Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages

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

Abstract

This paper describes a general approach for automatic and accurate space and space-bound analyses for high-level languages, considering stack space, heap allocation and live heap space usage of programs. The approach is based on program analysis and transformations and is fully automatic. The analyses produce accurate upper bounds in the presence of partially known input structures. The analyses have been implemented, and experimental results confirm the accuracy.

1 Introduction

Time and space analysis of computer programs is important for virtually all computer applications, especially in embedded systems, real-time or reactive systems, and interactive systems. In particular, space analysis is becoming increasingly important due to the increasing uses of high-level languages with garbage collection, such as Java [40], the importance of cache memories in performance [54], and the stringent space requirements in the ever-growing area of embedded applications [46]. For example, space analysis is needed to verify that an embedded application can run with certain memory resources, not running out of memory or being given more memory than necessary; it can help determine patterns of space usage and thus help analyze cache misses or page swapping; and it can determine memory allocation and garbage collection behavior.

Space analysis is also important for accurate prediction of running time [20]. For example, analysis of worst-case execution time in real-time systems often uses loop bounds or recursion depths [39, 2]; the former is often determined by the size of the data being processed, and the latter corresponds to the maximum size of the call stack. Also, memory allocation and garbage collection, as well as cache misses and page swapping, 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 [30]. The latter is mostly in the form of tracing memory behavior or analyzing cache effects at the machine level [36, 54]. What has been lacking is analysis of stack space and heap space for high-level languages, in particular, automatic and accurate techniques for live heap space analysis for languages with garbage collection.

This paper describes a general approach for automatic accurate analysis of stack space and heap space based on program analysis and transformations. We consider three important metrics of space usage:

stack space: the maximum size of the function call stack during execution. This is the minimum amount of stack space needed to run the program.

heap allocation: the total amount of heap space allocated during execution. If garbage collection were disabled, this would be the minimum amount of heap space needed to run the program.

*The authors gratefully acknowledge the support of ONR under grants N00014-99-1-0132 and N00014-99-1-0358 and of NSF under grants CCR-9711253 and CCR-9876058. Authors' address: Computer Science Department, 215 Lindley Hall, Indiana University, Bloomington, IN 47405. Email: {lunnikri,stoller,liu}@cs.indiana.edu. Phone: (812)855-{9761,7979,4373}. Fax: (812)855-4829.

live heap space: 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 can be performed at every point during execution.

Our approach starts with a given program written in a high-level language, such as Java, ML, or Scheme. The first step is to automatically construct a *space function* that takes the same input as the original program and returns the amount of space used, either in place of or in addition to the original return value. The space function has an additional argument, which is a vector of *primitive parameters*. The primitive parameters are numbers giving the size of (i.e., amount of space occupied by) each basic data construct used in the program, such as data constructors and stack frame sizes of functions defined in the program.

Since the goal is to calculate space usage without being given particular inputs, the calculation must be based on certain assumptions about inputs. Hence, a *space-bound function* is constructed. This function takes as input a characterization of a set of inputs of the original program and returns an upper bound on the space used by the original program on any input in that set. A key problem is how to characterize the input data and exploit this information in the analysis.

In traditional analysis of the time and space complexity of algorithms, inputs are characterized by their size. Accommodating this requires manual or semi-automatic transformation of the time or space function [53, 32, 55]. 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 [53, 5, 32, 45, 12]. 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* [45]. For example, instead of replacing an input list l with its length n, as done in algorithm analysis, we simply use as input a list of n unknown elements. A special value *unknown* is introduced for this purpose. At control points where decisions depend on unknown values, the maximum of all possible 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 time-bound functions.

We use automatic transformations to construct space-bound functions from the original program. The resulting functions have two kinds of arguments: *input parameters*, which are parameters characterizing partially known input structures, and a vector of primitive parameters, as described above. The only caveat here is that in some cases, the space-bound function might not terminate even though the original program does. Nontermination occurs only if the recursive/iterative structure of the original program depends on unknown parts of the given partially known input structures.

Stack space and heap allocation analyses are similar to the time-bound analysis proposed by Liu and Gómez [34]. Constructions of the space and space-bound functions for analyzing live heap space are based on reference counting [28]. We are analyzing functional programs, so reference counting provides an accurate basis for the analysis. If imperative updates to data constructions are allowed (e.g., *setcdr*! in Scheme), reference counting can be used to obtain upper bounds on the space usage, but the results would sometimes be inexact (i.e., not tight), because reference counting does not recognize that data structures containing cycles can be garbage.

Our analyses can easily be modified to determine related metrics. For example, consider an embedded system in which garbage collection is performed only at fixed points in the program but disabled at other program points in order to avoid delays during time-critical operations. A simple variant of our analysis can determine the live heap space used by programs executed in this way.

We may also use transformations that enable more accurate space bounds to be computed: lifting conditions, simplifying conditionals, and inlining nonrecursive functions, as described in [34]. These transformations should be applied to the original program before the space-bound function is constructed. They may result in larger code size, but they allow subcomputations based on the same control conditions to be merged, leading to more accurate space bounds, which can be computed more efficiently as well.

We choose to start by analyzing a functional subset of Scheme [1, 8] for three reasons. First, high-level languages with features like automatic garbage collection are becoming increasingly widely used. Second, existing work on analysis of functional programs, including complexity analysis, can be exploited in our work. Third, the resulting techniques can be extended later to handle imperative languages.

Our analyses 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.

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, primitive operations on primitive types and structured data (e.g., Boolean and arithmetic operations and data selectors), conditionals, bindings, and mutually recursive function calls. 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[1]

$$\begin{array}{lll} e ::= & v & \text{variable reference} \\ \mid & l & \text{literal} \\ \mid & c(e_1, ..., e_n) & \text{data construction} \\ \mid & p(e_1, ..., e_n) & \text{primitive operation} \\ \mid & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \text{conditional expression} \\ \mid & \textbf{let } v = e_1 \textbf{ in } e_2 & \text{binding expression} \\ \mid & f(e_1, ..., e_n) & \text{function application} \end{array}$$

For binary primitive operations, we sometimes use infix notation. $[x_1, \ldots, x_n]$ denotes a list containing the specified elements, e.g., $[x_1, x_2]$ abbreviates $cons(x_1, cons(x_2, nil))$. Here is a program written in the above language. The program selects the least element in a non-empty list.

$$\begin{array}{rl} least(x) = & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\ & \textbf{else let } s = least(cdr(x)) \textbf{in} \\ & \qquad \textbf{if } car(x) \le s \textbf{ then } car(x) \textbf{ else } s \end{array}$$

The language contains only three kinds of primitive operations that can take constructed data as argument; our analysis can easily be extended to handle other such operations.

Testers. $c?(v)$ returns *true* iff v has outermost constructor c.

Selectors. $c^{-i}(v)$ returns the i'th component of a data construction v with outermost constructor c. c^{-i} aborts if its argument does not have outermost constructor c.

Equality predicates. As in Lisp and Scheme, for data constructions v_1 and v_2, $eq?(v_1, v_2)$ tests whether v_1 and v_2 refer to the same data construction, and $equal?(v_1, v_2)$ tests for structural equality of v_1 and v_2. For primitive values, $eq?$ and $equal?$ both test equality of values. We sometimes write $eq?(v_1, v_2)$ as $v_1 = v_2$.

Input programs to our analysis are assumed to be purely functional, but transformed programs for live heap space analysis use arrays and imperative update. As in Lisp, *setcdr*! updates the second field in a *cons* cell. As in C, v++ and v−− increment and decrement, respectively, the value of an integer variable. A sequential composition $e_1; e_2$ returns the value of e_2.

3 Stack Space Analysis

We measure stack space using *frame count vectors*, which are vectors of integers with one element corresponding to each function (in a given program). Let i_f denote the index associated with function f. Let P_F

[1] The keywords are taken from ML [37]. Our implementation uses Scheme syntax.

be a vector of primitive parameters such that $P_F[i_f]$ is the size of a stack frame for f. Stack frame sizes of functions can be accurately determined based on knowledge about the specific compiler used. Alternatively, these sizes may be conservatively estimated, independent of any particular compiler.

Stack Space Function. The stack space function f_S for function f takes the same inputs as f and returns frame count vectors such that for all inputs x and all functions g, during evaluation of $f(x)$, the size of the stack is at most $f_S(x) \cdot P_F$, where $\cdot$ denotes dot product (also called inner product). The transformation $\mathcal{S}$ that produces f_S appears in Figure 1. Function calls increment the appropriate component of the frame count vector, and the maximum of the stack sizes needed to compute the subexpressions is returned. For clarity of presentation, the transformation for **if** expressions evaluates $\mathcal{S}[\![e_1]\!]$ and e_1 separately. This causes redundancy that can increase the running time of the transformed program by an exponential factor. This is easily avoided by tupling [41]; specifically, the transformation is modified so that $\mathcal{S}[\![e]\!]$ return a pair containing the return value of e and a frame count vector. With this optimization, the asymptotic time complexity of the transformed program is the same as that of the original program.

function def:	$f_S(v_1, \ldots, v_n)$	$= \mathcal{S}[\![e]\!]$ where e is the body of function f, i.e., $f(v_1, \ldots, v_n) = e$
variable:	$\mathcal{S}[\![v]\!]$	$= V_0$
prim. value:	$\mathcal{S}[\![l]\!]$	$= V_0$
prim. operation:	$\mathcal{S}[\![p(e_1, ..., e_n)]\!]$	$= max_F(\mathcal{S}[\![e_1]\!], ..., \mathcal{S}[\![e_n]\!])$
data construction:	$\mathcal{S}[\![c(e_1, ..., e_n)]\!]$	$= max_F(\mathcal{S}[\![e_1]\!], ..., \mathcal{S}[\![e_n]\!])$
conditional:	$\mathcal{S}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$= max_F(\mathcal{S}[\![e_1]\!], \textbf{if } e_1 \textbf{ then } \mathcal{S}[\![e_2]\!] \textbf{ else } \mathcal{S}[\![e_3]\!])$
binding:	$\mathcal{S}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]$	$= max_F(\mathcal{S}[\![e_1]\!], \textbf{let } v = e_1 \textbf{ in } \mathcal{S}[\![e_2]\!])$
function call:	$\mathcal{S}[\![f(e_1, ..., e_n)]\!]$	$= max_F(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!], inc(f_S(e_1, \ldots, e_n), i_f)$

Figure 1: Transformation that produces stack space functions f_S. V_0 denotes a frame count vector with all elements initialized to zero. $inc(v, i)$ returns (a copy of) vector v with the value of the i'th component incremented by 1. max_F is defined by: $max_F(v_1, v_2) = \textbf{if } v_1 \cdot P_F \geq v_2 \cdot P_F \textbf{ then } v_1 \textbf{ else } v_2$.

Stack Space Bound Function. The stack space-bound function f_{Sb} for function f takes partially known values as arguments and returns frame count vectors such that for all partially known values x_{pk} and all values x represented by x_{pk}, during evaluation of $f(x)$, the size of the stack is at most $f_S(x_{pk}) \cdot P_F$. The transformations $\mathcal{S}_e$ and $\mathcal{S}_b$ in Figure 2 produce stack space-bound functions. $\mathcal{S}_e$ is applied to expressions in the original program in order to define functions f_{Se} that take and return partially known values. $\mathcal{S}_b$ is applied to expressions already transformed by $\mathcal{S}$ in order to define stack space-bound functions f_{Sb} that take partially known values as arguments and return frame count vectors. $\mathcal{S}_b$ does not need to be defined for variable references or data constructors, because expressions returned by $\mathcal{S}$ contain variable references and data constructors only in the subexpression e_1 of a binding or conditional, and $\mathcal{S}_e$, not $\mathcal{S}_b$, is applied to those subexpressions.

The main differences between $\mathcal{S}$ and $\mathcal{S}_e$ are in the transformations for **if** expressions and primitive functions. For an **if** expression, if the value of the condition is *unknown*, then both branches must be considered. The programs produced by $\mathcal{S}_b$ compute frame count vectors for both branches and then take the maximum. The programs produced by $\mathcal{S}_e$ compute return values of both branches and then take the least upper bound. The least upper bound function *lub* compares the structure of its arguments, returns known values for substructures where the two arguments are equal, and returns *unknown* for other substructures. $lub(v_1, v_2)$ is v_1 if $eq?(v_1, v_2)$, otherwise it is defined as follows. If v_1 or v_2 is a primitive value, then $lub(v_1, v_2)$ is v_1 if $v_1 = v_2$ and is *unknown* otherwise. For data constructions, $lub(c_1(v_1, \ldots, v_n), c_2(w_1, \ldots, w_m))$ is $c_1(lub(v_1, w_1), \ldots, lub(v_n, w_n))$ if $c_1 = c_2$ and is *unknown* otherwise.

For each primitive function p other than *eq?*, uses of p are replaced with uses of a function p_u, defined as follows. $p_u(v_1, \ldots, v_n)$ returns *unknown* if any of its arguments is *unknown*; otherwise, it returns $p(v_1, \ldots, v_n)$. Special care is needed if an argument to *eq?* is a data construction allocated by *lub*; if it is, then *eq?* returns *unknown*. For example, $\mathcal{S}_e[\![\mathbf{let}\ v = \mathit{cons}(0,0)\ \mathbf{in}\ \mathit{eq?}(v, \mathbf{if}\ \mathit{unknown}\ \mathbf{then}\ v\ \mathbf{else}\ \mathit{cons}(1,1))]\!]$ returns *unknown* but would incorrectly return false if *eq?* were not treated specially. Implementing this requires tagging data constructions allocated by *lub*; such tagging is not shown explicitly in Figure 2.

$\mathcal{S}_e$, not $\mathcal{S}_b$, is used in the definition of $\mathcal{S}_b[\![f_S(e_1, \ldots, e_n)]\!]$, because the space-bound function f_{Sb} takes partially known values, not frame count vectors, as arguments. f_{Sb} uses the partially known values in conditions of **if** expressions to try to determine which branch is taken. This is the motivation for using $\mathcal{S}_e$, not $\mathcal{S}_b$, in the definitions of $\mathcal{S}_b[\![\mathbf{if}\ e_1 \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]$ and $\mathcal{S}_b[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!]$.

The space-bound function may be asymptotically slower than the original function by a factor that is exponential in the size of the input; intuitively, this reflects the fact that a partially known value of size n can represent an exponential number of known values.

Optimization of tail recursion affects the contents of the stack. The analysis presented here does not reflect the effect of such optimization but could be modified to do so.

$$
\begin{array}{lcl}
f_{Se}(v_1, \ldots, v_n) & = & \mathcal{S}_e[\![e]\!] \qquad \text{where } e \text{ is the body of function } f, \text{ i.e., } f(v_1, \ldots, v_n) = e \\
f_{Sb}(v_1, \ldots, v_n) & = & \mathcal{S}_b[\![\mathcal{S}[\![e]\!]]\!] \qquad \text{where } e \text{ is the body of function } f, \text{ i.e., } f(v_1, \ldots, v_n) = e \\
\mathcal{S}_e[\![l]\!] & = & l \\
\mathcal{S}_e[\![v]\!] & = & v \\
\mathcal{S}_e[\![c(e_1, \ldots, e_n)]\!] & = & c(\mathcal{S}_e[\![e_1]\!], \ldots, \mathcal{S}_e[\![e_n]\!]) \\
\mathcal{S}_e[\![p(e_1, \ldots, e_n)]\!] & = & p_u(\mathcal{S}_e[\![e_1]\!], \ldots, \mathcal{S}_e[\![e_n]\!]) \\
p_u(v_1, \ldots, v_n) & = & \mathbf{if}\ (v_1 = \mathit{unknown})\ \mathit{or}\ \cdots\ \mathit{or}\ (v_n = \mathit{unknown})\ \mathbf{then}\ \mathit{unknown}\ \mathbf{else}\ p(v_1, \ldots, v_n) \\
\mathcal{S}_e[\![\mathit{eq?}(e_1, e_2)]\!] & = & \mathbf{let}\ v_1 = \mathcal{S}_e[\![e_1]\!]\ \mathbf{in} \\
 & & \quad \mathbf{let}\ v_2 = \mathcal{S}_e[\![e_2]\!]\ \mathbf{in} \\
 & & \qquad \mathbf{if}\ (v_1 \text{ or } v_2 \text{ is data allocated by } \mathit{lub})\ \mathbf{then}\ \mathit{unknown}\ \mathbf{else}\ \mathit{eq?}_u(v_1, v_2) \\
\mathcal{S}_e[\![\mathbf{if}\ e_1 \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & \mathbf{let}\ v = \mathcal{S}_e[\![e_1]\!]\ \mathbf{in} \\
 & & \quad \mathbf{if}\ v = \mathit{unknown}\ \mathbf{then}\ \mathit{lub}(\mathcal{S}_e[\![e_2]\!], \mathcal{S}_e[\![e_3]\!])\ \mathbf{else\ if}\ v\ \mathbf{then}\ \mathcal{S}_e[\![e_2]\!]\ \mathbf{else}\ \mathcal{S}_e[\![e_3]\!] \\
\mathit{lub}(v_1, v_2) & = & \mathbf{if}\ \mathit{primVal}(v_1)\ \mathit{or}\ \mathit{primVal}(v_2) \\
 & & \mathbf{then\ if}\ v_1 = v_2\ \mathbf{then}\ v_1\ \mathbf{else}\ \mathit{unknown} \\
 & & \mathbf{else\ if}\ v_1 \text{ and } v_2 \text{ have the same outer constructor } c \text{ with arity } n \\
 & & \qquad \mathbf{then}\ c(\mathit{lub}(c^{-1}(v_1), c^{-1}(v_2)), \ldots, \mathit{lub}(c^{-n}(v_1), c^{-n}(v_2))) \\
 & & \qquad \mathbf{else}\ \mathit{unknown} \\
\mathcal{S}_e[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] & = & \mathbf{let}\ v = \mathcal{S}_e[\![e_1]\!]\ \mathbf{in}\ \mathcal{S}_e[\![e_2]\!] \\
\mathcal{S}_e[\![f(e_1, \ldots, e_n)]\!] & = & f_{Se}(\mathcal{S}_e[\![e_1]\!], \ldots, \mathcal{S}_e[\![e_n]\!]) \\
\mathcal{S}_b[\![l]\!] & = & l \\
\mathcal{S}_b[\![p(e_1, \ldots, e_n)]\!] & = & \mathit{max}_F(\mathcal{S}_b[\![e_1]\!], \ldots, \mathcal{S}_b[\![e_n]\!]) \\
\mathcal{S}_b[\![\mathbf{if}\ e_1 \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] & = & \mathbf{let}\ v = \mathcal{S}_e[\![e_1]\!]\ \mathbf{in} \\
 & & \quad \mathbf{if}\ v = \mathit{unknown}\ \mathbf{then}\ \mathit{max}_F(\mathcal{S}_b[\![e_2]\!], \mathcal{S}_b[\![e_3]\!])\ \mathbf{else\ if}\ v\ \mathbf{then}\ \mathcal{S}_b[\![e_2]\!]\ \mathbf{else}\ \mathcal{S}_b[\![e_3]\!] \\
\mathcal{S}_b[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] & = & \mathbf{let}\ v = \mathcal{S}_e[\![e_1]\!]\ \mathbf{in}\ \mathcal{S}_b[\![e_2]\!] \\
\mathcal{S}_b[\![f_S(e_1, \ldots, e_n)]\!] & = & f_{Sb}(\mathcal{S}_e[\![e_1]\!], \ldots, \mathcal{S}_e[\![e_n]\!])
\end{array}
$$

Figure 2: Transformations that produce stack space-bound functions f_{Sb}. p denotes any primitive operation. q denotes any primitive operation other than *eq?*.

Component-Wise Analysis. The above analyses requires that P_F be known during the analysis. Variants of these analyses, which we call *component-wise analyses*, can be performed without knowledge of P_F. "Component" here refers to a component of a frame count vector, i.e., to the stack frame size of a function. The transformations for the component-wise analyses are the same as in Figures 1 and 2, except with max_F replaced with component-wise maximum. The resulting component-wise space and space-bound functions, denoted f_S^{cw} and f_{Sb}^{cw}, respectively, satisfy the same specifications as f_S and f_{Sb}, respectively. Furthermore, f_S^{cw} satisfies: for all inputs x and all functions g, during evaluation of $f(x)$, the stack contains at most $f_S(x)[i_g]$ stack frames for g. f_{Sb}^{cw} satisfy an analogous property. Thus, component-wise analysis provides more information about maximum nesting depth of recursive calls to each function. However, if different components of the frame count vector achieve their maximum values at different points during execution or in different branches of conditionals, it may provide looser bounds on overall stack size.

4 Heap Allocation Analysis

We measure heap space using *constructor count vectors*, which are vectors of integers with one element corresponding to each data constructor (in a given program). Let i_c denote the index associated with data constructor c. Let P_C be a vector of primitive parameters such that $P_C[i_c]$ is the size of an instance of c, including space taken to store the components of c and the type tag, if any.

Heap Allocation Function. The heap allocation function f_H for function f takes the same inputs as f and returns constructor count vectors such that for all inputs x, evaluation of $f(x)$ allocates at most $f_H(x)[i_c]$ instances of c. The transformation $\mathcal{H}$ that produces f_H appears in Figure 3. $\mathcal{H}$ is similar to $\mathcal{S}$; the main differences are that max_F is replaced with add, and components of the count vector are incremented by data constructions, not function calls. As for stack space analysis, the transformation for **if** expressions, as presented in Figure 3, introduces redundancy that can easily be eliminated by tupling. With this optimization, the asymptotic time complexity of the transformed program is the same as that of the original program.

function def:	$f_H(v_1, \ldots, v_n)$	$=$	$\mathcal{H}[\![e]\!]$ where e is the body of function f, i.e., $f(v_1, \ldots, v_n) = e$
variable:	$\mathcal{H}[\![v]\!]$	$=$	V_0
prim. value:	$\mathcal{H}[\![l]\!]$	$=$	V_0
data constr.:	$\mathcal{H}[\![c(e_1, \ldots, e_n)]\!]$	$=$	$inc(add(\mathcal{H}[\![e_1]\!], \ldots, \mathcal{H}[\![e_n]\!]), i_c)$
prim. operation:	$\mathcal{H}[\![p(e_1, \ldots, e_n)]\!]$	$=$	$add(\mathcal{H}[\![e_1]\!], \ldots, \mathcal{H}[\![e_n]\!])$
conditional:	$\mathcal{H}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$=$	$add(\mathcal{H}[\![e_1]\!], \textbf{if } e_1 \textbf{ then } \mathcal{H}[\![e_2]\!] \textbf{ else } \mathcal{H}[\![e_3]\!])$
binding:	$\mathcal{H}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]$	$=$	$add(\mathcal{H}[\![e_1]\!], \textbf{let } v = e_1 \textbf{ in } \mathcal{H}[\![e_2]\!])$
function call:	$\mathcal{H}[\![f(e_1, \ldots, e_n)]\!]$	$=$	$add(\mathcal{H}[\![e_1]\!], \ldots, \mathcal{H}[\![e_n]\!], f_H(e_1, \ldots, e_n))$

Figure 3: Transformation that produces heap allocation functions f_H. V_0 denotes a constructor count vector with all elements initialized to zero. add denotes component-wise addition of vectors. inc is as defined in Figure 1. max_C is defined by: $max_C(v_1, v_2) = \textbf{if } v_1 \cdot P_C \geq v_2 \cdot P_C \textbf{ then } v_1 \textbf{ else } v_2$.

Heap Allocation Bound Function. The heap allocation bound function f_{Hb} for function f takes partially known values as arguments and returns constructor count vectors such that for all partially known values x_{pk} and all values x represented by x_{pk}, evaluation of $f(x)$ allocates at most $f_H(x_{pk}) \cdot P_C$ heap space. The transformations $\mathcal{H}_e$ and $\mathcal{H}_b$ that produce f_{Hb} appear in Figure 4. $\mathcal{H}_e$ is applied to expressions in the original program in order to define functions f_{He} that take and return partially known values. $\mathcal{H}_b$ is applied to expressions already transformed by $\mathcal{H}$ in order to define heap allocation bound functions f_{Hb} that take partially known values and return constructor count vectors. The handling of primitive functions and **if** expressions in these transformations is similar to their treatment in $\mathcal{S}_e$ and $\mathcal{S}_b$.

$$
\begin{array}{lcl}
f_{He}(v_1,\ldots,v_n) & = & \mathcal{H}_e[\![e]\!] \qquad \text{where } e \text{ is the body of function } f\text{, i.e., } f(v_1,\ldots,v_n)=e \\
f_{Hb}(v_1,\ldots,v_n) & = & \mathcal{H}_b[\![\mathcal{H}[\![e]\!]]\!] \quad \text{where } e \text{ is the body of function } f\text{, i.e., } f(v_1,\ldots,v_n)=e \\
\mathcal{H}_e[\![v]\!] & = & v \\
\mathcal{H}_e[\![c(e_1,...,e_n)]\!] & = & c(\mathcal{H}_e[\![e_1]\!],...,\mathcal{H}_e[\![e_n]\!]) \\
\mathcal{H}_e[\![p(e_1,...,e_n)]\!] & = & p_u(\mathcal{H}_e[\![e_1]\!],...,\mathcal{H}_e[\![e_n]\!]) \\
\mathcal{H}_e[\![eq?(e_1,e_2)]\!] & = & \textbf{let } v_1=\mathcal{H}_e[\![e_1]\!] \textbf{ in} \\
 & & \quad \textbf{let } v_2=\mathcal{H}_e[\![e_2]\!] \textbf{ in} \\
 & & \quad\quad \textbf{if } (v_1 \text{ or } v_2 \text{ is data allocated by } lub) \textbf{ then } unknown \textbf{ else } eq?_u(v_1,v_2) \\
\mathcal{H}_e[\![\textbf{if } e_1 \textbf{then } e_2 \textbf{ else } e_3]\!] & = & \textbf{let } v=\mathcal{H}_e[\![e_1]\!] \textbf{ in} \\
 & & \quad \textbf{if } v=unknown \textbf{ then } lub(\mathcal{H}_e[\![e_2]\!],\mathcal{H}_e[\![e_3]\!]) \textbf{ else if } v \textbf{ then } \mathcal{H}_e[\![e_2]\!] \textbf{ else } \mathcal{H}_e[\![e_3]\!] \\
\mathcal{H}_e[\![\textbf{let } v=e_1 \textbf{ in } e_2\ ]\!] & = & \textbf{let } v=\mathcal{H}_e[\![e_1]\!] \textbf{ in } \mathcal{H}_e[\![e_2]\!] \\
\mathcal{H}_e[\![f(e_1,...,e_n)]\!] & = & f_{He}(\mathcal{H}_e[\![e_1]\!],...,\mathcal{H}_e[\![e_n]\!]) \\
\mathcal{H}_b[\![l]\!] & = & l \\
\mathcal{H}_b[\![add(e_1,...,e_n)]\!] & = & add(\mathcal{H}_b[\![e_1]\!],...,\mathcal{H}_b[\![e_n]\!]) \\
\mathcal{H}_b[\![\textbf{if } e_1 \textbf{then } e_2 \textbf{ else } e_3]\!] & = & \textbf{let } v=\mathcal{H}_e[\![e_1]\!] \textbf{ in} \\
 & & \quad \textbf{if } v=unknown \\
 & & \quad \textbf{then } max_C(\mathcal{H}_b[\![e_2]\!],\mathcal{H}_b[\![e_3]\!]) \\
 & & \quad \textbf{else if } v \textbf{ then } \mathcal{H}_b[\![e_2]\!] \textbf{ else } \mathcal{H}_b[\![e_3]\!] \\
\mathcal{H}_b[\![\textbf{let } v=e_1 \textbf{ in } e_2]\!] & = & \textbf{let } v=\mathcal{H}_e[\![e_1]\!] \textbf{ in } \mathcal{H}_b[\![e_2]\!] \\
\mathcal{H}_b[\![f_H(e_1,...,e_n)]\!] & = & f_{Hb}(\mathcal{H}_e[\![e_1]\!],...,\mathcal{H}_e[\![e_n]\!])
\end{array}
$$

Figure 4: Transformations that produce heap allocation bound functions f_{Hb}. p, q, p_u, and lub have the same meanings as in Figure 2.

Component-Wise Analysis. A component-wise variant of heap allocation bound analysis is defined by replacing max_C with component-wise maximum in Figure 4. The resulting component-wise space-bound function f_{Hb}^{cw} satisfies the same specification as f_{Hb} and the property: for all partially known values x_{pk}, all values x represented by x_{pk}, and all data constructors c, evaluation of $f(x)$ allocates at most $f_{Hb}(x)[i_c]$ instances of c. max_C is not used in Figure 3, so f_H satisfies an analogous component-wise property. For programs that use only one constructor (e.g., Lisp or Scheme programs that use only *cons*), the original and component-wise analyses are equivalent. For programs that use multiple constructors, the component-wise analysis provides more information about the maximum number of instances of each constructor; however, if different components of the constructor count vector achieve their maximum values in different branches of conditionals, it may provide looser bounds on the overall amount of heap allocation.

5 Live Heap Space Analysis

Our analysis is based on reference counting, defined as follows.

> The *reference count* (abbreviated rc) 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 on the heap, created by data constructions.

We measure heap space using *constructor count vectors*, which are vectors of integers with one element corresponding to each data constructor (in a given program). Let i_c denote the index associated with data constructor c. Let P_C be a vector of primitive parameters such that $P_C[i_c]$ is the size of an instance of c, including space taken to store the components of c and the type tag, if any. Let $\cdot$ denote dot product (also called inner product) of vectors. The maximum $\max(v_1, v_2)$ of constructor count vectors v_1 and v_2 is v_1 if $v_1 \cdot P_C \geq v_2 \cdot P_C$ and is v_2 otherwise.

The transformations introduce two global variables, *live* and *maxlive*, that contain constructor count vectors and satisfy the invariants: (1) for each constructor i, $live[i_c]$ 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 $ml \cdot P_C$ where ml is the value of *maxlive* after evaluation of the space or space-bound function for f.

Our implementation of reference counting uses an abstract data type whose signature contains four functions (or macros). $new(c(x_1, \ldots, x_n))$ 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. $setrc(v, i)$ sets the reference count associated with v to i. $incrc(v)$ and $decrc(v)$ increment and decrement, respectively, the reference count associated with v. We require that *setrc*, *incrc* and *decrc* be no-ops if the argument is a primitive value.

This abstraction can be implemented in various ways. In our implementation, each data construction is paired (using *cons*) with its reference count; thus, [2] $new(c(x_1, \ldots, x_n))$ is $cons(c(x_1, \ldots, x_n), 0)$. It follows that *data* is *car*, *rc* is *cdr*, and the remaining functions are easily implemented using *cons*? and *setcdr*!.

Updating Reference Counts. The reference count of a data construction v is initialized to zero, because pointers to v of the kinds described above do not yet exist. Of course, some temporary storage location—perhaps a register—must point to v, but according to the above definition of reference count, such pointers are not counted (*cf.* the comments in the next paragraph). Thus, the reference count of v is incremented when v is bound to a variable or function parameter, or a data construction containing v as a child is created. The reference count of v is decremented when the scope of a **let** binding for v ends, a function call with an argument equal to v returns, or a data construction containing v as a child becomes garbage. Note that the reference count is defined to count pointers that exist during execution of the original program, not the transformed program.

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. Data is garbage if there is no pointer to it from the stack or heap and no such pointer can be created in the future (*cf.* the discussion of temporary increments, below). 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 can become garbage in three ways:

- Data created in the argument of a selector, tester, or equality predicate is lost to the program after the result of the selection, test, or equality predicate is obtained. For example,

  $$cons(cdr(cons(0,1))^{\dagger *}, 2)$$
  $$cons?(cons(0,1))^{\dagger *} \ or \ equal?(1,2)^{*}$$
  $$\textbf{if } eq?(cons(0,1), nil)^{\dagger *} \textbf{ then } 1 \textbf{ else } 2$$

  The dagger indicates where $cons(0,1)$ becomes garbage. The asterisk indicates where *gc* is called. Note that $cons(0,1)$ has reference count 0 when *gc* is called. *gc* decrements the reference count of $cons(0,1)$ to -1 before concluding that $cons(0,1)$ is garbage; that decrement is unnecessary but harmless. It is harmless because the data construction is garbage, so its reference count will never be examined again.

- Data bound to variables through **let** expressions or formal parameters of functions may become garbage

[2] We use ML-like notation here, although our implementation is in Scheme, which has different syntax.

when the variables go out of scope. Here are 3 examples:

$$\begin{array}{l} \mathbf{let}\ v = cons(0,1)\ \mathbf{in}\ 1^{\dagger *} \\ \mathbf{let}\ v = cons(2,3)\ \mathbf{in}\ v^{*} \\ cons(f(cons(0,1), cons(2,3))^{\dagger *}, 4) \quad \text{where } f(x,y) = y \end{array} \tag{1}$$

The dagger and asterisk have the same meaning as above. In the first and third examples, when *gc* is called, $cons(0,1)$ has reference count 1 and becomes garbage. In the second and third examples, $cons(2,3)$ does not become garbage, because it appears in the return value; this point is discussed further below.

- Data constructions that have become garbage may have components that refer to data constructions to which there are no other pointers. These data constructions also become garbage. They have reference count 1 when *gc* is called on them.

Temporary Increments of *rc*. In order to avoid v being considered as garbage when only temporary storage locations point to v, $rc(v)$ is temporarily incremented at certain points in the transformed program. Specifically, temporary increments of *rc* are required for selectors, function calls, and **let** expressions, since these expressions may produce garbage, but the garbage produced depends on what data construction (if any) is referenced by the return value. Since the return value is initially stored in a temporary storage location(until it is discarded or an enclosing function, **let** expression, or data construction stores it on the stack or heap), a temporary increment of its *rc* is needed. Accordingly, letting r denote the return value, we increment $rc(r)$ immediately before calling *gc* and decrement $rc(r)$ immediately afterwards. For example, in $car(cons(cons(0,1), nil))^{*}$, the value of $rc(cons(0,1))$ is temporarily incremented while the outer *cons* is garbage collected. The second and third lines of (1) provide examples involving **let** and function call; $rc(cons(2,3))$ is temporarily incremented to 2 before *gc* is called and is decremented to 0 afterwards.

We regard these temporary increments as transient deviations between the value of *rc* and the definition of reference count. Making them part of the definition would be problematic, because the exact number and scopes of temporary storage locations is compiler-dependent.

The transformation $\mathcal{L}$ in Figure 5 produces live heap space functions. For convenience, the transformation for **if** expressions assumes that the condition is Boolean-valued (otherwise, the program aborts). The transformation for function calls assumes that the evaluation order for **let** bindings and function arguments is the same; for example, this is true in Chez Scheme [3]. This assumption is easily avoided, by modifying the transformation so that the **let** bindings appear in the function arguments. Live heap space may depend on the evaluation order. Our transformation is correct regardless of the evaluation order (e.g., left-to-right or right-to-left), assuming the same compiler is used to run the original and transformed programs.

Optimization of tail recursion affects the contents of the stack and hence reference counts. The analysis presented here does not reflect the effect of such optimization but could be modified to do so.

6 Live Heap Space Bound Function

The transformation $\mathcal{L}_b$ in Figures 8-10 produces live heap space-bound functions. This transformation ensures that at every point during the execution of $\mathcal{L}_b[\![f]\!](x)$ the value of global variable *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, global variable *maxlive* contains the maximum value of *live* so far during execution, with one exception, described below.

Conditional expressions whose tests evaluate to *unknown* are the only source of uncertainty in bounds analysis. In stack space and heap allocation bound analyses, such expressions return data that is the least upper bound of the values of the branches. This does not work well for live space bound analysis whose correctness depends on keeping track of all references and reference counts meticulously. We need to keep references from the result r of such a conditional expression, to the results r_1 and r_2 of its branches. If we did not, and if r stays live longer than r_1 and r_2, when r_1 and r_2 become garbage, we might incorrectly subtract their sizes from *live* although one of them is actually live.

$$
\begin{aligned}
&f_L(v_1,\ldots,v_n) = \mathcal{L}[\![e]\!] \quad \text{where } e \text{ is the body of function } f\text{, i.e., } f(v_1,\ldots,v_n) = e\\
&\mathcal{L}[\![v]\!] = v\\
&\mathcal{L}[\![l]\!] = l\\
&\mathcal{L}[\![c(e1,\ldots,en)]\!] = \mathit{live}[i_c]{+}{+};\, \mathit{maxlive} = \max(\mathit{live},\mathit{maxlive});\\
&\qquad\qquad\qquad\quad \textbf{let } r_1 = \mathcal{L}[\![e_1]\!],\ldots,r_n = \mathcal{L}[\![e_n]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\, \mathit{new}(c(r_1,\ldots,r_n))\\
&\mathcal{L}[\![q(e_1,\ldots,e_n)]\!] = q(\mathcal{L}[\![e_1]\!],\ldots,\mathcal{L}[\![e_n]\!])\\
&\mathcal{L}[\![q'(e_1,\ldots,e_n)]\!] = \textbf{let } x_1 = \mathcal{L}[\![e_1]\!],\ldots,x_n = \mathcal{L}[\![e_n]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad \textbf{let } r = q'(\mathit{data}(x_1),\ldots,\mathit{data}(x_n)) \textbf{ in}\\
&\qquad\qquad\qquad\qquad\quad \textbf{if } \mathit{not}(\mathit{primVal}(x_1)) \textit{ and } \mathit{rc}(x_1) = 0 \textbf{ then } \mathit{gc}(x_1);\\
&\qquad\qquad\qquad\qquad\quad \ldots\\
&\qquad\qquad\qquad\qquad\quad \textbf{if } \mathit{not}(\mathit{primVal}(x_n)) \textit{ and } \mathit{rc}(x_n) = 0 \textbf{ then } \mathit{gc}(x_n);\\
&\qquad\qquad\qquad\qquad\quad r\\
&\mathcal{L}[\![c^{-i}(e)]\!] = \textbf{let } x = \mathcal{L}[\![e]\!] \textbf{ in}\\
&\qquad\qquad\quad \textbf{let } r = c^{-i}(\mathit{data}(x)) \textbf{ in}\\
&\qquad\qquad\qquad \textbf{if } \mathit{not}(\mathit{primVal}(x)) \textit{ and } \mathit{rc}(x) = 0 \textbf{ then } \mathit{incrc}(r);\, \mathit{gc}(x);\, \mathit{decrc}(r);\\
&\qquad\qquad\qquad r\\
&\mathcal{L}[\![\textbf{if } e1 \textbf{ then } e2 \textbf{ else } e3]\!] = \textbf{if } \mathcal{L}[\![e1]\!] \textbf{ then } \mathcal{L}[\![e2]\!] \textbf{ else } \mathcal{L}[\![e3]\!]\\
&\mathcal{L}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] = \textbf{let } v = \mathcal{L}[\![e_1]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad\quad \mathit{incrc}(v);\\
&\qquad\qquad\qquad\qquad\quad \textbf{let } r = \mathcal{L}[\![e_2]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{incrc}(r);\, \mathit{gc}(v);\, \mathit{decrc}(r);\, r\\
&\mathcal{L}[\![f(e_1,\ldots,e_n)]\!] = \textbf{let } r_1 = \mathcal{L}[\![e_1]\!],\ldots,r_n = \mathcal{L}[\![e_n]\!] \textbf{ in}\\
&\qquad\qquad\qquad\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\\
&\qquad\qquad\qquad\qquad \textbf{let } r = f_L(r_1,\ldots,r_n) \textbf{ in}\\
&\qquad\qquad\qquad\qquad\quad \mathit{incrc}(r);\, \mathit{gc}(r_1);\ldots;\mathit{gc}(r_n);\, \mathit{decrc}(r);\, r\\
&\mathit{gc}(v) = \textbf{if } \mathit{not}(\mathit{primVal}(v))\\
&\qquad\quad \textbf{then } \mathit{decrc}(v);\\
&\qquad\qquad\quad \textbf{if } \mathit{rc}(v) \le 0\\
&\qquad\qquad\quad \textbf{then } \mathit{live}[\mathit{consType}(v)]{-}{-};\\
&\qquad\qquad\qquad\quad \textbf{for } i = 1..\mathit{arity}(v)\ \mathit{gc}(c^{-i}(\mathit{data}(v)))
\end{aligned}
$$

Figure 5: Transformation that produces live heap space functions f_L. q denotes any primitive operation other than a selector, tester, or equality predicate. q' denotes a tester or equality predicate. $\mathit{primVal}(v)$ returns *true* iff v is primitive data. $\mathit{consType}(v)$ 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 $\mathit{data}(v)$. *consType* and *arity* abort (with a run-time type error) if the argument is not an abstract data type representing a data construction.

By maintaining these references from r to r_1 and r_2, we run the risk of obtaining loose upper bounds, since *live* might include the sizes of both branches when only one is live. We deal with this by subtracting from *live*, when appropriate, the size of the smaller branch. With an expression containing nested conditional expressions, we subtract from *live*, when appropriate, the sizes of all the data returned by the branches of the conditional expressions, other than the size of a single largest data structure that could be returned by the entire expression.

6.1 Abstract Data Types

Two abstract data types, lubData and constrData, are used. lubData is used to represent data that may be different data constructions for different inputs to the program being analyzed. lubData are results of either conditional expressions whose tests evaluate to *unknown* or selectors applied to lubData. Each lubData l has a list $branches(l)$ containing references to the data constructions that l represents, i.e., that l could be in an execution of the original program. Since conditional expressions are the only branching expressions, lubData have at most two branches. If l represents a choice between primitive and non-primitive data, then $branches(l)$ has only one element. If the choice is between two non-primitive data, then $branches(l)$ has two elements. Each lubData l has a rc $rc(l)$; functions *setrc*, *incrc*, and *decrc* apply to lubData. Each lubData l has a list $lubParents(l)$ of *lubparents*, which are lubData v such that $l \in branches(v)$. $addToLubParents(v, l')$ and $remFromLubParents(v, l')$ adds l' to and removes l' from, respectively, v's list of lubparents. Each lubData l has an associated constructor count vector $min(l)$, which is the amount subtracted from *live* in order to obtain the effect of keeping only a largest branch of l alive. When l becomes garbage, $min(l)$ is added to *live* just before garbage collecting the branches of l. $newlub(b)$ creates a lubData l with a list b of branches, and with $rc(l)$ initialized to 0, $lubParents(l)$ initialized to *nil*, and $min(l)$ initialized to the zero vector, which is denoted V_0.

constrData is an extension of the ADT described in Section 5. Each constrData has a rc and a list of lubparents. Functions *new*, *rc*, *setrc*, *incrc*, *decrc*, *lubParents*, *addToLubParents*, and *remFromLubParents* apply to constrData.

6.2 Conditionals and Least Upper Bounds

Consider a conditional expression $(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^\dagger$ whose condition evaluates to *unknown*. The value of *live* needed at $\dagger$ is the larger of the values of *live* obtained by executing the two branches of the conditional expression. Therefore, both branches are executed by the transformed program. Suppose the value of *live* after the evaluation of e_1 is l_1. e_2 is evaluated first, with the value of *live* set to l_1. Let l_2 denote the resulting value of *live*. The value of *live* is reset to l_1 and e_3 is evaluated. Let l_3 denote the resulting value of *live*. The required value of *live* at $\dagger$ is $\max(l_2, l_3)$. We achieve this by first setting the value of *live* to

$$l_1 + (l_2 - l_1) + (l_3 - l_1) \tag{2}$$

and relying on *lub*, as described in case (d) below, to subtract $\min(l_2 - l_1, l_3 - l_1)$ from *live*. The intermediate value of *live* computed using (2) does not satisfy the invariant for *maxlive*; this is the exception mentioned above.

The result of the conditional expression is computed by $lub(r_2, r_3, l_2 - l_1, l_3 - l_1)$, where r_2 and r_3 are the results of e_2 and e_3, respectively. Let r denote the result of this invocation. There are four possible situations.

a) r_2 and r_3 are primitive data. Then r is their least upper bound, i.e., r_2 if r_2 and r_3 are the same and *unknown*, otherwise.

b) r_2 and r_3 are the same non-primitive data. Then r is that non-primitive data.

c) Exactly one of r_2 and r_3 is non-primitive. Suppose, without loss of generality, that r_2 is non-primitive. Then, r is $newlub([r_2])$. The reference from $branches(r)$ to r_2 keeps r_2 alive for at least as long as r is live, i.e., r_2 is garbage collected and its size subtracted from *live* only after or when r gets garbage collected. $rc(r_2)$ is incremented and r is added to $lubParents(r_2)$ to reflect the reference from $branches(r)$ to r_2. Since r_3 is primitive, it does not use heap space and we do not keep track of it. This creates an approximation similar to the one in case (a). $rc(r)$ is 0 since there is no reference to r. Notice that $rc(r)$ is a bound on the number of references made to the result of the conditional expression and does not include any other references to r_2. $lubParents(r)$ is *nil*. The value of *live* at $\dagger$ computed using (2) is l_2. This being the maximum possible value of *live* at $\dagger$, nothing needs to be subtracted from *live*, so $min(r)$ is V_0.

d) r_2 and r_3 are non-primitive and different. r is $newlub([r_2, r_3])$. The references from $branches(r)$ to r_2 and r_3 keep them alive for at least as long as r stays live. The following paragraphs and Section 6.5 describe how tight bounds are attained despite keeping both branches live. $rc(r_2)$ and $rc(r_3)$ are incremented and r is added to $lubParents(r_2)$ and $lubParents(r_3)$ to reflect the references from $branches(r)$. The values of $rc(r)$ and $lubParents(r)$ are as in case (c). Recall that $min(l)$ for lubData l is the amount subtracted from *live*, if any, in order to obtain the effect of keeping only a largest branch of l alive. $min(r)$ is set to $\min((l_2 - l_1), (l_3 - l_1))$. This value is explained as follows.

i) Suppose at least one of $l_2 - l_1$ and $l_3 - l_1$ is a zero vector. $l_2 - l_1$ and $l_3 - l_1$ are the amounts of new data in r_2 and r_3, respectively. This means that at least one of r_2 and r_3 contains no new data, so keeping both of them alive (by references from r) does not cause any over-approximation at this point in the execution, so $min(r)$ is V_0.

ii) Suppose both $l_2 - l_1$ and $l_3 - l_1$ are non-zero vectors. This implies that both r_2 and r_3 contain new data whose sizes total to $l_2 - l_1$ and $l_3 - l_1$, respectively. It is impossible for both sets of new data to be live in an execution of the original program. So, we subtract from *live* the size of the smaller set, i.e., $\min(l_2 - l_1, l_3 - l_1)$ and set $min(r)$ to this amount. As a result of this subrtraction, the original value of *live* computed using (2) reduces to $\max(l_2, l_3)$, as required. Thus, we keep the new data in both r_2 and r_3 live but include only one of their sizes in *live*. Old data in r_2 and r_3 are created before the conditional expression and are referenced from data other than r_2 and r_3. They are live at † because of these other references. Hence, the sizes of such data are not included in the value of $min(r)$, even if they occur in only one of the branches of r.

In the following examples, we examine the values of *live* and *maxlive* at different points of $\mathcal{L}_b[e]$, for an expression e. It is easy to verify that these values are tight upper bounds on the possible values of *live* and *maxlive* at the corresponding points in $\mathcal{L}[e']$, for any expression e' obtained from e by substituting either $true$ or $false$ for *unknown*.

Example 1. This example illustrates case (c).

$$\begin{array}{l} \textbf{let } u = (\textbf{let } v = cons(1, nil) \textbf{ in} \\ \qquad\qquad (\textbf{if } unknown \textbf{ then } v \textbf{ else } 0)^{\dagger}) \textbf{ in} \\ \quad cons(2, u)^{*} \end{array}$$

Before the evaluation of the conditional,
$live = \langle 1\rangle$ (since the size of $cons(1, nil)$ is $\langle 1\rangle$)
$maxlive = \langle 1\rangle$
At †, letting r denote the result of the conditional,
v is bound to v'.
$v' = \langle cons(1, nil),\ rc = 2,\ lubParents = nil\rangle$
(references are from r and the binding for v)
$r = \langle branches = [v'],\ rc = 0,\ lubParents = nil,\ min = V_0\rangle$
($min(r)$ is V_0 since r has only one non-primitive branch)
$live = \langle 1\rangle$, $maxlive = \langle 1\rangle$

At $*$,
Let s denote the result of the entire expression.
u is bound to r.
$r = \langle branches = [v'],\ rc = 2,\ lubParents = nil,\ min = V_0\rangle$
(references are from s and the binding for u)
$s = \langle cons(2, r),\ rc = 0,\ lubParents = nil\rangle$
$live = \langle 2\rangle$, $maxlive = \langle 2\rangle$

The value of *live* at † is the same irrespective of the branch executed. The value of *live* at $*$ is affected by the branch taken; the larger value of *live* at $*$ is obtained by keeping v' live till $*$, using a reference from $branches(r)$ to v'; without this reference, v' would incorrectly be considered garbage when v goes out of scope.

Example 2. This example illustrates case d(ii).

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

Before the conditional expression, $live = \langle 2\rangle$ and $maxlive = \langle 2\rangle$. We have $l_1 = \langle 2\rangle$, $l_2 = \langle 3\rangle$ and $l_3 = \langle 4\rangle$, where l_1, l_2 and l_3 are as defined earlier. The value of *live* at † before the call to *lub* is $\langle 5\rangle$. The *min* value of the result r of the conditional expression is $\min(\langle 1\rangle, \langle 2\rangle) = \langle 1\rangle$. The value of *live* after the call to *lub* is $\langle 5\rangle - \langle 1\rangle = \langle 4\rangle$.

6.3 Selectors, Testers and Equality Predicates

Selectors and testers return *unknown* if given *unknown* arguments. If the argument to a tester q is a lubData with a single branch b, then, since one of the branches is primitive data on which the tester must return *false*, $lub(false, q(b))$ is returned. If the argument to a selector q is a lubData with a single branch, then, since applying q to any primitive data must cause an error, q is applied to *false*, causing the analysis to abort with an error. If the argument to a selector or tester q is a lubData with two branches b_1 and b_2, then $lub(q(b_1), q(b_2))$ is returned. For testers, the result of this *lub* may be *true*, *false* or *unknown*. For selectors, the result is (a) *unknown* if the selected fields in both branches are different primitive data, (b) data d if the selected fields in both branches contain d or (c) a new lubData otherwise. Suppose l is a lubData created in case (c) by applying a selector to a lubData l'. When l is created, l' is still live. Hence, applying a selector to l' and as a result, creating new references to data that appear in elements of $branches(l')$ does not affect the liveness of these data; these data are live regardless of the references from l and l has its *min* value set to V_0. In all calls to *lub* above, the third and fourth arguments to *lub*, which have been omitted, are *nil*. *lub* uses these arguments to determine the *min* value of the lubData, if any, that it creates. In all the above calls to *lub*, either no lubData is created or the *min* value of the created lubData is V_0.

Values of equality predicates applied to lubData are defined similarly. For lubData l and constrData v, if l has two branches b_1 and b_2, then $eq?(l, v)$ is $lub(eq?(b_1, v), eq?(b_2, v))$; otherwise, l has only one branch b_1, and the implicit second branch of l represents primitive data and therefore is not equal to v, so $eq?(l, v)$ is $lub(eq?(b_1, v), false)$. For lubData l_1 and l_2, $eq?(l_1, l_2)$ is true iff l_1 and l_2 are the same lubData; otherwise, if l_2 has two branches b_{21} and b_{22}, then the result is $lub(eq?(l_1, b_{21}), eq?(l_1, b_{22}))$; otherwise, l_2 has only one branch, and the result is *unknown*, since the primitive data represented by the implicit second branch is not available to compare with. *equal?* is defined similarly. Equality predicates are not used in our examples, so we omit transformations for them from Figure 8.

6.4 Recomputing the Min Value of LubData

Recall that $min(l)$ is the amount subtracted from *live* in order to obtain the effect of keeping only a largest branch of lubData l alive. The *min* value of a lubData is computed and initialized when the lubData is created. It may need to be recomputed at certain later points to make *live* tight. Consider a lubData l that is the result of a conditional expression of case (d) described in Section 6.2. l has two non-primitive branches b_1 and b_2. Suppose one or both of b_1 and b_2 are constructed before the conditional expression and hence are live without references from l, when l is constructed. Then $min(l)$ is initially set to V_0, and *live* contains the sizes of both b_1 and b_2. When all references to b_1 and b_2, except for those from l, disappear, we may conclude that only one of b_1 and b_2 is live at the corresponding point in an execution of the original program. In order to include the size of exactly one largest branch of l in *live*, $min(l)$ is recomputed. Notice that $min(l)$ must remain V_0 as long as one or both of b_1 and b_2 are live without references from l. If one branch, say b_1, of l has references from l alone and the other branch, b_2, has references from data other than l, then we cannot conclude that only one of b_1 and b_2 is live in the original program. Both b_1 and b_2 may be live; b_1 through l, i.e., as the result of the conditional, and b_2 through data other than l.

Example. Consider the following conditional expression of case d(i), whose result is a lubData r.

$$\begin{array}{l}\textbf{let } x = cons(1, nil) \textbf{ in}\\ \quad \textbf{if } unknown \textbf{ then } x \textbf{ else } cons(2, nil)\end{array}$$

Just after the evaluation of the conditional expression, *live* is $\langle 2 \rangle$ and includes the size of both $cons(1, nil)$ and $cons(2, nil)$; $min(r)$ is V_0. This is correct since the former is live through the binding for x. After the evaluation of the **let** expression, x goes out of scope, and the only reference to $cons(1, nil)$ is from r. Now, *live* should contain the size of exactly one of $cons(1, nil)$ and $cons(2, nil)$. Hence, $min(r)$ is recomputed and set to $\langle 1 \rangle$. Once the new value of $min(r)$ is subtracted from *live*, *live* becomes $\langle 1 \rangle$, as required.

Suppose some data construction v, created before a conditional expression that evaluates to lubData l, is part of a branch of l. The size of v remains included in *live* even after the initial computation of $min(l)$ when l is created. At this point, this does not cause any looseness in *live* since v is indeed live. When all references to v except for the ones from l disappear, then *live* may in fact be loose since it still includes the size of v which may not be part of the larger sized branch of l. At this point, $min(l)$ must be recomputed.

Example. In example (3), $cons(1, nil)$ and $cons(2, nil)$ are part of one or the other branch of the result r of the expression. At †, *live* includes the sizes of both these data constructions. This is correct since both $cons(1, nil)$ and $cons(2, nil)$ are live through bindings for u and v, respectively. v goes out of scope after the evaluation of the **let** expression that binds v. Now, $cons(2, nil)$ is live through r alone but since it occurs in the larger branch of r, *live* is still tight. u goes out of scope after the evaluation of the **let** expression that binds u. Now, since $cons(1, nil)$ is live only through r and is part of the smaller branch of r, its size is an excess in the current value of *live*, so $min(r)$ is recomputed and the value of *live* updated.

The *min* value of a lubData l with only one non-primitive branch never needs to be computed. As explained under case (c) in Section 6.2, since the primitive branch of l occupies no heap space, l does not create any looseness in the value of *live* by keeping its non-primitive branch live. Hence, *live* is tight as is and $min(l)$ is always V_0.

6.5 Definition of the Min Value of LubData

If all the descendants of a lubData l with two branches are constrData then it is easy to see that l represents a choice between exactly two data structures and that $min(l)$ is just the size of smaller of the two data structures. If l has lubData descendants, then in effect, l represents more than 2 possible data structures. $min(l)$ includes the size of all these data structures except a largest one. Sizes of parts of these data structures may be counted in *min* values of descendant lubData of l and hence already be subtracted from *live*. Sizes of those parts are not included in $min(l)$.

We consider the stack and live heap as a graph: formal parameters of functions, local variables placed on the stack and data in the heap are vertices, and references from a variable to a datum or from one datum to another, including, for lubData l, references in $branches(l)$ but excluding references in $lubParents(l)$, are edges. We say that u is *contained-in* v if v is an ancestor of u in every path into u. For a lubData l, let C_l denote the set of all data contained in l, and let G_l denote the graph comprising vertices and edges reachable from l. A *lub-path* of l is a subgraph of G_l containing l and constructed from G_l by selecting at every lubData descendant l' of l, including l itself, exactly one branch of l' and eliminating unreachable vertices and edges. A lub-path is not a *path* in the traditional sense since it contains all outedges of component constrData. Lub-paths of l correspond to data structures represented by l. $conTypeVec(u)$ for a constrData u is a constructor count vector in which the count of the constructor type of u is 1 and all other components are 0. $maxLubPath(l)$ is the maximum of the sizes of all lub-paths of l. If G_b is a lub-path of l, then

$$size(G_b) = \sum_{u \text{ is a constrData in } G_b \cap C_l} conTypeVec(u)$$

For a lubData l, if $branches(l)$ has two elements, both of which are contained-in l, $min(l)$ is defined as follows(lubData for which these conditions don't hold have *min* values of V_0).

$$min(l) = total(l) - sub(l) - maxLubPath(l) \tag{4}$$

$$total(l) = \sum_{u \text{ is a constrData in } C_l} conTypeVec(u) \tag{5}$$

$$sub(l) = \sum_{u \text{ is a lubData in } C_l} min(u) \tag{6}$$

All quantities in (4) are constructor count vectors. The sum and difference of vectors in (4) are computed component-wise. This is safe, i.e., does not result in vectors with negative counts, since *sub*(l) and *maxLubPath*(l) count data in disjoint subsets of C_l. This is explained as follows : if the lub-path P which contributes to *maxLubPath*(l) contains a lubData l', then P contains a largest lub-path of l' and $min(l')$ counts data in the other lub-paths of l'. Hence, $min(l')$, for any descendant lubData l' of l, counts data in lub-paths that are not part of P. Informally, (4) says "subtract from *live* everything except a largest lub-path and nodes that have already been subtracted from *live*".

6.6 Computing the Min Value of LubData

We first deal with computing the *min* value of a lubData l when both elements b_1 and b_2 of $branches(l)$ become contained-in l. This happens immediately after a decrement of $rc(b_1)$ or $rc(b_2)$ in *gc*. *gc* may access l through $lubParents(b_1)$ or $lubParents(b_2)$. *gc* checks whether the reference counts of b_1 and b_2 are 1; if so, b_1 and b_2 are contained-in l, so *gc* calls $computeMin(l)$. This conservative approximation to checking contained-in is faster than an exact check and suffices to obtain tight bounds for all of our examples.

The functions used to compute min values of lubData appear in Figure 9. *computeMin* is called only on lubData whose branches have two elements both of which are contained-in the lubData; this is conservatively checked using a call to $containedIn_0$. *computeMin* may not be used to compute *min* values of any other kind of lubData. *computeMin* calls three functions, each of which makes one pass through G_l, where l is the argument to *computeMin*. The first pass, made by *totalSub*, is used to compute $total(l)$ and $sub(l)$. Each time *totalSub* visits a node v, it decrements the rc of v. v's contribution to $total(l)$ and $sub(l)$ is counted only when v's rc reduces to zero; this ensures that v's contribution is counted only once even if there are multiple paths from l to v. In the second pass, *maxLubPath* determines the maximum of the sizes of lub-paths of l in a bottom-up manner, first computing the maximum lub-path sizes of descendant lubData of l and then using these results to compute the maximum lub-path sizes of ancestors. Because of the decrement in the first pass, nodes contained in l are easily identified as those with rc 0. In the third pass, *recIncrc* restores rc's of nodes in G_l to their original values.

A conditional expression of type d(ii) described in Section 6.2 evaluates to a lubData l such that, the elements of $branches(l)$ are contained-in l when l is created. For efficiency, instead of calling *computeMin* to determine $min(l)$, *lub* sets $min(l)$ to $\min(l_2 - l_1, l_3 - l_1)$, where l_1, l_2, l_3 are the values of *live* after the evaluation of the test, true branch and false branch, respectively. The two computations yield the same result. Intuitively, this is because $\min(l_2 - l_1, l_3 - l_1)$ is the lesser of the amounts of new data in the results of the two branches and the new data in the results are the only data that can be contained-in the resulting lubData of the conditional expression.

Example 1. This example demonstrates the computation of the *min* value of a lubData that has only constrData descendants.

$$\begin{array}{l}
\textbf{let } x = cons(1, nil) \textbf{ in} \\
\quad \textbf{let } r = (\textbf{let } u = cons(2, x) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } v = cons(3, x) \textbf{ in} \\
\qquad\qquad\quad\;\; \textbf{if } unknown \textbf{ then } u \textbf{ else } v)^{\dagger} \textbf{ in} \\
\quad\;\; \ldots
\end{array} \tag{7}$$

Let r' be the result of the conditional expression. The elements of $branches(r')$ become contained-in r' at †. Figure 6 shows lubData r' and its lub-paths at † and the computation of $min(r')$.

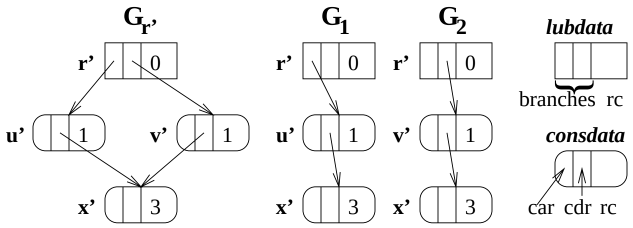
Example 2. This example demonstrates the computation of the *min* value of a lubData that has lubData descendants.

$$\begin{array}{l}
\textbf{let } x = cons(1, nil) \textbf{ in} \\
\quad \textbf{let } r = (\textbf{let } r1 = (\textbf{if } unknown \textbf{ then } cons(2, nil) \textbf{ else } x) \textbf{ in} \\
\qquad\qquad\quad \textbf{let } r2 = (\textbf{if } unknown \textbf{ then } cons(3, nil) \textbf{ else } x) \textbf{ in} \\
\qquad\qquad\quad\;\; \textbf{if } unknown \textbf{ then } r1 \textbf{ else } r2)^{\dagger} \textbf{ in} \\
\quad\;\; \ldots
\end{array}$$

Let r' be the result of the last conditional expression to be evaluated. The elements of $branches(r')$ become contained-in r' at †. Figure 7 shows lubData r' and its lub-paths at † and the computation of $min(r')$.

We now deal with computing the *min* value of a lubData l when a descendant v of l that is not in $branches(l)$ becomes contained-in l. This happens immediately after a decrement to $rc(v)$. We next discuss how such an ancestor l of v may be found since there is no direct way of doing so.

The following discussion explains how v may become contained-in a lubData l for the first time. We will see later that once v becomes contained-in l, if $min(l)$ is computed and v's contribution to $min(l)$ considered, then we need not be concerned about v's contribution to $min(l)$ again; so, we deal with only the first time that v becomes contained-in l and not any subsequent times. v's rc is decremented when



At †, before $min(r')$ is computed,
x, u and v are bound to x', u' and v', respectively.
r is not bound yet.
$x' = \langle cons(1, nil),\ rc = 3,\ lubParents = nil\rangle$
(references are from u', v' and the binding for x)
$u' = \langle cons(2, x'),\ rc = 1,\ lubParents = [r']\rangle$
$v' = \langle cons(3, x'),\ rc = 1,\ lubParents = [r']\rangle$
$r' = \langle branches = [u', v'],\ rc = 0,\ lubParents = nil,\ min = V_0\rangle$
$live = \langle 3\rangle$

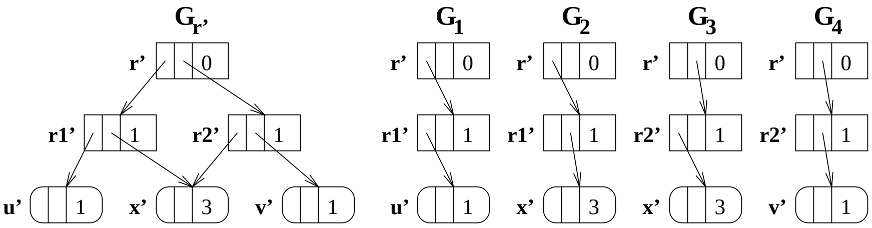
Computing $min(r')$ at †,
$C_{r'} = \{r', u', v'\}$
$size(G_1) = size(G_2) = \langle 1\rangle$
$min(r') = total(r') - sub(r') - maxLubPath(r')$
$= \langle 2\rangle - \langle 0\rangle - \langle 1\rangle$
$= \langle 1\rangle$
$live = \langle 3\rangle - \langle 1\rangle = \langle 2\rangle$

Figure 6: Computing $min(r')$ at †. $G_{r'}$ is the graph of nodes and edges reachable from r'; G_1 and G_2 are lub-paths of r'. lubData are represented by rectangles and constrData by rounded rectangles. Some fields of both types of data have been omitted from the pictorial representations since they are not relevant to the discussion.

1. a variable or parameter referencing v goes out of scope.
2. a data construction u containing references to v becomes garbage in one of the following ways:
   (a) the last variable or parameter referencing u goes out of scope.
   (b) after the evaluation of an expression $q(e)$ where q is a selector, tester, or equality predicate and e is an expression that evaluates to data y such that u is contained-in y and y is new data (i.e., data with reference count 0) and q (if it is a selector) does not select u.
   (c) the last data that contains a reference to u becomes garbage; this can happen only if there exists some ancestor x of u which becomes garbage through (a) or (b).

In case (1), v may become contained-in some lubData that is part of the result of the **let** expression or function call whose bound variable or parameter, respectively, is going out of scope. We recompute the *min* values of such lubData. This scheme also handles case (2a) and instances of case (2c) in which x becomes garbage through (2a). In case (2b), v may become contained-in the part of y that is selected by q. *recomputeMin* should be called on the results of such applications of selectors to new data. The current implementation does not make this invocation. Since none of our examples have instances of such applications, there is no loss of accuracy in these examples.

recomputeMin is called at the end of **let** expressions and function calls on the result of the expression. *recomputeMin*(v), where v is a constrData, calls *recomputeMin* recursively on the fields of v. For a lubData l, *recomputeMin*(l) should call *computeMin*(l) if *branches*(l) has two elements, both of which are contained-in l. Instead, it conservatively calls *computeMin*(l) if *branches*(l) has two elements with rc's equal to 1. If the check fails, then *recomputeMin* is called recursively on the elements of *branches*(l). In the first case, if *computeMin*(l) is greater than *min*(l), then *min*(l) and *live* are updated appropriately. *computeMin*(l) is less than *min*(l) iff there exists a v, such that (a) v was contained-in l when *min*(l) was last computed but is not currently contained-in l and (b) the size of v contributes to the last value of *min*(l). If u is contained-in a lubData l', it is possible to get a reference to u iff u occurs at the same position in all lub-paths of l', i.e., iff there exists a sequence of selectors $q_1, \ldots, q_n$ such that applying the sequence of selectors to any lub-path of l' yields u. For example, in (7), $cdr(r)$ returns a reference to x. (b) implies that v does not



At †, before $min(r')$ is computed,
x, $r1$ and $r2$ are bound to x', $r1'$ and $r2'$, respectively.
r is not bound yet.
$x' = \langle cons(1, nil),\ rc = 3,\ lubParents = [r1', r2'] \rangle$
$u' = \langle cons(2, nil),\ rc = 1,\ lubParents = [r1'] \rangle$
$v' = \langle cons(3, nil),\ rc = 1,\ lubParents = [r2'] \rangle$
$r1' = \langle branches = [u', x'],\ rc = 1,\ lubParents = [r'],\ min = V_0 \rangle$
$r2' = \langle branches = [v', x'],\ rc = 1,\ lubParents = [r'],\ min = V_0 \rangle$
(the min values of $r1'$ and $r2'$ are V_0 since one branch of both lubData viz. x', is not contained-in either.)
$r' = \langle branches = [r1', r2'],\ rc = 0,\ lubParents = nil,\ min = V_0 \rangle$
$live = \langle 3 \rangle$

Computing $min(r')$ at †,
$C_{r'} = \{r', r1', r2', u', v'\}$
(x' is not in $C_{r'}$ because of the binding for x)
$size(G_1) = size(G_4) = \langle 1 \rangle$
$size(G_2) = size(G_3) = \langle 0 \rangle$
$min(r') = total(r') - sub(r') - maxLubPath(r')$
$= \langle 2 \rangle - \langle 0 \rangle - \langle 1 \rangle$
$= \langle 1 \rangle$
$live = \langle 3 \rangle - \langle 1 \rangle = \langle 2 \rangle$

Figure 7: Computing $min(r')$ at †. $G_{r'}$ is the graph of nodes and edges reachable from r'; G_1, G_2, G_3 and G_4 are lub-paths of r'.

occur in the lub-path whose size is $maxLubPath(l)$, so it is impossible to get a direct reference to v after v becomes contained-in l. Henceforth, every new reference to v is in the $branches(l')$, for some lubData l'; such lubData and references from them to v are created by applying selectors to l. These new references to v cause v to not be contained-in l. According to definition (4) of min, this affects $min(l)$. However, applying selectors to l does not immediately affect the value of $live$, since l stays live even after the application, and references that keep fragments of l alive have no direct impact on $live$ until l becomes garbage. Therefore, if $computeMin(l)$ is less than $min(l)$, $min(l)$ and $live$ are not updated. This also explains why we are only concerned with the first time that v becomes contained-in l. After this first time, $min(l)$ is not affected by v's becoming contained-in or not contained-in l.

Once the rc's of the elements of $branches(l)$ reduce to 1, no new references can be made to them since selectors applied to l can only select fields of these elements. Thus, for $b \in branches(l)$, if $rc(b)$ becomes 1, it remains 1 until b becomes garbage, which occurs when l becomes garbage.

As seen in Figure 10, $gc(x)$ might call $computeMin(l)$, where l is a lubparent of x. The result of $computeMin(l)$ is subtracted from $live$ and assigned to $min(l)$. The old value of $min(l)$ is not first added to $live$. This is explained as follows. In the call to $gc(x)$, the decrement to $rc(x)$ might cause $rc(x)$ to reduce to 1 and x to become contained-in in l for the first time. If so, since $rc(x)$ is greater than 0 until now and x is an element of $branches(l)$, the old value of $min(l)$ must be V_0. Ignoring such a value of $min(l)$ is harmless. It follows from the arguments in the previous paragraph that once $rc(x)$ becomes 1, any subsequent call to $gc(x)$ occurs because l has become garbage. In such a case, $gc(x)$ does not call $computeMin(l)$. Thus, a call to $computeMin(l)$ from $gc(x)$ occurs at most once: at the first point when both elements of $branches(l)$ have rc 1. $computeMin(l)$ is not called at all from $gc(x)$ if both branches of l have rc 1 when l is constructed.

Calls to $recomputeMin(l)$, if any, occur only after the call to $computeMin(l)$, if any, from $gc(x)$, where x is the element of $branches(l)$ whose rc is the last to reduce to 1. This is because $gc(x)$ is called before the call to $recomputeMin$ in transformed **let** expressions and function calls. Further, the call to $recomputeMin$ on the results of these expressions will not affect the min value of l unless $computeMin$ has already done so.

6.7 Improvements

The current transformation does not achieve complete accuracy. We describe some improvements that are aimed at obtaining more accuracy. Although none of these have been implemented yet, we get accurate results on all our examples.

computeMin(l) should be called only after checking if elements b_1 and b_2 of *branches*(l) are contained-in l. However, we simply check if $rc(b_1)$ and $rc(b_2)$ are 1. This check is conservative. A thorough check may be implemented as follows : modify the definition of *computeMin* so that after calling *totalSub* on its branches, it checks if $rc(b_1)$ and $rc(b_2)$ are 0. If so, then the computation is completed. If not, V_0 is returned.

recomputeMin should be called on results of selectors also. This is described in Section 6.6.

According to the current definition of *recomputeMin*, if *recomputeMin*(l) calls *computeMin*(l) for lubData l, then *recomputeMin* is not recursively called on the elements of *branches*(l). Hence, the *min* value of a descendant lubData l' of l is not recomputed. Any excess in *live* caused by l' which has not already been subtracted away by *min*(l') will be included in *min*(l); this follows from the definition of *min*. Thus, at this point, l and its descendants cause no looseness in *live*. However, if at some later point l becomes garbage while l' stays live, *min*(l) which includes the excess caused by l' is added to *live*. The recursive call to $gc(l')$ will not cause this excess in l' just added to *live* to be subtracted from *live* since l' is still live. At this point, *min*(l') should be recomputed and *live* updated. In more general terms, when garbage collecting lubData, we should recompute the *min* values of descendant lubData that do not become garbage.

We do not correlate lubData results of duplicate evaluations of $q(l)$ or lubData results of evaluations of $q_1(l)$ and $q_2(l)$; l is a lubData, q, q_1 and q_2 are selectors, $q_1 \neq q_2$. Each time $q(l)$ is called, a new lubData is returned. Suppose l_1 and l_2 are results of two such calls to $q(l)$. If l becomes garbage and l_1 and l_2 stay live, since the common elements of *branches*(l_1) and *branches*(l_2) are not contained-in either l_1 or l_2, the sizes of both elements are included in *live* even though, in an execution of the original program, only one element is live. A similar situation arises in the case of the results of $q_1(l)$ and $q_2(l)$. This is roughly analogous to the false path problem.

Example. Consider the expression

$$\begin{array}{l}(\textbf{let } r = \textbf{if } \mathit{unknown} \\ \qquad\qquad\quad \textbf{then } \mathit{cons}(1, \mathit{cons}(2, \mathit{nil})) \\ \qquad\qquad\quad \textbf{else } \mathit{cons}(3, \mathit{cons}(4, \mathit{nil})) \textbf{ in} \\ \quad \mathit{cons}(\mathit{cdr}(r), \mathit{cdr}(r)))^{\dagger}\end{array}$$

The two calls to $cdr(r)$ return two different lubData l_1 and l_2. *branches*(l_1) and *branches*(l_2) contain references to $cons(2, nil)$ and $cons(4, nil)$. At †, r becomes garbage. Now, the *min* values of l_1 and l_2 are recomputed. $cons(2, nil)$ and $cons(4, nil)$ are not contained-in l_1 or l_2 because of the references from the other. Their *min* values are set to V_0 even though only one of $cons(2, nil)$ and $cons(4, nil)$ is live.

The problem may be partially solved by returning a unique lubData when selecting from some other lubData. The first time a selector is applied to lubData l, a reference to the resulting lubData l' is saved in l. Subsequent applications of the same selector to l return l'.

6.8 Garbage Collection

gc works on constrData the same way as before. *gc* also handles lubData. When a lubData l becomes garbage, before *gc* calls itself recursively on the elements of *branches*(l), it adds *min*(l) to *live*, so that *live* includes the sizes of all descendants of l when those descendants are garbage collected through recursive calls to *gc*. These recursive calls will remove from *live* the sizes of only those descendants that do become garbage. Without the last two improvements mentioned in Section 6.7, the total amount subtracted from *live* by these recursive calls may not be as large as *min*(l).

6.9 Component-Wise Analysis

The above analyses require that P_C be known during the analysis. Variants of these analyses, which we call *component-wise analyses*, can be performed without knowledge of P_C, simply by using component-wise

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

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

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

$$\mathcal{L}_b\,[\![c(e_1,\ldots,e_n)]\!] = \text{same as } \mathcal{L}\,[\![c(e_1,\ldots,e_n)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b$$

$$\mathcal{L}_b\,[\![q(e1,\ldots,en)]\!] = q_u(\mathcal{L}_b\,[\![e1]\!],\ldots,\mathcal{L}_b\,[\![en]\!])$$

$$q_u(v_1,\ldots,v_n) = \textbf{if } v_1 = \mathit{unknown}\ \mathit{or}\ \cdots\ \mathit{or}\ v_n = \mathit{unknown}\ \textbf{then}\ \mathit{unknown}\ \textbf{else}\ q(v_1,\ldots,v_n)$$

$$\mathcal{L}_b\,[\![q'(e)]\!] = \text{same as } \mathcal{L}\,[\![q'(e)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b\text{, and replace } q' \text{ with } q'_u$$

$$\begin{array}{l}
q'_u(v) = \textbf{if } v = \mathit{unknown}\ \textbf{then}\ \mathit{unknown}\\
\qquad\qquad \textbf{else if } \mathit{lubData}(v)\\
\qquad\qquad\qquad \textbf{then if } \mathit{length}(\mathit{branches}(v)) = 1\\
\qquad\qquad\qquad\qquad \textbf{then } \mathit{lub}(\mathit{false}, q'_u(\mathit{first}(\mathit{branches}(v))), \mathit{nil}, \mathit{nil})\\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{lub}(q'_u(\mathit{first}(\mathit{branches}(v))), q'_u(\mathit{second}(\mathit{branches}(v))), \mathit{nil}, \mathit{nil})\\
\qquad\qquad\qquad \textbf{else } q'(\mathit{data}(v))
\end{array}$$

$$\mathcal{L}_b\,[\![c^{-i}(e)]\!] = \text{same as } \mathcal{L}\,[\![c^{-i}(e)]\!]\text{, except replace } \mathcal{L} \text{ with } \mathcal{L}_b\text{, and replace } c^{-i} \text{ with } c_u^{-i}$$

$$\begin{array}{l}
c_u^{-i}(v) = \textbf{if } v = \mathit{unknown}\ \textbf{then}\ \mathit{unknown}\\
\qquad\qquad \textbf{else if } \mathit{lubData}(v)\\
\qquad\qquad\qquad \textbf{then if } \mathit{length}(\mathit{branches}(v)) = 1\\
\qquad\qquad\qquad\qquad \textbf{then } c^{-i}(\mathit{false})\\
\qquad\qquad\qquad\qquad \textbf{else } \mathit{lub}(c_u^{-i}(\mathit{first}(\mathit{branches}(v))), c_u^{-i}(\mathit{second}(\mathit{branches}(v))), \mathit{nil}, \mathit{nil})\\
\qquad\qquad\qquad \textbf{else } c^{-i}(\mathit{data}(v))
\end{array}$$

$$\begin{array}{l}
\mathcal{L}_b\,[\![\textbf{if } e_1\ \textbf{then } e_2\ \textbf{else } e_3]\!] = \textbf{let } r = \mathcal{L}_b\,[\![e_1]\!]\ \textbf{in}\\
\qquad \textbf{if } r = \mathit{unknown}\\
\qquad \textbf{then let } l0 = \mathit{copy}(\mathit{live}),\ l1 = \mathit{copy}(\mathit{live})\ \textbf{in}\\
\qquad\qquad \textbf{let } r2 = \mathcal{L}_b\,[\![e_2]\!]\ \textbf{in}\\
\qquad\qquad\quad \textbf{let } \mathit{diffl}2 = \mathit{vectorSub}(\mathit{live}, l1)\ \textbf{in}\\
\qquad\qquad\qquad \mathit{live} = l0;\\
\qquad\qquad\qquad \textbf{let } r3 = \mathcal{L}_b\,[\![e_3]\!]\ \textbf{in}\\
\qquad\qquad\qquad\quad \textbf{let } \mathit{diffl}3 = \mathit{vectorSub}(\mathit{live}, l1)\ \textbf{in}\\
\qquad\qquad\qquad\qquad \mathit{live} = \mathit{vectorAdd}(l1, \mathit{diffl}2, \mathit{diffl}3);\ \mathit{lub}(r2, r3, \mathit{diffl}2, \mathit{diffl}3)\\
\qquad \textbf{else if } r\ \textbf{then } \mathcal{L}_b\,[\![e_2]\!]\ \textbf{else } \mathcal{L}_b\,[\![e_3]\!]
\end{array}$$

$$\begin{array}{l}
\mathcal{L}_b\,[\![\textbf{let } v = e_1\ \textbf{in } e_2]\!] = \textbf{let } v = \mathcal{L}_b\,[\![e_1]\!]\ \textbf{in}\\
\qquad \mathit{incrc}(v);\\
\qquad \textbf{let } r = \mathcal{L}_b\,[\![e_2]\!]\ \textbf{in}\\
\qquad\quad \mathit{incrc}(r); \mathit{gc}(v); \mathit{decrc}(r); \mathit{recomputeMin}(r); r
\end{array}$$

$$\begin{array}{l}
\mathcal{L}_b\,[\![f(e_1,\ldots,e_n)]\!] = \textbf{let } r_1 = \mathcal{L}_b\,[\![e_1]\!],\ldots,r_n = \mathcal{L}_b\,[\![e_n]\!]\ \textbf{in}\\
\qquad \mathit{incrc}(r_1);\ldots;\mathit{incrc}(r_n);\\
\qquad \textbf{let } r = f_{Lb}(r_1,\ldots,r_n)\ \textbf{in}\\
\qquad\quad \mathit{incrc}(r); \mathit{gc}(r_1);\ldots;\mathit{gc}(r_n); \mathit{decrc}(r); \mathit{recomputeMin}(r); r
\end{array}$$

Figure 8: Transformation that produces live heap space-bound functions f_{Lb}. q is a primitive operation other than a selector, tester, or equality predicate. q' is a tester. *copy* copies a vector. *vectorAdd* and *vectorSub* determine the component-wise sum and difference, respectively, of their vector arguments; the first argument is modified in place to hold the sum or difference and is returned as the result. As an optimization, in the transformation for **if** expressions, it is easy to avoid transforming e_2 and e_3 twice, by saving and re-using the results of those transformations.

$$
\begin{array}{l}
\mathit{computeMin}(v) = \mathbf{let}\ \langle \mathit{total}, \mathit{sub}, \mathit{maxBr}\rangle = \langle \mathit{newZeroVec}(), \mathit{newZeroVec}(), \mathit{newZeroVec}()\rangle\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \langle \mathit{total}, \mathit{sub}\rangle \mathrel{+}= \mathit{totalSub}(u); \\
\qquad\qquad\qquad \mathbf{for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \mathit{maxBr} = \max(\mathit{maxBr}, \mathit{maxLubPath}(u)); \\
\qquad\qquad\qquad \mathbf{for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \mathit{recIncrc}(v); \\
\qquad\qquad\qquad (\mathit{total} - \mathit{sub} - \mathit{maxBr}) \\
\mathit{totalSub}(v) = \mathbf{if}\ \mathit{primVal}(v)\ \mathbf{then}\ \langle \mathit{newZeroVec}(), \mathit{newZeroVec}()\rangle \\
\qquad\qquad \mathbf{else}\ \mathit{decrc}(v); \\
\qquad\qquad\quad \mathbf{if}\ \mathit{rc}(v) = 0 \\
\qquad\qquad\quad \mathbf{then\ let}\ \langle \mathit{total}, \mathit{sub}\rangle = \langle \mathit{newZeroVec}(), \mathit{newZeroVec}()\rangle\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{if}\ \mathit{lubData}(v) \\
\qquad\qquad\qquad\quad \mathbf{then\ for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \langle \mathit{total}, \mathit{sub}\rangle \mathrel{+}= \mathit{totalSub}(u); \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{min}(v) \neq \mathit{nil}\ \mathbf{then}\ \mathit{vectorAdd}(\mathit{sub},\ \mathit{min}(v)) \\
\qquad\qquad\qquad\quad \mathbf{else\ for}\ i = 1..\mathit{arity}(v)\ \langle \mathit{total}, \mathit{sub}\rangle \mathrel{+}= \mathit{totalSub}(c^{-i}(\mathit{data}(v))); \\
\qquad\qquad\qquad\qquad \mathit{total}[\mathit{consType}(v)]{+}{+} \\
\qquad\qquad\qquad\quad \langle \mathit{total}, \mathit{sub}\rangle \\
\qquad\qquad\quad \mathbf{else}\ \langle \mathit{newZeroVec}(), \mathit{newZeroVec}()\rangle \\
\mathit{maxLubPath}(v) = \mathbf{if}\ \mathit{primVal}(v)\ \mathbf{then}\ \mathit{newZeroVec}() \\
\qquad\qquad\qquad \mathbf{else\ if}\ \mathit{rc}(v) = 0 \\
\qquad\qquad\qquad\quad \mathbf{then\ let}\ \mathit{maxBr} = \mathit{newZeroVec}()\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{lubData}(v) \\
\qquad\qquad\qquad\qquad \mathbf{then\ for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \mathit{maxBr} = \max(\mathit{maxBr}, \mathit{maxLubPath}(u)) \\
\qquad\qquad\qquad\qquad \mathbf{else\ for}\ i = 1..\mathit{arity}(v)\ \mathit{vectorAdd}(\mathit{maxBr}, \mathit{maxLubPath}(c^{-i}(\mathit{data}(v)))); \\
\qquad\qquad\qquad\qquad\quad \mathit{maxBr}[\mathit{consType}(v)]{+}{+} \\
\qquad\qquad\qquad\qquad \mathit{maxBr} \\
\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{newZeroVec}() \\
\mathit{recIncrc}(v) = \mathbf{if}\ \mathit{not}(\mathit{primVal}(v))\ \mathbf{then} \\
\qquad\qquad \mathit{incrc}(v); \\
\qquad\qquad \mathbf{if}\ \mathit{rc}(v) = 1\ \mathbf{then\ if}\ \mathit{lubData}(v) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then\ for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \mathit{recIncrc}(u) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else\ for}\ i = 1..\mathit{arity}(v)\ \mathit{recIncrc}(c^{-i}(\mathit{data}(v))) \\
\mathit{recomputeMin}(v) = \mathbf{if}\ \mathit{not}(\mathit{primVal}(v))\ \mathbf{then} \\
\qquad\qquad\qquad \mathbf{if}\ \mathit{lubData}(v) \\
\qquad\qquad\qquad \mathbf{then\ if}\ \mathit{length}(\mathit{branches}(v)) = 2\ \mathit{and}\ \mathit{containedIn}_0(\mathit{branches}(v)) \\
\qquad\qquad\qquad\quad \mathbf{then\ let}\ \mathit{newmin} = \mathit{computeMin}(v)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ \mathit{newmin} > \mathit{min}(v) \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \mathit{vectorSub}(\mathit{vectorAdd}(\mathit{live}, \mathit{min}(v)), \mathit{newmin}); \mathit{setmin}(v, \mathit{newmin}) \\
\qquad\qquad\qquad\quad \mathbf{else\ for}\ u\ \mathbf{in}\ \mathit{branches}(v)\ \mathit{recomputeMin}(u) \\
\qquad\qquad\qquad \mathbf{else\ for}\ i = 1..\mathit{arity}(v)\ \mathit{recomputeMin}(c^{-i}(\mathit{data}(v)))
\end{array}
$$

Figure 9: Auxiliary functions used to compute the *min* value of lubData. $\mathit{newZeroVec}()$ returns a new constructor count vector in which all counts are 0. $\langle t, s\rangle \mathrel{+}= \langle t', s'\rangle$ is the same as $\mathit{vectorAdd}(t, t'); \mathit{vectorAdd}(s, s')$.

maximum when computing the maximum of constructor count vectors. For programs that use multiple constructors, component-wise analysis provides more information about the maximum number of live instances of each constructor; however, if different components of the constructor count vector achieve their maximum values in different branches of conditionals, it may provide looser bounds on the overall amount of live data. Our examples use only one constructor (namely, *cons*); for such programs, component-wise analysis is

```
lub(v1, v2, live1, live2) = if eq?(v1, v2) then v1
                            else if primVal(v1)
                                 then if primVal(v2) then unknown
                                      else let result = newlub([v2]) in
                                             incrc(v2); addToLubParents(v2, result); result
                                 else if primVal(v2)
                                      then let result = newlub([v1]) in
                                             incrc(v1); addToLubParents(v1, result); result
                                      else let result = newlub([v1, v2]) in
                                            incrc(v1); addToLubParents(v1, result);
                                            incrc(v2); addToLubParents(v2, result);
                                            if live1 ≠ nil and live2 ≠ nil
                                            then let m = min(live1, live2) in
                                                   vectorSub(live, m); setmin(result, m);
                                            result
gc(v) = if not(primVal(v))
        then decrc(v);
             if length(lubParents(v)) = 1
             then let u = first(lubParents(v)) in
                    if length(branches(u)) = 2 and
                       containedIn0(branches(u))
                    then let m = computeMin(u) in
                           vectorSub(live, m); setmin(u, m)
             if rc(v) ≤ 0
             then if lubData(v)
                  then if min(v) ≠ nil
                       then live = vectorAdd(live, min(v));
                       for u in branches(v)
                         remFromLubParents(u, v); gc(u)
                  else live[consType(v)]−−;
                      for i = 1..arity(v) gc(c^{-i}(data(v)))
containedIn0(ls) = if null(ls) then true
                   else if rc(car(ls)) = length(lubParents(car(ls))) = 1
                        then containedIn0(cdr(ls))
                        else false
```

Figure 10: Auxiliary functions *lub*, *gc* and $containedIn_0$.

equivalent to the original analysis.

7 Experiments

We have implemented the above analyses using Chez Scheme [3] for a subset of Scheme and measured the results for several standard list and tree processing programs. These programs do not contain *eq*?; we did not implement tagging of data allocated by *lub* in stack space and heap allocation analyses. Comparisons of results of space functions and space-bound functions of all three analyses show that bound functions produce accurate results for all these examples. The results of live heap space-bound functions are also consistent with the expected asymptotic space complexities of the functions. All bound functions are either asymptotically faster or several times faster than applying the corresponding space functions to all possible inputs. Non-termination is not a problem for any of these examples.

Figures 11, 12, 13 and 14 contain the results of stack space, component-wise stack space, heap-allocation and live heap space analysis on some example programs. We do not show the results of both space functions on worst-case inputs and corresponding bound functions on partially known inputs since the two sets of results are the same in all examples, for all analyses. List reversal is the standard linear-time version of reverse; reversal with 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 [43, 6] 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 *unknown*; those for longest common subsequence and string edit are two such lists of equal length n. The bound function for binary-tree insertion inserts *unknown* into a complete binary tree of known depth n with unknown elements; the results are the same for inserting *unknown* into a tree with known elements or inserting a known value into a tree with unknown elements, since the test which checks if the given item is to be inserted into the left or right subtree evaluates to *unknown* in both these cases. Binomial coefficient uses integer arguments, n and m, where $m < n$. It was observed, using the analysis, that for a given n, binomial coefficient uses maximum stack space on inputs n and $n-1$. We use known inputs n and $n-1$ for all bound functions of binomial coefficient.

reversal		reversal w/app.			insertion sort			selection sort				merge sort				
n	*rev*	n	*rev*	*app*	n	*sort*	*insert*	n	*sort*	*least*	*remove*	n	*sort*	*even*	*odd*	*merge*
10	11	10	11	0	10	11	0	10	11	0	0	1	1	0	0	0
20	21	20	21	0	20	21	0	20	21	0	0	2	1	2	1	0
50	51	50	51	0	50	51	0	50	51	0	0	5	1	3	3	0
100	101	100	101	0	100	101	0	100	101	0	0	10	1	6	5	0
200	201	200	201	0	200	201	0	200	201	0	0	15	1	8	8	0
300	301	300	301	0	300	301	0	300	301	0	0	20	1	11	10	0
500	501	500	501	0	500	501	0	500	501	0	0	25	1	13	13	0
1000	1001	1000	1001	0	1000	1001	0	1000	1001	0	0	30	1	16	15	0
1500	1501	1500	1501	0	1500	1501	0	1500	1501	0	0	40	1	21	20	0
2000	2001	2000	2001	0	2000	2001	0	2000	2001	0	0	50	1	26	25	0

binomial coefficient				longest common subsequence					string edit distance				
n	*b*	*bhat*	*bhat'*	n	*lcs*	*lcshat*	*lcshat'*	*strref*	n	*se*	*sehat*	*sehat'*	*strref*
10	1	1	10	10	1	1	10	10	10	1	1	10	10
20	1	1	20	20	1	1	20	20	20	1	1	20	20
50	1	1	50	50	1	1	50	50	50	1	1	50	50
100	1	1	100	100	1	1	100	100	100	1	1	100	100
200	1	1	200	150	1	1	150	150	150	1	1	150	150
300	1	1	300	200	1	1	200	200	200	1	1	200	200
500	1	1	500	250	1	1	250	250	250	1	1	250	250
1000	1	1	1000	300	1	1	300	300	300	1	1	300	300
1500	1	1	1500	400	1	1	400	400	400	1	1	400	400
2000	1	1	2000	500	1	1	500	500	500	1	1	500	500

Figure 11: Results of stack space functions on worst-case inputs. These are also the results of stack space-bound functions; the two are equal for all of these examples. n is the input size.

Recall that the result of stack space analysis is a frame count vector whose elements are counts of the stack frames of functions defined in the program being analyzed. In the tables in Figures 11 and 12, the names of the functions defined in each program appear in the second line. For all programs, the first function listed is the top-level function. In list reversal with append, *app* appends a given list to another given list. *insert* in insertion sort inserts an element into a sorted list, in the correct position. In selection sort, *least* returns the least element in a given list, and *remove* removes a specified element from a list. In merge sort, *odd*(ls) and *even*(ls) return lists containing the elements of list ls at odd and even positions, respectively. The *hat*

reversal		reversal w/app.			insertion sort			selection sort				merge sort				
n	*rev*	n	*rev*	*app*	n	*sort*	*insert*	n	*sort*	*least*	*remove*	n	*sort*	*even*	*odd*	*merge*
10	11	10	11	10	10	11	10	10	11	10	10	1	1	0	0	0
20	21	20	21	20	20	21	20	20	21	20	20	2	2	2	2	2
50	51	50	51	50	50	51	50	50	51	50	50	5	4	3	3	5
100	101	100	101	100	100	101	100	100	101	100	100	10	5	6	6	10
200	201	200	201	200	200	201	200	200	201	200	200	15	5	8	8	15
300	301	300	301	300	300	301	300	300	301	300	300	20	6	11	11	20
500	501	500	501	500	500	501	500	500	501	500	500	25	6	13	13	25
1000	1001	1000	1001	1000	1000	1001	1000	1000	1001	1000	1000	30	6	16	16	30
1500	1501	1500	1501	1500	1500	1501	1500	1500	1501	1500	1500	40	7	21	21	40
2000	2001	2000	2001	2000	2000	2001	2000	2000	2001	2000	2000	50	7	26	26	50

binomial coefficient				longest common subsequence					string edit distance				
n	*b*	*bhat*	*bhat′*	n	*lcs*	*lcshat*	*lcshat′*	*strref*	n	*se*	*sehat*	*sehat′*	*strref*
10	1	2	10	10	1	11	11	10	10	1	21	20	10
20	1	2	20	20	1	21	21	20	20	1	41	40	20
50	1	2	50	50	1	51	51	50	50	1	101	100	50
100	1	2	100	100	1	101	101	100	100	1	201	200	100
200	1	2	200	150	1	151	151	150	150	1	301	300	150
300	1	2	300	200	1	201	201	200	200	1	401	400	200
500	1	2	500	250	1	251	251	250	250	1	501	500	250
1000	1	2	1000	300	1	301	301	300	300	1	601	600	300
1500	1	2	1500	400	1	401	401	400	400	1	801	800	400
2000	1	2	2000	500	1	501	501	500	500	1	1001	1000	500

Figure 12: Results of component-wise stack space functions on worst-case inputs. These are also the results of component-wise stack space-bound functions; the two are equal for all of these examples. n is the input size.

and *hat′* functions in binomial coefficient, longest common subsequence and string edit are constructed using methods described in [35]; the top-level function simply calls the *hat* function and extracts the appropriate part of its return value; the recursively-defined *hat* function calls the *hat′* function to compute the return value incrementally in the recursive case; in other words, the *hat′* function exploits the results of smaller sub-computations. *strref* in longest common subsequence and string edit returns the element at a given position in a list; in both programs, strings are represented as lists.

For all examples, stack space-bound analysis is tight relative to stack space analysis; in other words, the stack space-bound function applied to an unknown input returns the same result as the stack space function applied to a known worst-case input. Similarly, for all of these examples, component-wise stack space-bound analysis is tight relative to component-wise stack space analysis. Intuitively, these results indicate that the behavior of conditionals in the presence of partially known values is being analyzed sufficiently accurately. Recall from Section 3 that component-wise stack space-bound analysis provides bounds on the number of nested calls to each function but may lead to loose bounds on the overall stack size. This looseness occurs in most of these examples, as one can see from Figures 11 and 12. For example, the result of stack space analysis applied to reversal with append on a list of size n is $\langle (n+1), 0\rangle$, while the result of the component-wise analysis is $\langle (n+1), n\rangle$; the first and second components of the tuples are numbers of calls to *rev* and *app*, respectively.

We measured the running times of the original programs and the stack analysis functions for two examples - insertion sort and merge sort. The running time of insertion sort is quadratic time in the size of the input. Its stack space and stack space-bound functions also run in quadratic time. Stack space functions have similar structures as the original programs, except for the following: test expressions in conditionals, bindings in **let** expressions and arguments of function calls are evaluated twice, once to obtain their value and again to determine the stack space usage. This might lead to asymptotic slowdown if the expressions involved take more than constant time. Since all such expressions in insertion sort take constant time, its stack space function is only a constant factor slower than insertion sort itself. The stack space-bound function explores all possible ways of inserting elements into intermediate sorted sublists that contain unknown elements.

Given such a list of size m, an element may be inserted into the list in $(m+1)$ different ways. The bound function creates the lub of these $(m+1)$ possible results in $O(m)$ time, the computation being similar to a worst-case scenario of inserting into a list of known elements in which a known element has to be inserted at the end of the list. Thus, the bound function is no worse than the worst case scenario for insertion sort.

Even though inserting into a list of size m with unknown elements leads to $(m+1)$ possible results, the use of *lub* ensures that the result is a single list with unknown elements. This prevents exponential blowups that arise when all possible results have to manipulated instead of a single lub value.

Merge sort and its stack space function take $O(n \log n)$ time. The stack space-bound function of merge sort has a larger time complexity than merge sort. This is because of steps during the computation of the bound function where two lists containing unknown elements are merged. Two sorted lists of size n and m may be merged in $(n+m)!/(n! \times m!)$ different ways. The bound function explores each of these different possibilities. Hence, its running time is at least exponential in the input size.

list reversal		reversal w/append		insertion sort		selection sort		merge sort		binomial coefficient		longest common subseq.		string edit	
n	heap	n	heap	n	heap	n	heap	n	heap	n	heap	n	heap	n	heap
10	10	10	55	10	55	10	55	1	0	10	29	10	121	10	341
20	20	20	210	20	210	20	210	2	3	20	59	20	441	20	1281
50	50	50	1275	50	1275	50	1275	5	20	50	149	50	2601	50	7701
100	100	100	5050	100	5050	100	5050	10	59	100	299	100	10201	100	30401
200	200	200	20100	200	20100	200	20100	15	104	200	599	150	22801	150	68101
300	300	300	45150	300	45150	300	45150	20	157	300	899	200	40401	200	120801
500	500	500	125250	500	125250	500	125250	25	212	500	1499	250	63001	250	188501
1000	1000	1000	500500	1000	500500	1000	500500	30	267	1000	2999	300	90601	300	271201
1500	1500	1500	1125750	1500	1125750	1500	1125750	40	393	1500	4499	400	160801	400	481601
2000	2000	2000	2001000	2000	2001000	2000	2001000	50	523	2000	5999	500	251001	500	752001

Figure 13: Results of heap allocation functions on the worst-case input. These are also the results of the heap allocation-bound functions; the two are equal for all of these examples. n is the input size.

Figure 13 contains results of heap allocation analysis on all examples. These examples use only one constructor (namely, *cons*), so component-wise heap allocation bound analysis provides no additional information about space usage. This applies to live heap space analysis also. Heap allocation does not include the space used by top-level arguments since they are not allocated by any of the functions in the program being analyzed. The running times of heap allocation and heap allocation bound functions are similar to those of the corresponding stack analysis functions.

Figure 14 shows results of live heap space analysis. These results include the space used by top-level arguments since these arguments are indeed live throughout the execution of the program. It is easy to

list reversal		reversal w/append		insertion sort		selection sort		merge sort		binary-tree insertion		longest common subseq.		string edit	
n	result	n	result	n	result	n	result	n	result	n	result	n	result	n	result
10	20	10	29	1	2	1	2	1	1	1	18	2	10	10	100
20	40	20	59	2	5	2	5	2	5	2	33	4	18	20	200
50	100	50	149	3	9	3	9	3	9	3	60	6	26	50	500
100	200	100	299	4	14	4	14	4	11	4	111	8	34	100	1000
200	400	200	599	5	20	5	20	5	16	5	210	10	42	200	2000
300	600	300	899	6	27	6	27	6	18	6	405	12	50	300	3000
500	1000	500	1499	7	35	7	35	8	23	7	792	14	58	500	5000
1000	2000	1000	2999	8	44	8	44	10	31	8	1563	16	66	1000	10000
1500	3000	1500	4499	9	54	9	54	12	36	9	3102	18	74	1500	15000
2000	4000	2000	5999	10	65	10	65	15	45	10	6177	20	82	2000	20000

Figure 14: Results of live heap space-bound functions on partially known inputs.

see from these results that list reversal, reversal with append, longest common subsequence, and string edit take linear space. Insertion sort and selection sort take quadratic space. Merge sort takes $O(n \log n)$ space. Binary-tree insertion takes $O(2^n + n)$ space, since we define live heap space to include the space taken by the arguments, and a complete binary tree of depth n takes $O(2^n)$ space.

We compared the running times of the original functions, the live heap space functions, and the live heap space-bound functions for two programs: insertion sort and string edit. For both of these examples, the live heap space functions typically ran a factor of 60 to 100 slower than the original programs. For string edit, the running times of the original, space and bound functions are quadratic in terms of the input size (cf. next paragraph). For insertion sort, the running time of the space-bound function appears to be $O(n^2 n!)$, i.e., the average case running time of the original function times the number of equivalence classes of inputs. Thus, the bound function does not appear to offer any asymptotic savings in running time compared to running the space function on all inputs. However, the bound function does achieve significant constant-factor savings : 50 to 1500 for inputs of size 5 to 10.

Given partially known inputs of the kind described before, bound functions of all examples other than those of reversal and string edit build lubData. Bound functions of reversal, when given lists of known size and *unknown* elements, do not contain any conditional expressions whose tests evaluate to *unknown*, so they do not create any lubData. The only conditional expression in the bound function of string edit, whose test evaluates to *unknown*, has branches that are primitive data. Such conditional expressions return *unknown*. Notice that since the branches of such conditional expressions are primitive data, evaluation of both branches increases the running time by only a constant factor. The running times of all three bound functions grow at the same rate as the corresponding space functions.

8 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., [53, 32, 11, 45, 51, 49, 12, 44, 33, 47, 34]. Some techniques for time analysis can be adapted for space analysis, for example, as we did for stack space and heap allocation analysis. Analysis of live heap space has an important difference from all these other analyses: 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., [10, 38, 54, 14, 33], much of which is at a lower language level, for compiler generated code, and much of which is for facilitating time analysis. Our analyses bound stack space and heap space and are completely at the source level; they can serve many more purposes in understanding and optimizing programs, as discussed at the beginning. Live heap space analysis is also a first step towards analyzing cache behavior in the presence of garbage collection.

Our analyses for stack space and heap allocation are closely related to the time-bound analysis of Liu and Gómez [34]. Heap allocation corresponds to the number of data constructions that their analysis counts. Stack space counts nested function calls, separately for different functions, and takes the maximum, rather than sum in their analysis, over subexpressions.

Persson's work on live memory analysis [40] has the same goal as our live heap space analysis. He presents his techniques in the context of an object-oriented language and discusses various problems involved, and he 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. Persson also mentions the use of size information for deriving bounds on loops for time analysis but does not explain how. The general method used in our analyses is able to compute various kinds of bounds based on input size only, without program annotations; it can do time analysis automatically as well, as shown by the work of Liu and Gómez [34].

A number of people have worked on static analysis for compile-time garbage collection [22, 21, 16, 29, 27, 19, 25, 50, 18, 52], and one of the approaches is static reference counting [22, 21]. For compile-time optimization, the reference count includes only pointers to memory cells that will be used in the rest of the execution [21]. This kind of reference count can not be used for analyzing live heap space, since a cell not to be used in the future may still be pointed to from the stack and thus would not yet be garbage collected by standard run-time garbage collectors. Thus, we had to develop a different reference counting method for the live space analysis.

Inoue and others [26] analyze functional programs to detect run-time garbage, by finding 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.

Our analysis combines static program analysis techniques with reference counting and is simple and powerful. Interestingly, Goyal and Paige [17] developed a method for avoiding unnecessary hidden copies by performing compile-time analysis and optimizations that facilitate dynamic reference counting; their method is also simple and powerful, compared to relying completely on dynamic reference counting or using only static analysis.

Several type systems [24, 23, 7] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [24, 7]. They require programmers to annotate their programs with cost functions as types. Furthermore, some programs must be rewritten to have feasible types [24, 23].

Chin and Khoo [4] propose a method for calculating sized types by inferring constraints on size and then simplifying the constraints using Omega [42]. Their analysis results do not correspond to stack space or heap space in general. Furthermore, 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. We believe that it can be extended to handle higher-order functions and side effects. The extension to higher-order functions should be similar to the recent extension of timing analysis by Gómez and Liu from first-order [34] to higher-order functions [15]. The extension to handle side effects in the presence of destructive update is more complicated, since using reference counting may give overly pessimistic bounds when cycles are present. Since we have also developed a shape-analysis based version for live heap space analysis, we think it can be combined with new techniques for shape analysis in the presence of destructive update [48] to address this challenge.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[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] Cadence Research Systems. *Chez Scheme System Manual.* Cadence Research Systems, Bloomington, Indiana, revision 2.4 edition, July 1994.

[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] J. Cohen. Computer-assisted microanalysis of programs. *Commun. ACM*, 25(10):724–733, Oct. 1982.

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

[7] 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.

[8] R. K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

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

[10] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

[11] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 201–212, Rome, Italy, July 1989. Springer-Verlag, Berlin.

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

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

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

[15] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. Technical Report TR 535, Computer Science Department, Indiana University, Nov. 1999.

[16] K. Gopinath and J. L. Hennessy. Copy elimination in functional languages. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 303–314. ACM, New York, Jan. 1989.

[17] D. Goyal and R. Paige. A new solution to the hidden copy problem. In G. Levi, editor, *Proceedings of the 5th International Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 327–348, Pisa, Italy, Sept. 1998. Springer-Verlag, Berlin.

[18] G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In H. G. Baker, editor, *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Sept. 1995.

[19] G. W. Hamilton and S. B. Jones. Compile-time garbage collection by necessity analysis. In S. Peyton Jones, G. A. Hutton, and C. K. Holst, editors, *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, BCS Workshops in Computing Series, pages 66–70. Springer-Verlag, Berlin, Aug. 1991.

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

[21] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis Horwood Series in Computers and Their Applications, pages 45–62. Ellis Horwood, Chichester; Halsted Press, New York, 1987.

[22] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, New York, Jan. 1985.

[23] 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.

[24] 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.

[25] S. P. Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, Aug. 1992.

[26] 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.

[27] T. P. Jensen and T. Mogensen. A backwards analysis for compile-time garbage collection. In ESOP 1990 [9], pages 227–239.

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

[29] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [13], pages 54–74.

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

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

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

[33] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, July 1995.

[34] 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.

[35] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, Berlin, Mar. 1999.

[36] 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.

[37] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[38] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

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

[40] P. Persson. Live memory analysis for garbage collection in embedded systems. In LCTES 1999 [31], pages 45–54.

[41] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. ACM, New York, Aug. 1984.

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

[43] P. W. Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

[44] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 65–78. ACM, New York, June 1994.

[45] M. Rosendahl. Automatic complexity analysis. In FPCA 1989 [13], pages 144–156.

[46] I. Ryu. Issues and challenges in developing embedded software for information appliances and telecommunication terminals. In LCTES 1999 [31], pages 104–120. Invited talk.

[47] R. H. Saavedra and A. J. Smith. Analysis of benchmark characterization and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, Nov. 1996.

[48] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, Jan. 1998.

[49] D. Sands. Complexity analysis for a lazy higher-order language. In ESOP 1990 [9], pages 361–376.

[50] A. V. S. Sastry, W. Clinger, and Z. Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture*, pages 266–275. ACM, New York, June 1993.

[51] A. Shaw. Reasoning about time in higher level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, July 1989.

[52] M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 184–193. IEEE CS Press, Los Alamitos, Calif., May 1998.

[53] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–538, Sept. 1975.

[54] 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.

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