Automatic Accurate Live Memory Analysis for Garbage-Collected Languages

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

ABSTRACT

This paper describes a general approach for automatic and accurate live heap space and live heap space-bound analyses for high-level languages. The approach is based on program analysis and transformations and is fully automatic. The space-bound analysis produces accurate (tight) upper bounds in the presence of partially known input structures. The analyses have been implemented and experimental results confirm their accuracy.

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. In particular, space analysis is becoming important due to the increasing uses of high-level languages with garbage collection, such as Java [22], the importance of cache memories in performance [28], and the stringent space requirements in the growing area 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 [9]. For example, analysis of worst-case execution time in real-time systems often uses loop bounds or recursion depths [21, 1] 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 [16]. 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 even if garbage collection is performed whenever garbage is created. 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 used to help analyze the space usage of some continuosly running processes that have cyclic behavior.

Our approach starts with a given 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. 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 [18, 29]. The analysis is mainly asymptotic. A theoretically 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 [18, 24, 6]. Although much progress has been made in this area, closed forms are known only for subclasses of functions. Thus, such optimization can not be automatically done for analyzing general programs.

*The authors gratefully acknowledge the support of ONR under grants N00014-99-1-0132, N00014-99-1-0358 and N00014-01-1-0109 and of NSF under grants CCR-9711253 and CCR-9876058. Authors' address: Computer Science Department, SUNY at Stony Brook, Stony Brook, NY 11794-4400 USA. Email: {leena,stoller,liu}@cs.sunysb.edu. Web: www.cs.sunysb.edu/~{leena,stoller,liu}. Phone: (631)632-1627. Fax: (631)632-8334.

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. 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.

One caveat of using partially known input structures is that the space-bound function might not terminate even if the original program does. This occurs only if the recursive/iterative structure of the original program depends on unknown parts of the given partially known input structures.

Our analysis uses reference counts [14]. We are analyzing purely functional programs, so reference counting provides an accurate basis for determining liveness. The analysis and transformations are performed at source level. This allows implementations to be independent of compilers and underlying systems and allows analysis results to be understood at source level. Our space bound analysis is an abstract interpretation, expressed conveniently as a program transformation. 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.

2. LANGUAGE

We use a first-order, call-by-value functional language that has literal values of primitive types (e.g., Boolean and integer constants), structured data, operations on primitive types (e.g., Boolean and arithmetic operations), 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

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

We sometimes use infix notation for binary primitive operations.

For brevity, we assume the language contains only two kinds of primitive operations that take data constructions as arguments. A tester application $c?(v)$ returns *true* iff v has outermost constructor c. A selector application $c^{-i}(v)$ returns the i'th component of a data construction v with outermost constructor c. Our analysis can easily be extended to handle other similar operations such as equality predicates.

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. In examples, we use a constructor *cons* with arity 2.

3. LIVE HEAP SPACE FUNCTION

To analyze the live heap space used by a program in the presence of completely 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 of data is ascertained using reference counts. The *reference count* for a data construction v is the number of pointers to v. These may be pointers on the stack, created by **let** bindings or bindings to formal parameters of functions, or pointers on the heap, created by data constructions.

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 1 produces live heap space functions. It 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 $ml \cdot P$ where 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) 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.

Updating Reference Counts. $rc(v)$, for a data construction 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. An auxiliary function gc ("garbage collect") is called whenever data can become garbage. For a data construction v, $gc(v)$ decrements $rc(v)$ and then, if $rc(v)$ is not positive, it decrements the appropriate element of *live* and calls gc recursively on the children of v. 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, $cons(0, 1)$ is garbage after the application of $cons^{-2}$ in $cons(cons^{-2}(cons(0, 1)), 2)$; 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. At the end of function calls and **let** expressions, values bound to parameters and variables should be garbage collected without garbage collecting the result of the function call or the **let** expression. After selector applications, data selected from should be garbage collected without garbage collecting the selected part, even if the reference count of that part becomes 0. Figure 5 contains an example of a live heap space function.

4. LIVE HEAP SPACE BOUND FUNCTION

The transformation $\mathcal{L}_b$ in Figures 2-3 produces live heap space-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. As before, *maxlive* contains the maximum value of *live* so far during execution. The presence of partially known inputs in bounds analysis causes uncertainty. For conditional expressions whose tests evaluate to *uk*, both branches are evaluated to determine the maximum live heap space usage.

Correctness of live heap analysis depends on keeping track of all references and reference counts meticulously. Summarizing the results of two branches into a single partially known structure that represents both results, as is done in timing analysis [19] and stack space and heap allocation analysis [27], does not work for live heap analysis because it would be impossible to keep track of reference counts accurately. So the result of a conditional whose test evaluates to *uk* is a separate entity, a join-value, that points to both possible results and has its own reference count. By keeping both results live, we run the risk of obtaining loose bounds, since *live* might include the sizes of both results when only one of them is live. To keep *live* as tight as possible, we examine join-values at appropriate points of execution and manipulate *live* so that it includes the size of only a single largest data structure pointed to by each join-value.

4.1 Abstract Data Types

Two abstract data types (ADTs), the join-value type and the con-value ("constructed-value") type, are 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 l has a list *branches*(l) containing references to con-values and/or join-values. Primitive values, if any, in the set represented by l are not stored in l. Thus, if *branches*(l) has only one element, l represents a choice between that element and some primitive value. l has an associated constructor count vector *exs*(l). Parts of the data constructions represented by l may be live regardless of l. 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 an excess and is stored in *exs*(l), as discussed in Section 4.3. *live* does not include *exs*(l). When l becomes garbage, *exs*(l) is added to *live* just before garbage collecting the branches of l.

The con-value type is an extension of the ADT described in Section 3. con-values and join-values have a reference count and a list of join-parents. A join-value l is a join-parent of v if *branches*(l) references v. Functions *rc*, *incrc*, *decrc*, *joinParents*, *addJoinParent*, and *delJoinParent* apply to both ADTs; the names indicate their meanings. *newjoin*(b) creates a join-value l with a list b of branches, and with *rc*(l) initialized to 0, *joinParents*(l) initialized to *nil*, and *exs*(l) initialized to the zero vector, denoted V_0.

4.2 Conditionals, Selectors and Testers

Consider a conditional expression $(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^\dagger$ whose condition 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 expression 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 r_3 are primitive, then r 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 r is r_2. Otherwise, r is a join-value, and *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).

Selectors and testers return *uk* if given *uk* arguments. For join-values, the selector or tester is first applied to the branches and the *join* of the results is returned. The *exs* field of a join-value l that is the result of applying a selector to another join-value l' is set to V_0, because when l is created, l' is live, and *exs*(l') already takes care of any excess.

4.3 Achieving Tightness of *live* in the Presence of Join-values

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

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

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 l, there is a possibility that *live* is loose because it includes the size of v when it should not. Note that this occurs only if l stays live even after the afore-mentioned

$$
\begin{aligned}
&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)]\!] = \mathit{live}[i_c]{+}{+};\ \mathit{maxlive} = \max(\mathit{live}, \mathit{maxlive});\\
&\qquad\qquad\qquad\quad \textbf{let } r_1 = \mathcal{L}[\![e_1]\!],\ldots,r_n = \mathcal{L}[\![e_n]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\ \mathit{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?(e)]\!] = \textbf{let } x = \mathcal{L}[\![e]\!] \textbf{ in}\\
&\qquad\qquad\quad \textbf{let } r = c?(\mathit{data}(x)) \textbf{ in}\\
&\qquad\qquad\qquad (\textbf{if } \mathit{not}(\mathit{isPrim}(x)) \mathit{ and } \mathit{rc}(x) = 0 \textbf{ then } \mathit{gc}(x));\ r\\
&\mathcal{L}\left[\!\left[c^{-i}(e)\right]\!\right] = \textbf{let } x = \mathcal{L}[\![e]\!] \textbf{ in}\\
&\qquad\qquad\qquad \textbf{let } r = c^{-i}(\mathit{data}(x)) \textbf{ in}\\
&\qquad\qquad\qquad\quad (\textbf{if } \mathit{not}(\mathit{isPrim}(x)) \mathit{ and } \mathit{rc}(x) = 0 \textbf{ then } \mathit{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}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] = \textbf{let } v = \mathcal{L}[\![e_1]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{incrc}(v);\\
&\qquad\qquad\qquad\qquad\qquad \textbf{let } r = \mathcal{L}[\![e_2]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{gcExcept}(v,r);\ r\\
&\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}\\
&\qquad\qquad\qquad\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\\
&\qquad\qquad\qquad\qquad \textbf{let } r = f_L(r_1,\ldots,r_n) \textbf{ in}\\
&\qquad\qquad\qquad\qquad\quad \mathit{gcExcept}(r_1,r);\ldots;\mathit{gcExcept}(r_n,r);\ r\\
&\mathit{gc}(v) = \textbf{if } \mathit{not}(\mathit{isPrim}(v))\\
&\qquad\quad \textbf{then } \mathit{decrc}(v);\\
&\qquad\qquad\quad \textbf{if } \mathit{rc}(v) \le 0\\
&\qquad\qquad\quad \textbf{then } \mathit{live}[\mathit{conType}(v)]{-}{-};\\
&\qquad\qquad\qquad\quad \textbf{for } i = 1..\mathit{arity}(v)\ \mathit{gc}(c^{-i}(\mathit{data}(v)))\\
&\mathit{gcExcept}(u,v) = \mathit{incrc}(v);\ \mathit{gc}(u);\ \mathit{decrc}(v)
\end{aligned}
$$

Figure 1: Transformation that produces live heap space functions f_L. $\mathit{isPrim}(v)$ returns *true* iff v is primitive. $\mathit{conType}(v)$ returns an integer i_c that uniquely identifies the outermost constructor c in $\mathit{data}(v)$. $\mathit{arity}(v)$ returns the arity of the outermost constructor in $\mathit{data}(v)$.

loss of references to v. 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 l which in case (1), is in the result of the function call or the **let** expression and in case (2), is in the result of the selector. Observe that v cannot be part of join-values that are not in the results of these expressions since we are dealing with functional languages with no destructive update. *recomputeExs*, defined in Figure 3, is called on the results of function calls, **let** expressions and selectors to compute the *exs* attributes of join-values in the results and adjust the value of *live* appropriately.

We now formally define the *exs* attribute of join-values. Consider the stack and live heap 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, including references in *branches* attributes but excluding references in *joinParents* attributes, are edges. We say that u is *contained-in* v if v is an ancestor of u in every path from a node for a parameter or variable to u. For a join-value l, let C_l denote the set of all con-values and join-values contained-in l, and let G_l denote the graph comprising vertices and edges reachable from l. A *join-path* of l is a connected subgraph of G_l containing l and constructed from G_l by selecting at every join-value l' reachable from l, exactly one branch of l' and then, after all selections have been made, eliminating unreachable vertices and edges. Figure 4 contains examples of join-paths. Join-paths of l correspond to data structures represented by

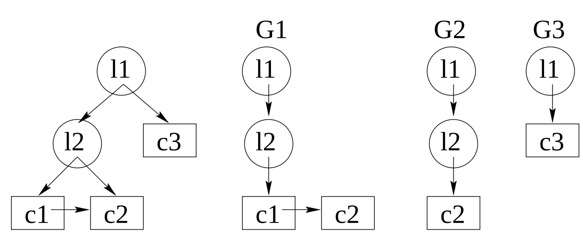

Figure 4: Examples of join-paths. $G1$, $G2$ and $G3$ are join-paths of join-value $l1$. Circles denote join-values and rectangles denote con-values.

l. $\mathit{conCountVec}(u)$ for a con-value u is a constructor count vector in which the count of the constructor type of u is 1 and all other counts are 0. $\mathit{maxJoinPath}(l)$ is the maxi-

$$
\begin{array}{l}
f_{Lb}(v_1,\ldots,v_n) = \mathcal{L}_b\,[\![e]\!] \qquad \text{where } e \text{ is the body of function } f\text{, i.e., } f(v_1,\ldots,v_n)=e \\
\mathcal{L}_b\,[\![v]\!] = v \\
\mathcal{L}_b\,[\![l]\!] = l \\
\mathcal{L}_b\,[\![c(e_1,\ldots,e_n)]\!] = \text{same as } \mathcal{L}\,[\![c(e_1,\ldots,e_n)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b \\
\mathcal{L}_b\,[\![p(e1,\ldots,en)]\!] = p_u(\mathcal{L}_b\,[\![e1]\!],\ldots,\mathcal{L}_b\,[\![en]\!]) \\
p_u(v_1,\ldots,v_n) = \textbf{if } v_1 = uk \textit{ or } \cdots \textit{ or } v_n = uk \textbf{ then } uk \textbf{ else } p(v_1,\ldots,v_n) \\
\mathcal{L}_b\,[\![c?(e)]\!] = \text{same as } \mathcal{L}[\![c?(e)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b\text{, and replace } c?(data(x)) \text{ with } c?_u(x) \\
c?_u(v) = \textbf{if } v = uk \textbf{ then } uk \\
\qquad\quad \textbf{else if } isJoin(v) \textbf{ then if } length(branches(v)) = 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } join(false, c?_u(first(branches(v)))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } join(c?_u(first(branches(v))), c?_u(second(branches(v)))) \\
\qquad\qquad \textbf{else } c?(data(v)) \\
\mathcal{L}_b\,[\![c^{-i}(e)]\!] = \text{same as } \mathcal{L}\,[\![c^{-i}(e)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b \text{ and } c^{-i}(data(x)) \text{ with } c_u^{-i}(x) \text{ and call } recomputeExs(r) \text{ after} \\
\qquad\qquad gcExcept(x,r) \text{ in the conditional} \\
c_u^{-i}(v) = \textbf{if } v = uk \textbf{ then } uk \\
\qquad\quad \textbf{else if } isJoin(v) \\
\qquad\qquad \textbf{then if } length(branches(v)) = 1 \textbf{ then } c^{-i}(false) \\
\qquad\qquad\quad \textbf{else } join(c_u^{-i}(first(branches(v))), c_u^{-i}(second(branches(v)))) \\
\qquad\qquad \textbf{else } c^{-i}(data(v)) \\
\mathcal{L}_b\,[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = \\
\quad \textbf{let } b = \mathcal{L}_b[\![e_1]\!] \textbf{ in} \\
\quad\; \textbf{if } b = uk \textbf{ then let } l_1 = copy(live) \textbf{ in} \\
\qquad\qquad\qquad \textbf{let } r_2 = \mathcal{L}_b\,[\![e_2]\!] \textbf{ in} \\
\qquad\qquad\qquad\; \textbf{let } l_2 = copy(live) \textbf{ in} \\
\qquad\qquad\qquad\quad live = l_1; \textbf{let } r_3 = \mathcal{L}_b\,[\![e_3]\!] \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let } l_3 = copy(live) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad\; 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 \\
\quad\; \textbf{else if } b \textbf{ then } \mathcal{L}_b[\![e_2]\!] \textbf{ else } \mathcal{L}_b[\![e_3]\!] \\
\mathcal{L}_b\,[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] = \textbf{let } v = \mathcal{L}_b\,[\![e_1]\!] \textbf{ in} \\
\qquad\qquad\qquad\qquad incrc(v); \textbf{let } r = \mathcal{L}_b[\![e_2]\!] \textbf{ in } (gcExcept(v,r); recomputeExs(r);\ r) \\
\mathcal{L}_b\,[\![f(e_1,\ldots,e_n)]\!] = \textbf{let } r_1 = \mathcal{L}_b\,[\![e_1]\!],\ldots,r_n = \mathcal{L}_b\,[\![e_n]\!] \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\; gcExcept(r_1,r);\ldots;gcExcept(r_n,r); recomputeExs(r);\ r
\end{array}
$$

Figure 2: 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.

mum of the sizes of all join-paths of l, where $size(P)$ for a join-path P of l is defined as

$$size(P) = \sum_{u \text{ is a con-value in } P \cap C_l} conCountVec(u)$$

$exs(l)$ is then defined as follows if both branches of l are non-primitive and contained-in l (otherwise, $exs(l)$ is V_0).

$$exs(l) = total(l) - sub(l) - maxJoinPath(l) \qquad (2)$$

$$total(l) = \sum_{u \text{ is a con-value in } C_l} conCountVec(u) \qquad (3)$$

$$sub(l) = \sum_{u \text{ is a join-value in } C_l} exs(u) \qquad (4)$$

The sums and differences of constructor count vectors are computed component-wise. (2) does not result in vectors with negative counts, since sub(l) and $maxJoinPath$(l) count data in disjoint subsets of C_l. This is justified as follows : if the join-path P which contributes to $maxJoinPath$(l) contains a join-value l', then P contains a largest join-path of l' and $exs(l')$ counts data in the other join-paths of l'. Hence, $exs(l')$, for any descendant join-value l' of l, counts data in join-paths that are not part of P. Informally, (2) says "subtract from $live$ everything except a largest join-path and nodes that have already been subtracted from $live$". In our implementation, $exs(l)$ is computed using (2) if the elements in $branches(l)$ have reference counts equal to 1; $exs(l)$ is conservatively set to V_0 otherwise.

A recomputed value of $exs(l)$ can be less than the existing value of $exs(l)$. This happens only if after the computation of the latter, selectors are applied to l creating a new join-value l' that references parts of l and a subsequent

$$
\begin{array}{l}
join(v_1, v_2) = \textbf{if } eq?(v_1, v_2) \textbf{ then } v_1 \\
\qquad\qquad\qquad \textbf{else if } isPrim(v_1) \\
\qquad\qquad\qquad\quad \textbf{then if } isPrim(v_2) \textbf{ then } uk \\
\qquad\qquad\qquad\qquad \textbf{else let } result = newjoin([v_2]) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad incrc(v_2); addJoinParent(v_2, result); result \\
\qquad\qquad\qquad\quad \textbf{else if } isPrim(v_2) \\
\qquad\qquad\qquad\qquad \textbf{then let } result = newjoin([v_1]) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad incrc(v_1); addJoinParent(v_1, result); result \\
\qquad\qquad\qquad\qquad \textbf{else let } result = newjoin([v_1, v_2]) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad incrc(v_1); addJoinParent(v_1, result); \\
\qquad\qquad\qquad\qquad\quad incrc(v_2); addJoinParent(v_2, result); \\
\qquad\qquad\qquad\qquad\quad result \\
gc(v) = \textbf{if } not(isPrim(v)) \\
\qquad \textbf{then } decrc(v); \\
\qquad\quad \textbf{if } rc(v) \leq 0 \\
\qquad\quad \textbf{then if } isJoin(v) \\
\qquad\qquad \textbf{then } live = live + exs(v); \\
\qquad\qquad\quad \textbf{for } u \textbf{ in } branches(v) \\
\qquad\qquad\quad\; delJoinParent(u, v); gc(u) \\
\qquad\qquad \textbf{else } live[conType(v)]{-}{-}; \\
\qquad\qquad\quad \textbf{for } i = 1..arity(v)\ gc(c^{-i}(data(v))) \\
recomputeExs(v) = \textbf{if } not(isPrim(v)) \textbf{ then} \\
\qquad\qquad\qquad \textbf{if } isJoin(v) \\
\qquad\qquad\qquad \textbf{then if } length(branches(v)) = 2 \textit{ and } containedIn_0(branches(v)) \\
\qquad\qquad\qquad\quad \textbf{then let } newexs = computeExs(v) \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{if } newexs > exs(v) \\
\qquad\qquad\qquad\qquad \textbf{then } live = live + exs(v) - newexs; setexs(v, newexs) \\
\qquad\qquad\qquad\quad \textbf{else for } u \textbf{ in } branches(v)\ recomputeExs(u) \\
\qquad\qquad\qquad \textbf{else for } i = 1..arity(v)\ recomputeExs(c^{-i}(data(v))) \\
containedIn_0(ls) = \textbf{if } null(ls) \textbf{ then } true \\
\qquad\qquad\qquad \textbf{else if } rc(cons^{-1}(ls)) = length(joinParents(cons^{-1}(ls))) = 1 \\
\qquad\qquad\qquad\quad \textbf{then } containedIn_0(cons^{-2}(ls)) \\
\qquad\qquad\qquad\quad \textbf{else } false
\end{array}
$$

Figure 3: Auxiliary functions *join*, *gc*, *recomputeExs* and $containedIn_0$. For brevity, we leave out the definition of *computeExs* which is based on (2) in Section 4.3.

garbage collection leads to the afore-mentioned recomputation of $exs(l)$. The references from l' to parts of l cause these parts to not be contained-in l and so (2) produces a smaller value than the existing $exs(l)$. But selection from l does not alter the fact that only one of the data constructions represented by l is live, so the new smaller value of $exs(l)$ is artificial and is ignored. [27] supplies a more detailed justification of how the contribution of a con-value v to $exs(l)$ does not change after v first becomes contained-in l despite any further references created to v. On a related note, garbage collections after testers do not lead to data becoming newly contained-in join-values, so *recomputeExs* is not called at those points. Figure 5 contains an example of a live heap space-bound function.

4.4 Optimizations

We use two optimizations that reduce the asymptotic complexity of live heap analysis for many programs. They are only briefly described here due to space constraints. The first optimization avoids calls to *recomputeExs* on data without join-value descendants. 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 reduces some join-values to con-values, thus avoiding expensive manipulations of the former. At point p, a join-value l with branches b_1 and b_2 and without any join-value descendants may be reduced to b_1 if b_1 leads to at least as much live heap usage as compared to b_2. Our optimization is conservative and reduces l to b_1 if b_1 and b_2 have the same shape, contain equal primitive values at corresponding locations, and for every descendant d_1 of b_1 that is not contained-in l, the corresponding descendant d_2 of b_2 is the same as d_1. These conditions ensure that at point p, b_1 contributes at least as much to *live* as compared to b_2 and further, that *live* is maximized at all future points.

5. EXPERIMENTS

We implemented the analyses and measured the results for several standard list and tree processing programs. Comparisons of results of space functions and bound functions show that bound functions produce accurate results (i.e., tight bounds) for all these examples. The results are consistent with the expected asymptotic space complexities of

$reverse(ls, revls) =$ **if** $null(ls)$ **then** $revls$
else $reverse(cons^{-2}(ls), cons(cons^{-1}(ls), revls))$

$reverse_L(ls, revls) =$ **if** (**let** $l = ls$ **in**
  let $l_{null?} = null(data(l))$ **in**
  if $not(isPrim(l))$ and $rc(l) = 0$ **then** $gc(l)$;
  $l_{null?}$)
then $revls$
else **let** $arg1 =$ (**let** $l = ls$ **in**
  let $l_{cdr} = cons^{-2}(data(l))$ **in**
  if $not(isPrim(l))$ and $rc(l) = 0$
  then $gcExcept(l, l_{cdr})$;
  $l_{cdr})^*$
 $arg2 =$ ($live[i_{cons}]{+}{+}; maxlive := \max(live, maxlive)$;
  let $ls_{car} =$ sub-expression $*$ with $cons^{-2}$ replaced by $cons^{-1}$
   $revl = revls$ **in**
  $incrc(ls_{car}); incrc(revl); new(cons(ls_{car}, revl)))$ **in**
 $incrc(arg1); incrc(arg2)$;
 let $r = reverse_L(arg1, arg2)$ **in**
  $gcExcept(arg1, r); gcExcept(arg2, r); r$

$reverse_{L_b}(ls, revls) =$ **let** $ls_{null?} =$ (**let** $l = ls$ **in**
  let $l_{null?} = null_u(l)$ **in**
  if $not(isPrim(l))$ and $rc(l) = 0$ **then** $gc(l)$;
  $l_{null?}$) **in**
if $ls_{null?} = uk$
then **let** $l_1 = copy(live)$ **in**
 let $branch_1 = revls$ **in**
 let $l_2 = copy(live)$ **in**
 $live := l_1$;
 let $branch_2 =$
  (**let** $arg1 =$ (**let** $l = ls$ **in**
   let $l_{cdr} = cons^{-2}{}_u(l)$ **in**
   if $not(isPrim(l))$ and $rc(l) = 0$
   then $incrc(l_{cdr}); gc(l); decrc(l_{cdr})$;
   $l_{cdr})^*$
  $arg2 =$ ($live[i_{cons}]{+}{+}; maxlive := \max(live, maxlive)$;
   let $ls_{car} =$ sub-expression $*$ with
    $cons^{-2}{}_u$ replaced by $cons^{-1}{}_u$
   $revl = revls$ **in**
   $incrc(ls_{car}); incrc(revl); new(cons(ls_{car}, revl)))$ **in**
  $incrc(arg1); incrc(arg2)$;
  let $r = reverse_{L_b}(arg1, arg2)$ **in**
  $gcExcept(arg1, r); gcExcept(arg2, r); recomputeExs(r); r)^\dagger$ **in**
 let $l_3 = copy(live)$ **in**
 $live := \max(l_2, l_3)$;
 let $r = join(branch_1, branch_2)$ **in**
 $setexs(r, \min(l_2 - l_1, l_3 - l_1)); r$
else **if** $ls_{null?}$
 then $revls$
 else sub-expression $\dagger$

Figure 5: Examples of space and space-bound functions. $reverse_L$ **and** $reverse_{L_b}$ **are the space and space-bound functions, respectively, of** $reverse$**.**

the programs. We measured the running times of space and bound functions of all examples. 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 caveat mentioned in Section 1 is not a problem for any of these examples.

Figure 6 contains the results of live heap space analysis on some examples. We do not show the results of space functions on worst-case inputs and bound functions on partially known inputs separately, because the two sets of results are the same in all examples. List reversal is the standard linear-time version; 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 [3] are used for binomial coefficient, 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. For binomial coefficient we use integer arguments, n and $n-2$, since it was found that for a given n, a value of $n-2$ for the second argument leads to maximum live heap usage.

The results in Figure 6 include the space used by top-level arguments since these arguments are indeed live throughout the execution of the program. Figure 7 contains running times of optimized live heap space analysis on a sampling of input sizes. For all examples, the live heap space function has the same asymptotic time complexity as the original function. The time complexities of the live heap space-bound functions of reverse, reverse using append, binomial coefficient, string edit and longest common subsequence are the same as the complexities of the corresponding original functions. The time complexities of the bound functions of insertion sort and selection sort are a linear factor more than those of the original functions. The linear factor is due to the computation involved in the reduction of 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.

6. DISCUSSION

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 is a challenge. However, from the results of bound analysis on smaller inputs, we may 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. Also, when closed forms or recurrence relations are known, we may use the results of the analysis to determine exact coefficients. The closed forms or recurrence relations may then be used to determine space bounds for large inputs.

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 eliminates this source of non-termination.

Although there are 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, among the several examples that we analyzed, in only one example, namely quicksort, is non-termination a problem.

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 partially known inputs that represent classes of actual inputs is straightforward and can easily be done by a simple program.

7. RELATED WORK

There has been a large amount of work on analyzing program cost or resource complexities, but the majority of it is on time analysis, e.g., [18, 24, 26, 19]. Stack space and heap allocation analysis [27] is similar to time analysis [19]. 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, 8], 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, as discussed in Section 1. Live heap space analysis is also a first step towards analyzing cache behavior in the presence of garbage collection.

Persson's work on live memory analysis [22] for an object-oriented language requires programmers to give annotations, including specific numbers as bounds for the size of recursive data structures. His work is preliminary: the presentation is informal, with a few formulas summarizing sizes of data in

list reversal		reversal u/append		insertion sort		selection sort		merge sort		binomial coefficient		longest common subseq.		string edit		binary tree insert		
n	result	n	result	n	result	n	result	n	result	n	result	n	result	n	result	h	n	result
50	100	50	149	50	1325	50	1325	2	5	50	48	50	202	50	500	1	3	18
100	200	100	299	100	5150	100	5150	5	16	100	98	100	402	100	1000	2	7	33
250	500	250	749	250	31625	250	31625	10	31	250	248	250	1002	250	2500	4	31	111
500	1000	500	1499	500	125750	500	125750	12	36	500	498	500	2002	500	5000	7	255	792
1000	2000	1000	2999	1000	501500	1000	501500	15	45	1000	998	1000	4002	1000	10000	9	1023	3102

Figure 6: Results of live heap space-bound functions on partially known inputs. These are also the results of live heap space functions; the two are equal for all of these examples. n is the input size except in the case of binomial coefficient, in which n is the first argument. For binary tree insert, h is the height of the complete binary tree and n is the number of nodes in the tree.

list reversal			reversal u/append			insertion sort			selection sort			merge sort		
n	space	bound	n	space	bound	n	space	bound	n	space	bound	n	space	bound
10	0	0	10	1.0	1.0	10	1.0	9.0	10	0	10.0	10	2.0	692.0
100	4.0	4.0	100	110.0	120.0	100	125.0	5.11 s	100	205.0	5.02 s	100	40.0	> 7 days
1000	35.0	35.0	1000	10.65 s	11.85 s	1000	12.77 s	1.52 H	1000	27.34 s	1.47 H	1000	645.0	> 7 days

binomial coefficient			longest common subseq.			string edit			binary tree insert			
n	space	bound	n	space	bound	n	space	bound	h	n	space	bound
10	5.0	5.0	10	10.0	44.0	10	15.0	20.0	2	7	0	3.0
100	105.0	105.0	100	6.57 s	24.03 s	100	7.13 s	7.41 s	6	127	4.0	115.0
1000	11.64 s	11.64 s	1000	2.01 H	7.12 H	1000	2.16 H	2.32 H	9	1023	34.0	1.22 s

Figure 7: Running times of live heap space and live heap space-bound functions. n and h are as in Figure 6. s is seconds and H is hours. Times without units associated with them are in milliseconds.

bytes based on the annotations, and only one example, summing a list, is given. Our analysis is able to compute bounds based on input size only, without program annotations.

Unlike static reference counting used in analysis for compile-time garbage collection [15, 13], our analysis uses a reference counting method similar to that in run-time garbage collection. While the former keeps track of pointers to memory cells that will be used later in the execution, the latter maintains pointers reachable from the stack at the current point in execution. Inoue and others [12] analyze functional programs to detect run-time garbage conservatively at compile-time. Their result is an approximation without any information about the input. Also, they do not compute the size of live space.

Several type systems [11, 10, 4] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [11, 4]. They require programmers to annotate their programs with cost functions as types. Furthermore, some programs must be rewritten to have feasible types [11, 10].

Chin and Khoo [2] 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.

To summarize, this work is a first attempt to analyze live heap space automatically and accurately using source-level program analysis and transformations. Imperative languages with destructive update allow data with cycles. Reference counting is not suitable for garbage collecting such data. The ideas in this paper may be combined with a suitable garbage collection algorithm such as mark and sweep, to obtain a live heap space analysis for imperative languages.

8. REFERENCES

[1] 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.

[2] 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.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[4] 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.

[5] *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, May 1990.

[6] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, Feb. 1991.

[7] *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, Sept. 1989.

[8] 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.

[9] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Department of Computer Science, Lund University, Sept. 1998.

[10] 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.

[11] 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.

[12] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Trans. Program. Lang. Syst.*, 10(4):555–578, Oct. 1988.

[13] T. P. Jensen and T. Mogensen. A backwards analysis for compile-time garbage collection. In ESOP 1990 [5], pages 227–239.

[14] R. Jones and R. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, New York, 1996.

[15] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [7], pages 54–74.

[16] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Mass., 1968.

[17] *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems.* ACM, New York, May 1999.

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

[19] 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, June 1998.

[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 [17], 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 [7], pages 144–156.

[25] I. Ryu. Issues and challenges in developing embedded software for information appliances and telecommunication terminals. In LCTES 1999 [17], pages 104–120. Invited talk.

[26] D. Sands. Complexity analysis for a lazy higher-order language. In ESOP 1990 [5], pages 361–376.

[27] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical Report 538, Computer Science Dept., Indiana University, Apr. 2000.

[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.