# Optimized Live Heap Bound Analysis

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

**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 improve 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. In particular, space analysis is becoming important due to the increasing uses of high-level languages with garbage collection, such as Java [26], the importance of cache memories in performance [32], and the stringent space requirements in the growing area of embedded applications [29]. 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 [12]. For example, analysis of worst-case execution time in real-time systems often uses loop bounds or recursion depths [25, 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 [19]. The latter is mostly in the form of tracing memory behavior or analyzing cache effects at the machine level [24, 32]. 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. 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. It can also be used to help analyze the space usage of some continuously 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

*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.

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 [21, 33]. 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 [21, 28, 8]. 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.

Rosendahl proposed characterizing inputs using *partially known input structures* [28]. 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. 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. They can thus be used to determine worst-case space usage of a program, given a particular metric such as input size. 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. Finding space bounds is undecidable, so space-bound functions may diverge. In our experience, this is rare.

While our approach can be applied to imperative languages, the analysis in this paper enjoys multiple benefits due to the functional nature of the source language. Associating reference counts with partially known structures forms an accurate basis for determining liveness. With reference counting, the heap never has to be examined in its entirety. In contrast, garbage collection algorithms for imperative languages generally examine the whole heap at once which is costly. Space-bound functions sometimes have to evaluate both branches of conditionals due to unknown values. Our analysis does not need to maintain multiple copies of the heap while evaluating the two branches exactly because of the absence of imperative update. Copying the heap would add significant complexity and overhead. It is necessary to keep the results of both branches. This could lead to loose bounds in a naive analysis, since both results seem live. Our analysis handles this by examining selected parts of the heap at a limited number of program points. If the source language were imperative, the analysis would need to examine much larger parts of the heap and do so more frequently.

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

We sometimes use infix notation for 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 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 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. Data construction $v$ is live if its reference count is greater than 0 or if it is the result of the expression just evaluated. Note that in the latter situation, $v$ is still accessible to the program even though there are no references to it.

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. Similarly, 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.

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

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

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

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

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

$$
\begin{aligned}
\mathcal{L}[\![c?(e)]\!] = {} & \textbf{let}\ x = \mathcal{L}[\![e]\!]\ \textbf{in} \\
& \textbf{let}\ r = c?(\mathit{data}(x))\ \textbf{in} \\
& \quad (\textbf{if}\ \mathit{not}(\mathit{isPrim}(x))\ \mathit{and}\ \mathit{rc}(x) = 0\ \textbf{then}\ \mathit{gc}(x));\ r
\end{aligned}
$$

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

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

$$
\begin{aligned}
\mathcal{L}[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] = {} & \textbf{let}\ v = \mathcal{L}[\![e_1]\!]\ \textbf{in} \\
& \mathit{incrc}(v); \\
& \textbf{let}\ r = \mathcal{L}[\![e_2]\!]\ \textbf{in} \\
& \quad \mathit{gcExcept}(v, r);\ r
\end{aligned}
$$

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

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

$\mathit{gcExcept}(u, v) = \mathit{incrc}(v); \mathit{gc}(u); \mathit{decrc}(v)$

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 $\mathit{data}(v)$.

# 4 Live Heap Space Bound Function

The transformation $\mathcal{L}_b$ in Figures 2-3 produces live heap space-bound functions. 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 $x$. 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 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 that represents both results, as is done in timing analysis [22] and stack space and heap allocation analysis [31], 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 $j$ 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 an excess and is stored in $exs(j)$, as discussed in Section 4.3. *live* does not include $exs(j)$. When $j$ becomes garbage, $exs(j)$ is added to *live* just before garbage collecting the branches of $j$.

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 $j$ is a join-parent of $v$ if $branches(j)$ 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 $j$ with a list $b$ of branches, and with $rc(j)$ initialized to 0, $joinParents(j)$ initialized to *nil*, and $exs(j)$ initialized to the zero vector, denoted $V_0$.

## 4.2 Conditionals, Selectors and Testers

Consider a conditional expression $(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)^\dagger$ 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 † 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 $exs(r)$.

Observe that in the transformation of conditional $(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)$, we evaluate $e_2$ and then $e_3$, making copies of only *live* in between. We do not have to copy the entire heap before or after evaluating either expression 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 just $(h_1 + r_2)$, where $r_2$ is the result of $e_2$. Similarly, $h_3$ is $(h_1 + r_3)$, $h_3$ and $r_3$ having the expected meanings. In other words, the heap after evaluation of the conditional is $(h_1 + (r_2\ \mathit{or}\ r_3))$. The choice between $r_2$ and $r_3$ is conveniently represented using a join-value that points to them both.

Selectors and testers return *uk* if given *uk* arguments. For join-values with two non-primitive branches, 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 $j$ 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. In fact, computing $exs(j)$ would yield $V_0$ as the result, because $j'$ references $j$'s children and as a result, none of the descendants of $j$ are contained-in $j$.

Applying any tester other than *null*? to join-values with one primitive branch and one non-primitive branch, results in the *join* of (a) $false$ (the result of applying any tester other than *null*? to any primitive branch) and (b) the result of applying the tester to the non-primitive branch. Recall that join-values do not save the values of their primitive branches. Applying *null*? to an unknown primitive branch yields *uk*, since the set of primitive values contains both a null value *nil* and non-null elements. So, the result of *null*? on

$$
\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)] = \mathit{live}[i_c]{+}{+};\ \textbf{if } (P\cdot \mathit{live} > P\cdot \mathit{maxlive}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then for } c \in \mathit{Constructors}\ \mathit{maxlive}[i_c] := \mathit{live}[i_c]; \\
\qquad\qquad\qquad \textbf{let } r_1 = \mathcal{L}_b\,[e_1],\ldots,r_n=\mathcal{L}_b\,[e_n] \textbf{ in} \\
\qquad\qquad\qquad\quad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\mathit{new}(c(r_1,\ldots,r_n)) \\
\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 = \mathit{uk}\ \mathit{or}\ \cdots\ \mathit{or}\ v_n = \mathit{uk} \textbf{ then } \mathit{uk} \textbf{ else } p(v_1,\ldots,v_n) \\
\mathcal{L}_b\,[c?(e)] = \textbf{let } x = \mathcal{L}_b\,[e] \textbf{ in} \\
\qquad\qquad \textbf{let } r = c?_u(x) \textbf{ in} \\
\qquad\qquad\quad (\textbf{if } \mathit{not}(\mathit{isPrim}(x))\ \mathit{and}\ \mathit{rc}(x)=0 \textbf{ then } \mathit{gc}(x));\ r \\
c?_u(v) = \textbf{if } v = \mathit{uk} \textbf{ then } \mathit{uk} \\
\qquad\quad \textbf{else if } \mathit{isJoin}(v) \textbf{ then if } \mathit{length}(\mathit{branches}(v)) = 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathit{join}(\mathit{false}, c?_u(\mathit{first}(\mathit{branches}(v)))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathit{join}(c?_u(\mathit{first}(\mathit{branches}(v))), c?_u(\mathit{second}(\mathit{branches}(v)))) \\
\qquad\qquad \textbf{else } c?(\mathit{data}(v)) \\
\mathcal{L}_b\,[\![c^{-i}(e)]\!] = \textbf{let } x = \mathcal{L}_b\,[e] \textbf{ in} \\
\qquad\qquad\quad \textbf{let } r = c_u^{-i}(x) \textbf{ in} \\
\qquad\qquad\qquad (\textbf{if } \mathit{not}(\mathit{isPrim}(x))\ \mathit{and}\ \mathit{rc}(x)=0 \textbf{ then } \mathit{gcExcept}(x,r);\mathit{recomputeExs}(r));\ r \\
c_u^{-i}(v) = \textbf{if } v = \mathit{uk} \textbf{ then } \mathit{uk} \\
\qquad\quad \textbf{else if } \mathit{isJoin}(v) \\
\qquad\qquad \textbf{then if } \mathit{length}(\mathit{branches}(v)) = 1 \textbf{ then } c^{-i}(\mathit{false}) \\
\qquad\qquad\qquad \textbf{else } \mathit{join}(c_u^{-i}(\mathit{first}(\mathit{branches}(v))), c_u^{-i}(\mathit{second}(\mathit{branches}(v)))) \\
\qquad\qquad \textbf{else } c^{-i}(\mathit{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 = \mathit{uk} \textbf{ then let } l_1 = \mathit{copy}(\mathit{live}) \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } r_2 = \mathcal{L}_b\,[e_2] \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let } l_2 = \mathit{copy}(\mathit{live}) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad \mathit{live} := l_1; \textbf{let } r_3 = \mathcal{L}_b\,[e_3] \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } l_3 = \mathit{copy}(\mathit{live}) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{live} := \max(l_2,l_3); \textbf{let } r = \mathit{join}(r_2,r_3) \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{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\quad \mathit{incrc}(v); \textbf{let } r = \mathcal{L}_b\,[e_2] \textbf{ in } (\mathit{gcExcept}(v,r);\mathit{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\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n); \\
\qquad\qquad\qquad\qquad \textbf{let } r = f_{Lb}(r_1,\ldots,r_n) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad \mathit{gcExcept}(r_1,r);\ldots;\mathit{gcExcept}(r_n,r);\mathit{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.

join-values with a primitive branch is *uk*, irrespective of whether the result of *null*? on the non-primitive branch is *true* or *false*.

When a selector $c^{-i}$ is applied to 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. Some test in a conditional in the original program must prevent $c^{-i}$ from being applied to the primitive branch. Thus, with this assumption, we could use the following alternative definition of $c_u^{-i}$.

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

Applications of selectors to join-values with primitive branches is in fact seen in only one of our examples, namely quicksort. For all of our examples, it is easy to see that for the expected input, selectors are never applied to primitive values. Hence, it is safe to use the above definition of $c_u^{-i}$ for all these examples. In general, a simple static analysis could be used to automatically ascertain such program properties.

## 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.

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

Let $r$ be the result of the conditional. Let $c_i$ denote the data construction with $\mathit{cons}^{-1}(c_i) = i$. 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 $j$, 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 $\mathit{rc}(v)$ 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 the function call or the **let** expression and in case (2), is in the result of the selector. *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.

Observe that $v$ may be a part of a join-value $j'$ that is not in the result of the function call or **let** expression or selector application. It can be shown that loss of references to $v$ at the completion of the function call, **let** expression or selector application, has no effect on $\mathit{exs}(j')$ and so we do not call $\mathit{recomputeExs}(j')$. This applies to tester applications also. Further, results of testers are boolean and hence may not contain any join-values. So, *recomputeExs* is not called after tester applications. Note that *recomputeExs* is used only to obtain tighter bounds, so calling or not calling it at any point in the analysis is safe, i.e., still yields an upper bound on the space usage.

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(v1, v2) = if eq?(v1, v2) then v1
               else if isPrim(v1)
                    then if isPrim(v2) then uk
                         else let result = newjoin([v2]) in
                                 incrc(v2); addJoinParent(v2, result); result
                    else if isPrim(v2)
                         then let result = newjoin([v1]) in
                                 incrc(v1); addJoinParent(v1, result); result
                         else let result = newjoin([v1, v2]) in
                                incrc(v1); addJoinParent(v1, result);
                                incrc(v2); addJoinParent(v2, result);
                                result

gc(v) = if not(isPrim(v))
        then decrc(v);
             if rc(v) ≤ 0
             then if isJoin(v)
                  then live = live + exs(v);
                       for u in branches(v)
                         delJoinParent(u, v); gc(u)
                  else live[conType(v)]−−;
                       for i = 1..arity(v) gc(c^{-i}(data(v)))

recomputeExs(v) = if not(isPrim(v)) then
                    if isJoin(v)
                    then if length(branches(v)) = 2 and containedIn_0(branches(v))
                         then let newexs = computeExs(v) in
                                if newexs > exs(v)
                                then live = live + exs(v) − newexs; setexs(v, newexs)
                         else for u in branches(v) recomputeExs(u)
                    else for i = 1..arity(v) recomputeExs(c^{-i}(data(v)))

containedIn_0(ls) = if null(ls) then true
                    else if rc(cons^{-1}(ls)) = length(joinParents(cons^{-1}(ls))) = 1
                         then containedIn_0(cons^{-2}(ls))
                         else false
```

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.

join-value $j$, let $C_j$ denote the set of all con-values and join-values contained-in $j$, and let $G_j$ denote the graph comprising vertices and edges reachable from $j$. A *join-path* of $j$ is a connected subgraph of $G_j$ containing $j$ and constructed from $G_j$ by selecting at every join-value $j'$ reachable from $j$, exactly one branch of $j'$ and then, after all selections have been made, eliminating unreachable vertices and edges. Figure 4 contains examples of join-paths. Join-paths of $j$ correspond to data structures represented by $j$. $conCountVec(u)$ 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. $maxJoinPath(j)$ is the maximum of the sizes of all join-paths of $j$, where $size(P)$ for a join-path $P$ of $j$ is defined as

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

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

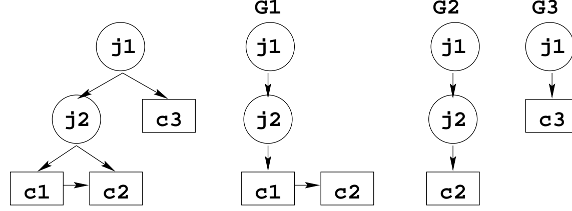



Figure 4: Examples of join-paths. G1, G2 and G3 are join-paths of join-value j1. Circles denote join-values and rectangles denote con-values.

is $V_0$).

$$
\begin{aligned}
\mathit{exs}(j) &= \mathit{total}(j) - \mathit{sub}(j) - \mathit{maxJoinPath}(j) && (2)\\
\mathit{total}(j) &= \sum_{u \text{ is a con-value in } C_j} \mathit{conCountVec}(u) && (3)\\
\mathit{sub}(j) &= \sum_{u \text{ is a join-value in } C_j} \mathit{exs}(u) && (4)
\end{aligned}
$$

The sums and differences of constructor count vectors are computed component-wise. (2) does not result in vectors with negative counts, since *sub*(j) and *maxJoinPath*(j) count data in disjoint subsets of $C_j$. This is justified as follows : if the join-path $P$ which contributes to *maxJoinPath*(j) contains a join-value $j'$, then $P$ contains a largest join-path of $j'$ and $\mathit{exs}(j')$ counts data in the other join-paths of $j'$. Hence, $\mathit{exs}(j')$, for any descendant join-value $j'$ of $j$, 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, $\mathit{exs}(j)$ is computed using (2) if the elements in $\mathit{branches}(j)$ have reference counts equal to 1; $\mathit{exs}(j)$ is conservatively set to $V_0$ otherwise.

A recomputed value $v_2$ of $\mathit{exs}(j)$ can be less than the existing value $v_1$ of $\mathit{exs}(j)$. This happens only if after the computation of $v_1$, selectors are applied to $j$ creating a new join-value $j'$ that references parts of $j$ and a subsequent garbage collection leads to the recomputation of $\mathit{exs}(j)$ yielding $v_2$. The references from $j'$ to parts of $j$ cause these parts to not be contained-in $j$ and so (2) produces a smaller value than the existing $\mathit{exs}(j)$. But selection from $j$ does not alter the fact that only one of the data constructions represented by $j$ is live, so the new smaller value of $\mathit{exs}(j)$ is artificial and is ignored. Figure 5 contains an example of a live heap space-bound function.

# 5 Optimizations

We use two optimizations that reduce the asymptotic complexity of live heap analysis for many programs. 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 any point $p$ during the execution of a space-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 $b_2$. The following discussion also holds with $b_1$ and $b_2$ interchanged. Our optimization is conservative and reduces $j$ 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 $j$, the corresponding descendant $d_2$ of $b_2$ is the same as $d_1$. If these conditions are true and additionally, if eq? is not used on $j$ or on data selected from $j$, then at point $p$ and thereafter, $b_1$ contributes as much or more to *live* as compared to $b_2$. In the following subsections, we formalize the notion of reducibility of join-values and prove that reduction of join-values is safe, i.e., the optimized space-bound analysis provides upper bounds on live heap usage.

```
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_Lb(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_Lb(arg1, arg2) in
                                               gcExcept(arg1, r); gcExcept(arg2, r); recomputeExs(r); r)† 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 †
```

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*.

An edge-ordered rooted directed acyclic graph (DAG) is a tuple $\langle V, E, r\rangle$ where $V$ is a set of vertices, $E$ is a set of edges and $r$ is the root of the DAG, i.e, a node from which all other nodes in the DAG are reached. An edge in $E$ is a tuple $\langle x, y, i\rangle$, where $x$ is the source node, $y$ is the destination node, and $i$ is the index of the edge amongst the out-edges of $x$.

Recall that the stack and live heap may be viewed as a graph. The subgraph $G_x = \langle V_x, E_x, x\rangle$ comprised of nodes and edges reachable from a node $x$ is an edge-ordered rooted DAG. It is acyclic because we are dealing with a first-order functional language without imperative update. The ordering of fields in data constructions imposes an ordering on the out-edges from nodes. For example, if $x$ is an instance of *cons* and $cons^{-2}(x) = y$ and $y$ is not a primitive value, then $G_x$ contains the edge $\langle x, y, 2\rangle$.

Two edge-ordered rooted DAGs $G_1 = \langle V_1, E_1, r_1\rangle$ and $G_2 = \langle V_2, E_2, r_2\rangle$ are isomorphic if there exists a one-to-one, onto mapping $f : V_1 \to V_2$ such that $\langle x, y, i\rangle \in E_1$ iff $\langle f(x), f(y), i\rangle \in E_2$, and $x$ and $f(x)$ are data constructions of the same type. If $f(x) = y$, we say that $x$ corresponds to $y$ and *vice versa*.

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

R2. Corresponding primitive values in $b_1$ and $b_2$ and their descendants are equal, taking $uk = uk$.

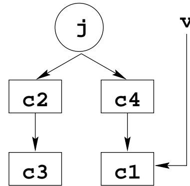
R3. For every node $d_1$ of $G_{b_1}$, if $d_1$ is not contained-in $j$, then $d_1$ and $f(d_1)$ are the same node.

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. For example, $b_1$ and $b_2$ may be two lists of the same length and containing the same primitive values. 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, R2 and R4 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 $b_2$; specifically, using $b_1$ *vs.* $b_2$ does not affect any other heap allocations or live heap usage of the program. 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 following expression.

$$\begin{array}{l}\textbf{let } v = cons(uk, nil) \textbf{ in} \\ \quad \textbf{if } uk \textbf{ then } cons(uk, cons(uk, nil)) \textbf{ else } cons(uk, v)\end{array}$$

The abstract heap at the point just after evaluating the conditional is shown above. Con-values `c1` through




`c4` are numbered according to the order of creation. The result `j` of the conditional satisfies conditions R0 through R4, 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 space-bound function, then it is safe to replace all references to $j$ with references to $b_1$, i.e., in the presence of these replacements the space-bound analysis still returns an upper bound on live heap usage.*

**Proof:** Based on the above arguments, it suffices to show that $b_1$ contributes at least as much to the live heap space as $b_2$.

Let $G_{b_1} = \langle V_{b_1}, E_{b_1}, b_1 \rangle$ and $G_{b_2} = \langle V_{b_2}, E_{b_2}, b_2 \rangle$ be the edge-ordered DAGs rooted at $b_1$ and $b_2$, respectively. Let $C_j$ be the set of nodes contained-in $j$ at $p_0$. The contribution of $b_i$ to *live* is the amount of data contained-in $j$ and referenced by $b_i$. Observe that data not contained-in $j$ is live regardless of whether a replacement of $j$ with $b_1$ or $b_2$ is performed. The contribution of $j$ to *live* is the maximum of the contributions of $b_1$ and $b_2$. At $p_0$,

$$\text{contrib. of } b_1 \text{ to } \mathit{live} = \sum_{x \in (V_{b_1} \bigcap C_j)} \mathit{conCountVec}(x) \tag{5}$$

$$\text{contrib. of } b_2 \text{ to } \mathit{live} = \sum_{x \in (V_{b_2} \bigcap C_j)} \mathit{conCountVec}(x) \tag{6}$$

First, we show that (5) is at least as large as (6). R1 states that $G_{b_1}$ and $G_{b_2}$ are isomorphic. Let $f : V_{b_1} \to V_{b_2}$ be the isomorphism. We show that at $p_0$, for every node $y \in (V_{b_2} \bigcap C_j)$, $f^{-1}(y) \in (V_{b_1} \bigcap C_j)$. By definition, $f^{-1}(y) \in V_{b_1}$. We prove by contradiction that $f^{-1}(y) \in C_j$. Suppose not. Then, according to R3, $f^{-1}(y) = y$. Hence, $y \notin C_j$. This contradicts the assumption that $y \in (V_{b_2} \bigcap C_j)$, so $f^{-1}(y) \in C_j$. Also, $y$ and $f^{-1}(y)$ are data constructions of the same type. Thus, for each element of the sum in (6), there is a corresponding and equal element of the sum in (5), so (5) is at least as large as (6).

Now, consider a point $p_1$ after $p_0$. If $j$ is dead at $p_1$, then the replacement has no more effect on the execution of the space-bound function; the indirect effect through parts of $j$ that were selected out and that are still live, is considered below. Suppose $j$ is live at $p_1$. Observe that $G_{b_1}$ and $G_{b_2}$ do not change between $p_0$ and $p_1$, due to the absence of imperative update, and that $f$ is still an isomorphism between $G_{b_1}$ and $G_{b_2}$. We use $C^p_x$ to refer to the set of nodes contained-in node $x$ at point $p$; $p$ is omitted when it is clear from context. We show that, for every node $y \in (V_{b_2} \bigcap C^{p_1}_j)$, $f^{-1}(y) \in C^{p_1}_j$. Case 1: $f^{-1}(y) \notin C^{p_0}_j$. Then, according to R3, $f^{-1}(y) = y$. By hypothesis, $y \in C^{p_1}_j$, so $f^{-1}(y) \in C^{p_1}_j$. Case 2: $f^{-1}(y) \in C^{p_0}_j$. We prove by contradiction that $f^{-1}(y) \in C^{p_1}_j$. $f^{-1}(y) \in C^{p_0}_j$ implies that after $p_0$, the only way to create references to $f^{-1}(y)$ from outside $G_j$ (the DAG rooted at $j$) is through selector applications to $j$. This is thus the only way for $f^{-1}(y)$ to become not contained-in $j$ after $p_0$. Since selector applications to $j$ return join-values whose branches are corresponding nodes in $V_{b_1}$ and $V_{b_2}$, such join-values that reference $f^{-1}(y)$ must also reference $y$. So, if $f^{-1}(y) \notin C_j$ (as hypothesized for the proof by contradiction), then $y \notin C_j$. This contradicts the assumption $y \in (V_{b_2} \bigcap C^{p_1}_j)$. Hence, $f^{-1}(y) \in C^{p_1}_j$.

We now show that after $p_0$, the result of a selector application to $b_1$ also contributes at least as much to *live* as compared to the result of the same selector application to $b_2$. Let the result of $c_1^{-i_1}(...(c_n^{-i_n}(j)))$ be $j' = \mathit{join}(b'_1, b'_2)$ where $b'_1 \in V_{b_1}$ and $b'_2 \in V_{b_2}$, and let $p_1$ be a point equal to or after $p_0$ at which $j'$ is live. At $p_1$,

$$\text{contrib. of } b'_1 \text{ to } \mathit{live} = \sum_{x \in (V_{b'_1} \bigcap C_{j'})} \mathit{conCountVec}(x) \tag{7}$$

$$\text{contrib. of } b'_2 \text{ to } \mathit{live} = \sum_{x \in (V_{b'_2} \bigcap C_{j'})} \mathit{conCountVec}(x) \tag{8}$$

Observe that the restriction of $f$ to any subgraph $G$ of $G_{b_1}$ is an isomorphism between $G$ and the corresponding subgraph of $G_{b_2}$. So, the restriction of $f$ to $G_{b'_1}$ is an isomorphism between $G_{b'_1}$ and $G_{b'_2}$. We refer to this restriction of $f$ as just $f$. We prove that for every $y \in (V_{b'_2} \bigcap C^{p_1}_{j'})$, $f^{-1}(y)$ is in $(V_{b'_1} \bigcap C^{p_1}_{j'})$. We consider two cases. Case 1: $f^{-1}(y) \notin C^{p_0}_j$. Then, $f^{-1}(y) = y$, because of R3. $y$ is in $C^{p_1}_{j'}$, so $f^{-1}(y)$ is in $C^{p_1}_{j'}$. Case 2: $f^{-1}(y) \in C^{p_0}_j$. We prove by contradiction that $f^{-1}(y) \in C^{p_1}_{j'}$. Suppose this is not so. Then, there exist references to $f^{-1}(y)$ from outside $G'_j$, the DAG rooted at $j'$. At $p_0$, $f^{-1}(y) \in C_j$, so any references to $f^{-1}(y)$ from outside $G'_j$ are created after $p_0$, implying $p_1$ is after (not equal to) $p_0$. Furthermore, these references must be due to selector applications to $j$. Thus, at $p_1$, references to $f^{-1}(y)$ from outside $G_{j'}$ are either from $j$ or from the results of selector applications to $j$. Since both $j$ and results of selector applications to $j$ are

join-values that reference corresponding nodes in $V_{b_1}$ and $V_{b_2}$, there exist references from outside $G_{j'}$ to $y$ also, so $y \notin C_{j'}^{p_1}$. This contradicts the assumption $y \in (V_{b'_2} \bigcap C_{j'}^{p_1})$. Hence, $f^{-1}(y)$ is in $C_{j'}^{p_1}$. Thus, for each element of the sum in (8), there is a corresponding and equal element of the sum in (7), so (7) is at least as large as (8).

# 6 Handling Tail Call Optimization

In some languages, the environment of the caller is discarded only after the completion of all function calls in its body, even if the body contains a function call in tail position. An expression is in *tail position* if it is the last thing that the function does before returning. A *tail call* is a function call in tail position. Tail call optimization [1] allows the caller's environment to be discarded right before executing a tail call. We have extended our analysis to reflect the effect of tail call optimization. In the presence of this optimization, the analysis described in Sections 3 and 4 yields safe but perhaps loose bounds on space usage.

Our extended analysis recognizes function calls in tail position and at the sites of these calls, garbage collects all variables in the current scope. In the original analysis, rc's of arguments are incremented just before function calls and arguments are garbage collected on return from the calls. In the extended analysis, both operations are performed within the function body: increments are done at the start of the function and garbage collection is done just before the result is returned. The latter is performed either just before a tail call or on completion of the last non-call expression in the function body. This structure provides for a simple transformation as well as simpler and more efficient space and bound functions. Figure 6 contains the new transformation that yields space functions. The right hand side uses quasi-quotes '...'; all expressions within quotes, except those of the form "$\mathcal{L}_x [\![e]\!] \ldots$", are to be taken literally. We make the exception to minimize clutter.

Figures 7, 8 and 9 contain the transformation that yields bound functions. The transformation of conditionals is more involved than $\mathcal{L}_b$. 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 live, so the effects of the earlier garbage collection have to be reversed before evaluating $e_3$. This is done by using *recIncrc* to increment rc's of $u_1, \ldots, u_m$ and all their descendants. Further, during the evaluation of $e_3$, $u_1, \ldots, u_m$ are garbage collected just before the tail call in $e_3$. Care needs to be taken to ensure that references from the result $r_2$ of $e_2$ to $u_1, \ldots, u_m$ do not cause the latter to be counted as live after this garbage collection. So, before evaluating $e_3$, the rc's of $r_2$ and all its descendants are decremented. Similar issues arise when only one of the two branches contain tail calls. Consider the case when neither of the branches contain tail calls. In each branch, $u_1, \ldots, u_m$ are garbage collected after the evaluation of the last non-call expression in the branch. Instead of garbage collecting $u_1, \ldots, u_m$ at the end of $e_2$, and again incrementing their rc's and those of their descendants before evaluating $e_3$, we simply evaluate $e_2$ and $e_3$ as if they were not in tail position and garbage collect $u_1, \ldots, u_m$ just before returning the result of the conditional.

*tlrec*? in Figure 9 is used to determine whether or not a given expression contains tail calls. More precisely, given an expression $e$ in tail position, *tlrec*?($e$) determines if all variables in the scope of $e$ are garbage collected in $\mathcal{L}_b [\![e]\!]$ itself or if they have to be garbage collected after $\mathcal{L}_b [\![e]\!]$. This more precise definion is particularly relevant to conditionals. Consider a conditional $e =$ (**if** $e_1$ **then** $e_2$ **else** $e_3$) in which one branch, say $e_2$, contains a tail call and the other does not. In $e_2$, $u_1, \ldots, u_m$ are garbage collected before the tail call. $l_2$ in $\mathcal{L}_{xb} [\![e]\!]$ is the value of live after the evaluation of $e_2$. We need to garbage collect $u_1, \ldots, u_m$ at the end of $e_3$ in order to ensure that the value of live after the evaluation of $e_3$ ($l_3$ in $\mathcal{L}_{xb} [\![e]\!]$) corresponds to the same "execution point" as $l_2$. If neither $e_2$ nor $e_3$ contained tail calls it would be sufficient to garbage collect $u_1, \ldots, u_m$ after the entire conditional is evaluated. Therefore, *tlrec*?($e$) is true exactly when either *tlrec*?($e_2$) or *tlrec*?($e_3$) is true.

$$
\begin{array}{l}
f_{L_x}(v_1,\ldots,v_n) = incrc(v_1);\ldots;incrc(v_n);\\
\qquad\qquad\qquad\quad \mathcal{L}_x\,[\![e]\!]\,true
\end{array}
$$

where $e$ is the body of function $f$, i.e., $f(v_1,\ldots,v_n)=e$

$$
\mathcal{L}_x\,[\![v]\!]\,tailpos? = \textbf{if}\ \ tailpos?\ \textbf{then}\ \ \text{`}gcExcept(u_1,v);\ldots;gcExcept(u_m,v);\ v\text{'}\ \textbf{else}\ \ \text{`}v\text{'}
$$

$$
\mathcal{L}_x\,[\![l]\!]\,tailpos? = \textbf{if}\ \ tailpos?\ \textbf{then}\ \ \text{`}gc(u_1);\ldots;gc(u_m);\ l\text{'}\ \textbf{else}\ \ \text{`}l\text{'}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![c(e1,\ldots,en)]\!]\,tailpos? = \text{`}live[i_c]{+}{+};\ \textbf{if}\ (P\cdot live > P\cdot maxlive)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then for}\ c\in Constructors\ maxlive[i_c] := live[i_c];\\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ r_1 = \text{'}\mathcal{L}_x\,[\![e_1]\!]\,false\text{`},\text{'}\ldots,\text{`}r_n = \text{'}\mathcal{L}_x\,[\![e_n]\!]\,false\ \text{`}\textbf{in}\\
\qquad\qquad\qquad\qquad\qquad incrc(r_1);\ldots;incrc(r_n);\text{'}\\
\qquad\qquad\qquad\qquad\qquad \textbf{if}\ tailpos?\\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \ \text{`}\textbf{let}\ r = new(c(r_1,\ldots,r_n))\ \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{`}gcExcept(u_1,r);\ldots;gcExcept(u_m,r);\ r\text{'}\\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \text{`}new(c(r_1,\ldots,r_n))\text{'}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![p(e_1,\ldots,e_n)]\!]\,tailpos? = \textbf{if}\ \ tailpos?\ \textbf{then}\ \text{`}\textbf{let}\ r = p(\mathcal{L}_x\,[\![e_1]\!]\,false,\ldots,\mathcal{L}_x\,[\![e_n]\!]\,false)\ \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad gc(u_1);\ldots;gc(u_m);\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r\text{'}\\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \text{`}p(\mathcal{L}_x\,[\![e_1]\!]\,false,\ldots,\mathcal{L}_x\,[\![e_n]\!]\,false)\text{'}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![c?(e)]\!]\,tailpos? = \text{`}\textbf{let}\ x = \text{'}\ \mathcal{L}_x\,[\![e]\!]\,false\ \text{`}\textbf{in}\\
\qquad\qquad\qquad\qquad \textbf{let}\ r = (\textbf{if}\ isPrim(x)\ \textbf{then}\ c?(x)\ \textbf{else}\ c?(data(x)))\ \textbf{in}\\
\qquad\qquad\qquad\qquad\ (\textbf{if}\ not(isPrim(x))\ and\ rc(x)=0\ \textbf{then}\ gc(x));\text{'}\\
\qquad\qquad\qquad\qquad\ \textbf{if}\ tailpos?\ \textbf{then}\ \text{`}gc(u_1);\ldots;gc(u_m);\ r\text{'}\\
\qquad\qquad\qquad\qquad\ \textbf{else}\ \text{`}r\text{'}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![c^{-i}(e)]\!]\,tailpos? = \text{`}\textbf{let}\ x = \text{'}\ \mathcal{L}_x\,[\![e]\!]\,false\ \text{`}\textbf{in}\\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ r = c^{-i}(data(x))\ \textbf{in}\\
\qquad\qquad\qquad\qquad\quad\ (\textbf{if}\ not(isPrim(x))\ and\ rc(x)=0\ \textbf{then}\ gcExcept(x,r));\text{'}\\
\qquad\qquad\qquad\qquad\quad\ \textbf{if}\ tailpos?\ \textbf{then}\ \text{`}gcExcept(u_1,r);\ldots;gcExcept(u_m,r);\ r\text{'}\\
\qquad\qquad\qquad\qquad\quad\ \textbf{else}\ \text{`}r\text{'}
\end{array}
$$

$$
\mathcal{L}_x\,[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]\,tailpos? = \text{`}\textbf{if}\ \ \mathcal{L}_x\,[\![e_1]\!]\,false\ \textbf{then}\ \mathcal{L}_x\,[\![e_2]\!]\,tailpos?\ \textbf{else}\ \mathcal{L}_x\,[\![e_3]\!]\,tailpos?\text{'}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!]\,tailpos? = \text{`}\textbf{let}\ v = \mathcal{L}_x\,[\![e_1]\!]\,false\ \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad incrc(v);\text{'}\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ tailpos?\ \textbf{then}\ \text{`}\mathcal{L}_x\,[\![e_2]\!]\,true\text{'}\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \text{`}\textbf{let}\ r = \mathcal{L}_x\,[\![e_2]\!]\,false\ \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad gcExcept(v,r);\ r\text{'}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}_x\,[\![f(e_1,\ldots,e_n)]\!]\,tailpos? = \textbf{if}\ \ tailpos?\\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \text{`}\textbf{let}\ r_1 = \mathcal{L}_x\,[\![e_1]\!]\,false,\ldots,r_n = \mathcal{L}_x\,[\![e_n]\!]\,false\ \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad incrc(r_1);\ldots;incrc(r_n);\\
\qquad\qquad\qquad\qquad\qquad\qquad gc(u_1);\ldots;gc(u_m);\\
\qquad\qquad\qquad\qquad\qquad\qquad decrc(r_1);\ldots;decrc(r_n);\\
\qquad\qquad\qquad\qquad\qquad\qquad f_{L_x}(r_1,\ldots,r_n)\text{'}\\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \text{`}f_{L_x}(\mathcal{L}_x\,[\![e_1]\!]\,false,\ldots,\mathcal{L}_x\,[\![e_n]\!]\,false)\text{'}
\end{array}
$$

Figure 6: Transformation that produces live heap space functions $f_{L_x}$ handling tail call optimization. $u_1$, ..., $u_m$ are variables in the current scope.

# 7 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

$f_{L_{xb}}(v_1,\ldots,v_n) = \text{`}incrc(v_1);\ldots;incrc(v_n);$
$\quad\quad \mathcal{L}_{xb}[\![e]\!]\,true\text{'}$

where $e$ is the body of function $f$, i.e., $f(v_1,\ldots,v_n) = e$

$\mathcal{L}_{xb}[\![v]\!]\,tailpos? = \mathbf{if}\ \ tailpos?\ \mathbf{then}\ \ \text{`}gcExcept(u_1,v);\ldots;gcExcept(u_m,v);\ v\text{'}\ \mathbf{else}\ \ \text{`}v\text{'}$

$\mathcal{L}_{xb}[\![l]\!]\,tailpos? = \mathbf{if}\ \ tailpos?\ \mathbf{then}\ \ \text{`}gc(u_1);\ldots;gc(u_m);\ l\text{'}\ \mathbf{else}\ \ \text{`}l\text{'}$

$\mathcal{L}_{xb}[\![c(e1,\ldots,en)]\!]\,tailpos? = \text{`}live[i_c]{+}{+};\ \mathbf{if}\ (P\cdot live > P\cdot maxlive)$
$\quad\quad \mathbf{then\ for}\ c \in Constructors\ maxlive[i_c] := live[i_c];$
$\quad \mathbf{let}\ r_1 = \text{'}\mathcal{L}_{xb}[\![e_1]\!]\,false\text{`},\text{'}\ldots,\text{`}r_n = \text{'}\mathcal{L}_{xb}[\![e_n]\!]\,false\ \text{`}\mathbf{in}$
$\quad\quad incrc(r_1);\ldots;incrc(r_n);\text{'}$
$\quad\quad \mathbf{if}\ tailpos?$
$\quad\quad \mathbf{then}\ \ \text{`}\mathbf{let}\ r = new(c(r_1,\ldots,r_n))\ \mathbf{in}$
$\quad\quad\quad \text{`}gcExcept(u_1,r);\ldots;gcExcept(u_m,r);\ \ recomputeExs(r);\ \ r\text{'}$
$\quad\quad \mathbf{else}\ \ \text{`}new(c(r_1,\ldots,r_n))\text{'}$

$\mathcal{L}_{xb}[\![p(e_1,\ldots,e_n)]\!]\,tailpos? = \mathbf{if}\ \ tailpos?\ \mathbf{then}\ \text{`}\mathbf{let}\ r = p_u(\mathcal{L}_{xb}[\![e_1]\!]\,false,\ldots,\mathcal{L}_{xb}[\![e_n]\!]\,false)\ \mathbf{in}$
$\quad\quad\quad gc(u_1);\ldots;gc(u_m);$
$\quad\quad\quad r\text{'}$
$\quad \mathbf{else}\ \text{`}p_u(\mathcal{L}_{xb}[\![e_1]\!]\,false,\ldots,\mathcal{L}_{xb}[\![e_n]\!]\,false)\text{'}$

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

$\mathcal{L}_{xb}[\![c?(e)]\!]\,tailpos? = \text{`}\mathbf{let}\ x = \text{'}\ \mathcal{L}_{xb}[\![e]\!]\,false\ \text{`}\mathbf{in}$
$\quad \mathbf{let}\ r = c?_u(x)\ \mathbf{in}$
$\quad\quad (\mathbf{if}\ \ not(isPrim(x))\ \ and\ \ rc(x) = 0\ \mathbf{then}\ gc(x));\text{'}$
$\quad\quad \mathbf{if}\ tailpos?\ \mathbf{then}\ \text{`}gc(u_1);\ldots;gc(u_m);\ r\text{'}$
$\quad\quad \mathbf{else}\ \text{`}r\text{'}$

$c?_u(v) = \mathbf{if}\ \ v = uk\ \mathbf{then}\ uk$
$\quad \mathbf{else\ if}\ \ isJoin(v)\ \mathbf{then\ if}\ \ length(branches(v)) = 1$
$\quad\quad\quad \mathbf{then}\ join(false, c?_u(first(branches(v))))$
$\quad\quad\quad \mathbf{else}\ join(c?_u(first(branches(v))), c?_u(second(branches(v))))$
$\quad\quad \mathbf{else}\ c?(data(v))$

$\mathcal{L}_{xb}\left[\!\left[c^{-i}(e)\right]\!\right]\,tailpos? = \text{`}\mathbf{let}\ x = \text{'}\ \mathcal{L}_{xb}[\![e]\!]\,false\ \text{`}\mathbf{in}$
$\quad \mathbf{let}\ r = c_u^{-i}(x)\ \mathbf{in}$
$\quad\quad (\mathbf{if}\ \ not(isPrim(x))\ \ and\ \ rc(x) = 0\ \mathbf{then}\ gcExcept(x,r));\text{'}$
$\quad\quad \mathbf{if}\ tailpos?\ \mathbf{then}\ \text{`}gcExcept(u_1,r);\ldots;gcExcept(u_m,r);recomputeExs(r)\ r\text{'}$
$\quad\quad \mathbf{else}\ \text{`}recomputeExs(r)\ \ r\text{'}$

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

$\mathcal{L}_{xb}[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!]\,tailpos? = \text{`}\mathbf{let}\ v = \mathcal{L}_{xb}[\![e_1]\!]\,false\ \mathbf{in}$
$\quad incrc(v);\text{'}$
$\quad \mathbf{if}\ tailpos?\ \mathbf{then}\ \text{`}\mathcal{L}_{xb}[\![e_2]\!]\,true\text{'}$
$\quad \mathbf{else}\ \text{`}\mathbf{let}\ r = \mathcal{L}_{xb}[\![e_2]\!]\,false\ \mathbf{in}$
$\quad\quad gcExcept(v,r);\ \ recomputeExs(r);\ \ r\text{'}$

$\mathcal{L}_{xb}[\![f(e_1,\ldots,e_n)]\!]\,tailpos? = \mathbf{if}\ \ tailpos?$
$\quad \mathbf{then}\ \text{`}\mathbf{let}\ r_1 = \mathcal{L}_{xb}[\![e_1]\!]\,false,\ldots,r_n = \mathcal{L}_{xb}[\![e_n]\!]\,false\ \mathbf{in}$
$\quad\quad incrc(r_1);\ldots;incrc(r_n);$
$\quad\quad gc(u_1);\ldots;gc(u_m);$
$\quad\quad decrc(r_1);\ldots;decrc(r_n);$
$\quad\quad recomputeExs(r_1);\ldots;recomputeExs(r_n);$
$\quad\quad f_{L_{xb}}(r_1,\ldots,r_n)\text{'}$
$\quad \mathbf{else}\ \text{`}f_{L_{xb}}(\mathcal{L}_{xb}[\![e_1]\!]\,false,\ldots,\mathcal{L}_{xb}[\![e_n]\!]\,false)\text{'}$



Figure 7: Transformation that produces live heap space-bound functions $f_{L_{xb}}$ handling tail call optimization.

$$
\begin{array}{l}
\mathcal{L}_{xb}\,[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\,\mathit{tailpos?} = \\
\quad \mathbf{if}\ \mathit{tailpos?} \\
\quad \mathbf{then\ if}\ \mathit{tlrec?}(e_2) \\
\qquad\quad \mathbf{then\ if}\ \mathit{tlrec?}(e_3) \\
\qquad\qquad\quad \mathbf{then}\ \text{`}\mathbf{let}\ b = \mathcal{L}_{xb}\,[\![e_1]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ b = \mathit{uk}\ \mathbf{then\ let}\ l_1 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ r_2 = \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{recincrc}(u_1); \ldots; \mathit{recincrc}(u_m); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{recdecrc}(r_2); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{live} := l_1; \mathbf{let}\ r_3 = \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ l_3 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{live} := \max(l_2, l_3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{recincrc}(r_2); \mathbf{let}\ r = \mathit{join}(r_2, r_3)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{setexs}(r, \mathit{computeExs}(r)); r \\
\qquad\qquad\qquad\quad \mathbf{else\ if}\ b\ \mathbf{then}\ \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{else}\ \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\text{'} \\
\qquad\qquad\quad \mathbf{else}\ \text{`}\mathbf{let}\ b = \mathcal{L}_{xb}\,[\![e_1]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ b = \mathit{uk}\ \mathbf{then\ let}\ l_1 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ r_3 = \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ l_3 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{recdecrc}(r_3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{live} := l_1; \mathbf{let}\ r_2 = \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{live} := \max(l_2, l_3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{recincrc}(r_3); \mathbf{let}\ r = \mathit{join}(r_2, r_3)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{setexs}(r, \mathit{computeExs}(r)); r \\
\qquad\qquad\qquad\quad \mathbf{else\ if}\ b\ \mathbf{then}\ \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{else}\ \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\text{'} \\
\qquad\quad \mathbf{else\ if}\ \mathit{tlrec?}(e_3) \\
\qquad\qquad \mathbf{then}\ \text{`}\mathbf{let}\ b = \mathcal{L}_{xb}\,[\![e_1]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ b = \mathit{uk}\ \mathbf{then\ let}\ l_1 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ r_2 = \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{recdecrc}(r_2); \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{live} := l_1; \mathbf{let}\ r_3 = \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{live} := \max(l_2, l_3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{recincrc}(r_2); \mathbf{let}\ r = \mathit{join}(r_2, r_3)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{setexs}(r, \mathit{computeExs}(r)); r \\
\qquad\qquad\qquad\quad \mathbf{else\ if}\ b\ \mathbf{then}\ \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{else}\ \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\text{'} \\
\qquad\qquad \mathbf{else}\ \text{`}\mathbf{let}\ b = \mathcal{L}_{xb}\,[\![e_1]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{if}\ b = \mathit{uk}\ \mathbf{then\ let}\ l_1 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ r_2 = \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{live} := l_1; \mathbf{let}\ r_3 = \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ l_3 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{live} := \max(l_2, l_3); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ r = \mathit{join}(r_2, r_3)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{setexs}(r, \min(l_2 - l_1, l_3 - l_1)); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{gcExcept}(u_1, r); \ldots; \mathit{gcExcept}(u_m, r); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r \\
\qquad\qquad\qquad \mathbf{else\ if}\ b\ \mathbf{then}\ \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{true}\ \mathbf{else}\ \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{true}\text{'} \\
\quad \mathbf{else}\ \text{`}\mathbf{let}\ b = \mathcal{L}_{xb}\,[\![e_1]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\quad \mathbf{if}\ b = \mathit{uk}\ \mathbf{then\ let}\ l_1 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ r_2 = \mathcal{L}_{xb}\,[\![e_2]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ l_2 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathit{live} := l_1; \mathbf{let}\ r_3 = \mathcal{L}_{xb}\,[\![e_3]\!]\,\mathit{false}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ l_3 = \mathit{copy}(\mathit{live})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{live} := \max(l_2, l_3); \mathbf{let}\ r = \mathit{join}(r_2, r_3)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{setexs}(r, \min(l_2 - l_1, l_3 - l_1)); r
\end{array}
$$

```
recincrc(v) = incrc(v);
              if  (rc(v) ≤ 1)
              then if isJoin(v)
                   then for u in branches(v) recincrc(u)
                   else for i = 1..arity(v) recincrc(c^{-i}(data(v)))
recdecrc(v) = decrc(v);
              if  (rc(v) ≤ 0)
              then if isJoin(v)
                   then for u in branches(v) recdecrc(u)
                   else for i = 1..arity(v) recdecrc(c^{-i}(data(v)))
tlrec?(e) = case e of
              l                        : false
              v                        : false
              p(e_1, ..., e_n)         : false
              c(e_1, ..., e_n)         : false
              c?(e)                    : false
              c^{-i}(e)                : false
              if e_1 then e_2 else e_3 : tlrec?(e_2) ∨ tlrec?(e_3)
              let v = e_1 in e_2       : tlrec?(e_2)
              f(e_1, ..., e_n)         : true
```

Figure 9: Auxiliary functions *recincrc*, *recdecrc* and *tlrec?*.

results (i.e., tight bounds) for all these examples. The results are consistent with the expected asymptotic space complexities of 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 issue mentioned in Section 1 is not a problem for any of these examples.

Figure 10 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 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 difference between the results of the space and bound functions of quicksort is explained as follows. Quicksort selects a pivot element $p$ from the given list $ls$ and splits the list into two lists : one containing all elements of $ls$ that are lesser than or equal to $p$ and the other containing elements that are greater than $p$. Suppose $ls$ has size $n$. Since all elements in $ls$ are *uk*, the bound function incorrectly concludes that each of the two lists has worst-case size $n-1$. In reality, the sum of the sizes of the two lists is $n-1$. Further, since quicksort is called recursively on the two lists, inaccuracies at every recursion level quickly add up resulting in the exponential growth of the bound function results.

In time analysis [22] and stack space analysis [31], the bound functions of quicksort run into the non-

termination problem. This is because results of conditionals are summarized into partially known structures and the recursion in quicksort depends on an unknown aspect of such a partially known structure. In live heap analysis, we retain more information about the results of branches of conditionals by using join-values rather than partially known structures. The mentioned recursion in quicksort now has sufficient information to terminate.

| reversal w/append | | insertion sort | | selection sort | | merge sort | | quick sort | | | binomial coefficient | | longest common subseq. | | string edit | | binary tree insert | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | result | $n$ | result | $n$ | result | $n$ | result | $n$ | space | bound | $n$ | result | $n$ | result | $n$ | result | $h$ | $n$ | result |
| 50 | 149 | 50 | 149 | 50 | 1325 | 2 | 5 | 1 | 1 | 1 | 50 | 194 | 50 | 202 | 50 | 500 | 1 | 3 | 18 |
| 100 | 299 | 100 | 299 | 100 | 5150 | 5 | 16 | 2 | 6 | 6 | 100 | 394 | 100 | 402 | 100 | 1000 | 2 | 7 | 33 |
| 250 | 749 | 250 | 749 | 250 | 31625 | 10 | 31 | 5 | 24 | 51 | 250 | 994 | 250 | 1002 | 250 | 2500 | 4 | 31 | 111 |
| 500 | 1499 | 500 | 1499 | 500 | 125750 | 12 | 36 | 6 | 32 | 100 | 500 | 1994 | 500 | 2002 | 500 | 5000 | 7 | 255 | 792 |
| 1000 | 2999 | 1000 | 2999 | 1000 | 501500 | 15 | 45 | 7 | 41 | 197 | 1000 | 3994 | 1000 | 4002 | 1000 | 10000 | 9 | 1023 | 3102 |

Figure 10: Results of live heap space analysis. $n$ 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.

| reversal w/append | | | | insertion sort | | | | selection sort | | | | merge sort | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | $10^1$ | 2.0 m | 0.9 s | 0.7 s |
| $10^2$ | 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 | $10^2$ | 40.0 m | | |
| $10^3$ | 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 | $10^3$ | 0.7 s | | |

| binomial coefficient | | | | longest common subseq. | | | | string edit | | | | binary tree insert | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n | S | B | Bopt | n | S | B | Bopt | n | S | B | Bopt | h | n | S | B | Bopt |
| $10^1$ | 5.0 m | 1.0 m | 5.0 m | $10^1$ | 10.0 m | 0.8 s | 44.0 m | $10^1$ | 15.0 m | 25.0 m | 20.0 m | 2 | 7 | 0 | 3.0 m | 3.0 m |
| $10^2$ | 0.1 s | 0.4 s | 0.1 s | $10^2$ | 6.6 s | | 24.0 s | $10^2$ | 7.1 s | 13.5 s | 7.4 s | 6 | 127 | 4.0 m | 0.1 s | 0.1 s |
| $10^3$ | 11.6 s | 4.7 M | 11.6 s | $10^3$ | 2.0 H | | 7.1 H | $10^3$ | 2.2 H | 3.7 H | 2.3 H | 9 | 1023 | 34.0 m | 1.2 s | 1.2 s |

Figure 11: 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 10. m is milliseconds, s is seconds, M is minutes and H is hours. Blank fields in B columns indicate analyses that were aborted after 7 days. Blank fields in Bopt columns indicate analyses that were aborted after 2 days.

The results in Figure 10 include the space used by top-level arguments since these arguments are indeed live throughout the execution of the program. Figure 11 contains running times of 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 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.

The first optimization in Section 5 improves the asymptotic complexities of reverse using append and binomial coefficient 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 11.

Figure 12 shows the results of the space analysis that deals with tail call optimization. The corresponding space-bound analysis is not yet implemented. List reversal is the standard linear-time version. The optimized Ackermann function example [23] is a systematically derived program which has much better time complexity

than the classical version. In all of these examples, the main function is tail recursive. The inputs for the first three examples are known lists of length $n$ that lead to the worst-case space usage. The inputs for Ackermann's function are known integers, $n$ and $m$. 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. For each value of $n$ for Ackermann's function in Figure 12, all values of the second argument in the range $[2, 10]$ produced the same space usage. While this does not prove $O(n)$ space usage, it does help confirm that, for a given $n$, the space usage is independent of $m$. Computing Ackermann's function for $n > 3$ is famously expensive.

| tail recursive list reversal | | tail recursive insertion sort | | tail recursive selection sort | | optimized Ackermann | |
|---|---|---|---|---|---|---|---|
| n | result | n | result | n | result | n | result |
| 50 | 51 | 50 | 100 | 50 | 100 | 0 | 2 |
| 100 | 101 | 100 | 200 | 100 | 200 | 1 | 8 |
| 250 | 251 | 250 | 500 | 250 | 500 | 2 | 15 |
| 500 | 501 | 500 | 1000 | 500 | 1000 | 3 | 25 |
| 1000 | 1001 | 1000 | 2000 | 1000 | 2000 | | |

Figure 12: Results of live heap space functions handling tail call optimization. $n$ is the input size except in the case of optimized Ackermann function, in which $n$ is the first argument; the second argument $m$ ranges over the interval $[2, 10]$, for each $n$.

Comparing the results in Figure 12 with the space usage of the corresponding non-tail-recursive programs, shown in Figure 10, we see that the tail recursive versions use less live heap space. For example, tail-recursive selection sort uses only $O(n)$ space, while non-tail-recursive selection sort uses $O(n^2)$ space. We also applied our extended analysis to the examples of Figure 10. The results were either the same or differed by a constant amount of 1 or 2 *cons* cells. The differences occur because the programs do contain some function calls in tail position.

Figure 13 contains closed forms or recurrence relations for the live heap space used by the examples. We obtained closed forms by using Matlab to fit polynomials, wherever applicable, to the data in Figure 10. The recurrence relation for merge sort and the closed form for binary tree insert were obtained by hand from the results in Figure 10.

| | |
|---|---|
| tail recursive reversal | $n+1$ |
| reversal w/append | $0, n=0$<br>$3n-1, n>0$ |
| insertion sort | $0, n=0$<br>$3n-1, n>0$ |
| tail rec. insertion sort | $2n, n>0$ |
| selection sort | $\frac{n(n+3)}{2}$ |
| tail rec. selection sort | $2n$ |
| merge sort | $n, n=0,1$<br>$S(\lceil \frac{n}{2} \rceil) + 2n - \lceil \frac{n}{2} \rceil, n>1$ |

| | |
|---|---|
| binomial coefficient | $1, m=0, n$<br>$2m+2, m=n-1$<br>$4m+2$, otherwise |
| longest common subseq. | $n+m+1, n=0$ or $m=0$<br>$2m+3, n=1$ and $m>0$<br>$n+3m+2$, otherwise |
| string edit | $3n+3m+1, n=0$ or $m=0$<br>$5n+5m$, otherwise |
| binary tree insert | $3(2^{h+1}+h+1)$ |

Figure 13: Closed forms or recurrence relations for live heap space used by example programs. $n$ and $m$ are the sizes of the first and second (if any) arguments. For binomial coefficient, $n$ and $m$ are themselves the first and second arguments. For binary tree insert, $h$ is the height of the complete binary tree.

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

**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. 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, non-termination is not a problem in any of the examples we analyzed.

**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 such 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 [30, 11].

**Correctness** A complete proof of correctness of the analysis is yet to be formulated. Included below is one part of the proof.

**Lemma 1** *For a join-value $j$, it is safe to ignore a smaller recomputed value of $exs(j)$.*

**Proof:** Suppose $exs(j)$ is computed at point $p$ and then recomputed at point $q$ and $exs(j)$ at $q$ is less than $exs(j)$ at $p$. We prove that this may happen only if after $p$, a selector is applied to $j$, thus creating a new join-value $j'$ which references parts of $j$. The referenced parts of $j$ are no longer contained-in $j$ and when $exs(j)$ is recomputed at $q$, it is smaller than the earlier value of $exs(j)$. It is indeed safe to ignore any decrease in $exs(j)$ caused by selector application because selection does not alter the fact that only one of the data constructions represented by $j$ is live. We keep only one such data construction live by recording the excess in the $exs$ field of $j$ and setting $exs(j')$, for any $j'$ selected from $j$, to $V_0$. This ensures that excesses are treated as such exactly once. From the definition of $exs$ (2), $exs(j)$ decreases only if one or more of the following occurs :

1. $total(j)$ decreases : this happens if some con-value $c \in C_j^p$ is not contained-in $j$ at $q$. Since $c$ is contained-in $j$ at $p$, there is no way to create a new reference to $c$ other than by applying a selector to $j$. As we explained earlier, we may safely ignore decreases of $exs(j)$ caused by selection from $j$.

2. $max(j)$ increases : this can happen only if some con-value $c \notin C_j^p$ is contained-in $j$ at $q$ and $c$ is in a maximum-sized join-path of $j$ at $q$. By definition, the size of $c$ is also included in $total(j)$ at $q$. Thus, $c$ being contained-in $j$ at $q$ (and not $p$) causes atleast as much increase in $total(j)$ as in $max(j)$. Thus, an increase in $max(j)$ doesn't actually contribute to a decrease in $exs(j)$.

3. $sub(j)$ increases : $sub(j)$ increases only if (a) for some join-value $j' \in C_j^p \bigcap C_j^q$, $exs(j')$ increases (after $p$) or (b) there exists a join-value $j' \in C_j^q / C_j^p$ such that $exs(j') > V_0$.

   (a) $j' \in C_j^p$ implies that from point $p$ onwards, it is impossible to obtain a direct reference to $j'$, i.e., no expression may evaluate to $j'$. Expressions may evaluate to results that contain $j'$ but even then every path from such a result $r$ to $j'$ must contain a join-value, either $j$ itself or a join-value obtained by selection from $j$. The analysis calls *recomputeExs* only on results of expressions. Thus, after point $p$, *recomputeExs* is never called directly on $j'$. It may be called on a result $r$ that satisfies the constraints just described. By definition, *recomputeExs*$(r)$ recomputes the exs fields of any join-value ancestors of $j'$ but does not recompute $exs(j')$. Hence, for all $j' \in C_j^p$, $exs(j')$ at $p' = exs(j')$ at $p$, for any point $p' > p$. (3a) does not cause any increase in $sub(j)$.

   (b) Let $j'$ be a join-value in $C_j^q \setminus C_j^p$ such that $exs(j')$ at $q > V_0$. $exs(j')$ includes only the sizes of con-values contained-in $j'$. Since $j'$ is contained-in $j$, any con-value $c \in C_j'^q$ is also contained-in $j$ at $q$. Therefore, $total(j)$ at $q$ also includes the size of $c$. Thus, any increase in $sub(j)$ is accompanied by atleast an equal increase in $total(j)$, effectively leaving $exs(j)$ unchanged or larger than before.

# 9 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., [21, 28, 30, 22]. Stack space and heap allocation analysis [31] is similar to time analysis [22]. 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., [32, 10], 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 [26] 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 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 [18, 16], 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. 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. Inoue and others [15] 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 [14, 13, 6] 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 [27]. 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. The analysis can be modified to reflect the effect of optimization of tail recursion. 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 [30, 22].

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

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

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

[10] 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.

[11] 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.

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

[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] 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.

[16] T. P. Jensen and T. Mogensen. A backwards analysis for compile-time garbage collection. In ESOP 1990 [7], pages 227–239.

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

[18] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [9], pages 54–74.

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

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

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

[22] 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.

[23] 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.

[24] 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.

[25] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.

[26] P. Persson. Live memory analysis for garbage collection in embedded systems. In LCTES 1999 [20], pages 45–54.

[27] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.

[28] M. Rosendahl. Automatic complexity analysis. In FPCA 1989 [9], pages 144–156.

[29] I. Ryu. Issues and challenges in developing embedded software for information appliances and telecommunication terminals. In LCTES 1999 [20], pages 104–120. Invited talk.

[30] D. Sands. Complexity analysis for a lazy higher-order language. In ESOP 1990 [7], pages 361–376.

[31] 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.

[32] 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.

[33] P. Zimmermann and W. Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. In *Computer and Information Sciences VI*. Elsevier, 1991.