Automatic Time-Bound Analysis for a Higher-Order Language

Gustavo Gómez* and Yanhong A. Liu*

November 1999

1 Introduction

Analysis of program running time is important for reactive systems, interactive environments, compiler optimizations, performance evaluation, and many other computer applications. It has been extensively studied in many fields of computer science: algorithms [21, 12, 13, 40], programming languages [38, 22, 31, 35, 34], and systems [36, 29, 33, 32]. Being able to predict accurate time bounds automatically and efficiently is particularly important for many applications, such as reactive systems. It is also particularly desirable to be able to do so for high-level languages [36, 29].

Since Shaw proposed timing schema for analyzing system running time based on high-level languages [36], a number of people have extended it for analysis in the presence of compiler optimizations [29, 9], pipelining [16, 23], cache memory [3, 23, 11], etc. However, there is still a serious limitation of the timing schema, even in the absence of low-level complications. This is the inability to provide loop bounds, recursion depths, or execution paths automatically and accurately for the analysis [28, 2]. For example, the inaccurate loop bounds cause the calculated worst-case time to be as much as 67% higher than the measured worst-case time in [29], while the manual way of providing such information is potentially an even larger source of error, in addition to being inconvenient [28]. Various program analysis methods have been proposed to provide loop bounds or execution paths [2, 10, 15, 17]. However, they apply only to some classes of programs or use approximations that are too crude for the analysis. Also, separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [27].

Liu and Gómez [24] studied a general approach for automatic and accurate time-bound analysis that combines methods and techniques studied in theory, languages, and systems. They call it a *language-based* approach since it primarily exploits methods and techniques for static program analysis and transformation. However, the particular analysis there handles only first-order functions. Being able to handle higher-order functions is important for analyzing most functional languages and for analyzing methods in object-oriented languages with inheritance.

This paper extends the language-based approach to a higher-order language. As before [24], the approach consists of transformations for building time-bound functions in the presence of partially known input structures, symbolic evaluation of the time-bound function based on input parameters, optimizations to make the analysis efficient as well as accurate, and measurements of primitive parameters, all at the source-language level. To handle higher-order functions, special transformations are needed to build lambda expressions for computing running times, to optimize the construction of the time lambda expressions, and to optimize the symbolic evaluation. We describe analysis and transformation algorithms and explain how they work. We have implemented this approach and performed a large number of experiments analyzing Scheme programs. The measured worst-case times are closely bounded by the calculated bounds. We describe our prototype system, ALPA, as well as the analysis and measurement results.

*This work is supported in part by ONR under grant N00014-99-1-0132 and NSF under grant CCR-9711253. Authors' address: Computer Science Department, 215 Lindley Hall, Indiana University, Bloomington, IN 47405. Email: {ggomezes,liu}@cs.indiana.edu. Phone: (812)855-{9761,4373}. Fax: (812)855-4829.

2 Language-based approach

Language-based time-bound analysis starts with a given program written in a high-level language, such as Java, ML, or Scheme. The first step is to build a *time function* that (takes the same input as the original program but) returns the running time in place of (or in addition to) the original return value. This is done by associating a parameter with each program construct representing its running time and by summing these parameters based on the semantics of the constructs [38, 5, 36]. This transformation is straightforward for all constructs except lambda abstraction, i.e., first-class function, where additional transformations are needed to build lambda expressions for computing running times, as proposed by Sands [35, 34]. We call parameters that describe the running times of program constructs *primitive parameters*. To calculate actual time bounds based on the time function, three difficult problems must be solved.

First, since the goal is to calculate running time without being given particular inputs, the calculation must be based on certain assumptions about inputs. Thus, the first problem is to characterize the input data and reflect them in the time function. In general, due to imperfect knowledge about the input, the time function is transformed into a *time-bound function*.

In algorithm analysis, inputs are characterized by their size; accommodating this requires manual or semi-automatic transformation of the time function [38, 22, 40]. The analysis is mainly asymptotic, and primitive parameters are considered independent of the input size, i.e., are constants while the computation iterates or recurses. Whatever values of the primitive parameters are assumed, a second problem arises, and it is theoretically challenging: optimizing the time-bound function to a closed form in terms of the input size [38, 5, 22, 31, 13]. 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.

In systems, inputs are characterized indirectly using loop bounds or execution paths in programs, and such information must in general be provided by the user [36, 29, 28, 23], even though program analyses can help in some cases [2, 10, 15, 17]. Closed forms in terms of parameters for these bounds can be obtained easily from the time function. This isolates the third problem, which is most interesting to systems research: obtaining values of primitive parameters for various compilers, run-time systems, operating systems, and machine hardwares. In recent years, much progress has been made in analyzing low-level dynamic factors, such as clock interrupt, memory refresh, cache usage, instruction scheduling, and parallel architectures [29, 3, 23, 11]. Nevertheless, the inability to compute loop bounds or execution paths automatically and accurately has led calculated bounds to be much higher than measured worst-case time.

In programming-language area, Rosendahl proposed using *partially known input structures* [31]. For example, instead of replacing an input list l with its length n, as done in algorithm analysis, or annotating loops with numbers related to n, as done in systems, we use as input a list of n unknown elements. We call parameters for describing partially known input structures *input parameters*. The time function is then transformed automatically into a time-bound function: at control points where decisions depend on unknown values, the maximum time of all possible branches is computed; otherwise, the time of the chosen branch is computed. Rosendahl concentrated on proving the correctness of this transformation. He assumed constant 1 for primitive parameters and relied on optimizations to obtain closed forms in terms of input parameters, but again closed forms can not be obtained for all time-bound functions. Also, Rosendahl handles only first-order functions. Sands studied time functions for higher-order functions [35, 34], but he did not address any of the three problems described above. In addition, his analysis is presented only for named functions, not general lambda abstractions.

Combining results from theory to systems, and exploring methods and techniques for static program analysis and transformation, we have developed a general approach for computing time bounds automatically, efficiently, and more accurately. The approach has four main components.

First, we use an automatic transformation to construct a time-bound function from the original program based on partially known input structures. The resulting function takes input parameters and primitive parameters as arguments. The only caveat here is that the time-bound function may not terminate. However, nontermination occurs only if the recursive/iterative structure of the original program depends on unknown parts in the given partially known input structures.

Then, to compute worst-case time bounds efficiently without relying on closed forms, we optimize the time-bound function symbolically with respect to given values of input parameters. This is based on partial evaluation and incremental computation. This symbolic evaluation always terminates provided that the time-bound function terminates. The resulting function can be used repeatedly to compute time bounds efficiently for different primitive parameters measured for different underlying systems.

A third component consists of transformations that enable more accurate time bounds to be computed: lifting conditions, simplifying conditionals, and inlining non-recursive functions. These transformations should be applied on the original program before the time-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 time bounds, which can be computed more efficiently as well.

Finally, we measure primitive parameters at the source-language level and use the best conservative estimations in computing the time bound. We have implemented these transformations and the measurement procedures for a higher-order functional subset of Scheme. All the transformations and measurements are done automatically, and the time bound is computed efficiently and accurately. Examples analyzed include various list processing and numerical programs.

The approach is general because all four components we developed are based on general methods and techniques. Each particular component requires relative small improvements or modifications to existing analyses or transformations, but the combination of them for the application of automatic and accurate time-bound analysis for high-level languages is powerful. We used a higher-order functional subset of Scheme [1, 7] for three reasons.

1) Functional programming languages, together with features like automatic garbage collection, have become increasingly widely used, but methods for calculating actual running time of functional programs have been lacking.
2) Much work has been done on analyzing and transforming functional programs, including complexity analysis, and it can be exploited for analyzing actual running times efficiently and accurately as well.
3) Analyses and transformations developed for functional languages can be applied to improve analyses of imperative languages as well [39].

All our analyses and transformations are performed at source level. This allows implementations to be independent of compilers and underlying systems. It also allows analysis results to be understood at source level.

Language. We use a high-order, call-by-value functional language that has structured data, primitive arithmetic, boolean, and comparison operations, conditionals, bindings, first-class functions, and mutually recursive function calls. A program is a set of mutually recursive definitions. Its syntax is given by the grammar below:

$$\begin{array}{lll@{\quad}l}
\mathit{program} & ::= & v_1 \triangleq\ e_1, ..., v_n \triangleq\ e_n & \\
e & ::= & v & \text{variable reference} \\
 & | & c(e_1, ..., e_n) & \text{data construction} \\
 & | & p(e_1, ..., e_n) & \text{primitive operation} \\
 & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 & \text{conditional expression} \\
 & | & \textbf{let}\ v = e_1\ \textbf{in}\ e_2 & \text{binding expression} \\
 & | & \textbf{letrec}\ v = e_1\ \textbf{in}\ e_2 & \text{recursive binding expression} \\
 & | & \textbf{lambda}\ (v_1, ..., v_n)\, e_0 & \text{first-class function} \\
 & | & e_0(e_1, ..., e_n) & \text{function application}
\end{array}$$

Constants are constructors of arity 0; for convenience, we write c instead of $c()$ for them. We use constructor *nil* to denote an empty list, with operator *null*? as the corresponding tester, and we use constructor *cons* to build a list from a head element and a tail list, with operators *car* and *cdr* as the corresponding selectors. For binary operations, we chose between infix and prefix notations depending on whichever is easier for the presentation. For simplicity of the presentation, we restrict the discussion to single-variable bindings, but the implementation handles multiple-variable bindings. For ease of analysis and transformation, we assume that a preprocessor gives a distinct name to each bound variable.

Figure 1 gives an example program with definitions *index* and *index-cps*. Function *index* takes an item and a list and returns the zero-based index of the item in the list, or -1 if the item is not in the list. It calls function *index-cps*, which uses continuation-passing style (CPS) to avoid unnecessary additions if the item is not in the list. We use this program as a small running example. To present various analysis results, we also use several other examples as described in Section 5.

```
index ≜ lambda (item, ls) index-cps(item, ls, lambda (x) x),
index-cps ≜ lambda (item, ls, k)
                if null?(ls) then −1
                else if item = car(ls) then k(0)
                     else index-cps(item, cdr(ls), lambda (v) k(v + 1))
```

Figure 1: Example program with definitions *index* and *index-cps*.

Even though this language is small, it is sufficiently powerful and convenient for writing sophisticated programs. Structured data is essentially records in Pascal, structs in C, and constructor applications in ML. Conditionals and bindings easily simulate conditional statements and assignments, and recursions subsume loops.

The absence of arrays and pointers in the language does not detract the generality of the method, since time analysis with them is not fundamentally harder. The running times of the program constructs for them can be analyzed in the same way as times of other constructs. For example, accessing an array element $a[i]$ takes the time of accessing i, offsetting the element address from that of a, and finally getting the value from that address. Note that side effects caused by these features often cause other analyses to be more difficult.

For pure functional languages, lazy evaluation is important. Time functions that accommodate it have been studied. The symbolic evaluation and optimizations we describe apply to it as well.

3 Constructing time-bound function

Constructing time functions. We first transform the original program to construct a time function, which takes the original input and primitive parameters as arguments and returns the running time. This can be done based on the semantics of each program construct. It is straightforward for all constructs except first-class functions, i.e., lambda expressions.

For example, a variable reference is transformed into a symbol T_{varref} representing the running time of a variable reference; a conditional statement is transformed into the time of the test plus, if the condition is true, the time of the true branch, otherwise, the time of the false branch, and plus the time for the transfers of control. We introduce a new prefix operator *add* to add two or more time expressions.

To handle lambda expressions, it is necessary to introduce new lambda expressions for computing the running times. A lambda expression evaluates to a closure, where the body of the lambda is evaluated only when the function represented by the closure is actually applied. Thus, the time for evaluating the body of a lambda can also only be computed when the function is actually applied and, therefore, we need to build a new lambda expression for computing the running time. The body of the time lambda expression will be based on the body of the original lambda expression, and the time lambda expression will be evaluated to a time closure. We introduce a special data constructor *lambda-pair* to build a pair of an original lambda expression and its time lambda expression, and we use *value* and *time* as the corresponding selectors.

The *time transformation* $\mathcal{T}$ embodies the overall algorithm and is given below. It takes an original program, builds lambda pairs for lambda expressions in each definition e_i using transformation $\mathcal{T}_v$, where subscript v is mnemonic for value, and builds the time component of each lambda pair based on the value component of the pair using transformation $\mathcal{T}_t$, where subscript t is mnemonic for time. To avoid clutter, we reuse identifiers $v_1, ..., v_n$ in the transformed program; this does cause any problem since the old meanings of theses identifiers are not used in the transformed program.

$$\mathcal{T}\left[\!\!\left[\begin{array}{l} v_1 \triangleq\ e_1, \\ ..., \\ v_n \triangleq\ e_n \end{array}\right]\!\!\right] \quad = \quad \begin{array}{l} v_1 \triangleq\ \mathcal{T}_v\,[\![e_1]\!], \\ ..., \\ v_n \triangleq\ \mathcal{T}_v\,[\![e_n]\!] \end{array}$$

$$\begin{array}{lll}
v_1: & \mathcal{T}_v\,[\![v]\!] & = v \\
v_2: & \mathcal{T}_v\,[\![c(e_1,...,e_n)]\!] & = c(\mathcal{T}_v\,[\![e_1]\!],...,\mathcal{T}_v\,[\![e_n]\!]) \\
v_3: & \mathcal{T}_v\,[\![p(e_1,...,e_n)]\!] & = p(\mathcal{T}_v\,[\![e_1]\!],...,\mathcal{T}_v\,[\![e_n]\!]) \\
v_4: & \mathcal{T}_v\,[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & = \textbf{if}\ \mathcal{T}_v\,[\![e_1]\!]\ \textbf{then}\ \mathcal{T}_v\,[\![e_2]\!]\ \textbf{else}\ \mathcal{T}_v\,[\![e_3]\!] \\
v_5: & \mathcal{T}_v\,[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{let}\ v = \mathcal{T}_v\,[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_v\,[\![e_2]\!] \\
v_6: & \mathcal{T}_v\,[\![\textbf{letrec}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{letrec}\ v = \mathcal{T}_v\,[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_v\,[\![e_2]\!] \\
v_7: & \mathcal{T}_v\,[\![\textbf{lambda}\ (v_1,...,v_n)\ e_0]\!] & = \textit{lambda-pair}(\textbf{lambda}\ (v_1,...,v_n)\ \mathcal{T}_v\,[\![e_0]\!], \\
 & & \qquad\qquad\qquad \textbf{lambda}\ (v_1,...,v_n)\ \mathcal{T}_t\,[\![\mathcal{T}_v\,[\![e_0]\!]]\!]) \\
v_8: & \mathcal{T}_v\,[\![e_0(e_1,...,e_n)]\!] & = \textit{value}(\mathcal{T}_v\,[\![e_0]\!])\,(\mathcal{T}_v\,[\![e_1]\!],...,\mathcal{T}_v\,[\![e_n]\!]) \\
 & & \\
t_1: & \mathcal{T}_t\,[\![v]\!] & = T_{varref} \\
t_2: & \mathcal{T}_t\,[\![c(e_1,...,e_n)]\!] & = \textit{add}(T_c, \mathcal{T}_t\,[\![e_1]\!],...,\mathcal{T}_t\,[\![e_n]\!]) \\
t_3: & \mathcal{T}_t\,[\![p(e_1,...,e_n)]\!] & = \textit{add}(T_p, \mathcal{T}_t\,[\![e_1]\!],...,\mathcal{T}_t\,[\![e_n]\!]) \\
t_4: & \mathcal{T}_t\,[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & = \textbf{if}\ e_1\ \textbf{then}\ \textit{add}(T_{if}, \mathcal{T}_t\,[\![e_1]\!], \mathcal{T}_t\,[\![e_2]\!]) \\
 & & \qquad\quad \textbf{else}\ \textit{add}(T_{if}, \mathcal{T}_t\,[\![e_1]\!], \mathcal{T}_t\,[\![e_3]\!]) \\
t_5: & \mathcal{T}_t\,[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{let}\ v = e_1\ \textbf{in}\ \textit{add}(T_{let}, \mathcal{T}_t\,[\![e_1]\!], \mathcal{T}_t\,[\![e_2]\!]) \\
t_6: & \mathcal{T}_t\,[\![\textbf{letrec}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{letrec}\ v = e_1\ \textbf{in}\ \textit{add}(T_{letrec}, \mathcal{T}_t\,[\![e_1]\!], \mathcal{T}_t\,[\![e_2]\!]) \\
t_7: & \mathcal{T}_t\,[\![\textit{lambda-pair}(e_1,e_2)]\!] & = T_{lambda} \\
t_8: & \mathcal{T}_t\,[\![\textit{value}(e_0)(e_1,...,e_n)]\!] & = \textit{add}(T_{funcall}, \mathcal{T}_t\,[\![e_0]\!], \mathcal{T}_t\,[\![e_1]\!],...,\mathcal{T}_t\,[\![e_n]\!], \\
 & & \qquad\quad \textit{time}(e_0)\,(e_1,...,e_n))
\end{array}$$

Rules v_1 to v_6 handle expressions other than lambda expressions or function calls, so they transform subexpressions recursively. Rule v_7 takes a lambda expression and creates a lambda pair; the first component is the body transformed recursively by $\mathcal{T}_v$, and the second component is the time body transformed further by $\mathcal{T}_t$. To make the transformation run in linear time, the resulting expression of $\mathcal{T}_v\,[\![e_0]\!]$ is shared. Rule v_8 takes an application of function e_0 and transforms subexpressions recursively; since $\mathcal{T}_v\,[\![e_0]\!]$ evaluates to a lambda pair, its value component is selected and applied to the transformed arguments.

Rule t_1 transforms a variable reference to the time of a variable reference T_{varref}. Rule t_2 (respectively t_3) sums the times of evaluating the arguments and the time of the primitive (respectively constructor). Rule t_4 sums the times of the conditional transfer, of evaluating the condition, and of evaluating the true branch, if the condition is true; otherwise, it sums the times of the conditional transfer, of evaluating the condition, and of evaluating the false branch. Rules t_5 and t_6 include the bindings unchanged, because the transform body may refer to the bound variable; they sum the times of making a binding, of evaluating the expression for the bound variable, and of evaluating the body. Rule t_7 just returns the time of evaluating a lambda abstraction; there is no need to go into the body of the lambda, because this time does not depend on the body. Rule t_8 sums the times of making a function call, of evaluating e_0 and all its argument expressions, and of evaluating the function; the function is given by the time component of the lambda pair.

Transformation $\mathcal{T}$ as described above runs in linear time in terms of the size of the given program. Intuitively, each subexpression is transformed at most twice: once by $\mathcal{T}_v$ and once by $\mathcal{T}_t$. A formal proof is done by an induction on the number of subexpressions in the program, and the number of nestings of first-class functions.

Figure 2 shows the result of this transformation applied to function *index-cps*. Shared code is presented with *where* clauses when this makes the code smaller. For ease of presentation, we give all constants the same symbol $T_{constant}$ for their times.

This transformation is similar to the local cost assignment [38], step-counting function [31], cost function [35], etc. in other work. Our transformation extends those methods with bindings and general first-class functions. It also makes all primitive parameters explicit at the source-language level. For example, each primitive operation p is given a different symbol T_p, and each constructor c is given a different symbol T_c.

```
index-cps ≜ lambda-pair(lambda (item, ls, k)
                          if null?(ls) then −1
                          else if item = car(ls) then value(k)(0)
                               else value(index-cps)(item, cdr(ls), lambda_1),
                        lambda (item, ls, k)
                          if null?(ls) then add($T_{if}$, add($T_{null?}$, $T_{varref}$), $T_{constant}$)
                          else add($T_{if}$, add($T_{null?}$, $T_{varref}$),
                                   if item = car(ls)
                                   then add($T_{if}$, add($T_{=}$, $T_{varref}$, add($T_{car}$, $T_{varref}$)),
                                             add($T_{funcall}$, time(k)(0), $T_{varref}$, $T_{constant}$))
                                   else add($T_{if}$, add($T_{=}$, $T_{varref}$, add($T_{car}$ $T_{varref}$)),
                                            add($T_{funcall}$, $T_{varref}$, $T_{varref}$, $T_{lambda}$
                                                time(index-cps)(item, cdr(ls), lambda_1),
                                                add($T_{cdr}$, $T_{varref}$)))))
  where lambda_1 is
    lambda-pair(lambda (v) value(k)(v + 1),
                lambda (v) add($T_{funcall}$, time(k)(v + 1), $T_{varref}$,
                               add($T_{+}$, $T_{constant}$, $T_{varref}$)))
```

Figure 2: Function *index-cps* after transformation $\mathcal{T}$.

Note that the time function terminates with the appropriate sum of primitive parameters if the original program terminates, and it runs forever to sum to infinity if the original program does not terminate, which is the desired meaning of a time function.

Constructing time-bound functions. Characterizing program inputs in the time function is difficult to automate [38, 22, 36]. However, partially known input structures provide a natural mean [31]. A special constant *unknown* is used to represent unknown values. For example, to represent all input lists of length n, the following partially known input structure can be used.

```
list ≜ lambda (n)
         if $n = 0$ then nil
         else cons(unknown, list($n-1$))
```

Similar structures can be used to describe an array of n elements, a matrix of m-by-n elements, etc.

Since partially known input structures give incomplete knowledge about inputs, the original functions need to be transformed to handle the special value *unknown*. In particular, for each primitive function p, we define a new primitive function f_p such that $f_p(v_1, ..., v_n)$ returns *unknown* if any v_i is *unknown* and returns $p(v_1, ..., v_n)$ as usual otherwise. We also define a new *least upper bound* function *lub* that takes two values and returns the most precise partially known structure that both values conform with.

```
$f_p$ ≜ lambda ($v_1, ..., v_n$)
         if $v_1$ = unknown
           ∨ ... ∨
           $v_n$ = unknown
         then unknown
         else $p(v_1, ..., v_n)$
```

```
lub ≜ lambda ($v_1, v_2$)
        if $v_1$ is $c_1(x_1, ..., x_i)$ ∧
           $v_2$ is $c_2(y_1, ..., y_j)$ ∧
           $c_1 = c_2$ ∧ $i = j$
        then $c_1(lub(x_1, y_1), ..., lub(x_i, y_i))$
        else unknown
```

Also, the time functions need to be transformed to compute an upper bound of the running time. If the truth value of a conditional test is known, then the time of the chosen branch is computed, otherwise, the maximum of the times of both branches is computed.

Because functions are first-class objects, their values can also be *unknown*. If we try to apply an *unknown* function, the result is *unknown*, and the time is *infinite*, as shown below by definitions *value_apply* and *time_apply*. We could keep more precise information than *unknown*. This can be a set of possible function values. Then the upper bound of the times of applying all functions in the set can be taken. This is easy to implement, but it may be expensive to compute if it is indeed needed. An important fact is that in all examples we have tried, this is not needed, i.e., the naturally given partially known input contains enough information to decide all lambdas at analysis time.

$$value\_apply \triangleq \textbf{lambda}\ (v_0, v_1, ..., v_n)\ \textbf{if}\ v_0 = unknown\ \textbf{then}\ unknown\ \textbf{else}\ value(v_0)\,(v_1, ..., v_n)$$

$$time\_apply \triangleq \textbf{lambda}\ (v_0, v_1, ..., v_n)\ \textbf{if}\ v_0 = unknown\ \textbf{then}\ infinite\ \textbf{else}\ time(v_0)\,(v_1, ..., v_n)$$

The *time-bound transformation* $\mathcal{T}_b$ embodies the overall algorithm. It takes a program obtained from time transformation $\mathcal{T}$ and builds the corresponding time-bound version. It uses two transformations: $\mathcal{T}_{vb}$ and $\mathcal{T}_{tb}$. $\mathcal{T}_{vb}$ transforms an expression that computes the original value, and $\mathcal{T}_{tb}$ transforms an expression that computes the running time. Again, identifiers $v_1, ..., v_n$ are reused in the transformed program.

$$\mathcal{T}_b \left[\!\!\left[\begin{array}{l} v_1 \triangleq e_1, \\ ..., \\ v_n \triangleq e_n \end{array} \right]\!\!\right] = \begin{array}{l} v_1 \triangleq \mathcal{T}_{vb}[\![e_1]\!], \\ ..., \\ v_n \triangleq \mathcal{T}_{vb}[\![e_n]\!] \end{array}$$

$$\begin{array}{rll}
vb_1: & \mathcal{T}_{vb}[\![v]\!] & = v \\
vb_2: & \mathcal{T}_{vb}[\![c(e_1, ..., e_n)]\!] & = c(\mathcal{T}_{vb}[\![e_1]\!], ..., \mathcal{T}_{vb}[\![e_n]\!]) \\
vb_3: & \mathcal{T}_{vb}[\![p(e_1, ..., e_n)]\!] & = f_p(\mathcal{T}_{vb}[\![e_1]\!], ..., \mathcal{T}_{vb}[\![e_n]\!]) \\
vb_4: & \mathcal{T}_{vb}[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & = \textbf{let}\ v = \mathcal{T}_{vb}[\![e_1]\!] \\
& & \quad \textbf{in if}\ v = unknown \\
& & \qquad \textbf{then}\ lub(\mathcal{T}_{vb}[\![e_2]\!], \mathcal{T}_{vb}[\![e_3]\!]) \\
& & \qquad \textbf{else if}\ v\ \textbf{then}\ \mathcal{T}_{vb}[\![e_2]\!]\ \textbf{else}\ \mathcal{T}_{vb}[\![e_3]\!] \\
vb_5: & \mathcal{T}_{vb}[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{let}\ v = \mathcal{T}_{vb}[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_{vb}[\![e_2]\!] \\
vb_6: & \mathcal{T}_{vb}[\![\textbf{letrec}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{letrec}\ v = \mathcal{T}_{vb}[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_{vb}[\![e_2]\!] \\
vb_7: & \mathcal{T}_{vb}[\![lambda\text{-}pair(\textbf{lambda}\ (v_1, ..., v_n)\, e_1, & = lambda\text{-}pair(\textbf{lambda}\ (v_1, ..., v_n)\, \mathcal{T}_{vb}[\![e_1]\!], \\
& \qquad \textbf{lambda}\ (v_1, ..., v_n)\, e_2)]\!] & \qquad \textbf{lambda}\ (v_1, ..., v_n)\, \mathcal{T}_{tb}[\![e_2]\!]) \\
vb_8: & \mathcal{T}_{vb}[\![value(e_0)\,(e_1, ..., e_n)]\!] & = value\_apply(\mathcal{T}_{vb}[\![e_0]\!], \mathcal{T}_{vb}[\![e_1]\!], ..., \mathcal{T}_{vb}[\![e_n]\!]) \\
\\
tb_1: & \mathcal{T}_{tb}[\![T]\!] & = T \\
tb_2: & \mathcal{T}_{tb}[\![add(e_1, ..., e_n)]\!] & = add(\mathcal{T}_{tb}[\![e_1]\!], ..., \mathcal{T}_{tb}[\![e_n]\!]) \\
tb_3: & \mathcal{T}_{tb}[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & = \textbf{let}\ v = \mathcal{T}_{vb}[\![e_1]\!] \\
& & \quad \textbf{in if}\ v = unknown \\
& & \qquad \textbf{then}\ max(\mathcal{T}_{tb}[\![e_2]\!], \mathcal{T}_{tb}[\![e_3]\!]) \\
& & \qquad \textbf{else if}\ v\ \textbf{then}\ \mathcal{T}_{tb}[\![e_2]\!]\ \textbf{else}\ \mathcal{T}_{tb}[\![e_3]\!] \\
tb_4: & \mathcal{T}_{tb}[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{let}\ v = \mathcal{T}_{vb}[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_{tb}[\![e_2]\!] \\
tb_5: & \mathcal{T}_{tb}[\![\textbf{letrec}\ v = e_1\ \textbf{in}\ e_2]\!] & = \textbf{letrec}\ v = \mathcal{T}_{vb}[\![e_1]\!]\ \textbf{in}\ \mathcal{T}_{tb}[\![e_2]\!] \\
tb_6: & \mathcal{T}_{tb}[\![time(e_0)\,(e_1, ..., e_n)]\!] & = time\_apply(\mathcal{T}_{vb}[\![e_0]\!], \mathcal{T}_{vb}[\![e_1]\!], ..., \mathcal{T}_{vb}[\![e_n]\!])
\end{array}$$

Rule vb_1 leaves variables unchanged, as they do not change with the introduction of the value *unknown*. Rule vb_2 transforms arguments of a constructor recursively. Rule vb_3 transforms the arguments recursively and replaces the primitive operator p by the new operator f_p that returns *unknown* if any of the arguments evaluates to *unknown*. Rule vb_4 transforms subexpressions recursively, builds an expression that binds the value of the transformed e_1 to a distinct variable v, and if the value of v is *unknown* returns the least upper bound of the values of the two transformed branches, otherwise returns the value of the appropriate branch based on the value of v. Rules vb_5 and vb_6 do not directly use the value *unknown*, so they simply transform subexpressions recursively. Rule vb_7 uses $\mathcal{T}_{vb}$ to transform the value component of the lambda pair and uses $\mathcal{T}_{tb}$ to transform the time component. Rule vb_8 uses function *value_apply* to apply the transformed function to the transformed arguments.

Rule tb_2 transforms subexpressions recursively. Rule tb_3 is similar to rule vb_4, except that it computes the maximum time instead of the least upper bound when the value of the condition is *unknown*. Rules tb_4 and tb_5 use $\mathcal{T}_{vb}$ to transform the binding expression, and recursively use $\mathcal{T}_{tb}$ to transform the body. Rule tb_6 uses *time_apply* to handle *unknown* functions; it uses $\mathcal{T}_{vb}$ to transform the argument expressions because the time lambda expression takes values as arguments.

Applying transformation $\mathcal{T}_b$ to function *index-cps* in Figure 2 yields function *index-cps* in Figure 3. Again, shared code is presented with *where* clauses.

```
index-cps ≜ lambda-pair(lambda (item, ls, k)
                          let $v_1 = f_{null?}(ls)$
                          in if $v_1$ = unknown then lub(−1, $exp_1$)
                             else if $v_1$ then −1 else $exp_1$,
                        lambda (item, ls, k)
                          let $v_2 = f_{null?}(ls)$
                          in if $v_2$ = unknown then
                               max($time_1$, $time_2$)
                             else if $v_2$ then $time_1$ else $time_2$)
         where $exp_1$ is let $v_3$ = item $f_=$ $f_{car}(ls)$
                        in if $v_3$ = unknown then lub(value_apply(k, 0), $exp_2$)
                           else if $v_3$ then time_apply(k, 0) else $exp_2$
                      where $exp_2$ is value_apply(index-cps, item, $f_{cdr}(ls)$, $lambda_1$)
               $time_1$ is add($T_{if}$, add($T_{null?}$, $T_{varref}$), $T_{constant}$)
               $time_2$ is add($T_{if}$, add($T_{null?}$, $T_{varref}$),
                           let $v_4$ = item $f_=$ $f_{car}(ls)$
                           in if $v_4$ = unknown then max($time_3$, $time_4$)
                              else if $v_4$ then $time_3$ else $time_4$)
                       where $time_3$ is add($T_{if}$, add($T_=$, $T_{varref}$, add($T_{car}$, $T_{varref}$))
                                         add($T_{funcall}$, time_apply(k, 0), $T_{varref}$, $T_{constant}$))
                           $time_4$ is add($T_{if}$, add($T_=$, $T_{varref}$, add($T_{car}$, $T_{varref}$))
                                         add($T_{funcall}$, time_apply(index-cps, item, $f_{cdr}(ls)$, $lambda_1$),
                                             $T_{varref}$, $T_{varref}$, add($T_{cdr}$, $T_{varref}$), Tlambda ))
                               where $lambda_1$ is lambda-pair(lambda (v) value_apply(k, v $f_+$ 1),
                                                               lambda (v) add($T_{funcall}$, time_apply(k, v $f_+$ 1),
                                                                              $T_{varref}$, add($T_+$, $T_{constant}$, $T_{varref}$)))
```

Figure 3: Function *index-cps* after time-bound transformation $\mathcal{T}_b$.

4 Optimizing time-bound function

Time-bound functions may be extremely inefficient to evaluate given values for their parameters. In fact, in the worst case, the evaluation takes exponential time in terms of the input parameters, since it essentially searches for the worst-case execution path for all inputs satisfying the partially known input structures.

This section describes symbolic evaluation and optimizations that make the computation of time bounds drastically more efficient so that it is feasible to compute them quickly for input sizes in the thousands. The transformations consist of partial evaluation, realized as global inlining, and incremental computation, realized as local optimization.

Partial evaluation of time-bound functions. In practice, values of input parameters are given for almost all applications. This is why time-analysis techniques used in systems can require loop bounds from the user before time bounds are computed. While in general it is not possible to obtain explicit loop bounds automatically and accurately, we can implicitly achieve the desired effect by evaluating the time-bound function symbolically in terms of primitive parameters given specific values of input parameters.

The evaluation simply follows the structures of time-bound functions. Specifically, the control structures determine conditional branches and make recursive function calls as usual. The only primitive operations are sums of primitive parameters and maximums among alternative sums, which can easily be done symbolically. Thus, the transformation simply inlines all function calls, sums all primitive parameters symbolically, determines conditional branches if it can, and takes maximum sums among all possible branches if it can not.

The symbolic evaluation $\mathcal{E}$ defined below performs the transformations. It takes as arguments an expression e and an environment ρ of variable bindings and returns as result a symbolic value that contains the primitive parameters. The evaluation starts with the application of the program to be analyzed to a partially known input structure, e.g., $index(unknown, list(100))$, and it starts with an empty environment. Assume add_s is a function that symbolically sums its arguments, and max_s is a function that symbolically takes the maximum of its arguments.

$$\begin{array}{lll}
se_1: & \mathcal{E}[\![v]\!]\rho & = \rho(v) \\
se_2: & \mathcal{E}[\![T]\!]\rho & = T \\
se_3: & \mathcal{E}[\![c(e_1,...,e_n)]\!]\rho & = c(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho) \\
se_4: & \mathcal{E}[\![p(e_1,...,e_n)]\!]\rho & = p(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho) \\
se_5: & \mathcal{E}[\![add(e_1,...,e_n)]\!]\rho & = add_s(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho) \\
se_6: & \mathcal{E}[\![max(e_1,...,e_n)]\!]\rho & = max_s(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho) \\
se_7: & \mathcal{E}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\rho & = \mathcal{E}[\![e_2]\!]\rho \quad \text{if } \mathcal{E}[\![e_1]\!]\rho = true \\
 & & \mathcal{E}[\![e_3]\!]\rho \quad \text{if } \mathcal{E}[\![e_1]\!]\rho = false \\
se_8: & \mathcal{E}[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{]}\!]\rho & = \mathcal{E}[\![e_2]\!]\rho[v \mapsto \mathcal{E}[\![e_1]\!]\rho] \\
se_9: & \mathcal{E}[\![\textbf{letrec } v = e_1 \textbf{ in } e_2 \textbf{]}\!]\rho & = \mathcal{E}[\![e_2]\!]\rho[v \mapsto \mathcal{E}[\![e_1]\!]\rho] \\
se_{10}: & \mathcal{E}[\![\textbf{lambda } (v_1,...,v_n)\, e_0]\!]\rho & = \langle e_0, \rho \rangle \\
se_{11}: & \mathcal{E}[\![e_0(e_1,...,e_n)]\!]\rho & = e'_0[v_1 \mapsto \mathcal{E}[\![e_1]\!]\rho,...,v_n \mapsto \mathcal{E}[\![e_n]\!]\rho]\,\rho' \\
 & & \text{if } \mathcal{E}[\![e_0]\!]\rho = \langle e'_0, \rho' \rangle
\end{array}$$

As an example, applying symbolic evaluation to the time-bound function for *index* on an *unknown* item and a list of size 100, we obtain the following result:

$$\begin{array}{lll}
\mathcal{E}[\![index(unknown, list(100))]\!]\emptyset & = & 101 * T_{constant} + 802 * T_{varref} + 201 * T_{if} \\
 & & + 201 * T_{funcall} + 101 * T_{lambda} + 100 * T_{car} \\
 & & + 100 * T_{cdr} + 101 * T_{null?} + 99 * T_{+} + 100 * T_{=}
\end{array}$$

Avoiding repeated summations over recursions. The symbolic evaluation above is a global optimization over all time-bound functions involved. During the evaluation, summations of symbolic primitive parameters within each function definition are performed repeatedly while the computation recurses. Thus, we can speed up the symbolic evaluation by first performing such summations in a preprocessing step. Specifically, we create a vector and let each element correspond to a primitive parameter. The transformation $\mathcal{S}$ performs this optimization. We introduce two new functions: add_{sv} performs symbolic addition by component-wise summation of the argument vectors, and max_{sv} computes the component-wise maximum of the argument vectors.

$$\begin{array}{llll}
\text{program:} & \mathcal{S}\left[\!\!\left[\begin{array}{l} v_1 \triangleq e_1, \\ ..., \\ v_n \triangleq e_n \end{array}\right]\!\!\right] & = & \begin{array}{l} v_1 \triangleq \mathcal{S}_t[\![e_1]\!], \\ ..., \\ v_n \triangleq \mathcal{S}_t[\![e_n]\!] \end{array} \\
\text{primitive parameter:} & \mathcal{S}_t[\![T]\!] & = & \text{create a vector of 0's except with the} \\
 & & & \text{component corresponding to } T \text{ set to 1} \\
\text{summation:} & \mathcal{S}_t[\![add(e_1,...,e_n)]\!] & = & add_{sv}(\mathcal{S}_t[\![e_1]\!],...,\mathcal{S}_t[\![e_n]\!]) \\
\text{maximum:} & \mathcal{S}_t[\![max(e_1,...,e_n)]\!] & = & max_{sv}(\mathcal{S}_t[\![e_1]\!],...,\mathcal{S}_t[\![e_n]\!]) \\
\text{all other:} & \mathcal{S}_t[\![e]\!] & = & e
\end{array}$$

Applying this optimization to the time-bound version of function *index-cps* in Figure 3 yields the definition in Figure 4.

This incrementalizes the computation in each recursive step to avoid repeated summation. As other transformations we have described, this is fully automatic and takes linear time, here in terms of the size of the time-bound function.

The result of this optimization is dramatic. For example, optimized symbolic evaluation of the same curried Ackermann with input $\langle 3, 7 \rangle$ takes only 1.68 seconds while unoptimized symbolic evaluation takes 127 seconds.

On small inputs, symbolic evaluation takes relatively much more time than direct evaluation, due to the relatively large overhead of vector setup; as inputs get larger, symbolic evaluation is almost as fast as direct evaluation for most examples. After the symbolic evaluation, time bounds can be computed in virtually no time given primitive parameters measured on any machine.

Time-bound functions can further be made more accurate by lifting conditions, simplifying conditionals, and inlining non-recursive functions, as done previously in [24].

```
index-cps
  ≜ lambda-pair(lambda (item, ls, k)
                      let $v_1$ = $f_{null?}(ls)$ in
                        if $v_1$ = unknown then lub(−1, $exp_1$)
                        else if $v_1$ then −1 else $exp_1$,
                    lambda (item, ls, k)
                      let $v_2$ = $f_{null?}(ls)$ in
                        if $v_2$ = unknown then $max_v$(<0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0>, $time_1$)
                        else if $v_2$ then <0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0> else $time_1$)
  where $exp_1$ is let $v_3$ = item $f_=$ $f_{car}(ls)$ in
                    if $v_3$ = unknown then lub(value_apply(k, 0), value(index-cps)(item, $f_{cdr}(ls)$, $lambda_1$))
                    else if $v_3$ then value_apply(k, 0) else value_apply(index-cps, item, $f_{cdr}(ls)$, $lambda_1$)
        $time_1$ is let $v_4$ = item $f_=$ $f_{car}(ls)$ in
                     if $v_4$ = unknown then
                       $max_v$($add_v$(<1 0 0 1 1 0 0 0 0 0 0 1 4 2 0 0 1 0>, time(k)(0)),
                              $add_v$(<1 1 0 1 1 0 0 0 0 0 0 0 6 2 0 0 1 0>, time(index-cps)(item, $f_{cdr}(ls)$, $lambda_1$)))
                     else if $v_4$ then $add_v$(<1 0 0 1 1 0 0 0 0 0 0 1 4 2 0 0 1 0>, time(k)(0))
                         else $add_v$(<1 1 0 1 1 0 0 0 0 0 0 0 6 2 0 0 1 0>, time(index-cps)(item, $f_{cdr}(ls)$, $lambda_1$))
            where $lambda_1$ is lambda-pair(lambda (v) value_apply(k, v $f_+$ 1),
                                           lambda (v) $add_v$(<0 0 0 0 0 1 0 0 0 0 0 1 2 0 0 0 1 0>, time(k)(v $f_+$ 1)))
```

Figure 4: Function *index-cps* after optimization for avoiding repeated summations.

5 Implementation and experimentation

We have implemented the analysis approach in a prototype system, ALPA (Automatic Language-based Performance Analyzer). We performed a large number of measurements and obtained encouraging good results.

The implementation is for a subset of Scheme. The prototype is implemented using Chez Scheme v6.0a compiler [8]. The input is a program as defined in Section 2, but with Scheme syntax. The output is an optimized time-bound function that takes an input size and returns the symbolic time bound of the program for inputs of that size.

The computer used to take the measurements is a Sun Enterprise 450 Model 4400 with four 400MHz cpu's, 1 GB of RAM, and 4.6 GB virtual memory.

Since the minimum running time of a program construct is about 0.1 microseconds, and the precision of the time function is 10 milliseconds, we use control/test loops that iterate 10,000,000 times, keeping measurement error under 0.001 microseconds, i.e., 1%. Such a loop is repeated 100 times, and the average value is taken to compute the primitive parameter for the tested construct (the variance is less than 10% in most cases). The calculation of the time bound is done by plugging these measured parameters into the optimized time-bound function. We then run each example program an appropriate number of times to measure its running time with less than 1% error.

All the measurements were done by starting a new Scheme process, loading the needed definitions, measuring the time of interest, and exiting Scheme. This ensures that only the time related to the given program is counted.

The example programs shown here are: *ack*: Ackermann function programmed using the standard first-order recursive definition; *ack-curried*: a curried version of Ackermann function that uses higher-order functions (and is almost twice as fast as the standard first-order function); *tak-cps*: the Takeuchi function in CPS, part of the Gabriel benchmark suite [14]; *reverse*: standard first-order list reverse function; *rev-cps*: a CPS version of reverse; *split*: taking a predicate and a list and returning two lists, one whose elements satisfy the predicate and another whose elements do not satisfy the predicate; *fix*: factorial function programmed using the Y combinator for a heavy use of higher-order functions; *map*: standard map function; *union*: taking two sets and returning the union; *index*: taking an item and a list and returning the index of the item in the list, or −1 if the item is not in the list.

Table 1 gives the results of symbolic evaluation of the time-bound functions for these example programs on inputs of various sizes. Several counts of the primitive operations are merged to fit the table on the page. All numbers are exact symbolic counts. They are verified by using a modified evaluator.

Table 2 shows the calculated and the measured worst-case running time for these programs with various input sizes. The item me/ca is the measured time expressed as a percentage of the calculated time. In

program	size	var ref	constant	cons	null?	car/cdr	+/-	compare	if	let(rec)	lambda	funcall
ack	⟨3,1⟩	472	328	0	0	0	153	164	164	0	0	106
	⟨3,5⟩	190848	127560	0	0	0	63533	63780	63780	0	0	42438
	⟨3,7⟩	3122332	2082904	0	0	0	1040439	1041452	1041452	0	0	693964
	⟨3,9⟩	50237624	33497192	0	0	0	16744513	16748596	16748596	0	0	11164370
ack curried	⟨3,1⟩	277	171	0	0	0	98	62	62	6	4	111
	⟨3,5⟩	105989	63787	0	0	0	42194	21346	21346	6	4	42443
	⟨3,7⟩	1734421	1041459	0	0	0	692954	347492	347492	6	4	693969
	⟨3,9⟩	27908901	16748603	0	0	0	11160290	5584230	5584230	6	4	11164375
tak-cps	⟨19,8,1⟩	16121904	1560183	0	0	0	1560183	2080245	2080245	1	1560185	3640430
	⟨19,9,1⟩	46538205	4503696	0	0	0	4503696	6004929	6004929	1	4503698	10508627
	⟨19,9,3⟩	2582251	249894	0	0	0	249894	333193	333193	1	249896	583089
	⟨19,10,1⟩	122680095	11872266	0	0	0	11872266	15829689	15829689	1	11872268	27701957
reverse	10	299	10	55	66	110	0	0	66	0	0	66
	20	1094	20	210	231	420	0	0	231	0	0	231
	50	6479	50	1275	1326	2550	0	0	1326	0	0	1326
	100	25454	100	5050	5151	10100	0	0	5151	0	0	5151
	200	100904	200	20100	20301	40200	0	0	20301	0	0	20301
	500	627254	500	125250	125751	250500	0	0	125751	0	0	125751
	1000	2504504	1000	500500	501501	1001000	0	0	501501	0	0	501501
	2000	10009004	2000	2001000	2003001	4002000	0	0	2003001	0	0	2003001
rev-cps	10	422	11	55	66	110	0	0	66	0	56	123
	20	1537	21	210	231	420	0	0	231	0	211	443
	50	9082	51	1275	1326	2550	0	0	1326	0	1276	2603
	100	35657	101	5050	5151	10100	0	0	5151	0	5051	10203
	200	141307	201	20100	20301	40200	0	0	20301	0	20101	40403
	500	878257	501	125250	125751	250500	0	0	125751	0	125251	251003
	1000	3506507	1001	500500	501501	1001000	0	0	501501	0	500501	1002003
	2000	14013007	2001	2001000	2003001	4002000	0	0	2003001	0	2001001	4004003
split	10	128	33	12	11	30	0	20	41	0	10	32
	20	248	63	22	21	60	0	40	81	0	20	62
	50	608	153	52	51	150	0	100	201	0	50	152
	100	1208	303	102	101	300	0	200	401	0	100	302
	200	2408	603	202	201	600	0	400	801	0	200	602
	500	6008	1503	502	501	1500	0	1000	2001	0	500	1502
	1000	12008	3003	1002	1001	3000	0	2000	4001	0	1000	3002
	2000	24008	6003	2002	2001	6000	0	4000	8001	0	2000	6002
fix	10	275	22	0	0	0	20	11	11	0	212	233
	20	545	42	0	0	0	40	21	21	0	422	463
	50	1355	102	0	0	0	100	51	51	0	1052	1153
	100	2705	202	0	0	0	200	101	101	0	2102	2303
	200	5405	402	0	0	0	400	201	201	0	4202	4603
	500	13505	1002	0	0	0	1000	501	501	0	10502	11503
	1000	27005	2002	0	0	0	2000	1001	1001	0	21002	23003
	2000	54005	4002	0	0	0	4000	2001	2001	0	42002	46003
map	10	84	2	10	11	20	10	0	11	0	1	22
	20	164	2	20	21	40	20	0	21	0	1	42
	50	404	2	50	51	100	50	0	51	0	1	102
	100	804	2	100	101	200	100	0	101	0	1	202
	200	1604	2	200	201	400	200	0	201	0	1	402
	500	4004	2	500	501	1000	500	0	501	0	1	1002
	1000	8004	2	1000	1001	2000	1000	0	1001	0	1	2002
	2000	16004	2	2000	2001	4000	2000	0	2001	0	1	4002
union	10	705	10	10	121	230	0	100	231	10	0	121
	20	2605	20	20	441	860	0	400	861	20	0	441
	50	15505	50	50	2601	5150	0	2500	5151	50	0	2601
	100	61005	100	100	10201	20300	0	10000	20301	100	0	10201
	200	242005	200	200	40401	80600	0	40000	80601	200	0	40401
	500	1505005	500	500	251001	501500	0	250000	501501	500	0	251001
	1000	6010005	1000	1000	1002001	2003000	0	1000000	2003001	1000	0	1002001
	2000	24020005	2000	2000	4004001	8006000	0	4000000	8006001	2000	0	4004001
index	10	72	11	0	11	20	9	10	21	1	12	21
	20	142	21	0	21	40	19	20	41	1	22	41
	50	352	51	0	51	100	49	50	101	1	52	101
	100	702	101	0	101	200	99	100	201	1	102	201
	200	1402	201	0	201	400	199	200	401	1	202	401
	500	3502	501	0	501	1000	499	500	1001	1	502	1001
	1000	7002	1001	0	1001	2000	999	1000	2001	1	1002	2001
	2000	14002	2001	0	2001	4000	1999	2000	4001	1	2002	4001

Table 1: Results of symbolic evaluation of time-bound functions (exact counts).

general, all measured times are closely bounded by the calculated times (with about 70-98% accuracy).

	ackermann			ackermann (curried)				takeuchi (CPS)		
size	calculated	measured	me/ca	calculated	measured	me/ca	size	calculated	measured	me/ca
$\langle 3,1\rangle$	0.03207	0.02861	89.2002	0.02059	0.01503	72.9892	$\langle 19,8,1\rangle$	576.058	509.402	88.4289
$\langle 3,5\rangle$	12.8462	10.6540	82.9348	7.89957	5.33000	67.4719	$\langle 19,9,1\rangle$	1662.87	1473.49	88.6111
$\langle 3,7\rangle$	210.051	174.943	83.2863	129.241	89.1780	69.0009	$\langle 19,9,3\rangle$	92.2675	81.6250	88.4655
$\langle 3,9\rangle$	3379.19	2888.33	85.4739	2079.53	1517.27	72.9621	$\langle 19,10,1\rangle$	4383.53	3912.50	89.2543

	reverse			reverse (CPS)			split		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.02410	0.01854	76.9136	0.02872	0.02395	83.3632	0.00877	0.00769	87.7389
20	0.08873	0.06615	74.5462	0.10591	0.08774	82.8435	0.01710	0.01489	87.0741
50	0.52675	0.38781	73.6221	0.63007	0.52147	82.7634	0.04211	0.03588	85.2049
100	2.07054	1.53300	74.0385	2.47907	2.06100	83.1357	0.08378	0.07103	84.7775
200	8.20967	6.03300	73.4864	9.83483	8.13700	82.7365	0.16713	0.14151	84.6698
500	51.0395	37.9980	74.4481	61.1641	50.6200	82.7609	0.41717	0.35321	84.6673
1000	203.797	158.995	78.0164	244.252	202.042	82.7185	0.83391	0.70749	84.8399
2000	814.470	656.137	80.5600	976.205	815.471	83.5348	1.66738	1.40501	84.2642

	fix			map			union			index		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.02059	0.01981	96.1887	0.00578	0.00476	82.2714	0.04512	0.03547	78.5966	0.00476	0.00344	72.26890
20	0.04087	0.03879	94.9079	0.01133	0.00900	79.4816	0.16680	0.13401	80.3414	0.00894	0.00647	72.37136
50	0.10169	0.09605	94.4445	0.02798	0.02169	77.5121	0.99204	0.80972	81.6212	0.02148	0.01561	72.67225
100	0.20308	0.19183	94.4597	0.05572	0.04360	78.2375	3.90155	3.08000	78.9429	0.04273	0.03073	71.91668
200	0.40584	0.38599	95.1080	0.11121	0.08781	78.9532	15.4734	12.1280	78.3794	0.08575	0.06166	71.90670
500	1.01413	0.97661	96.3001	0.27768	0.22843	82.2614	96.2121	75.1470	78.1055	0.21951	0.15412	70.21092
1000	2.02794	1.93700	95.5154	0.55513	0.48007	86.4776	384.186	315.918	82.2304	0.43402	0.31197	71.87917
2000	4.05556	4.00700	98.8024	1.11003	0.95652	86.1700	1535.42	1260.83	82.1163	1.05827	0.77977	73.68346

Table 2: Calculated and measured worst-case times (in milliseconds.)

6 Related work and conclusion

An overview of comparison with related work in time analysis appears in Section 2. Certain detailed comparisons have also been discussed while presenting our method. This section summarizes them, compares with other related work, and concludes.

Compared to work in algorithm analysis and program complexity analysis [22, 35, 34, 40], this work consistently pushes through symbolic primitive parameters, so it allows us to calculate actual time bounds and validate the results with experimental measurements. There is also work on analyzing average-case complexity [13], which has a different goal than worst-case bounds. Compared to work in systems [36, 29, 28, 23], this work explores program analysis and transformation techniques to make the analysis automatic, efficient, and accurate, overcoming the difficulties caused by the inability to obtain loop bounds, recursion depths, or execution paths automatically and precisely. There is also work for measuring primitive parameters of Fortran programs for the purpose of general performance prediction [33, 32], where information about execution paths was obtained by running the programs on a number of inputs; for programs such as insertion sort whose best-case and worst-case execution times differ greatly, the predicted time using that method could be very inaccurate.

Reistad and Gifford [30] studied static analysis that helps estimating running times in the presence of first-class procedures, and the results of the estimation were used for dynamic parallelization. Their analysis produces only a formula that needs to be computed at run time after information about the particular input is available; they do not analyze time bounds in the presence of incomplete knowledge about the input as we do. Also, their cost systems do not handle user-defined recursive procedures as we do; as pointed out by Hughes and others [19], the extension to user-defined recursive procedures is a major one that affects the entire system. They also mention that they handle imperative constructs, but the analysis and transformations given do not handle mutable data, so relevant constructs can be simulated easily using bindings.

Several type systems [19, 18, 6] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [19, 6]. These do not analyze cost, or build cost functions. Programmers are required to annotate their programs with cost functions as types; some programs have to be rewritten to have feasible types [19, 18].

A number of techniques have been studied for obtaining loop bounds or execution paths for analyzing time bound [28, 2, 10, 15, 17]. Manual annotations [28, 23] are inconvenient and error-prone [2]. Automatic analysis of such information has two main problems. First, even when a precise loop bound can be obtained by symbolic evaluation of the program [10], separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [27]. Second, approximations for merging paths from loops, or recursions, very often lead to nontermination of the time analysis, not just looser bounds [10, 15, 27]. Some new methods, while powerful, apply only to certain classes of programs [17]. In contrast, our method allows recursions, or loops, to be considered naturally in the overall execution-time analysis based on partially known input structures. In addition, our method does not merge paths from recursions, or loops; this may cause exponential time complexity in the analysis, but our experiments on test programs show that the analysis is still tractable for input sizes in the thousands. We have also studied simple but powerful optimizations to speed up the analysis.

In the analysis for cache behavior by Ferdinand and others [11], loops are transformed into recursive calls, and a predefined *callstring* level determines how many times the fixed point analysis iterates and thus how the analysis results are approximated. Our method allows the analysis to perform the exact number of recursions, or iterations, for the given partial input data structures. Recent work by Lundqvist and Stenstrom [27] is based on essentially the same ideas as ours. They apply the ideas at machine instruction level and can more accurately take into account the effects of instruction pipelining and data caching, but their method for merging paths for loops would lead to nonterminating analysis for many programs, for example, a program that computes the union of two lists with no repeated elements. We apply the ideas at source-level, and our experiments show that we can calculate more accurate time bound and for many more programs than merging paths, and the calculation is still efficient.

The idea of using partially known input structures originates from Rosendahl [31]. We have extended it to manipulate primitive parameters, to handle binding constructs, and most importantly, to include higher-order functions. The power of our method also lies in the optimizations of the time-bound function using partial evaluation, incremental computation, and transformations of conditionals to make the analysis more efficient and more accurate. Partial evaluation [4, 20], incremental computation [26, 25], and other transformations have been studied intensively in programming languages. Their applications in our time-bound analysis are particularly simple and clean; the resulting transformations are fully automatic and efficient.

We have started to explore a suite of new language-based techniques for time analysis, in particular, analyses and optimizations for further speeding up the evaluation of the time-bound function. To make the analysis even more accurate and efficient, we can automatically generate measurement programs for all maximum subexpressions that do not include transfers of control; this corresponds to the large atomic-blocks method [29]. We also believe that the lower-bound analysis is symmetric to the upper-bound analysis, by replacing maximum with minimum at all conditional points; there, special pruning actually allows us to speed up the analysis even further. Finally, we plan to accommodate more lower-level dynamic factors for timing at the source-language level [23, 11]. In particular, we have started applying our general approach to analyze space consumption [37] and hence to help predict garbage-collection and caching behavior.

In conclusion, the approach we developed is based entirely on program analysis and transformations at the source level. The methods and techniques are intuitive; together they produce automatic tools for analyzing time bounds efficiently and accurately. We find the accuracy of the experimental results very encouraging, especially considering that we are analyzing recursive programs at source-level, with garbage collection, and currently without special treatment for instruction pipelining or cache effects.

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] R. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 13th IEEE Real-Time Systems Symposium.* IEEE CS Press, Los Alamitos, Calif., 1994.

[4] B. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, Amsterdam, 1988.

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

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

[7] R. K. Dybvig. *The Scheme Programming Language*, Second edition. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[8] R. K. Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998

[9] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems*, Berlin, Germany, June 1998.

[10] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *In Proceedings of Euro-Par'97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1298–1307. Springer-Verlag, Berlin, Aug. 1997.

[11] 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.

[12] 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.

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

[14] R. P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press series in computer systems. MIT Press, Cambridge, MA, 1985

[15] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2), June 1998.

[16] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE CS Press, Los Alamitos, Calif., Dec. 1992.

[17] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the IEEE Real-Time Applications Symposium*. IEEE CS Press, Los Alamitos, Calif., June 1998.

[18] 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.

[19] 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.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, N.J., 1993.

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

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

[23] 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.

[24] 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.

[25] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

[26] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[27] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. Technical Report No. 98-3, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1998.

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

[29] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Comput.*, 24(5):48–57, 1991.

[30] 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.

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

[32] 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.

[33] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12):1659–1679, Dec. 1989. Special issue on Performance Evaluation.

[34] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, London, U.K., Sept. 1990.

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

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

[37] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate stack space and heap space analysis for high-level languages. Technical report, Computer Science Department, Indiana University. To appear.

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

[39] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, Jan. 1994.

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