scheme.

This paper describes a general approach for automatic accurate analysis of live heap space based on program analysis and transformations. The analysis determines the maximum size of the live data on the heap during execution. This is the minimum amount of heap space needed to run the program if garbage collection is performed whenever garbage is created. This metric is useful for evaluating other garbage collection schemes, just like the performance of an optimal cache replacement algorithm is useful for evaluating other replacement algorithms. 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.

Our approach starts with a program written in a high-level functional language with garbage collection. We construct (i) a *space function* that takes the same input as the original program and returns the amount of space used and (ii) a *space-bound function* that takes as input a characterization of a set of inputs of the original program and returns an upper bound on the space used by the original program on any input in that set. Finding tight space bounds is undecidable, so space-bound functions may diverge. In our experience, this is rare.

A key problem is how to characterize the input data and exploit this information in the analysis. In traditional complexity analysis, inputs are characterized by their size. Accommodating this requires manual or semi-automatic transformation of the time or space function [17, 29]. The analysis is mainly asymptotic. A challenging problem that arises in this approach is optimizing the time-bound or space-bound function to a closed form in terms of the input size [17, 24, 7]. But closed forms are known only for subclasses of functions. Thus, such optimization can not be done automatically for analyzing general programs.

Rosendahl proposed characterizing inputs using *partially known input structures* [24]. For example, instead of replacing an input list l with its length n, we simply use as input a list of n unknown elements. A special value uk ("unknown") is introduced for this purpose. It represents unknown primitive values; if it represented constructed data, we wouldn't know how much space it used. At control points where decisions depend on unknown values, the maximum space usage of all branches is computed. Rosendahl concentrated on proving the correctness of this transformation for time-bound analysis. He relied on optimizations to obtain closed forms, but closed forms can not be obtained for all bound functions.

Our analysis and transformations are performed at source level. Our goal is a language-based analysis that abstracts from details of particular language implementations. For a particular language implementation, lower-level issues may need to be considered to determine the exact minimum heap space needed to run a given program.

Profiling, like space functions, measures the program's behavior on one input at a time; space-bound functions can efficiently analyze the program's behavior

on a set of inputs at once. They can thus be used to determine worst-case space usage of a program. Alternatively, worst-case space usage may be determined by applying the space function to a worst-case input. But in general, it is non-trivial to determine such an input.

Our approach combines program analysis and model checking and is called *model analysis*.

Live heap space-bound analysis is an abstract interpretation of live heap space analysis. While the latter works in the domain of concrete heaps, the former works in the domain of abstract heaps that represent the different possible heaps at a program point. Instead of performing a fixed-point computation of the abstract function like traditional abstract interpretations, we simply execute the abstract space-bound function on a given partially known input. Our analysis could be cast as a fixed-point calculation. While this may help achieve termination, the memory needs would be far greater due to the large number of required subcomputations and the large size of abstract heaps.

Space-bound analysis may also be viewed as a specialized model checking algorithm that searches a program's state space to determine its worst-case space usage. Our analysis is similar in some ways to an explicit-state model-checker applied to a program in which primitive values have been abstracted. But there is an important difference: traditional model checkers work on unstructured transition systems and can check a large range of properties, while our analysis works directly on programs in a high-level language and incorporates optimizations that exploit program structure and are possible only because the analysis is specifically targeted to determine heap space usage. In particular, it is easy to see that the two states resulting from the two branches of a conditional in a space-bound function are the same in all ways, except for a well-delineated region of their heaps. Our analysis uses a join operation that merges such states into a single state. This reduces the number of states and the space needed to store them. It also reduces the number of transitions that need to be explored, since they may now be explored from a single merged state.

A main contribution of this paper is an extremely effective optimization in space-bound analysis that limits the state space search to paths leading to maximal heap usage. For many examples, this optimization improves the asymptotic complexity of the analysis from greater than polynomial to polynomial.

2 Language

We use a first-order, call-by-value functional language that has literal values of primitive types, structured data, operations on primitive types, testers, selectors, conditionals, bindings, and function calls. These are fundamental program constructs that have analogues in all programming languages. A program is a set of mutually recursive function definitions of the form $f(v_1, ..., v_n) = e$, where an expression e is given by the grammar in Figure 1. We sometimes use infix notation for primitive operations. A tester application $c?(v)$ returns *true* iff v has outermost constructor c. A selector application $c^{-i}(v)$ returns the i'th com-

$$
\begin{array}{llll}
e ::= & v & & \text{variable reference} \\
& | & l & \text{literal} \\
& | & c(e_1, ..., e_n) & \text{constructor application} \\
& | & p(e_1, ..., e_n) & \text{operation on primitive types} \\
& | & c?(e) & \text{tester application} \\
& | & c^{-i}(e) & \text{selector application} \\
& | & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \text{conditional expression} \\
& | & \textbf{let } v = e_1 \textbf{ in } e_2 & \text{binding expression} \\
& | & f(e_1, ..., e_n) & \text{function application}
\end{array}
$$

Fig. 1. Input language.

ponent of a data construction v with outermost constructor c. Input programs to our analysis are assumed to be purely functional, but transformed programs use arrays and imperative update. A sequential composition $e_1; e_2$ returns the value of e_2.

3   Live Heap Space Functions

To analyze the live heap space used by a program on a known input, we transform the program into one that performs all the computations of the original program and keeps track of the total amount of live data. Liveness is ascertained using reference counts. The *reference count* for a data construction v is the number of pointers to v. These may be pointers from the stack, created by **let** bindings or bindings to formal parameters of functions, or pointers from the heap, created by data constructions. Data construction v is live if its reference count is greater than 0 or if it is the result of the expression just evaluated.

A *constructor count vector* v has one element $v[i_c]$ corresponding to each data constructor c used in a given program. Let $P[i_c]$ be the size of an instance of c. Let $\cdot$ denote dot product of vectors. The maximum $\max(v_1, v_2)$ of constructor count vectors v_1 and v_2 is v_1 if $v_1 \cdot P \geq v_2 \cdot P$ and is v_2 otherwise.

The transformation $\mathcal{L}$ in Figure 2 produces live heap space functions. The transformation of testers and **let** expressions is elided for brevity. The complete transformation appears in [27]. $\mathcal{L}$ introduces two global variables, *live* and *maxlive*, that satisfy: (1) for each constructor c, $live[i_c]$ is the number of live instances of c; (2) *maxlive* is the maximum value of *live* so far during execution. The maximum live space used during evaluation of function f is at most $ml \cdot P$; ml is the value of *maxlive* after evaluation of the space or bound function for f.

Our implementation of reference counting is based on an abstract data type (ADT) called con-value ("constructed-value") that defines five functions. $new(c(x_1, \ldots, x_n))$ returns a value v representing a new data construction $c(x_1, \ldots, x_n)$, whose reference count is initialized to zero. $data(v)$ returns the data construction $c(x_1, \ldots, x_n)$. $rc(v)$ returns the reference count associated with v. $incrc(v)$ and

$decrc(v)$ increment and decrement, respectively, the reference count associated with v. $incrc$ and $decrc$ are no-ops if the argument is a primitive value.

$$f_L(v_1, \ldots, v_n) = \mathcal{L}[\![e]\!] \quad \text{where } e \text{ is the body of function } f, \text{ i.e., } f(v_1, \ldots, v_n) = e$$

$$\mathcal{L}[\![v]\!] = v$$

$$\mathcal{L}[\![l]\!] = l$$

$$\mathcal{L}[\![c(e1, \ldots, en)]\!] = live[i_c]{+}{+}; \; \textbf{if } (P \cdot live > P \cdot maxlive)$$
$$\textbf{then for } c \in Constructors \; maxlive[i_c] := live[i_c];$$
$$\textbf{let } r_1 = \mathcal{L}[\![e_1]\!], \ldots, r_n = \mathcal{L}[\![e_n]\!] \textbf{ in}$$
$$incrc(r_1); \ldots; incrc(r_n); new(c(r_1, \ldots, r_n))$$

$$\mathcal{L}[\![p(e_1, \ldots, e_n)]\!] = p(\mathcal{L}[\![e_1]\!], \ldots, \mathcal{L}[\![e_n]\!])$$

$$\mathcal{L}[\![c^{-i}(e)]\!] = \textbf{let } x = \mathcal{L}[\![e]\!] \textbf{ in}$$
$$\textbf{let } r = c^{-i}(data(x)) \textbf{ in}$$
$$(\textbf{if } not(isPrim(x)) \; and \; rc(x) = 0 \textbf{ then } gcExcept(x, r)); \, r$$

$$\mathcal{L}[\![\textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3]\!] = \textbf{if } \; \mathcal{L}[\![e1]\!] \textbf{ then } \mathcal{L}[\![e2]\!] \textbf{ else } \mathcal{L}[\![e3]\!]$$

$$\mathcal{L}[\![f(e_1, \ldots, e_n)]\!] = \textbf{let } r_1 = \mathcal{L}[\![e_1]\!], \ldots, r_n = \mathcal{L}[\![e_n]\!] \textbf{ in}$$
$$incrc(r_1); \ldots; incrc(r_n);$$
$$\textbf{let } r = f_L(r_1, \ldots, r_n) \textbf{ in}$$
$$gcExcept(r_1, r); \ldots; gcExcept(r_n, r); \, r$$

$$gc(v) = \textbf{if } not(isPrim(v))$$
$$\textbf{then } decrc(v); \; \textbf{if } rc(v) \leq 0$$
$$\textbf{then } live[conType(v)]{-}{-};$$
$$\textbf{for } i = 1..arity(v) \; gc(c^{-i}(data(v)))$$

$$gcExcept(u, v) = incrc(v); gc(u); decrc(v)$$

Fig. 2. Transformation that produces live heap space functions f_L. $isPrim(v)$ is *true* iff v is primitive. $conType(v)$ returns an integer i_c that uniquely identifies the outermost constructor c in $data(v)$. $arity(v)$ returns the arity of the outermost constructor in $data(v)$.

Updating Reference Counts. $rc(v)$ is incremented when v is bound to a variable or function parameter, or a data construction containing v as a child is created. $rc(v)$ is decremented when the scope of a **let** binding for v ends, a function call with an argument bound to v returns, or a data construction containing v as a child becomes garbage.

Updating *live* and *maxlive*. Whenever new data is constructed, *live* is incremented, and *maxlive* is recomputed. *live* is decremented by an auxiliary function gc ("garbage collect") which is called whenever data can become garbage. A data construction may become garbage (1) because of a decrement of its reference count or (2) because it is created in the argument of a selector or tester and is lost to the program after the result of the selection or test is obtained. For example, $cons(0, nil)$ is garbage after the application of $cons^{-1}$ in

$cons^{-1}(cons(0, nil))$; note that its reference count is always 0. $gcExcept(u, v)$ is called when u should be garbage collected, v should not be garbage collected and v might be a descendant of u.

4   Live Heap Space Bound Functions

The transformation $\mathcal{L}_b$ in Figure 3 produces live heap space-bound functions. The transformation of certain expressions is elided for brevity; for variables, literals and constructor applications $\mathcal{L}_b$ works the same way as $\mathcal{L}$. The complete transformation appears in [27]. We sometimes refer to space-bound functions simply as bound functions. At every point during the execution of $\mathcal{L}_b [\![f]\!] (x)$, the value of $live$ is an upper bound on the possible values of $live$ at the corresponding point in executions of $\mathcal{L} [\![f]\!] (x')$, for all x' in the set represented by x. The presence of partially known inputs in bound analysis causes uncertainty. For conditionals whose tests evaluate to uk, both branches are evaluated to determine the maximum live heap space usage.

Live heap bound analysis depends on keeping track of all references and reference counts meticulously. Summarizing the results of two branches into a single partially known structure, as is done in time analysis [18], does not work for live heap bound analysis because it would be impossible to keep track of references accurately. So the result of a conditional whose test evaluates to uk is a separate entity, a join-value, that points to both results and has its own reference count.

Abstract Data Types. In addition to the con-value type, an ADT called join-value is also used. A join-value represents a set of possible results. Join-values are created by conditional expressions whose tests evaluate to uk and by selectors applied to join-values. Each join-value j has a list $branches(j)$ containing references to con-values and/or join-values. Primitive values, if any, in the set represented by j are not stored in j. Thus, if $branches(j)$ has only one element, j represents a choice between that element and some primitive value. j has an associated constructor count vector $exs(j)$. Parts of the data constructions represented by j may be live regardless of j. Of the other parts, only those occurring in a single largest branch are live in a worst case (i.e., maximal live heap space) execution of the original program. The sum of the other parts that are not in the largest branch is stored in $exs(j)$. $live$ does not include $exs(j)$. When j becomes garbage, $exs(j)$ is added to $live$ just before garbage collecting the branches of j. Like con-values, join-values have reference counts and related functions rc, $incrc$ and $decrc$. $newjoin(b)$ creates a join-value j with a list b of branches; $rc(j)$ is initialized to 0, and $exs(j)$ is initialized to the zero vector, denoted V_0.

Conditionals, Selectors and Testers. Consider a conditional expression $(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^{\dagger}$ whose test evaluates to uk. Suppose l_1, l_2 and l_3 are the values of $live$ after evaluating e_1, $e_1; e_2$ and $e_1; e_3$, respectively. The value of $live$ at $\dagger$ is set to $\max(l_2, l_3)$. The result r of the conditional is computed by $join(r_2, r_3)$, where r_2 and r_3 are the results of e_2 and e_3, respectively. If r_2 and

$f_{Lb}(v_1, \ldots, v_n) = \mathcal{L}_b \llbracket e \rrbracket$ where e is the body of function f, i.e., $f(v_1, \ldots, v_n) = e$

$\mathcal{L}_b \llbracket p(e1, \ldots, en) \rrbracket = p_u(\mathcal{L}_b \llbracket e1 \rrbracket, \ldots, \mathcal{L}_b \llbracket en \rrbracket)$

$p_u(v_1, \ldots, v_n) = \textbf{if } v_1 = uk \textbf{ or } \cdots \textbf{ or } v_n = uk \textbf{ then } uk \textbf{ else } p(v_1, \ldots, v_n)$

$\mathcal{L}_b \llbracket c^{-i}(e) \rrbracket = \textbf{let } x = \mathcal{L}_b \llbracket e \rrbracket \textbf{ in}$
$\qquad\qquad \textbf{let } r = c_u^{-i}(x) \textbf{ in}$
$\qquad\qquad\quad (\textbf{if } not(isPrim(x)) \; and \; rc(x) = 0 \textbf{ then } gcExcept(x, r); recomputeExs(r)); r$

$c_u^{-i}(v) = \textbf{if } v = uk \textbf{ then } c^{-i}(false)$
$\qquad\quad \textbf{else if } isJoin(v)$
$\qquad\qquad\quad \textbf{then if } length(branches(v)) = 1 \textbf{ then } c^{-i}(false)$
$\qquad\qquad\qquad\quad \textbf{else } join(c_u^{-i}(first(branches(v))), c_u^{-i}(second(branches(v))))$
$\qquad\qquad\quad \textbf{else } c^{-i}(data(v))$

$\mathcal{L}_b \llbracket \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \rrbracket =$
  $\textbf{let } b = \mathcal{L}_b \llbracket e_1 \rrbracket \textbf{ in}$
    $\textbf{if } b = uk \textbf{ then let } l_1 = copy(live) \textbf{ in}$
$\qquad\qquad\qquad \textbf{let } r_2 = \mathcal{L}_b \llbracket e_2 \rrbracket \textbf{ in}$
$\qquad\qquad\qquad\quad \textbf{let } l_2 = copy(live) \textbf{ in}$
$\qquad\qquad\qquad\qquad live := l_1; \textbf{let } r_3 = \mathcal{L}_b \llbracket e_3 \rrbracket \textbf{ in}$
$\qquad\qquad\qquad\qquad\quad \textbf{let } l_3 = copy(live) \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad live := \max(l_2, l_3); \textbf{let } r = join(r_2, r_3) \textbf{ in}$
$\qquad\qquad\qquad\qquad\qquad\qquad setexs(r, \min(l_2 - l_1, l_3 - l_1)); r$
$\qquad\quad \textbf{else if } b \textbf{ then } \mathcal{L}_b \llbracket e_2 \rrbracket \textbf{ else } \mathcal{L}_b \llbracket e_3 \rrbracket$

$\mathcal{L}_b \llbracket f(e_1, \ldots, e_n) \rrbracket = \textbf{let } r_1 = \mathcal{L}_b \llbracket e_1 \rrbracket, \ldots, r_n = \mathcal{L}_b \llbracket e_n \rrbracket \textbf{ in}$
$\qquad\qquad\qquad incrc(r_1); \ldots; incrc(r_n);$
$\qquad\qquad\qquad \textbf{let } r = f_{Lb}(r_1, \ldots, r_n) \textbf{ in}$
$\qquad\qquad\qquad\quad gcExcept(r_1, r); \ldots; gcExcept(r_n, r); recomputeExs(r); r$

$gc(v) = \textbf{if } not(isPrim(v))$
$\qquad\quad \textbf{then } decrc(v); \textbf{if } rc(v) \leq 0$
$\qquad\qquad\qquad \textbf{then if } isJoin(v)$
$\qquad\qquad\qquad\quad \textbf{then } live = live + exs(v);$
$\qquad\qquad\qquad\qquad \textbf{for } u \textbf{ in } branches(v) \; gc(u)$
$\qquad\qquad\qquad\quad \textbf{else } live[conType(v)]--;$
$\qquad\qquad\qquad\qquad \textbf{for } i = 1..arity(v) \; gc(c^{-i}(data(v)))$

Fig. 3. Transformation that produces live heap space-bound functions f_{Lb}. *copy* copies a vector. $+$ and $-$, when applied to vectors, denote component-wise sum and difference.

r_3 are primitive, then $join(r_2, r_3)$ is r_2 if $r_2 = r_3$ and uk otherwise. If r_2 and r_3 are not primitive and are the same, then $join(r_2, r_3)$ is r_2. Otherwise, $join(r_2, r_3)$ is a join-value pointing to r_2 and r_3. $exs(r)$ is set to $\min(l_2 - l_1, l_3 - l_1)$. $l_2 - l_1$ and $l_3 - l_1$ are the amounts of new data in r_2 and r_3, i.e., the amounts of data created by e_2 and e_3. r_2 and r_3 may contain old data too, i.e., data created before evaluating e_2 and e_3. Old data are live regardless of r. Between the sets

of new data in r_2 and r_3, only one set is live. We keep the larger set live; the size of the other set is $exs(r)$.

Observe that in the transformation of (**if** e_1 **then** e_2 **else** e_3), we evaluate e_2 and then e_3, making copies of only *live* in between. We do not need to make copies of the heap because the source language does not contain imperative update. Thus, if h_1 is the heap after the evaluation of e_1, then e_2 and e_3 modify h_1 only by adding new con-values to it. Informally, h_2, the heap after evaluation of e_2, is $(h_1 \cup r_2)$. Similarly, h_3 is $(h_1 \cup r_3)$, h_3 having the expected meaning. In other words, the heap after evaluation of the conditional is $(h_1 \cup (r_2 \text{ or } r_3))$. The choice between r_2 and r_3 is conveniently represented using a join-value that points to them both.

For join-values with two non-primitive branches, selectors and testers are first applied to the branches and the *join* of the results is returned. The *exs* field of a join-value j that is the result of applying a selector to another join-value j' is set to V_0, because when j is created, j' is live, and $exs(j')$ already takes care of any excess.

When a selector c^{-i} is applied to uk or a join-value j with a primitive branch, it simply aborts by attempting to apply the selector to an arbitrary primitive value. However, if we assume that the given program never applies selectors to primitive values, then the occurrence of $c^{-i}(j)$ in the analysis corresponds to the application of c^{-i} to the non-primitive branch of j in the original program. With this assumption, $c^{-i}(j)$ is simply the result of applying c^{-i} to the non-primitive branch. Applications of selectors to join-values with primitive branches is in fact seen in only one of our examples, namely quicksort.

Achieving Tightness. The following example illustrates why *live* may not be as tight as desired.

$$
\begin{aligned}
&\textbf{let } u = cons(1, nil) \textbf{ in}\\
&\quad \textbf{let } v = cons(2, nil) \textbf{ in}\\
&\quad\quad (\textbf{if } uk \textbf{ then } cons(3, v) \textbf{ else } cons(4, cons(5, u)))
\end{aligned}
\tag{1}
$$

Let r be the result of the conditional. Let c_i denote the data construction with $cons^{-1}(c_i) = i$. Just after the conditional is evaluated, *live* includes the sizes of both c_1 and c_2. *live* excludes the size of c_3 because the result of the alternative branch containing c_4 and c_5 is larger; so *live* includes the latter instead of the former. Once v goes out of scope, c_2 is live only through the reference from r. At this point in any execution of the original program, either c_2 and c_3 are live or c_4 and c_5 are live; c_1 is definitely live because of the binding for u. But in the analysis, because of the reference from r, c_2 is kept live and its size is included in *live*. Thus, join-value r causes *live* to be loose by one *cons*.

In general, at any point at which all references to a data construction v are lost except for references from a join-value, there is a possibility that *live* is loose because it includes the size of v when it should not. These points arise immediately after decrements to $rc(v)$ caused by (1) a variable or parameter going out of scope or (2) parts of data becoming garbage after the application of a selector. v may then be an excess in *live* caused by a join-value j which in

case (1), is in the result of a function call or **let** expression and in case (2), is in the result of the selector. The *exs* attributes of join-values in the results of function calls, **let** expressions and selectors are recomputed and the value of *live* adjusted appropriately. Observe that v may be part of a join-value j' that is not in the result of these expressions. It can be shown that loss of references to v at the completion of the expressions, has no effect on $exs(j')$ and so we do not recompute it. Note that recomputing *exs* is used only to obtain tighter bounds, so calling or not calling it at any point in the analysis is safe.

5   Optimizations

We use two optimizations that reduce the asymptotic complexity of live heap analysis for many programs. The first optimization avoids looking at data structures without join-value descendants when recomputing *exs* attributes. This is done by adding to con-values and join-values a boolean attribute that indicates the presence of join-value descendants. The second optimization is as follows: at any point p during the execution of a bound function, a join-value j with branches b_1 and b_2 and without any join-value descendants may be reduced to b_1 if b_1 leads to equal or greater live heap usage as compared to b_2. This holds with b_1 and b_2 interchanged also.

The stack and live heap can be viewed as a graph: con-values and join-values in the heap and formal parameters of functions and **let**-bound variables on the stack are vertices; references from variables, con-values and join-values to con-values and join-values are edges. The subgraph comprised of nodes and edges reachable from a node x is an edge-ordered DAG G_x rooted at x. It is acyclic because we are dealing with a first-order functional language. The ordering of fields in data constructions imposes an ordering on the out-edges from nodes. For example, if $x = cons(1, y)$ and y is not a primitive value, then G_x contains the edge $\langle x, y, 2 \rangle$. We say that a vertex u is *contained-in* a vertex v if v is an ancestor of u in every path from a node for a parameter or variable to u.

Reducibility of Join-values. $j = join(b_1, b_2)$ is reducible to b_1 at a point p_0 during execution of a bound function if at p_0

R0. j does not have any join-value descendants.
R1. G_{b_1} and G_{b_2}, the DAGs rooted at b_1 and b_2, are isomorphic. Let f be an isomorphism between G_{b_1} and G_{b_2}.
R2. Corresponding primitive values in b_1 and b_2 and their descendants are equal, taking $uk = uk$.
R3. For every node d_1 of G_{b_1}, if d_1 is not contained-in j, then d_1 and $f(d_1)$ are the same node.

R0 implies that j represents exactly two data structures: b_1 and b_2. R1 and R2 state that b_1 and b_2 have the same structure and contents. The only possible difference between b_1 and b_2 is the particular heap locations they use. No operation in our language can distinguish b_1 and b_2; recall that we don't consider eq?. Thus, R1 and R2 ensure that the program's execution is the same regardless
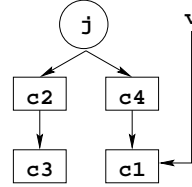
of whether b_1 or b_2 is used, except for the heap space used by b_1 and b_2. R3 asserts that b_1 always contributes at least as much to the live heap space as b_2. For example, this happens if b_2 references data constructions that are live even without references from b_2 and the corresponding data constructions in b_1 are live only because of the references from b_1.

As an example, consider the expression below. Also shown is the abstract heap at the point just after the evaluation of the conditional. Con-values c1

let $v = cons(uk, nil)$ **in**
  if uk
  then $cons(uk, cons(uk, nil))$
  else $cons(uk, v)$

through c4 are numbered according to their syntactic order of appearance. The join-value result j of the conditional satisfies conditions R0 through R3, and hence may be reduced to its left branch.

Theorem 1. *If $j = join(b_1, b_2)$ is reducible to b_1 at a point p_0 during execution of a bound function, then it is safe to replace all references to j with references to b_1, i.e., space-bound analysis still returns an upper bound on live heap usage.*

Sketch of Proof: Based on the above arguments, it suffices to show that b_1 contributes at least as much to the live heap space as b_2, at p_0 and thereafter. The contribution of b_i to *live* is the amount of data contained-in j and referenced by b_i. At p_0, because of R1 and R3, for every descendant of b_2 that is contained-in j, there exists a unique descendant of b_1 that is contained-in j. So, at p_0, b_1 contributes at least as much to *live* as b_2. It can be shown that this continues to be the case after p_0. The complete proof is in [26].

6 Handling Tail Call Optimization

Tail call optimization [1] is handled in our analysis by recognizing function calls in tail position and at the sites of these calls, garbage collecting all variables in the current scope. The transformation for space functions is straight-forward but that for bound functions is more involved. Consider a conditional (**if** uk **then** e_2 **else** e_3) that is in tail position. Both e_2 and e_3 need to be evaluated. Suppose both e_2 and e_3 contain tail calls. During the evaluation of e_2, all environment variables $u_1, \ldots, u_m$ are garbage collected just before the tail call. But at the start of evaluation of e_3, $u_1, \ldots, u_m$ are still live, so the effects of the earlier garbage collection have to be reversed before evaluating e_3. Also, references from the result of e_2 to $u_1, \ldots, u_m$ should not hinder their garbage collection before the tail call in e_3. Similar issues arise when only one or none of the two branches contain tail calls.

reversal w/append		insertion sort		selection sort		merge sort		quick sort			longest common subseq.		string edit		binary tree insert		
n	result	n	result	n	result	n	result	n	space	bound	n	result	n	result	h	n	result
100	299	100	299	100	5150	5	16	2	6	6	100	402	100	1000	2	7	33
500	1499	500	1499	500	125750	12	36	6	32	100	500	2002	500	5000	7	255	792
1000	2999	1000	2999	1000	501500	15	45	7	41	197	1000	4002	1000	10000	9	1023	3102

Fig. 4. Results of live heap space analysis. n is the input size. For binary tree insert, h is the height of the complete binary tree and n is the number of nodes in the tree.

7   Experiments

We implemented the analyses and measured the results for several standard list and tree processing programs. Comparisons of results of space and bound functions show that bound functions produce exact bounds for all but one example. For most of the examples, the bound functions have the same asymptotic time complexities as the corresponding space functions. For all examples, a comparison of the running times of bound functions and the running times of space functions multiplied by the number of represented inputs showed that the bound functions are asymptotically faster than applying the corresponding space functions to all represented inputs. The non-termination issue mentioned in Section 1 is not a problem for any of these examples.

Figure 4 contains the results of live heap space analysis on some examples. For all examples except quicksort, we show only the results of bound functions on partially known inputs, because they are the same as the results of the space functions on worst-case input. Reversal using append is the standard quadratic-time version. The version of merge sort tested is the one that splits the input list into sublists containing the elements at odd and even positions. Dynamic programming algorithms [5] are used for longest common subsequence and string edit. Binary-tree insertion involves insertion of an item into a complete binary tree in which each node is a list containing an element and left and right subtrees.

The partially known inputs for the bound functions of reversal and sorting are lists of known lengths n where all elements are uk; those for longest common subsequence and string edit are two such lists of equal length n. The bound function for binary-tree insertion inserts uk into a complete binary tree of known height h with unknown elements.

The difference between the results of the space and bound functions of quicksort is due to the use of uk; every uk element of the input list is both greater and less than every other uk element. The space-bound function of quicksort terminates, while the time-bound function for quicksort in [18] diverges. This is essentially because join-values retain more information than partially known structures. See [26] for more details.

The results in Figure 4 include the space used by top-level arguments since these arguments are indeed live throughout the execution of the program. Figure 5 contains running times of live heap analysis of some examples; more results appear in [27, 26]. For all examples, the live heap space function has the same

reversal w/append				insertion sort				selection sort				longest common subseq.			
n	S	B	Bopt	n	S	B	Bopt	n	S	B	Bopt	n	S	B	Bopt
10^1	1.0 m	0	1.0 m	10^1	1.0 m	1.0 M	9.0 m	10^1	0	37.3 s	10.0 m	10^1	10.0 m	0.8 s	44.0 m
10^2	0.1 s	0.3 s	0.1 s	10^2	0.1 s		5.1 s	10^2	0.2 s		5.0 s	10^2	6.6 s		24.0 s
10^3	10.7 s	3.5 M	11.9 s	10^3	12.8 s		1.5 H	10^3	27.3 s		1.5 H	10^3	2.0 H		7.1 H

Fig. 5. Running times of live heap space and live heap space-bound functions. Columns S, B and Bopt contain times of space, unoptimized space-bound and optimized space-bound functions, respectively. n and h are as in Figure 4. m is milliseconds, s is seconds, M is minutes and H is hours. Blank fields in B and Bopt columns indicate terminating but long analyses that were aborted after a few days.

asymptotic time complexity as the original function. The time complexities of the live heap bound functions of reverse using append, string edit and longest common subsequence are the same as the complexities of the corresponding original functions. The time complexities of the optimized bound functions of insertion sort and selection sort are a linear factor more than those of the original functions due to the computation involved in reducing join-values. The running time of the bound function of merge sort is more than polynomial in the size of the input. This is because the analysis examines all $(n+m)!/(n! \times m!)$ ways in which two sorted lists of sizes n and m may be merged in sorted order. The running time of the bound function of binary tree insert is polynomial in the size of the input. The first optimization in Section 5 improves the asymptotic complexity of reverse using append by a linear factor. The second optimization improves the asymptotic complexities of insertion sort, selection sort and longest common subsequence from greater than polynomial to polynomial. These speedups are shown in Figure 5.

We applied live heap space analysis handling tail call optimization to tail-recursive versions of reverse, insertion sort, selection sort and an optimized Ackermann's function. Comparing the results with those of the corresponding non-tail-recursive programs showed that tail call optimization does significantly reduce heap usage. For example, tail-recursive insertion sort uses only $O(n)$ space, while non-tail-recursive insertion sort uses $O(n^2)$ space. The optimized Ackermann's function [19] is a systematically derived program which has much better time complexity than the classical version. Let n and m be the first and second arguments to the function. The space complexity of this program was worked out by hand to be $O(n)$, but it is hard to see this because of the complicated space usage of the program. The results of our analysis for $n \in [0,3]$, $m \in [2,10]$ do not prove $O(n)$ space usage but helped confirm that, for a given n, the space usage is independent of m. Computing Ackermann's function for $n > 3$ is famously expensive.

We applied our analysis to a 600-line `calendar` benchmark. The partially known inputs used are partially known dates. The analysis takes only a few seconds to complete and yields tight bounds, providing preliminary evidence for the scalability of our method. We plan to analyze more benchmarks. We have also used the analysis in teaching programming languages courses.

8 Discussion

Correctness. More detailed correctness arguments appear in [26]. We are also working on more formal proofs.

Termination. The space function terminates iff the original program terminates. The bound function might not terminate, even when the original program does if the recursive structure of the original program directly or indirectly depends on unknown parts of a partially known input structure. For example, if the given partially known input structure is uk, then the bound function for any recursive program does not terminate; if such a bound function counts new space, then the original program might indeed take an unbounded amount of space. Indirect dependency on unknown data can be caused by an imprecise join operation. Making the join operation more precise might eliminate this source of non-termination. Another strategy that could be especially effective to detect non-termination of bound functions corresponding to terminating programs is as follows: for every call to a bound function, a check is made to see if the same call with equivalent arguments is on the call stack. If so, the analysis stops and reports that the bound function diverges.

Although there are other methods to deal with non-termination, incorporating such methods in our analysis could result in loose bounds on space usage, even for programs for which non-termination is not a problem. Further, non-termination is not a problem in any of the examples we analyzed.

Scalability. For large programs or programs with sophisticated control structures, the analysis is efficient if the input parameters are small, but for larger parameters, efficiency might be a challenge. One approach is to improve efficiency by memoizing calls to bound functions and reusing the memoized results wherever possible. Another strategy is to use the results of space-bound analysis on smaller inputs to semi-automatically derive closed forms and/or recurrence relations that describe the program's space usage, by fitting a given functional form to the analysis results. The closed forms or recurrence relations may then be used to determine space bounds for large inputs.

Inputs to Bound Functions. To analyze space usage with respect to some property of the input, we need to formulate sets of partially known inputs that represent all actual inputs with that characteristic, e.g., all lists with length n, all binary trees of height h or all binary trees with n nodes. As an example, $\{(uk, (uk, nil, nil), nil), (uk, nil, (uk, nil, nil)), (uk, (uk, nil, nil), (uk, nil, nil))\}$ represents all binary trees of height 1, each node being a list of the element and left and right subtrees. Often, formulating such sets of partially known inputs is straightforward but tedious for the user to do by hand. However, it is easy to write programs that generate sets of partially known inputs.

Imperative Update and Higher-Order Functions. The ideas in this paper may be combined with reference-counting garbage collection extended to handle cycles [3] or with other garbage collection algorithms, such as mark and sweep, to obtain a live heap space analysis for imperative languages. They may also be combined with techniques for analysis of higher-order functions [10].

9 Related Work

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis, e.g., [17, 24, 18]. Analysis of live heap space is different because it involves explicit analysis of the graph structure of the data.

Most of the work related to analysis of space is on analysis of cache behavior, e.g., [28, 9], much of which is at a lower language level and does not consider liveness. Live heap analysis is a first step towards analysis of cache behavior in the presence of garbage collection.

Persson's work on live memory analysis [22] requires programmers to give annotations, including numerical bounds on the size of recursive data structures. His work is preliminary: the presentation is informal, and only one example, summing a list, is given. Our analysis does not require annotations.

Unlike static reference counting used for compile-time garbage collection [15], our analysis uses a reference counting method similar to that in run-time garbage collection. The former keeps track of pointers to memory cells that will be used later in the execution. Our analysis could be modified so that $decrc(v)$ is called when a parameter or **let**-variable won't be used again (instead of waiting until v goes out of scope). Our current analysis corresponds to the garbage collection behavior in, e.g., JVMs from Sun, IBM, and Transvirtual.

Several type systems [14, 13, 6, 12] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [14, 6]. They require programmers to annotate their programs with cost functions as types. Furthermore, some programs must be rewritten to have feasible types [14, 13].

Chin and Khoo [4] propose a method for calculating sized types by inferring constraints on size and then simplifying the constraints using Omega [23]. Their analysis results do not correspond to live heap space in general. Further, Omega can only reason about constraints expressed as linear functions.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 2 edition, 1996.
2. P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the 8th EuroMicro Workshop on Real-Time Systems*, pages 102–107, L'Aquila, June 1996.
3. D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A non-intrusive multiprocessor garbage collector. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2001.
4. W.-N. Chin and S.-C. Khoo. Calculating sized types. In *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72. ACM, New York, Jan. 2000.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press/McGraw-Hill, 1990.
6. K. Crary and S. Weirich. Resource bound certification. In *Conference Record of the 27th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 2000.

7. P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, Feb. 1991.
8. *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, Sept. 1989.
9. 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.
10. G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–86. ACM Press, 2002.
11. R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund University, Sept. 1998.
12. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*. ACM Press, Jan. 2003.
13. 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, New York, Sept. 1999.
14. 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, New York, Jan. 1996.
15. S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [8], pages 54–74.
16. *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, May 1999.
17. D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.
18. Y. A. Liu and G. Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, Dec. 2001.
19. Y. A. Liu and S. D. Stoller. Optimizing Ackermann's function by incrementalization. Technical Report DAR 01-1, Computer Science Department, SUNY Stony Brook, Jan. 2001.
20. M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 248–259. ACM, New York, 1992.
21. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.
22. P. Persson. Live memory analysis for garbage collection in embedded systems. In LCTES 1999 [16], pages 45–54.
23. W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.
24. M. Rosendahl. Automatic complexity analysis. In FPCA 1989 [8], pages 144–156.
25. I. Ryu. Issues and challenges in developing embedded software for information appliances and telecommunication terminals. In LCTES 1999 [16], pages 104–120. Invited talk.
26. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. Technical Report DAR 01-2, Computer Science Dept., SUNY at Stony Brook, Oct. 2001, Available at http://www.cs.sunysb.edu/~stoller/dar012.html.
27. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 102–111. ACM Press, 2001.
28. 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, June 1998.
29. P. Zimmermann and W. Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. In *Computer and Information Sciences VI*. Elsevier, 1991.