AUTOMATIC TIME-BOUND ANALYSIS FOR HIGH-LEVEL LANGUAGES

Gustavo Gomez

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
June, 2006

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

David S. Wise, Ph.D.

Yanhong A. Liu, Ph.D.

R. Kent Dybvig, Ph.D.

Gregory J. E. Rawlins, Ph.D.

May 2, 2005

To Eliana

To Andrés and Lucía

Acknowledgements

I am extremely grateful to my advisor Annie Liu for all her support and advice, encouragement and guidance. Her help made possible all this work.

I can not be more grateful to my "adoptive" advisor David Wise. Thanks for all the help and important advice. I wish to thank the other members of my committee as well: Kent Dybvig and Gregory Rawlins. Thanks for the excellent comments and important questions.

Thanks to all the friends that made all this time easier, richer, more enjoyable, and in general, better. The CS folks: Byron, Cris, Heather, Jenett, Jerry, Jim, Kaushik, Kyle, Leena, Nirmal, Pantelis, Peter, Rutvik, Steve, Xing Chun, Yuchen. The Knee-High family: Amy, Anne, Antonia, Beth, Brian, Cal, Choonhyun, Corey, Dmitri, Hilary, James, Janette, Jason, Jen, Joan, Joanne, Lore, Majd, Molly, Patrick, Saya, Shorouq, Steve, Tracy, Will, Abbey, Allie, Anisa, Anneke, Brooke, Cam, Carter, Cate, Cleo, Dahin, Eve, Finn, Grayson, Isaac, Isaia, Ivy, Jacob, Layth, Logan, Olivia, Sean, Shannon, Sophie, Thais, Yuki, Jen, Mary Corinne, Maribeth, Nikkie, Rose, Stacy. The Spanish Playgroup: Bárbara, Daniel, Enriquín, Enrique, Eri,, Giselle, Jose, Juan Pablo, Leonardo, Lucía, Mónica, Mari, Mladen, Nancy, Paty, Ron, Sonia, Tomás.

A big thanks to my family. They never stopped believing that some day I would finish my dissertation. Special thanks to my mom and my dad, and to my suegro and my suegra.

My wife deserves so much more than "thanks". The trite phrase "I could not have done it without her" is the actual truth. Thank you.

This work was partially supported by the Consejo Nacional de Ciencia y Tecnología, Mexico, the National Science Foundation under Grant CCR-9711253 and the Office of Naval Research under Grant N00014-99-1-0132.

Abstract

Analysis of program running time is important for reactive systems, interactive environments, compiler optimizations, performance evaluation, and many other computer applications. Automatic and efficient prediction of accurate time bounds is particularly important, and being able to do so for high-level languages is particularly desirable. This dissertation presents a general approach for automatic and accurate time-bound analysis for high-level languages, combining methods and techniques studied in theory, languages, and systems. 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. 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.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Analysis of program running time is important for reactive systems, interactive environments, compiler optimizations, performance evaluation, and many other computer applications. This analysis has been extensively studied in many fields of computer science: algorithms [**60, 32, 33, 104**], programming languages [**100, 61, 84, 88, 87**], and systems [**91, 76, 86, 85**]. Being able to predict accurate time bounds automatically and efficiently is particularly important for many time-sensitive applications, such as reactive systems. It is also particularly desirable to be able to do so for high-level languages [**91, 76**].

Since Shaw proposed a timing schema for analyzing system running time based on high-level languages [**91**], a number of people have extended it for analysis in the presence of compiler optimizations [**76, 28**], pipelining [**46, 62**], cache memory [**4, 62, 31**], etc. However, there is still a serious limitation of this 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 [**75, 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 [**76**], whereas the manual way of providing such information is potentially an even larger source of error, in addition to being inconvenient [**75**]. Various program analysis methods have been proposed to provide loop bounds or execution paths [**2, 29, 44, 47**]. However, they apply only to some classes of programs or use approximations that are too crude for accurate analysis. Also, separating the loop and path information from the rest of the analysis is in general less accurate than performing an integrated analysis [**70**].

This dissertation describes a general approach for automatic and accurate time-bound analysis that combines methods and techniques studied in the areas of systems, languages, and theory. It is a *language-based* approach since it primarily exploits methods and techniques for static program analysis and transformation, and uses techniques from systems and theory to improve its accuracy and efficiency.

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 overall the analysis efficient as well as accurate, and measurements of primitive parameters, all at the source-language level.

This approach is powerful because it is general in three senses. First, it works for other kinds of cost analysis as well, such as space analysis and output-size analysis. Second, the basic ideas also apply to any programming language. I implemented the approach on a functional, on an imperative and on a high-order language. And third, the implementation is independent of the underlying systems – compilers, operating systems, and hardware.

The rest of the dissertation is organized as follows. This chapter describes the language-based approach. Chapters 2, 3 and 4 present the approach used with a first-order functional language, an imperative language and a higher-order functional language respectively. Each chapter describes analysis and transformation algorithms and explain how they work, as well as the implementation and experiments analyzing Scheme programs. The measured worst-case times are closely bounded by the calculated bounds. I describe our prototype system, ALPA, as well as the analysis and measurement results. Chapter 3 presents the approach with an imperative language in two ways. First it uses a program transformation to convert the imperative program into a functional one using Storage-Passing-Style so the approach from Chapter 2 is applicable. The second way is to have a new transformation specific for the imperative language. Chapter 5 presents an application of the approach to automatically obtain a worst-case input, an input that will make the program exhibit its worst-case execution time. Chapter 6 discusses advantages and disadvantages compared to related work, limitations and future work.

1. 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 [**100, 12, 91**]. The parameters that describe the running times of program constructs are called *primitive parameters*. To calculate actual time bounds based on the time function, three difficult problems must be solved: characterize the input data, optimize the time-bound function and obtain the values of the primitive parameters.

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 [**100, 61,**

104]. 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 [**100, 12, 61, 84, 33**]. Although much progress has been made in this area, closed forms are known only for subclasses of functions. Thus, such optimization cannot 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 [**91, 76, 75, 62**], even though program analyses can help in some cases [**2, 29, 44, 47**]. 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 [**76, 4, 62, 31**]. 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 the area of programming-languages, Rosendahl proposed using *partially known input structures* [**84**]. 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, a partially-known input structure approach would use as input a list of n unknown elements. The parameters for describing partially known input structures are called *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 cannot be obtained for all time-bound functions. Also, Rosendahl handles only first-order functions. Sands studied time functions for higher-order functions [**88, 87**], 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, I have developed a general approach for computing time bounds automatically, efficiently, and more accurately. The approach has four main components.

First, an automatic transformation is used 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, the time-bound function is optimized 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, I measure primitive parameters at the source-language level and use the best conservative estimations in computing the time bound. I 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 developeded 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.

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. Our analysis scales well with program size, as the transformations take linear time in terms of program size, but depending on program structures, the analysis might not scale well with input size used in partially known input structures.

CHAPTER 2

Analysis of a Functional Language

This chapter will discuss the approach using a functional language. The chapter is organized as follows. Section 1 gives a formal definition of the language used. Sections 2, 3, and 4 present the analysis and transformation methods and techniques. Section 5 describes our implementation and experimental results.

1. Language definition

The language used is a first-order, call-by-value functional language that has structured data, primitive arithmetic, Boolean, and comparison operations, conditionals, bindings, and mutually recursive function calls. A program is a set of mutually recursive function definitions. Its syntax is given by the grammar in Figure 1 and its semantics is the corresponding subset of Scheme[**58, 25**].

For example, the program in Figure 2 selects the least element in a non-empty list.

$$
\begin{array}{lcll}
\textit{program} & ::= & (\textbf{define}\ (f_1\ v_{1_1}\ \ldots\ v_{1_n})\ e_1) & \\
 & & \qquad\qquad \ldots & \\
 & & (\textbf{define}\ (f_m\ v_{m_1}\ \ldots\ v_{m_n})\ e_m) & \\
e & ::= & v & \text{variable reference} \\
 & | & (c\ e_1\ \ldots\ e_n) & \text{data construction} \\
 & | & (p\ e_1\ \ldots\ e_n) & \text{primitive operation} \\
 & | & (\textbf{if}\ e_1\ e_2\ e_3) & \text{conditional expression} \\
 & | & (\textbf{let}\ ((v\ e_1))\ e_2) & \text{binding expression} \\
 & | & (f\ e_1\ \ldots\ e_n) & \text{function application}
\end{array}
$$

FIGURE 1. Definition of the functional language.

I use *least* as a small running example. To present various analysis results, I also use several other examples: insertion sort, selection sort, merge sort, set union, list

```
(define (least x)
  (if (null? (cdr x))
      (car x)
      (let ((s (least (cdr x))))
        (if (< (car x) s)
            (car x)
            s))))
```

FIGURE 2. Program *least*, which selects the smallest element from a list

reversal (the standard linear-time version), and reversal with append (the standard quadratic-time version).

Even though this language is small, it is sufficiently powerful and convenient to write 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 can simulate loops.

2. Constructing time-bound functions

2.1. Constructing timing functions. The first step is to transform the original program to construct a timing function, which takes the original input and primitive parameters as arguments and returns the running time. This is straightforward based on the semantics of the program constructs.

Given an original program, a set of timing functions are added, one for each original function, which simply count the time while the original program executes. The algorithm, given in Figure 3, is presented as a transformation $\mathcal{T}$ on the original program, which calls a transformation $\mathcal{T}_e$ to recursively transform subexpressions. 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. The function tf denotes the timing function for f.

$$\text{program: } \mathcal{T}\left[\!\!\left[\begin{array}{l}(\mathbf{define}\ (f_1\ v_{1_1}\ ...\ v_{1_n})\ e_1)\\ ...\\ (\mathbf{define}\ (f_m\ v_{m_1}\ ...\ v_{m_n})\ e_m)\end{array}\right]\!\!\right] = \begin{array}{l}(\mathbf{define}\ (f_1\ v_{1_1}\ ...\ v_{1_n})\ e_1)\\ ...\\ (\mathbf{define}\ (f_m\ v_{m_1}\ ...\ v_{m_n})\ e_m)\\ (\mathbf{define}\ (tf_1\ v_{1_1}\ ...\ v_{n_1})\ \mathcal{T}_e[\![e_1]\!])\\ ...\\ (\mathbf{define}\ (tf_m\ v_{m_1}\ ...\ v_{m_n})\ \mathcal{T}_e[\![e_m]\!])\end{array}$$

$$\begin{array}{lll}
\text{variable reference:} & \mathcal{T}_e[\![v]\!] & = T_{varref}\\
\text{data construction:} & \mathcal{T}_e[\![(c\ e_1\ ...\ e_n)]\!] & = (+\ T_c\ \mathcal{T}_e[\![e_1]\!]\ ...\ \mathcal{T}_e[\![e_n]\!])\\
\text{primitive operation:} & \mathcal{T}_e[\![(p\ e_1\ ...\ e_n)]\!] & = (+\ T_p\ \mathcal{T}_e[\![e_1]\!]\ ...\ \mathcal{T}_e[\![e_n]\!])\\
\text{conditional:} & \mathcal{T}_e[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!] & = (+\ T_{if}\ \mathcal{T}_e[\![e_1]\!]\ (\mathbf{if}\ e_1\ \mathcal{T}_e[\![e_2]\!]\ \mathcal{T}_e[\![e_3]\!]))\\
\text{binding:} & \mathcal{T}_e[\![(\mathbf{let}\ ([v\ e_1])\ e_2)]\!] & = (+\ T_{let}\ \mathcal{T}_e[\![e_1]\!]\ (\mathbf{let}\ ([v\ e_1])\ \mathcal{T}_e[\![e_2]\!]))\\
\text{function call:} & \mathcal{T}_e[\![(f\ e_1\ ...\ e_n)]\!] & = (+\ T_{call}\ \mathcal{T}_e[\![e_1]\!]\ ...\ \mathcal{T}_e[\![e_n]\!]\ (tf\ e_1\ ...\ e_n))
\end{array}$$

FIGURE 3. Rules for timing transformation $\mathcal{T}$.

Applying this transformation to the program *least*, we obtain function *least* as originally given and timing function *tleast* shown in Figure 4. Note that various T's are indeed arguments to the timing function *tleast*; but are omitted from argument positions for ease of reading.

$$\begin{array}{l}
(\mathbf{define}\ (tleast\ x)\\
\quad (+\ T_{if}\ (+\ T_{null?}\ T_{cdr}\ T_{varref})\\
\quad\quad (\mathbf{if}\ (null?\ (cdr\ x))\\
\quad\quad\quad (+\ T_{car}\ T_{varref})\\
\quad\quad\quad (+\ T_{let}\ (+\ T_{call}\ (+\ T_{cdr}\ T_{varref})\ (tleast\ (cdr\ x)))\\
\quad\quad\quad\quad (\mathbf{let}\ ((s\ (least\ (cdr\ x))))\\
\quad\quad\quad\quad\quad (+\ T_{if}\ (+\ T_{\le}\ (+\ T_{car}\ T_{varref})\ T_{varref})\\
\quad\quad\quad\quad\quad\quad (\mathbf{if}\ (<\ (car\ x)\ s)\\
\quad\quad\quad\quad\quad\quad\quad (+\ T_{car}\ T_{varref})\\
\quad\quad\quad\quad\quad\quad\quad T_{varref})))))))
\end{array}$$

FIGURE 4. Function *tleast* after transformation $\mathcal{T}$

This transformation is similar to the local cost assignment [**100**], step-counting function [**84**], cost function [**88**], etc. in other work. Our transformation extends those methods with bindings, and 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. Note that the timing 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 timing function.

2.2. Constructing time-bound functions. Characterizing program inputs in the time function is difficult to automate [**100, 61, 91**]. However, partially known input structures provide a natural means [**84**]. 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.

```
(define (list n)
  (if (= n 0)
      '()
      (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, a new primitive function f_p is defined 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. A new *least upper bound* function *lub* is also defined that takes two values and returns the most precise partially known structure that both values conform with.

```
(define (f_p v_1 ... v_n)
  (if (or (unknown? v_1) ...)
      'unknown
      (p v_1 ... v_n)))
(define (lub v_1 v_2)
  (if (equal? v_1 v_2)
      v_1
      (if (and v_1 is (c_1 x_1 ... x_i)
               v_2 is (c_2 y_1 ... y_j)
               c_1 = c_2
               i = j)
          (c_1 (lub x_1 y_1) ... (lub x_i y_i))
          'unknown)))
```

Also, the timing 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 normally, otherwise, the maximum of the times of both branches is computed. Transformation $\mathcal{C}$, given in Figure 5, embodies these algorithms, where $\mathcal{C}_e$ transforms an expression in the original functions, and $\mathcal{C}_t$ transforms an expression in the timing functions. The variable uf denotes function f extended with the value *unknown*, and the variable tbf denotes the time-bound function for f.

$$
R_{\mathcal{C}0}:\ \mathcal{C}\left[\!\!\left[\begin{array}{l}
(\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \dots\ v_{1_k})\\
\quad exp_1)\\
(\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \dots\ v_{2_k})\\
\quad exp_2)\\
\qquad\vdots\\
(\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \dots\ v_{n_k})\\
\quad exp_n)\\
(\textbf{define}\ (tf_1\ v_{1_1}\ v_{1_2}\ \dots\ v_{1_k})\\
\quad exp'_1)\\
(\textbf{define}\ (tf_2\ v_{2_1}\ v_{2_2}\ \dots\ v_{2_k})\\
\quad exp'_2)\\
\qquad\vdots\\
(\textbf{define}\ (tf_n\ v_{n_1}\ v_{n_2}\ \dots\ v_{n_k})\\
\quad exp'_n)
\end{array}\right]\!\!\right]
=
\begin{array}{l}
(\textbf{define}\ (uf_1\ v_{1_1}\ v_{1_2}\ \dots\ v_{1_k})\\
\quad \mathcal{C}_e[\![exp_1]\!])\\
(\textbf{define}\ (uf_2\ v_{2_1}\ v_{2_2}\ \dots\ v_{2_k})\\
\quad \mathcal{C}_e[\![exp_2]\!])\\
\qquad\vdots\\
(\textbf{define}\ (uf_n\ v_{n_1}\ v_{n_2}\ \dots\ v_{n_k})\\
\quad \mathcal{C}_e[\![exp_n]\!])\\
(\textbf{define}\ (tbf_1\ v_{1_1}\ v_{1_2}\ \dots\ v_{1_k})\\
\quad \mathcal{C}_t[\![exp'_1]\!])\\
(\textbf{define}\ (tbf_2\ v_{2_1}\ v_{2_2}\ \dots\ v_{2_k})\\
\quad \mathcal{C}_t[\![exp'_2]\!])\\
\qquad\vdots\\
(\textbf{define}\ (tbf_n\ v_{n_1}\ v_{n_2}\ \dots\ v_{n_k})\\
\quad \mathcal{C}_t[\![exp'_n]\!])
\end{array}
$$

$$
\begin{array}{ll}
R_{\mathcal{C}_e1}: \mathcal{C}_e[\![v]\!] & = v\\
R_{\mathcal{C}_e2}: \mathcal{C}_e[\![(c\ e_1\ ...\ e_n)]\!] & = (c\ \mathcal{C}_e[\![e_1]\!]\ ...\ \mathcal{C}_e[\![e_n]\!])\\
R_{\mathcal{C}_e3}: \mathcal{C}_e[\![(p\ e_1\ ...\ e_n)]\!] & = (f_p\ \mathcal{C}_e[\![e_1]\!]\ ...\ \mathcal{C}_e[\![e_n]\!])\\
R_{\mathcal{C}_e4}: \mathcal{C}_e[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] & = (\textbf{let}\ ((v\ \mathcal{C}_e[\![e_1]\!]))\\
 & \quad (\textbf{if}\ (unknown?\ v)\\
 & \qquad (lub\ \mathcal{C}_e[\![e_2]\!]\ \mathcal{C}_e[\![e_3]\!])\\
 & \qquad (\textbf{if}\ v\ \mathcal{C}_e[\![e_2]\!]\ \mathcal{C}_e[\![e_3]\!])))\\
R_{\mathcal{C}_e5}: \mathcal{C}_e[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] & = (\textbf{let}\ ((v\ \mathcal{C}_e[\![e_1]\!]))\ \mathcal{C}_e[\![e_2]\!])\\
R_{\mathcal{C}_e6}: \mathcal{C}_e[\![(f\ e_1\ ...\ e_n)]\!] & = (uf\ \mathcal{C}_e[\![e_1]\!]\ ...\ \mathcal{C}_e[\![e_n]\!])\\
\\
R_{\mathcal{C}_t1}: \mathcal{C}_t[\![T]\!] & = T\\
R_{\mathcal{C}_t2}: \mathcal{C}_t[\![(+\ e_1\ ...\ e_n)]\!] & = (+\ \mathcal{C}_t[\![e_1]\!]\ ...\ \mathcal{C}_t[\![e_n]\!])\\
R_{\mathcal{C}_t3}: \mathcal{C}_t[\![(\textbf{if}\ e_1\ e_2\ e_3]\!] & = (\textbf{let}\ ((v\ \mathcal{C}_e[\![e_1]\!]))\\
 & \quad (\textbf{if}\ (unknown?\ v)\\
 & \qquad (max\ \mathcal{C}_t[\![e_2]\!]\ \mathcal{C}_t[\![e_3]\!])\\
 & \qquad (\textbf{if}\ v\ \mathcal{C}_t[\![e_2]\!]\ \mathcal{C}_t[\![e_3]\!])))\\
R_{\mathcal{C}_t4}: \mathcal{C}_t[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] & = (\textbf{let}\ ((v\ \mathcal{C}_t[\![e_1]\!]))\ \mathcal{C}_t[\![e_2]\!])\\
R_{\mathcal{C}_t5}: \mathcal{C}_t[\![(tf\ e_1\ ...\ e_n)]\!] & = (tbf\ \mathcal{C}_t[\![e_1]\!]\ ...\ \mathcal{C}_t[\![e_n]\!])
\end{array}
$$

FIGURE 5. Rules for Time-Bound transformation $\mathcal{C}$

Applying this transformation on functions *least* and *tleast* yields functions *uleast* and *tbleast* in Figure 6, where function f_p for each primitive operator p and function *lub* are as given above.

```
(define (uleast x)
  (let ((v0 (f_null? (f_cdr x))))
    (if (unknown? v0)
        (lub (f_car x)
          (let ((s (uleast (f_cdr x))))
            (let ((v1 (f_<= (f_car x) s)))
              (if (unknown? v1)
                  (lub (f_car x) s)
                  (if v1 (f_car x) s))))))
        (if v0
            (f_car x)
            (let ((s (uleast (f_cdr x))))
              (let ((v1 (f_<= (f_car x) s)))
                (if (unknown? v1)
                    (lub (f_car x) s)
                    (if v1 (f_car x) s))))))))
(define (tbleast x)
  (+ T_if T_null? T_cdr T_varref
    (let ((v2 (f_null? (f_cdr x))))
      (if (unknown? v2)
          (max (+ T_car T_varref)
            (+ T_let T_call T_cdr T_varref (tbleast (cdr x))
              (let ((s (uleast (f_cdr x))))
                (+ T_if T_<= T_car T_varref T_varref
                  (let ((v3 (f_<= (f_car x) s)))
                    (if (unknown? v3)
                        (max (+ T_car T_varref) T_varref)
                        (if v3 (+ T_car T_varref) T_varref))))))))
          (if v2
              (+ T_car T_varref)
              (+ T_let T_call T_cdr T_varref (tbleast (cdr x))
                (let ((s (uleast (f_cdr x))))
                  (+ T_if T_<= T_car T_varref T_varref
                    (let ((v3 (f_<= (f_car x) s)))
                      (if (unknown? v3)
                          (max (+ T_car T_varref) T_varref)
                          (if v3 (+ T_car T_varref) T_varref)))))))))))
```

FIGURE 6. Function *least* after Time-Bound transformation $\mathcal{C}$.

The resulting time-bound function takes as arguments partially known input structures, such as $list(n)$, which take as arguments input parameters, such as n.

Therefore, the resulting function takes as arguments input parameters and primitive parameters and computes the most accurate time bound possible.

Both transformations $\mathcal{T}$ and $\mathcal{C}$ take linear time in terms of the size of the program, so they are extremely efficient, as also seen in our prototype system ALPA. Note that the resulting time-bound function may not terminate, but this occurs only if the recursive structure of the original program depends on unknown parts in the partially known input structure. As a trivial example, if partially known input structure given is *unknown*, then the corresponding time-bound function for any recursive function does not terminate, since the original program does take infinite time in the worst case.

3. Optimizing time-bound functions

This section describes symbolic evaluation and optimizations that make computation of time bounds more efficient. The transformations consist of partial evaluation, realized as global inlining, and incremental computation, realized as local optimization.

The 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.

3.1. 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

$$
\begin{array}{ll}
R_{\mathcal{E}1} : \mathcal{E}\,[\![v]\!]\,\rho & = \rho(v) \text{ look up binding in environment} \\
R_{\mathcal{E}2} : \mathcal{E}\,[\![T]\!]\,\rho & = T \\
R_{\mathcal{E}3} : \mathcal{E}\,[\![(c\ e_1\ ...\ e_n)]\!]\,\rho & = (c\ \mathcal{E}\,[\![e_1]\!]\,\rho\ ...\ \mathcal{E}\,[\![e_n]\!]\,\rho) \\
R_{\mathcal{E}4} : \mathcal{E}\,[\![(p\ e_1\ ...\ e_n)]\!]\,\rho & = (p\ \mathcal{E}\,[\![e_1]\!]\,\rho\ ...\ \mathcal{E}\,[\![e_n]\!]\,\rho) \\
R_{\mathcal{E}5} : \mathcal{E}\,[\![(+\ e_1\ ...\ e_n)]\!]\,\rho & = (symbAdd\ \mathcal{E}\,[\![e_1]\!]\,\rho\ ...\ \mathcal{E}\,[\![e_n]\!]\,\rho) \\
R_{\mathcal{E}6} : \mathcal{E}\,[\![(max\ e_1\ ...\ e_n)]\!]\,\rho & = (symbMax\ \mathcal{E}\,[\![e_1]\!]\,\rho\ ...\ \mathcal{E}\,[\![e_n]\!]\,\rho) \\
R_{\mathcal{E}7} : \mathcal{E}\,[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!]\,\rho & = \mathcal{E}\,[\![e_2]\!]\,\rho \text{ if } \mathcal{E}\,[\![e_1]\!]\,\rho = true \\
 & \quad \mathcal{E}\,[\![e_3]\!]\,\rho \text{ if } \mathcal{E}\,[\![e_1]\!]\,\rho = false \\
R_{\mathcal{E}8} : \mathcal{E}\,[\![(\mathbf{let}\ ((v\ e_1))\ e_2)]\!]\,\rho & = \mathcal{E}\,[\![e_2]\!]\,\rho[v \mapsto \mathcal{E}\,[\![e_1]\!]\,\rho] \text{ bind } v \text{ in environment} \\
R_{\mathcal{E}9} : \mathcal{E}\,[\![(f\ e_1\ ...\ e_n)]\!]\,\rho & = e[v_1 \mapsto \mathcal{E}\,[\![e_1]\!]\,\rho, ..., v_n \mapsto \mathcal{E}\,[\![e_n]\!]\,\rho] \\
 & \quad \text{where } f \text{ is defined by } f(v_1, ..., v_n) = e
\end{array}
$$

FIGURE 7. Rules for symbolic evaluation of programs

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, and 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 in Figure 7 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 unknown input structure, e.g., $mergesort(list(250))$, and it starts with an empty environment. Assume *symbAdd* is a function that symbolically sums its arguments, and $symbMax$ is a function that symbolically takes the maximum of its arguments.

As an example, applying symbolic evaluation to *tbleast* on a list of size 100, we obtain the following result:

$$\begin{aligned} tbleast(list(100)) = \ & 497 * T_{varref} + 100 * T_{null?} + 199 * T_{car} + 199 * T_{cdr} \\ & +99 * T_{\leq} + 199 * T_{if} + 99 * T_{let} + 99 * T_{call} \end{aligned}$$

Table 1 gives the results of symbolic evaluation of the timing functions for other example programs on inputs of various sizes. The last column lists the sums for every rows. All numbers are exact symbolic counts. They are verified by using a modified evaluator.

This symbolic evaluation is exactly a specialized partial evaluation. It is fully automatic and computes the most accurate time bound possible with respect to the given program structure. It always terminates as long as the time-bound function terminates.

The symbolic evaluation given only values of input parameters is inefficient compared to direct evaluation given values of both input parameters and particular primitive parameters, but the resulting function takes virtually constant time given any values of primitive parameters. For example, directly evaluating a quadratic-time reverse function (that uses append) on input of size 20 takes about 0.96 milliseconds, whereas the symbolic evaluation takes 670 milliseconds, hundreds of times slower. However, the resulting function can be evaluated in virtually no time given values of primitive parameters measured for any underlying systems. I propose further optimizations below that greatly speed up the symbolic evaluation.

3.2. 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

TABLE 1. Results of symbolic evaluation of time-bound functions (exact counts) for a functional language.

example	size	varref	nil	cons	null?	car	cdr	$\leq$	if	let	call
insertion sort	10	321	11	55	66	100	55	45	111	0	65
	20	1241	21	210	231	400	210	190	421	0	230
	50	7601	51	1275	1326	2500	1275	1225	2551	0	1325
	100	30201	101	5050	5151	10000	5050	4950	10101	0	5150
	200	120401	201	20100	20301	40000	20100	19900	40201	0	20300
	300	270601	301	45150	45451	90000	45150	44850	90301	0	45450
	500	751001	501	125250	125751	250000	125250	124750	250501	0	125750
	1000	3002001	1001	500500	501501	1000000	500500	499500	1001001	0	501500
	2000	12004001	2001	2001000	2003001	4000000	2001000	1999000	4002001	0	2003000
selection sort	10	576	11	55	121	190	200	90	211	55	120
	20	2251	21	210	441	780	800	380	821	210	440
	50	13876	51	1275	2601	4950	5000	2450	5051	1275	2600
	100	55251	101	5050	10201	19900	20000	9900	20101	5050	10200
	200	220501	201	20100	40401	79800	80000	39800	80201	20100	40400
	300	495751	301	45150	90601	179700	180000	89700	180301	45150	90600
	500	1376251	501	125250	251001	499500	500000	249500	500501	125250	251000
	1000	5502501	1001	500500	1002001	1999000	2000000	999000	2001001	500500	1002000
	2000	22005001	2001	2001000	4004001	7998000	8000000	3998000	8002001	2001000	4004000
merge-sort	10	456	28	69	192	119	112	25	217	0	138
	20	1154	58	177	468	315	284	69	537	0	340
	50	3680	148	573	1440	1047	908	237	1677	0	1054
	100	8562	298	1345	3284	2491	2116	573	3857	0	2412
	200	19526	598	3089	7372	5779	4832	1345	8717	0	5428
	300	31354	898	4977	11748	9355	7764	2189	13937	0	8660
	500	56354	1498	8977	20948	16955	13964	3989	24937	0	15460
	1000	124710	2998	19953	45900	37907	30928	8977	54877	0	33924
	2000	273422	5998	43905	99804	83811	67856	19953	119757	0	73852
set union	10	582	10	10	121	120	110	100	231	10	120
	20	2162	20	20	441	440	420	400	861	20	440
	50	12902	50	50	2601	2600	2550	2500	5151	50	2600
	100	50802	100	100	10201	10200	10100	10000	20301	100	10200
	200	201602	200	200	40401	40400	40200	40000	80601	200	40400
	300	452402	300	300	90601	90600	90300	90000	180901	300	90600
	500	1254002	500	500	251001	251000	250500	250000	501501	500	251000
	1000	5008002	1000	1000	1002001	1002000	1001000	1000000	2003001	1000	1002000
	2000	20016002	2000	2000	4004001	4004000	4002000	4000000	8006001	2000	4004000
list reversal	10	43	1	10	11	10	10	0	11	0	11
	20	83	1	20	21	20	20	0	21	0	21
	50	203	1	50	51	50	50	0	51	0	51
	100	403	1	100	101	100	100	0	101	0	101
	200	803	1	200	201	200	200	0	201	0	201
	300	1203	1	300	301	300	300	0	301	0	301
	500	2003	1	500	501	500	500	0	501	0	501
	1000	4003	1	1000	1001	1000	1000	0	1001	0	1001
	2000	8003	1	2000	2001	2000	2000	0	2001	0	2001
reversal with app	10	231	11	55	66	55	55	0	66	0	65
	20	861	21	210	231	210	210	0	231	0	230
	50	5151	51	1275	1326	1275	1275	0	1326	0	1325
	100	20301	101	5050	5151	5050	5050	0	5151	0	5150
	200	80601	201	20100	20301	20100	20100	0	20301	0	20300
	300	180901	301	45150	45451	45150	45150	0	45451	0	45450
	500	501501	501	125250	125751	125250	125250	0	125751	0	125750
	1000	2003001	1001	500500	501501	500500	500500	0	501501	0	501500
	2000	8006001	2001	2001000	2003001	2001000	2001000	0	2003001	0	2003000

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 $\mathcal{S}$, given in Figure 8, performs this optimization. Variable $vtbf$ denotes the transformed time-bound function of f that operates on vectors.

$$
\begin{array}{lll}
\text{program:} & \mathcal{S}\left[\!\!\left[\begin{array}{l}(\textbf{define}\ (tbf_1\ v_{1_1}\ ...\ v_{1_k})\\ \quad exp_1)\\ (\textbf{define}\ (tbf_2\ v_{2_1}\ ...\ v_{2_k})\\ \quad exp_2)\\ \qquad\vdots\\ (\textbf{define}\ (tbf_n\ v_{n_1}\ ...\ v_{n_k})\\ \quad exp_n)\end{array}\right]\!\!\right] & = \begin{array}{l}(\textbf{define}\ (vtbf_1\ v_{1_1}\ ...\ v_{1_k})\\ \quad \mathcal{S}_t\,[\![exp_1]\!])\\ (\textbf{define}\ (vtbf_2\ v_{2_1}\ ...\ v_{2_k})\\ \quad \mathcal{S}_t\,[\![exp_2]\!])\\ \qquad\vdots\\ (\textbf{define}\ (vtbf_n\ v_{n_1}\ ...\ v_{n_k})\\ \quad \mathcal{S}_t\,[\![exp_n]\!])\end{array}\\
\begin{array}{l}\text{primitive}\\ \text{parameter:}\end{array} & \mathcal{S}_t\,[\![T]\!] & = \begin{array}{l}\text{create a vector of 0's except with the}\\ \text{component corresponding to } T \text{ set to 1}\end{array}\\
\text{summation:} & \mathcal{S}_t\,[\![(+\ e_1\ ...\ e_n)]\!] & = \begin{array}{l}\text{component-wise summation of all the}\\ \text{vectors among } \mathcal{S}_t\,[\![e_1]\!]\,, ..., \mathcal{S}_t\,[\![e_n]\!]\end{array}\\
\text{maximum:} & \mathcal{S}_t\,[\![(max\ e_1\ ...\ e_n)]\!] & = \begin{array}{l}\text{component-wise maximum of all the}\\ \text{vectors among } \mathcal{S}_t\,[\![e_1]\!]\,, ..., \mathcal{S}_t\,[\![e_n]\!]\end{array}\\
\text{all other:} & \mathcal{S}_t\,[\![e]\!] & = e
\end{array}
$$

FIGURE 8. Transformation $\mathcal{S}$ to optimize repeated summations

Let V be the following vector of primitive parameters:

$$\langle T_{varref}, T_{nil}, T_{cons}, T_{null?}, T_{car}, T_{cdr}, T_{\leq}, T_{if}, T_{let}, T_{call}\rangle$$

Applying the above transformation on function *tbleast* yields function *vtbleast* shown in Figure 9, where components of the vectors correspond to the components of V.

The time-bound function $tbleast(x)$ is simply the dot product of $vtbleast(x)$ and V.

This transformation incrementalizes the computation over recursions to avoid repeated summation. Again, this is fully automatic and takes time linear in terms of the size of the cost-bound function.

The result of this optimization is dramatic. For example, optimized symbolic evaluation of the same quadratic-time reverse takes only 2.55 milliseconds, while direct evaluation takes 0.96 milliseconds, resulting in less than 3 times slow-down. Table 2

$$
\begin{array}{l}
(\textbf{define}\ (\mathit{vtbleast}\ x) \\
\hspace{1em}(+\ \langle 1,0,0,1,0,1,0,1,0,0\rangle \\
\hspace{2em}(\textbf{let}\ ((v_0\ (f_{\mathit{null?}}\ (f_{\mathit{cdr}}\ x)))) \\
\hspace{3em}(\textbf{if}\ (\mathit{unknown?}\ v_0) \\
\hspace{4em}(\mathit{max}\ \langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{5em}(+\ \langle 1,0,0,0,0,1,0,0,1,1\rangle \\
\hspace{6em}(\mathit{vtbleast}\ (\mathit{cdr}\ x)) \\
\hspace{6em}(\textbf{let}\ ((s\ (\mathit{uleast}\ (f_{\mathit{cdr}}\ x)))) \\
\hspace{7em}(+\ \langle 2,0,0,0,1,0,1,1,0,0\rangle \\
\hspace{8em}(\textbf{let}\ ((v_1\ (f_{\leq}\ (f_{\mathit{car}}\ x)\ s))) \\
\hspace{9em}(\textbf{if}\ (\mathit{unknown?}\ v_1) \\
\hspace{10em}\langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{10em}(\textbf{if}\ v_1 \\
\hspace{11em}\langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{11em}\langle 1,0,0,0,0,0,0,0,0,0\rangle))))))) \\
\hspace{4em}(\textbf{if}\ v_0 \\
\hspace{5em}\langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{5em}(+\ \langle 1,0,0,0,0,1,0,0,1,1\rangle \\
\hspace{6em}(\mathit{vtbleast}\ (\mathit{cdr}\ x)) \\
\hspace{6em}(\textbf{let}\ ((s\ (\mathit{uleast}\ (f_{\mathit{cdr}}\ x)))) \\
\hspace{7em}(+\ \langle 2,0,0,0,1,0,1,1,0,0\rangle \\
\hspace{8em}(\textbf{let}\ ((v_1\ (f_{\leq}\ (f_{\mathit{car}}\ x)\ s))) \\
\hspace{9em}(\textbf{if}\ (\mathit{unknown?}\ v_1) \\
\hspace{10em}\langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{10em}(\textbf{if}\ v_1 \\
\hspace{11em}\langle 1,0,0,0,1,0,0,0,0,0\rangle \\
\hspace{11em}\langle 1,0,0,0,0,0,0,0,0,0\rangle))))))))))))
\end{array}
$$

FIGURE 9. Function *tbleast* after transformation $\mathcal{S}$.

compares the times of direct evaluation of timing functions, with each primitive parameter set to 1, and the times of optimized symbolic evaluation, obtaining the exact symbolic counts as in Figure 1. These measurements are taken on a Sun Ultra 1 with 167MHz CPU and 64MB memory. They include garbage-collection time. The times without garbage-collection times are all about 1% faster, so they are not shown here.

For merge sort, it takes several days for inputs of size 50 or larger. A special but simple optimization can be done, and resulting symbolic evaluation takes only seconds. For all other examples, it takes at most 2.7 hours. Note that, 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. Again, after the symbolic evaluation, time bounds can be computed in virtually no time given primitive parameters measured on any machines.

TABLE 2. Times of direct evaluation vs. optimized symbolic evaluation (in milliseconds).

size	insertion sort		selection sort		merge sort		set union		list reversal		reversal w/app.	
	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic
10	0.49328	1.89057	0.71550	3.04985	1.43136	14.6666	1.44601	4.28571	0.0113	0.1391	0.25637	1.32877
20	1.93942	4.79452	3.89051	14.2352	605.714	8500.00	5.02935	10.6274	0.0211	0.2649	0.96215	2.55132
50	56.6666	87.4193	46.6666	106.451	xxxxxx	xxxxxx	134.516	192.666	0.0498	0.6422	23.2283	44.1269
100	451.428	557.142	338.571	571.428	xxxxxx	xxxxxx	1026.66	1176.66	0.0973	1.2603	178.000	231.333
500	58240.0	58080.0	39480.0	46050.0	xxxxxx	xxxxxx	125910.	117240.	0.5030	6.2426	21540.0	22180.0
2000	4024730	4039860	2666290	2761410	xxxxxx	xxxxxx	9205680	9690370	3.6070	27.401	1810280	1711650

4. Making time-bound functions accurate

While loops and recursions affect time bounds most, the accuracy of the time bounds calculated also depends on the handling of the conditionals in the original program, which is reflected in the time-bound function. For conditionals whose test results are known to be true or false at the symbolic-evaluation time, the appropriate branch is chosen; so other branches, which may even take longer, are not considered for the worst-case time. This is a major source of accuracy for our worst-case bound.

For conditionals whose test results are not known at symbolic-evaluation time, we need to take the maximum time among all alternatives. The only case in which this would produce inaccurate time bound is when the test in a conditional in one subcomputation implies the test in a conditional in another subcomputation. For example, consider an expression e_0 whose value is *unknown* and

$$e_1 = (\textbf{if}\ e_0\ \mathsf{1}\ (\textit{fibonacci}\ \mathsf{1000}))$$

$$e_2 = (\textbf{if}\ e_0\ (\textit{fibonacci}\ \mathsf{2000})\ \mathsf{2})$$

If we compute the time bound for $(+\ e_1\ e_2)$ directly, the result is at least $(+\ (\textit{tfibonacci}\ \mathsf{1000})\ (\textit{tfibonacci}\ \mathsf{2000}))$. However, if we consider only the two realizable execution paths, we know that the worst case is $(\textit{tfibonacci}\ \mathsf{2000})$ plus some small constants. This is known as the false-path elimination problem [**2**].

Two transformations, *lifting conditions* and *simplifying conditionals*, allow us to achieve the accurate analysis results above. In each function definition, the former lifts conditions to the out-most scope that the test does not depend on, and the latter simplifies conditionals according to the lifted condition. For $e_1 + e_2$ in the above example, lifting the condition for e_1, we obtain

$$(\textbf{if}\ e_0\ (+\ \mathsf{1}\ e_2)\ (+\ (\textit{fibonacci}\ \mathsf{1000})\ e_2))$$

Simplifying the conditionals in the two occurrences of e_2 to (*fibonacci* 2000) and 2, respectively, we obtain

$$(\textbf{if}\ e_0\ (+\ \mathsf{1}\ (\textit{fibonacci}\ \mathsf{2000}))\ (+\ (\textit{fibonacci}\ \mathsf{1000})\ \mathsf{2}))$$

To facilitate these transformations, we inline all function calls where the function is not defined recursively.

The power of these transformations depends on reasonings used in simplifying the conditionals, as have been studied in many program transformation methods [**101, 89, 94, 34, 68**]. At least syntactic equality can be used, which identifies the most obvious source of inaccuracy. These optimizations also speed up the symbolic evaluation, since now obviously infeasible execution paths are not searched.

5. Implementation and experimentation

I have implemented the analysis approach in a prototype system, ALPA (Automatic Language-based Performance Analyzer). I performed a large number of measurements and obtained encouraging good results. I also used the system to obtain the exact symbolic counts and the performance measurements shown in Section 3.

The implementation is for a subset of Scheme. An editor for the source programs is implemented using the Synthesizer Generator [**83**], and thus we can easily change the syntax for the source programs. For example, the current implementation supports both the syntax used in this Chapter and the syntax used in [**66**].

Time-bound functions are constructed using SSL, a simple functional language used in the Synthesizer Generator. Lifting conditions, simplifying conditionals, and inlining non-recursive calls are also implemented in SSL; they can be applied on the source program before constructing the time-bound function. The symbolic evaluation and optimizations, as well as measurements of primitive parameters, are written in Scheme. The measurements and analyses are performed for source programs compiled with Chez Scheme compiler [**24**]. The particular numbers below are taken on a Sun Ultra 1 with 167MHz UltraSPARC CPU and 64MB main memory, but the analysis were performed for several other kinds of SPARC stations, and the results are similar.

I tried to avoid compiler optimizations by setting the optimization level to 0. To handle garbage-collection time, I performed two sets of experiments: one set excludes garbage-collection times in both calculations and measurements, while the other includes them in both. The source program does not use any library; in particular, no numbers are large enough to trigger the bignum implementation of Chez Scheme. Our current system does not handle the effects of cache memory or instruction pipelining; thus I tried to avoid producing large data in the example programs to minimize possible cache effects.

Since the minimum running time of a program construct is about 0.1 microseconds, and the precision of the timing function is 10 milliseconds, I 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.

Table A.1 shows the calculated and measured worst-case times for six example programs on inputs of size 10 to 2000. For the set union example, we used inputs where both arguments were of the given sizes. These times do not include garbage-collection times. The item me/ca is the measured time expressed as a percentage of the calculated time. In general, all measured times are closely bounded by the calculated times (with about 90-95% accuracy) except when inputs are extremely small (10 or 20, in 1 case) or extremely large (2000, in 3 cases), which is analyzed below.

Table A.2 shows the calculated and measured worst-case times that include garbage-collection times. The results are similar to whose when garbage-collection times are excluded, except that the percentages are consistently higher than in Table A.1. In particular, underestimations occur more often for extremely small inputs, for inputs of size 1000 as well as 2000 on some examples, and for a few other inputs (about 1-2%, in 2 cases). We believe that this is the effect of garbage collection, which we have only measured in general but not analyzed specifically.

In general, the measured worst-case times are closely bounded by calculated upper bounds for all inputs of medium sizes (up to 500 for measurements including garbage-collection time, up to 1000 excluding garbage-collection time, and even larger for faster programs or programs that use less space). Figure 10 depicts the numbers in Table A.1. Examples such as sorting are classified as complex examples in previous study [**76, 62**], where calculated time is as much as 67% higher than measured time, and where only the result for one sorting program on a single input (of size 10 [**76**] or 20 [**62**]) is reported in each experiment.

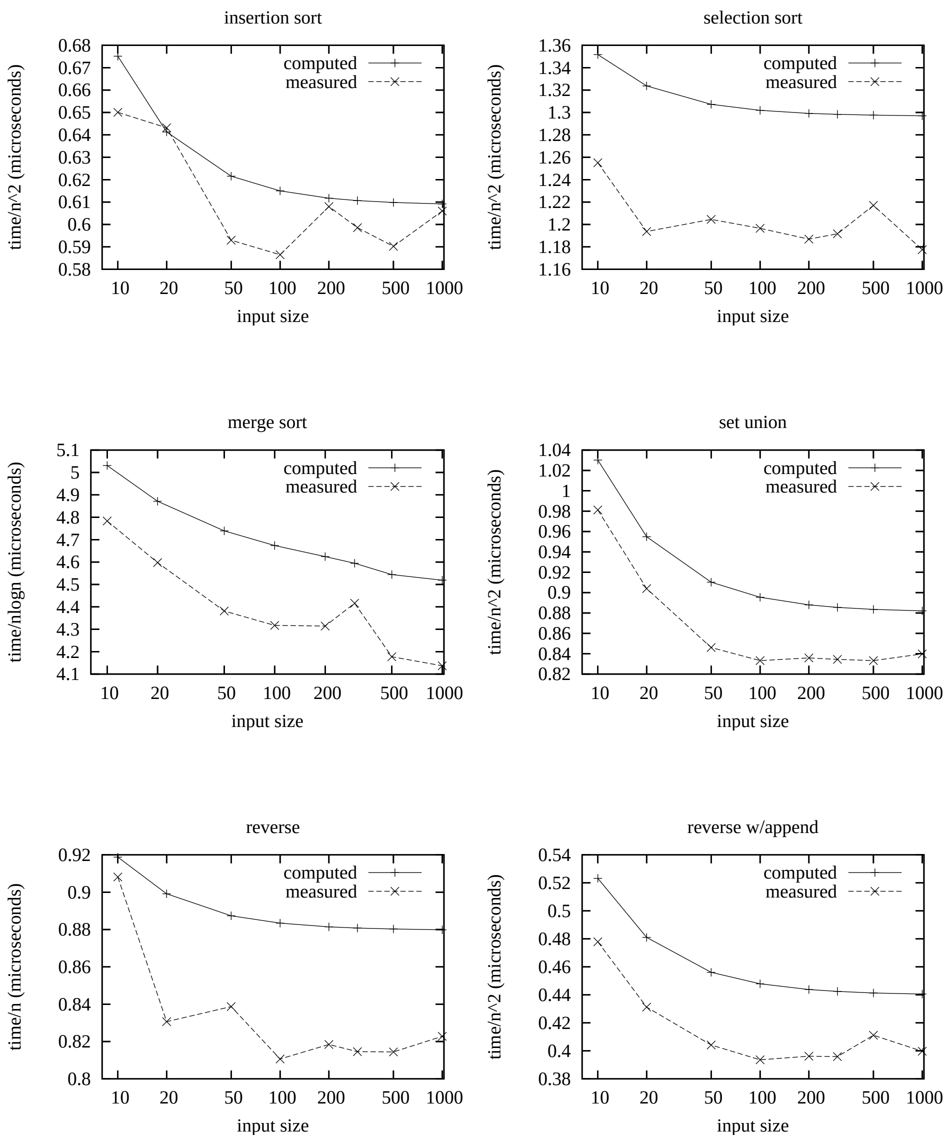

FIGURE 10. Comparison of calculated and measured worst-case times for the functional language, without garbage collection.

We found that when inputs are extremely small (10 or 20), the measured time is occasionally above the calculated time for some examples. Also, when inputs are large (1000 for measurements including garbage-collection time, or 2000 excluding garbage-collection time), the measured times for some examples are above the calculated time.

We attribute these to cache memory effects, and this is further confirmed by measuring programs, such as Cartesian product, that use extremely large amount of space even on small inputs (50-200); for example, on input of size 200, the measured time is 65% higher than the calculated time. While this shows that cache effects need to be considered for larger applications, it also helps validate that our calculated results are accurate relative to our current model.

Among fifteen programs we have analyzed using ALPA, two of them did not terminate. One is quick sort, and the other is a contrived variation of sorting; both diverge because the recursive structure for splitting a list depends on the values of unknown list elements. We have found a different symbolic-evaluation strategy that uses a kind of incremental path selection, and the evaluation would terminate for both examples, as well as all other examples, giving accurate worst-case bounds. We are implementing that algorithm. We also noticed that static analysis can be exploited to identify sources of nontermination.

CHAPTER 3

Analysis of an Imperative Language

This chapter extend the time-bound analysis of functional programs to include assignment and vectors.

In order to analyze programs in the presence of assignments we transform the imperative program into a functional one using Storage Passing Style (SPS). We then do the analysis of this new functional program with the technique from the previous chapter, but with a few necessary changes, to cover the fact that now a store is a value and it has to be dealt with appropriately.

1. Language definition

We use the same language described in Chapter 2, extended with an assignment expression, a loop expression, a sequencing expression and with side-effecting primitives (vector and record update). Its syntax is given by the grammar in Figure 1, and its semantics corresponds to the appropriate subset of Scheme [**58, 25**] where the loop expression is defined as in Figure 2. For example, the program in Figure 3 adds all the elements in a vector of numbers. For ease of analysis and transformation, we assume that a preprocessor gives a distinct name to each bound variable.

We use *vector-sum* as a small running example.

2. Converting the imperative program to a functional program

2.1. Assignment elimination. We first transform the original program to eliminate assignments, in order to avoid having a mutable environment. This way only

$$
\begin{array}{lcll}
\textit{program} & ::= & (\textbf{define}\ (f_1\ v_{1_1}\ \ldots\ v_{1_n})\ e_1) & \\
 & & \quad\cdots & \\
 & & (\textbf{define}\ (f_m\ v_{m_1}\ \ldots\ v_{m_n})\ e_m) & \\
e & ::= & v & \text{variable reference} \\
 & | & (c\ e_1\ \ldots\ e_n) & \text{data construction} \\
 & | & (p\ e_1\ \ldots\ e_n) & \text{primitive operation} \\
 & | & (\textbf{if}\ e_1\ e_2\ e_3) & \text{conditional expression} \\
 & | & (\textbf{set!}\ v\ e) & \text{assignment expression} \\
 & | & (\textbf{let}\ ((v\ e_1))\ e_2) & \text{binding expression} \\
 & | & (\textbf{begin}\ e_1\ e_2) & \text{sequencing} \\
 & | & (\textbf{while}\ e_1\ e_2) & \text{loop} \\
 & | & (f\ e_1\ \ldots\ e_n) & \text{function application}
\end{array}
$$

FIGURE 1. Definition of the imperative language.

```
(define-syntax while
  (syntax-rules ()
    ((while test body)
     (let loop ()
       (if test
           (begin body (loop))
           'void)))))
```

FIGURE 2. Definition of the while expression in Scheme.

```
(define (vector-sum v)
  (let ([sum 0])
    (let ([i 0])
      (begin
        (while (< i (vector-length v))
          (begin
            (set! sum (+ sum (vector-ref v i)))
            (set! i (+ i 1))))
        sum))))
```

FIGURE 3. Program *vector-sum*, which returns the addition of all the numbers in a vector

the store is mutable, and it will greatly simplify the conversion to a functional program. In order to eliminate assignment, we first look for variables that are subject to assignment, using the algorithm shown in Figure 4. For each such variable, we change the definition to a vector, we change the references to vector references and we change the assignments to vector updates. This transformation is shown in Figure 5. For example, Figure 6 shows the program *vector-sum* after the assignment elimination step.

$$
\begin{array}{lll}
R_{\mathcal{T}_{sv}1} : \mathcal{T}_{sv}\,[\![v]\!] & = \emptyset \\
R_{\mathcal{T}_{sv}2} : \mathcal{T}_{sv}\,[\![(c\ e_1\ \ldots\ e_n)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \ldots \cup \mathcal{T}_{sv}\,[\![e_n]\!] \\
R_{\mathcal{T}_{sv}3} : \mathcal{T}_{sv}\,[\![(p\ e_1\ \ldots\ e_n)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \ldots \cup \mathcal{T}_{sv}\,[\![e_n]\!] \\
R_{\mathcal{T}_{sv}4} : \mathcal{T}_{sv}\,[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \mathcal{T}_{sv}\,[\![e_2]\!] \cup \mathcal{T}_{sv}\,[\![e_3]\!] \\
R_{\mathcal{T}_{sv}5} : \mathcal{T}_{sv}\,[\![(\mathbf{set!}\ v\ e)]\!] & = \{v\} \cup \mathcal{T}_{sv}\,[\![e]\!] \\
R_{\mathcal{T}_{sv}6} : \mathcal{T}_{sv}\,[\![(\mathbf{let}\ ((v\ e_1))\ e_2)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \mathcal{T}_{sv}\,[\![e_2]\!] \\
R_{\mathcal{T}_{sv}7} : \mathcal{T}_{sv}\,[\![(\mathbf{begin}\ e_1\ e_2)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \mathcal{T}_{sv}\,[\![e_2]\!] \\
R_{\mathcal{T}_{sv}8} : \mathcal{T}_{sv}\,[\![(\mathbf{while}\ e_1\ e_2)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \mathcal{T}_{sv}\,[\![e_2]\!] \\
R_{\mathcal{T}_{sv}9} : \mathcal{T}_{sv}\,[\![(f\ e_1\ \ldots\ e_n)]\!] & = \mathcal{T}_{sv}\,[\![e_1]\!] \cup \ldots \cup \mathcal{T}_{sv}\,[\![e_n]\!]
\end{array}
$$

FIGURE 4. Algorithm to find the variables subject to assignment. This algorithm assumes that each variable has a distinct name.

$$
R_{\mathcal{T}_{ae}0} : \mathcal{T}_{ae}\left[\!\!\left[\begin{array}{l}
(\mathbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\
\quad exp_1) \\
(\mathbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\
\quad exp_2) \\
\qquad \vdots \\
(\mathbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\
\quad exp_n)
\end{array}\right]\!\!\right] = \begin{array}{l}
(\mathbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\
\quad (\mathbf{let}\ ((v_{1_i}\ (\mathit{vector}\ v_{1_i}))\ \ldots\ ) \\
\qquad \mathcal{T}_{ae_e}\,[\![exp_1]\!]\,(\mathcal{T}_{sv}\,[\![exp_1]\!]))) \\
(\mathbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\
\quad (\mathbf{let}\ ((v_{2_i}\ (\mathit{vector}\ v_{2_i}))\ \ldots\ ) \\
\qquad \mathcal{T}_{ae_e}\,[\![exp_2]\!]\,(\mathcal{T}_{sv}\,[\![exp_2]\!]))) \\
\qquad \vdots \\
(\mathbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\
\quad (\mathbf{let}\ ((v_{n_i}\ (\mathit{vector}\ v_{n_i}))\ \ldots\ ) \\
\qquad \mathcal{T}_{ae_e}\,[\![exp_n]\!]\,(\mathcal{T}_{sv}\,[\![exp_n]\!])))
\end{array}
$$

$$
\begin{array}{lll}
R_{\mathcal{T}_{ae_e}1} : \mathcal{T}_{ae_e}\,[\![v]\!]\,s & = \begin{cases} v & , v \notin s \\ (\mathit{vector\text{-}ref}\ v\ 0), & v \in s \end{cases} \\
R_{\mathcal{T}_{ae_e}2} : \mathcal{T}_{ae_e}\,[\![(c\ e_1\ \ldots\ e_n)]\!]\,s & = (c\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \ldots\ \mathcal{T}_{ae_e}\,[\![e_n]\!]\,s) \\
R_{\mathcal{T}_{ae_e}3} : \mathcal{T}_{ae_e}\,[\![(p\ e_1\ \ldots\ e_n)]\!]\,s & = (p\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \ldots\ \mathcal{T}_{ae_e}\,[\![e_n]\!]\,s) \\
R_{\mathcal{T}_{ae_e}4} : \mathcal{T}_{ae_e}\,[\![(\mathbf{if}\ e_1\ e_2\ e_3)]\!]\,s & = (\mathbf{if}\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \mathcal{T}_{ae_e}\,[\![e_2]\!]\,s\ \mathcal{T}_{ae_e}\,[\![e_3]\!]\,s) \\
R_{\mathcal{T}_{ae_e}5} : \mathcal{T}_{ae_e}\,[\![(\mathbf{set!}\ v\ e)]\!]\,s & = (\mathit{vector\text{-}set!}\ v\ 0\ \mathcal{T}_{ae_e}\,[\![e]\!]\,s) \\
R_{\mathcal{T}_{ae_e}6} : \mathcal{T}_{ae_e}\,[\![(\mathbf{let}\ ((v\ e_1))\ e_2)]\!]\,s & = \begin{cases} (\mathbf{let}\ ((v\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s))\ \mathcal{T}_{ae_e}\,[\![e_2]\!]\,s) & , v \notin s \\ (\mathbf{let}\ ((v\ (\mathit{vector}\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s)))\ \mathcal{T}_{ae_e}\,[\![e_2]\!]\,s), & v \in s \end{cases} \\
R_{\mathcal{T}_{ae_e}7} : \mathcal{T}_{ae_e}\,[\![(\mathbf{begin}\ e_1\ e_2)]\!]\,s & = (\mathbf{begin}\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \mathcal{T}_{ae_e}\,[\![e_2]\!]\,s) \\
R_{\mathcal{T}_{ae_e}8} : \mathcal{T}_{ae_e}\,[\![(\mathbf{while}\ e_1\ e_2)]\!]\,s & = (\mathbf{while}\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \mathcal{T}_{ae_e}\,[\![e_2]\!]\,s) \\
R_{\mathcal{T}_{ae_e}9} : \mathcal{T}_{ae_e}\,[\![(f\ e_1\ \ldots\ e_n)]\!]\,s & = (f\ \mathcal{T}_{ae_e}\,[\![e_1]\!]\,s\ \ldots\ \mathcal{T}_{ae_e}\,[\![e_n]\!]\,s)
\end{array}
$$

FIGURE 5. Assignment elimination transformation

2.2. Lambda lifting step. Since our original functional language has no loops, we need to remove the loops using the lambda lifting technique [**54, 56, 78**]. Each loop will now be a function, with all the free variables as arguments. Figure 7 shows the program *vector-sum* after the lambda lifting step.

```
(define (vector-sum v)
  (let ([sum (vector 0)])
    (let ([i (vector 0)])
      (begin
        (while (< (vector-ref i 0) (vector-length v))
          (begin
            (vector-set! sum
              (+ (vector-ref sum 0) (vector-ref v (vector-ref i 0))))
            (vector-set! i (+ (vector-ref i 0) 1))))
        (vector-ref sum 0)))))
```

FIGURE 6. Program *vector-sum*, after the assignment elimination step

```
(define (vector-sum v)
  (let ([sum (vector 0)])
    (let ([i (vector 0)])
      (begin
        (vector-sum-loop0 sum i v)
        (vector-ref sum 0)))))
(define (vector-sum-loop0 sum i v)
  (if (< (vector-ref i 0) (vector-length v))
     (begin
       (vector-set! sum
         (+ (vector-ref sum 0) (vector-ref v (vector-ref i 0))))
       (vector-set! i (+ (vector-ref i 0) 1))
       (vector-sum-loop0 sum i v))
     'void))
```

FIGURE 7. Program *vector-sum*, after the lambda lifting step

2.3. Storage passing style step. The last step to make the program functional is to convert it to Storage Passing Style (SPS). With this transformation, every expression will now return two values: the original value and the store, which is a mapping from locations to values. We need to change all primitive and construction operations to accept a store as argument, and to return a new store along with the original value. We call this new primitives "store-aware" primitives and are named by prepending "sa-" to the name of the original primitive. For primitives that access the store, we use the fully persistent data structure techniques presented in [**21, 22**]. For the rest of the primitives, we just add a new argument *store* and return it unchanged, as shown in Figure 8 for a two argument primitive. In the following description, the expression $\langle exp_1\ exp_2\rangle$ is used for brevity of the presentation instead of the expression $(cons\ exp_1\ exp_2)$. Similarly, the expression $(\textbf{let}\ ((\langle v\ s\rangle\ exp_0))\ exp_1)$ is used instead of $(\textbf{let}\ ((tmp\ exp_0))\ (\textbf{let}\ ((v\ (car\ tmp)))\ (\textbf{let}\ ((s\ (cdr\ tmp)))\ exp_1)))$.

The transformation algorithm, given in Figure 9, is presented as a transformation $\mathcal{T}_{sps}$ on the original program, which calls a transformation $\mathcal{T}_{sps_e}$ to recursively transform subexpressions.

$$
\begin{array}{l}
(\textbf{define}\ (sa\text{-}quotient\ arg_1\ arg_2\ store)\\
\quad \langle (quotient\ arg_1\ arg_2)\ store\rangle)
\end{array}
$$

FIGURE 8. Redefinition of primitive *quotient* to accept and return a store argument

$$
\begin{array}{lll}
R_{\mathcal{T}_{sps}0}: \ \mathcal{T}_{sps}\left[\!\!\left[\begin{array}{l}(\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k})\\ \quad exp_1)\\ (\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k})\\ \quad exp_2)\\ \qquad\vdots\\ (\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k})\\ \quad exp_n)\end{array}\right]\!\!\right] & = & \begin{array}{l}(\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}\ store)\\ \quad \mathcal{T}_{sps_e}[\![exp_1]\!])\\ (\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}\ store)\\ \quad \mathcal{T}_{sps_e}[\![exp_2]\!])\\ \qquad\vdots\\ (\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}\ store)\\ \quad \mathcal{T}_{sps_e}[\![exp_n]\!])\end{array}\\
R_{\mathcal{T}_{sps_e}1}: \mathcal{T}_{sps_e}[\![v]\!] & = & \langle v\ store\rangle\\
R_{\mathcal{T}_{sps_e}2}: \mathcal{T}_{sps_e}[\![(c\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ store\rangle\ \mathcal{T}_{sps_e}[\![e_1]\!]))\\
 & & \quad\ddots\\
 & & \qquad(\textbf{let}\ ((\langle v_n\ store\rangle\ \mathcal{T}_{sps_e}[\![e_n]\!]))\\
 & & \qquad\quad(sa\text{-}c\ v_1\ \ldots\ v_n\ store))\cdots)\\
R_{\mathcal{T}_{sps_e}3}: \mathcal{T}_{sps_e}[\![(p\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ store\rangle\ \mathcal{T}_{sps_e}[\![e_1]\!]))\\
 & & \quad\ddots\\
 & & \qquad(\textbf{let}\ ((\langle v_n\ store\rangle\ \mathcal{T}_{sps_e}[\![e_n]\!]))\\
 & & \qquad\quad(sa\text{-}p\ v_1\ \ldots\ v_n\ store))\cdots)\\
R_{\mathcal{T}_{sps_e}4}: \mathcal{T}_{sps_e}[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] & = & (\textbf{let}\ ((\langle v\ store\rangle\ \mathcal{T}_{sps_e}[\![e_1]\!]))\\
 & & \quad(\textbf{if}\ v\ \mathcal{T}_{sps_e}[\![e_2]\!]\ \mathcal{T}_{sps_e}[\![e_3]\!]))\\
R_{\mathcal{T}_{sps_e}5}: \mathcal{T}_{sps_e}[\![(\textbf{begin}\ e_1\ e_2)]\!] & = & (\textbf{let}\ ((\langle ignored\ store\rangle\ \mathcal{T}_{sps_e}[\![e_1]\!]))\\
 & & \quad\mathcal{T}_{sps_e}[\![e_2]\!])\\
R_{\mathcal{T}_{sps_e}6}: \mathcal{T}_{sps_e}[\![(f\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ store\rangle\ \mathcal{T}_{sps_e}[\![e_1]\!]))\\
 & & \quad\ddots\\
 & & \qquad(\textbf{let}\ ((\langle v_n\ store\rangle\ \mathcal{T}_{sps_e}[\![e_n]\!]))\\
 & & \qquad\quad(f\ v_1\ \ldots\ v_n\ store))\cdots)
\end{array}
$$

FIGURE 9. Transformation $\mathcal{T}_{sps}$

Rule $R_{\mathcal{T}_{sps}0}$ adds a *store* argument to each function and it transforms the expressions recursively using $\mathcal{T}_{sps_e}$. Notice that the variable *store* should not be used by the original program, and it is the same variable through all the transformed program.

Rule $R_{\mathcal{T}_{spse}}1$ transforms a variable reference to a pair variable–store.

Rules $R_{\mathcal{T}_{spse}}2$, $R_{\mathcal{T}_{spse}}3$ and $R_{\mathcal{T}_{spse}}6$ evaluate the arguments from left to right, carrying the store from argument evaluation to argument evaluation, where $v_1 \dots v_n$ are fresh variables. At the end we just call the original primitive/constructor/function. Notice that the bindings of *store* will shadow all the previous bindings, making the new store the only accessible store.

Rule $R_{\mathcal{T}_{spse}}4$ evaluates the test expression, binding the value to a fresh variable v, and shadowing the binding for *store*.

Rule $R_{\mathcal{T}_{spse}}5$ evaluates the first expression in a sequence, ignores the value and uses the store to evaluate the second expression.

Applying transformation $\mathcal{T}_{sps}$to the program *vector-sum*, we obtain the function shown in Figure 10.

```
(define (vector-sum v store)
  (let ([⟨sum store⟩ (sa-vector 0 store)])
    (let ([⟨i store⟩ (sa-vector 0 store)])
      (let ([⟨ignored_0 store⟩ (vector-sum-loop_0 sum i v store)])
        (sa-vector-ref sum 0 store)))))
(define (vector-sum-loop_0 sum i v store)
  (let ([⟨v_0 store⟩ (sa-vector-ref i 0 store)])
    (let ([⟨v_1 store⟩ (sa-vector-length v store)])
      (let ([⟨v_2 store⟩ (sa-< v_0 v_1 store)])
        (if v_2
            (let ([⟨v_3 store⟩ (sa-vector-ref sum 0 store)])
              (let ([⟨v_4 store⟩ (sa-vector-ref i 0 store)])
                (let ([⟨v_5 store⟩ (sa-vector-ref v v_4 store)])
                  (let ([⟨v_6 store⟩ (sa-+ v_3 v_5 store)])
                    (let ([⟨ignored_1 store⟩ (sa-vector-set! sum v_6 store)])
                      (let ([⟨v_7 store⟩ (sa-vector-ref i 0 store)])
                        (let ([⟨v_8 store⟩ (sa-+ v_7 1 store)])
                          (let ([⟨ignored_2 store⟩ (sa-vector-set! i v_8 store)])
                            (vector-sum-loop_0 sum i v store)))))))))
            ⟨'void store⟩)))))
```

FIGURE 10. Program *vector-sum*, after transformation $\mathcal{T}_{sps}$.

3. Optimizing the SPS function

The resulting function is Storage Passing Style has now many tuple construction and destruction that can be avoided, to speed up the execution of the analysis. The idea is to create a tuple only when it is absolutely necessary to do so. We accomplish this by looking at each primitive and decide if its job is to modify the store, return a value, or both. For example, (*vector-ref* v n) only returns a value, (*vector-set!* v n e) only modifies the store, and (*vector* e_1 ...) both returns a value and modifies the store.

To avoid tuple creation, we modify the store-aware primitives such that value-returning procedures (*car*, *cdr*, *vector-ref*, *vector-length*, *null?*, *eq?*, $+$, $-$, $*$, $>$, $<$, $=$) return only the value, store-modifying procedures (*set-car!*, *set-cdr!*, *vector-set!*) return only a new store, and the rest (*cons*, *make-vector*, *vector*) return a tuple as before. Then, we modify the tuple bindings accordingly, as shown in Figure 11.

$$\begin{array}{l}(\textbf{let}\ ([\langle v\ \mathit{store}\rangle\ \mathit{exp}_1])\\ \quad \mathit{exp}_2)\end{array} = \begin{cases} \begin{array}{l}(\textbf{let}\ ([v\ \mathit{exp}_1])\\ \quad \mathit{exp}_2)\end{array} & \text{if } \mathit{exp}_1 \text{ returns a value}\\ \begin{array}{l}(\textbf{let}\ ([\mathit{store}\ \mathit{exp}_1])\\ \quad \mathit{exp}_2)\end{array} & \text{if } \mathit{exp}_1 \text{ returns a store}\\ \begin{array}{l}(\textbf{let}\ ([\langle v\ \mathit{store}\rangle\ \mathit{exp}_1])\\ \quad \mathit{exp}_2)\end{array} & \text{if } \mathit{exp}_1 \text{ returns both}\end{cases}$$

FIGURE 11. Optimizing the SPS functions.

After the optimization, there is also a copy propagation step, since some bindings will be now redundant. Figure 12 shows the function *vector-sum* after this optimization.

4. Running the functional analysis

Once the program is in a functional style, it can be given to the analysis described in Chapter 2. However, there are some things to consider. First, the store is now a user variable, which means that when execution paths are unknown, we need to do a

```
(define (vector-sum v store)
  (let ([⟨sum store⟩ (sa-vector 0 store)])
    (let ([⟨i store⟩ (sa-vector 0 store)])
      (let ([store (vector-sum-loop0 sum i v store)])
        (sa-vector-ref sum 0 store)))))
(define (vector-sum-loop0 sum i v store)
  (let ([v0 (sa-vector-ref i 0 store)])
    (let ([v1 (sa-vector-length v store)])
      (let ([v2 (sa-< v0 v1 store)])
        (if v2
            (let ([v3 (sa-vector-ref sum 0 store)])
              (let ([v4 (sa-vector-ref i 0 store)])
                (let ([v5 (sa-vector-ref v v4 store)])
                  (let ([v6 (sa-+ v3 v5 store)])
                    (let ([store (sa-vector-set! sum v6 store)])
                      (let ([v7 (sa-vector-ref i 0 store)])
                        (let ([v8 (sa-+ v7 1 store)])
                          (let ([store (sa-vector-set! i v8 store)])
                            (vector-sum-loop0 sum i v store)))))))))
            store)))))
```

FIGURE 12. Program *vector-sum*, after SPS optimization.

least-upper bound on stores. Since we know the format of a store, we can do a better job getting the least upper bound than the general *lub* function. In fact, the general *lub* will return an object that is not a store when given two different stores. The new *lub* function is shown in Figure 13.

```
(define (lub x y)
  (if (store? x)
      (lub-store x y)
      (if (equal? x y)
          x
          'unknown)))
```

where

$$lub\text{-}store(s_1, s_2) = s_3$$

such that

$$s_3(loc) = lub(s_1(loc), s_2(loc))$$

FIGURE 13. The new function *lub* appropriate for functional programs converted from imperative.

Second, with the transformation we introduced code that wasn't there at the beginning, which is going to have an impact in the resulting bound. The most notable source of new code is the SPS transformation, which introduces many tuple creation and tuple destruction of value/store pairs. The solution is to assign a cost of zero to this new code. To handle the tuple creation and destruction, we introduce new primitives *pair*, $1st$ and $2nd$ with zero cost. Since now every function call has a new argument *store*, the new cost of a function call is now $T_{call} - T_{varref}$ and for every primitive, the new cost is $T_{prim} - T_{varref}$.

5. Implementation and experimentation

I have implemented this analysis approach in our prototype system, ALPA, obtaining encouraging good results. The methodology is similar to the methodology used in Section 8. The symbolic evaluation and optimizations, as well as measurements of primitive parameters, are written in Scheme. The measurements and analyses are performed for source programs compiled with Chicken Scheme compiler [**24**]. The particular numbers below are taken on a Apple Dual G5 CPU and 4GB main memory, but the analysis were performed for several kinds of SPARC stations, and the results are similar.

Optimization was set to 0, and there were two sets of experiments to account for the presence and absence of garbage-collection in the timings. To avoid possible cache effects, cache was disabled when running the experiments.

Since the minimum running time of a program construct is about 0.1 microseconds, and the precision of the timing function is 1 millisecond, I use control/test loops that iterate 1,000,000 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.

Table A.3 shows the calculated and measured worst-case times for six example programs on inputs of size 10 to 2000. These times include garbage-collection times.

The item me/ca is the measured time expressed as a percentage of the calculated time. In general, all measured times are bounded by the calculated times (with about 60-95% accuracy). The programs are insertion, selection and merge sort on vectors, vector sum, destructive list reversal, and destructive merge sort on lists.

In general, the measured worst-case times are closely bounded by calculated upper bounds for all inputs. Figure 14 depicts the numbers in Table A.3, normalized on the asymptotic growth of the respective functions.

6. Direct transformation

One drawback of using a transformation to a functional program to analyze imperative programs is that we lose accuracy. In particular, all the computed costs have more function calls and variable references than the measured costs. To overcome this inaccuracy, a new transformation is presented directly from the imperative program to the time-bound function, to avoid introducing new code in the analyzed program.

In order to analyze programs in the presence of assignment, we use, as before, a transformation $\mathcal{T}$ that transform the original program into a new program that computes the original value plus the time-bound to compute that value and the resulting environment. The environment is needed in the intermediate steps to help keep track of the value of the variables before an evaluation, in case we need those same values to evaluate a different expression in the original context, for example, with a conditional expression we would like to evaluate both the true branch and the else branch with the same environment.

7. Constructing time-bound function

For the first-order functional language, we had two transformation, a timing transformation and a time-bound transformation. However in this case we need to keep

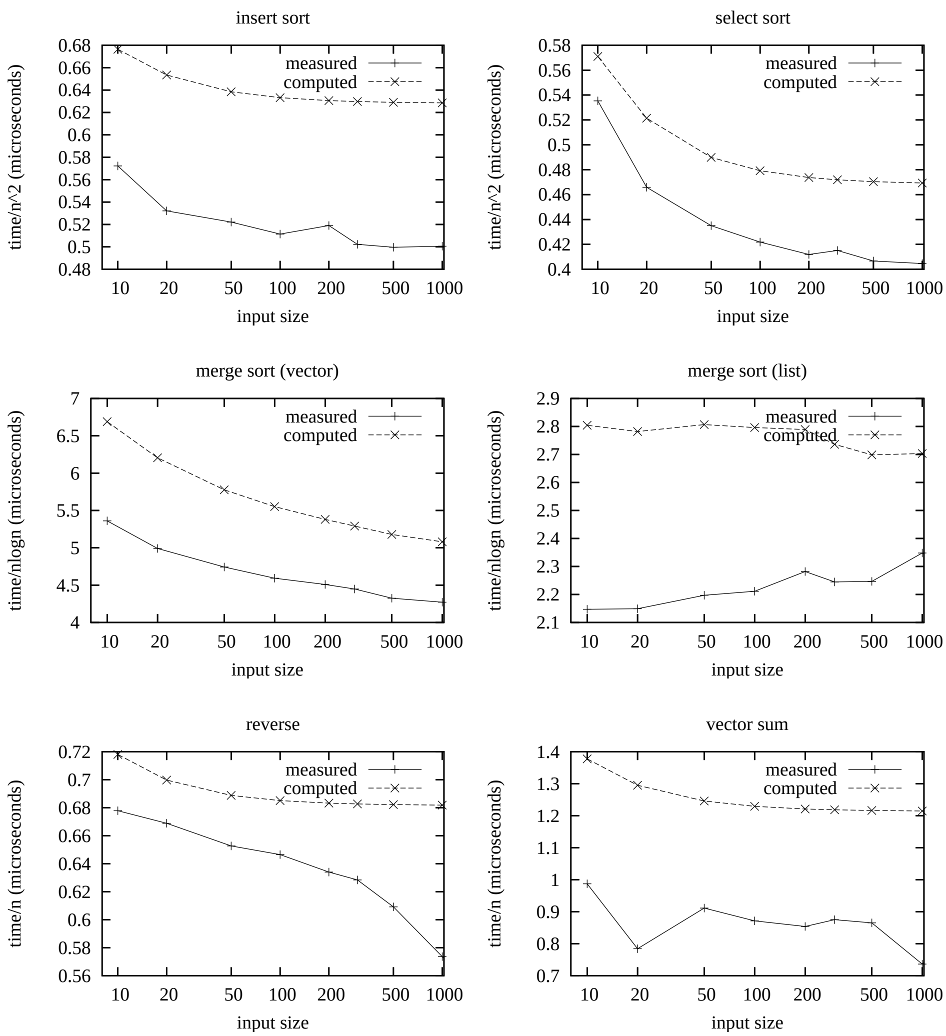

FIGURE 14. Comparison of calculated and measured worst-case times for the imperative language, using SPS.

track of the environment and the storage, which makes the separation of the two transformation steps more complex, instead of simpler. For this reason, we have only one transformation step which makes a time-bound function directly from the original function.

Transformation $\mathcal{T}$ is defined by the rules shown in Figure 15

$$
R_{\mathcal{T}0}:\ \mathcal{T}\left\llbracket\begin{array}{l}(\mathbf{define}\ (f_1\ v_{1_1}\ \ldots\ v_{1_k})\ \ exp_1)\\(\mathbf{define}\ (f_2\ v_{2_1}\ \ldots\ v_{2_k})\ \ exp_2)\\\qquad\vdots\\(\mathbf{define}\ (f_n\ v_{n_1}\ \ldots\ v_{n_k})\ \ exp_n)\end{array}\right\rrbracket=\begin{array}{l}(\mathbf{define}\ (f_1^*\ v_{1_1}\ \ldots\ v_{1_k}\ \rho\ \sigma)\ \ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma))\\(\mathbf{define}\ (f_2^*\ v_{2_1}\ \ldots\ v_{2_k}\ \rho\ \sigma)\ \ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho\ \sigma))\\\qquad\vdots\\(\mathbf{define}\ (f_n^*\ v_{n_1}\ \ldots\ v_{n_k}\ \rho\ \sigma)\ \ (\mathcal{T}_e\llbracket exp_n\rrbracket\ \rho\ \sigma))\end{array}
$$

$$
\begin{array}{l}
R_{\mathcal{T}_e1}:\mathcal{T}_e\llbracket v\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\ \langle(\rho\ v)\ \ T_{var}\ \rho\ \sigma\rangle)\\
R_{\mathcal{T}_e2}:\mathcal{T}_e\llbracket c\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\ \langle c\ \ T_c\ \rho\ \sigma\rangle)\\
R_{\mathcal{T}_e3}:\mathcal{T}_e\llbracket(\mathbf{set!}\ v\ exp)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v_1\ t_1\ \rho_1\ \sigma_1\rangle\ (\mathcal{T}_e\llbracket exp\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\quad\langle v_1\ (+\ t_1\ \ T_{set!})\ \rho_1[v\mapsto v_1]\ \sigma_1\rangle))\\
R_{\mathcal{T}_e4}:\mathcal{T}_e\llbracket(\mathbf{begin}\ exp_1\ \ exp_2)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v_1\ t_1\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\quad(\mathbf{let}\ (((\langle v_2\ t_2\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\qquad\langle v_2\ (+\ t_1\ \ t_2\ \ T_{begin})\ \rho\ \sigma\rangle))))\\
R_{\mathcal{T}_e5}:\mathcal{T}_e\llbracket(\mathbf{if}\ exp_1\ \ exp_2\ \ exp_3)\rrbracket=\\
\quad(\mathbf{lambda}\ (\rho\ \sigma)\\
\quad\ (\mathbf{let}\ (((\langle v_1\ t_1\ \rho_1\ \sigma_1\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\quad\ \ (\mathbf{if}\ (unknown?\ v_1)\\
\quad\ \ \ (\mathbf{let}\ (((\langle v_2\ t_2\ \rho_2\ \sigma_2\rangle\ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho_1\ \sigma_1)))\\
\quad\ \ \ \ (\mathbf{let}\ (((\langle v_3\ t_3\ \rho_3\ \sigma_3\rangle\ (\mathcal{T}_e\llbracket exp_3\rrbracket\ \rho_1\ \sigma_1)))\\
\quad\ \ \ \ \ \langle(lub_v\ v_2\ v_3)\ (+\ T_{if}\ t_1\ (max\ t_2\ t_3))\ (lub_e\ \rho_2\ \rho_3)\ (lub_s\ \sigma_2\ \sigma_3)\rangle))\\
\quad\ \ \ (\mathbf{if}\ v_1\\
\quad\ \ \ \ (\mathbf{let}\ (((\langle v_2\ t_2\ \rho_2\ \sigma_2\rangle\ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho_1\ \sigma_1)))\\
\quad\ \ \ \ \ \langle v_2\ (+\ T_{if}\ t_1\ t_2)\ \rho_2\ \sigma_2\rangle)\\
\quad\ \ \ \ (\mathbf{let}\ (((\langle v_3\ t_3\ \rho_3\ \sigma_3\rangle\ (\mathcal{T}_e\llbracket exp_3\rrbracket\ \rho_1\ \sigma_1)))\\
\quad\ \ \ \ \ \langle v_3\ (+\ T_{if}\ t_1\ t_3)\ \rho_3\ \sigma_3\rangle))))))\\
R_{\mathcal{T}_e6}:\mathcal{T}_e\llbracket(prim\ exp_1\ \ldots\ exp_n)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v_1\ t_1\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\quad\ddots\\
\qquad\qquad\qquad(\mathbf{let}\ (((\langle v_n\ t_n\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_n\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\qquad\ (\mathbf{let}\ (((\langle v_0\ t_0\ \rho\ \sigma\rangle\ (prim^*\ v_1\ \ldots\ v_n\ \rho\ \sigma)))\\
\qquad\qquad\qquad\quad\langle v_0\ (+\ t_0\ t_1\ \ldots\ t_n)\ \rho\ \sigma\rangle)))))\\
R_{\mathcal{T}_e7}:\mathcal{T}_e\llbracket(f\ \ exp_1\ \ldots\ exp_n)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v_1\ t_1\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\quad\ddots\\
\qquad\qquad\qquad(\mathbf{let}\ (((\langle v_n\ t_n\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_n\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\qquad\ (\mathbf{let}\ (((\langle v_0\ t_0\ \rho\ \sigma\rangle\ (f^*\ v_1\ \ldots\ v_n\ \rho\ \sigma)))\\
\qquad\qquad\qquad\quad\langle v_0\ (+\ T_{call}\ t_0\ t_1\ \ldots\ t_n)\ \rho\ \sigma\rangle)))))\\
R_{\mathcal{T}_e8}:\mathcal{T}_e\llbracket(\mathbf{while}\ exp_1\ \ exp_2)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v\ t\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\ (\mathbf{while}\ v\\
\qquad\qquad\quad(\mathbf{let}\ (((\langle ignored\ t_2\ \rho_2\ \sigma_2\rangle\ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\quad\ (\mathbf{let}\ (((\langle v_1\ t_1\ \rho_1\ \sigma_1\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho_2\ \sigma_2)))\\
\qquad\qquad\qquad(\mathbf{begin}\\
\qquad\qquad\qquad\ (\mathbf{set!}\ v\ v_1)\ (\mathbf{set!}\ t\ (+\ t\ t_1\ t_2\ T_{loop}))\\
\qquad\qquad\qquad\ (\mathbf{set!}\ \rho\ \rho_1)\ (\mathbf{set!}\ \sigma\ \sigma_1)))))\\
\qquad\qquad\ (\mathbf{if}\ (unknown?\ v)\\
\qquad\qquad\quad(\mathbf{abort}\ \text{'infinity})\\
\qquad\qquad\quad\langle\text{'void}\ (+\ T_{while}\ t)\ \rho\ \sigma\rangle))))\\
R_{\mathcal{T}_e9}:\mathcal{T}_e\llbracket(\mathbf{let}\ ((v\ \ exp_1))\ \ exp_2)\rrbracket=(\mathbf{lambda}\ (\rho\ \sigma)\\
\qquad\qquad(\mathbf{let}\ (((\langle v_1\ t_1\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_1\rrbracket\ \rho\ \sigma)))\\
\qquad\qquad\ (\mathbf{let}\ (((\langle v_2\ t_2\ \rho\ \sigma\rangle\ (\mathcal{T}_e\llbracket exp_2\rrbracket\ \rho[v\mapsto v_1]\ \sigma)))\\
\qquad\qquad\quad\langle v_2\ (+\ T_{let}\ t_1\ t_2)\ \rho\ \sigma\rangle))))
\end{array}
$$

FIGURE 15. Rules for *time-bound* transformation $\mathcal{T}$

Rule $R_{\mathcal{T}0}$ adds the environment and the storage as new arguments to each function definition, and it applies that argument to the transformation $\mathcal{T}_e$ of the original body.

Transformation $\mathcal{T}_e$ builds a function that takes the environment as argument and returns a quadruple $\langle$*value time new-env new-storage*$\rangle$ where value is the value computed by the original function, time is the time-bound to compute that function, new-env is the resulting environment after the evaluation of the expression and new-storage is the resulting storage after the evaluation of the expression. Notice that now every function call, including the primitive procedures, receive the environment and the store, and return the quadruple. The primitive procedures are redefined in a similar fashion to Figure 8, including the environment as argument.

Rule $R_{\mathcal{T}_e1}$ creates a functional expression which returns a quadruple with the value of v in the environment, the constant T_{var} which represents the time associated with a variable lookup, the unchanged environment and the original store.

Rule $R_{\mathcal{T}_e2}$ creates a functional expression which returns a quadruple with the constant c, the constant T_c which represents the time associated with the constant c, the original environment and the original store.

In rule $R_{\mathcal{T}_e3}$ we bind the resulting quadruple of applying the transformation of the expression exp to the environment to a new fresh quadruple $\langle\ v_1\ t_1\ \rho_1\ \sigma_1\rangle$, and the resulting quadruple contains the value v_1, the time-bound of the evaluation of the expression plus the time associated with a variable assignment, the new environment with the variable associated to the new value v_1, and the new store.

Rule $R_{\mathcal{T}_e4}$ evaluates sequentially the expressions, where the resulting quadruple has the value and the environment resulting from the second expression, and the time is the sum of the time-bounds for both expressions plus the time associated with the sequencing operation.

Rule $R_{\mathcal{T}_e 5}$ we first evaluate the condition expression, and if the value of the expression is unknown then we evaluate both the true branch and the false branch, and the resulting quadruple contains the least upper bound of the values of the two branches, the time associated with a conditional expression plus the time-bound of the condition expression plus the maximum of the time-bounds of the true and false branches, and the least upper bound of the environments, which is the natural extension of the least upper bound function for values. If the value of the expression is known then we take the appropriate branch and add the time-bounds accordingly.

Rule $R_{\mathcal{T}_e 6}$ applies to primitives, and it evaluates the arguments in order, using the resulting environment of the previous argument to evaluate the current argument, and the resulting quadruple contains the application of the primitive to the values, the sum of all the time-bounds plus the time associated with the constructor, and the resulting environment of the evaluation of the last argument. If the primitive is not a constructor, the function $prim^*$ handles unknown objects in a similar way as in Chapter 2.

Rule $R_{\mathcal{T}_e 7}$ is similar to rules $R_{\mathcal{T}_e 5}$ and $R_{\mathcal{T}_e 6}$, but instead of calling function f, it calls the transformed function f^* with the environment and store as the extra arguments.

Rule $R_{\mathcal{T}_e 8}$ uses fresh variables v, t, ρ and σ to keep the value, time-bound, environment and store at the end of each loop, so when exiting the loop we can use the values. If we exited the loop because the condition is unknown, then we abort with a result of infinity, which means we cannot produce a good bound. This didn't happen in any of the examples used.

After transformation $\mathcal{T}$, the function *vector-sum* is 79 lines long, so Figure 16 shows only part of the function *vector-sum* after the transformation.

As before, this version has many tuple construction and destruction that are not necessary. We can use the same technique we used with the SPS approach. Also, all

```
(define (vector-sum v ρ σ)
  ((lambda (ρ σ)
     (let ([⟨v_0 t_0 ρ σ⟩
            ((lambda (ρ σ) ⟨0 T_c ρ σ⟩) ρ σ)])
       (let ([⟨v_1 t_1 ρ σ⟩
              ((lambda (ρ σ)
                 (let ([⟨v_2 t_2 ρ σ⟩
                        ((lambda (ρ σ) ⟨0 T_c ρ σ⟩) ρ σ)])
                   (let ([⟨v_3 t_3 ρ σ⟩
                          ((lambda (ρ σ)
                             (let ([⟨v_4 t_4 ρ σ⟩
                                    ((lambda (ρ σ)
                                       [... while-loop ...])
                                     ρ
                                     σ)])
                               (let ([⟨v_5 t_5 ρ σ⟩
                                      ((lambda (ρ σ)
                                         ⟨(ρ 'sum) T_var ρ σ⟩)
                                       ρ
                                       σ)])
                                 ⟨v_5 (+ t_4 t_5 T_seq) ρ σ⟩)))
                           (env:extend 'i v_2 ρ)
                           σ)])
                     ⟨v_3 (+ t_2 t_3 T_let) ρ σ⟩)))
               (env:extend 'sum v_0 ρ)
               σ)])
         ⟨v_1 (+ t_0 t_1 T_let) ρ σ⟩)))
   ρ σ))
```

FIGURE 16. Fragment of program *vector-sum*, after transformation $\mathcal{T}$

those ((**lambda** (ρ σ) ...) ...) can be transformed into (**let** ([ρ ...] [σ ...]) ...) and then a simple copy-propagation step will get rid of the useless bindings. The code segment in Figure 16 is shown after this optimization in Figure 17.

```
(define (vector-sum v ρ σ)
  (let ([⟨v_1 t_1 ρ σ⟩
         (let ([ρ (env:extend 'sum 0 ρ)])
           (let ([⟨v_3 t_3 ρ σ⟩
                  (let ([ρ (env:extend 'i 0 ρ)])
                    (let ([⟨v_4 t_4 ρ σ⟩
                           [... while-loop ...]])
                      ⟨(ρ 'sum) (+ t_4 T_var T_seq) ρ σ⟩))])
             ⟨v_3 (+ T_c t_3 T_let) ρ σ⟩))])
    ⟨v_1 (+ T_c t_1 T_let) ρ σ⟩))
```

FIGURE 17. Program *vector-sum*, after optimization.

The optimizations discussed in Chapter 2 also apply to this analysis.

8. Implementation and experimentation

I have implemented the analysis approach in our prototype system, ALPA, obtaining encouraging good results. The methodology is similar to the methodology used in Chapter 2. The symbolic evaluation and optimizations, as well as measurements of primitive parameters, are written in Scheme. The measurements and analyses are performed for source programs compiled with Chicken Scheme compiler [**24**]. The particular numbers below are taken on a Dual Apple G5 CPU and 4GB main memory, but the analysis were performed for several other kinds of SPARC stations, and the results are similar.

As in the functional case, optimization was set to 0, and there were two sets of experiments to account for the presence and absence of garbage-collection in the timings. No numbers are large enough to trigger the bignum implementation. To avoid possible cache effects, cache was disabled when running the experiments.

Since the minimum running time of a program construct is about 0.1 microseconds, and the precision of the timing function is 1 milliseconds, I use control/test loops that iterate 1,000,000 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.

Table A.4 shows the calculated and measured worst-case times for six example programs on inputs of size 10 to 2000. These times include garbage-collection times.

The item me/ca is the measured time expressed as a percentage of the calculated time. In general, all measured times are closely bounded by the calculated times (with about 80-99% accuracy). The programs are insertion, selection and merge sort on vectors, vector sum, destructive list reversal, and destructive merge sort on lists.

In general, the measured worst-case times are closely bounded by calculated upper bounds for all inputs of medium sizes. Figure 18 depicts the numbers in Table A.4 and Table A.3, showing clearly that the direct approach yields more accurate results.

As expected, this approach yields more accurate symbolic counts. In all the examples tested, the symbolic counts were the exact actual running counts, as opposed to the SPS approach which over predicted the variable references and the function calls. The running time of the analysis was just a little higher –between 2% and 5%– with this approach.

9. Experimentation on a real world program

It is important to verify that this approach scales to larger programs. I tested this with two programs: `arcode` [**19**] (a file compressor using arithmetic coding) and `cruft` [**13**] (a file encryptor).

9.1. Language translation. Both programs are written in C and they must be translated to the language defined here. For that I used `Evil` [**71**], a C++ to Scheme Compiler. This compiler works in two steps: a C++ parser that generates Scheme S-expressions, and the compiler proper. I use only the first step to get the C program in a S-expression syntax, and then a small program I wrote to change from the S-expression syntax used in `Evil` to the syntax used here.

9.2. Measurements. The program `cruft` ran without problems, and it gives the results shown below. The program `arcode`, however, suffers from the same problem

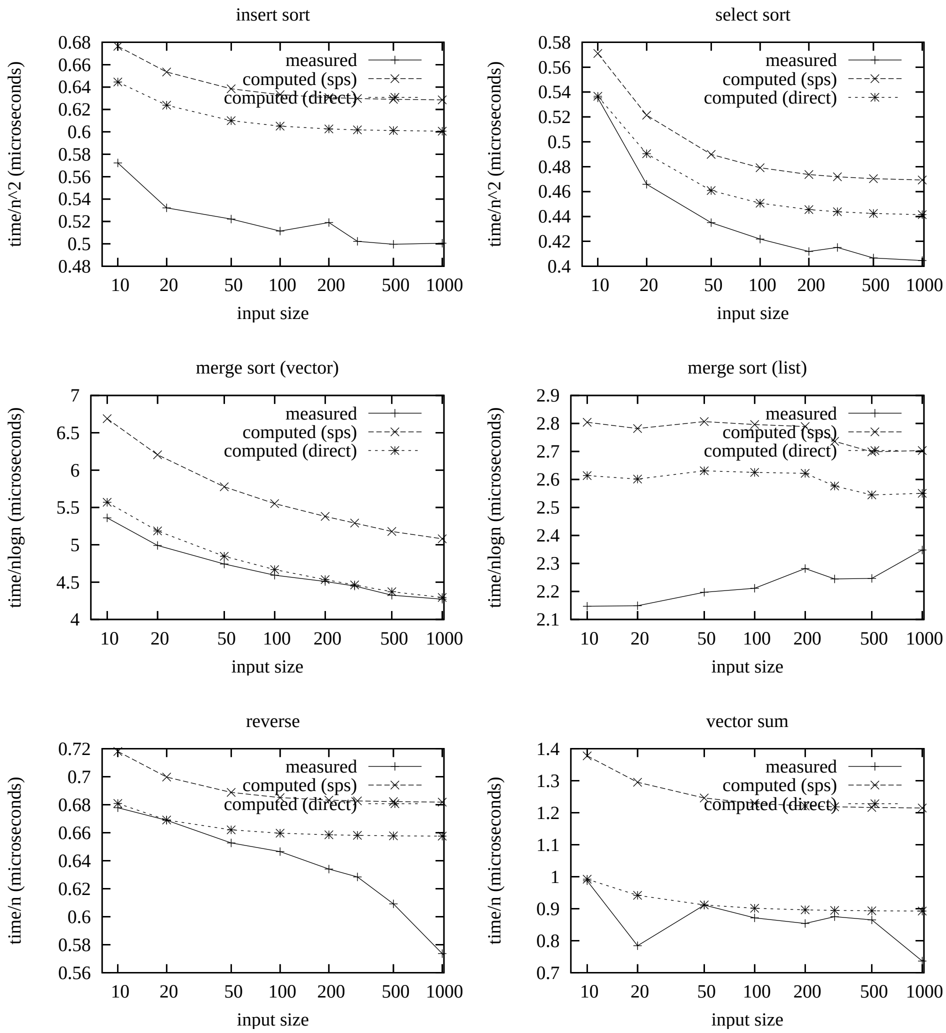

FIGURE 18. Comparison of calculated and measured worst-case times for the imperative language using the direct approach.

as quick sort, where the size of the problem depends on the unknown parts of the input.

The measurements were taken on a Apple Dual G5 CPU and 4GB main memory using the compiler GCC 4.0.1. The times for the primitive functions were measured as in the previous section. Since I don't know the shape of the worst case input, I used 25 different input files per input size: one file with only the NUL character, one file with

ASCII characters in alphabetical order, three files with ASCII characters in mostly alphabetical order (90% chance the character is in the correct position), eight files with ASCII characters in random order, one file with ordered binary characters, three files with mostly ordered binary characters and eight files with binary characters in random order. I run the testcases 100 times, where a testcase would run the program in a loop 2000 times, and for each size I used the time of the file with the worst case.

Table A.5 shows the calculated and measured worst-case times for `cruft` on inputs of size 10 to 2000. The item me/ca is the measured time expressed as a percentage of the calculated time. Again, all measured times are closely bounded by the calculated times (with about 87% accuracy). Figure 19 depicts the numbers in Table A.5 with the numbers normalized on the asymptotic growth of cruft.

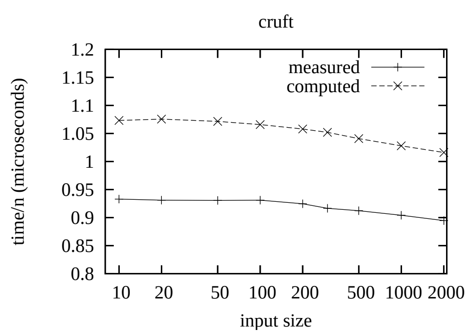

FIGURE 19. Comparison of calculated and measured worst-case times for the imperative language on program `cruft`, using the direct approach.

CHAPTER 4

Analysis of a Higher-Order Language

This chapter extends the language-based approach to a higher-order language. As before, 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.

1. Language definition

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 in Figure 1.

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

$$
\begin{array}{lcll}
\textit{program} & ::= & (\textbf{define}\ f_1\ e_1) & \\
 & & \quad\quad\quad\quad \ldots & \\
 & & (\textbf{define}\ f_m\ e_m) & \\
e & ::= & v & \text{variable reference} \\
 & | & (c\ e_1\ \ldots\ e_n) & \text{data construction} \\
 & | & (p\ e_1\ \ldots\ e_n) & \text{primitive operation} \\
 & | & (\textbf{if}\ e_1\ e_2\ e_3) & \text{conditional expression} \\
 & | & (\textbf{let}\ ((v\ e_1))\ e_2) & \text{binding expression} \\
 & | & (\textbf{letrec}\ ((v\ e_1))\ e_2) & \text{recursive binding exp} \\
 & | & (\textbf{lambda}\ (v_1\ \ldots\ v_n)\ e) & \text{first-class function} \\
 & | & (f\ e_1\ \ldots\ e_n) & \text{function application}
\end{array}
$$

FIGURE 1. Definition of the functional language.

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 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 2 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 4.

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.

```
(define index
  (lambda (item ls)
    (index-cps item ls (lambda (x) x))))
(define index-cps
  (lambda (item ls k)
    (if (null? ls)
        -1
        (if (= item (car ls))
            (k 0)
            (index-cps item (cdr ls)
              (lambda (v) (k (+ v 1))))))))
```

FIGURE 2. Example program with definitions *index* and *index-cps*.

2. Constructing time-bound function

2.1. 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. Partially known input structures may be given by a user or constructed automatically for typical input structures parameterized by information such as the length of a list or the height of a complete binary tree.

For example, a variable reference is transformed into a symbol T_{var} 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 function $+_t$ 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

$$
\begin{array}{l}
R_{\mathcal{T}0} : \mathcal{T}\left[\!\!\left[\begin{array}{c}(\textbf{define}\ v_1\ e_1)\\ \vdots \\ (\textbf{define}\ v_n\ e_n)\end{array}\right]\!\!\right] = \begin{array}{c}(\textbf{define}\ v_1\ \mathcal{T}_v[e_1])\\ \vdots \\ (\textbf{define}\ v_n\ \mathcal{T}_v[e_n])\end{array}\\
R_{\mathcal{T}_v1} : \mathcal{T}_v[v] = v\\
R_{\mathcal{T}_v2} : \mathcal{T}_v[(c\ e_1\ ...\ e_n)] = (c\ \mathcal{T}_v[e_1]\ ...\ \mathcal{T}_v[e_n])\\
R_{\mathcal{T}_v3} : \mathcal{T}_v[(p\ e_1\ ...\ e_n)] = (p\ \mathcal{T}_v[e_1]\ ...\ \mathcal{T}_v[e_n])\\
R_{\mathcal{T}_v4} : \mathcal{T}_v[(\textbf{if}\ e_1\ e_2\ e_3)] = (\textbf{if}\ \mathcal{T}_v[e_1]\ \mathcal{T}_v[e_2]\ \mathcal{T}_v[e_3])\\
R_{\mathcal{T}_v5} : \mathcal{T}_v[(\textbf{let}\ ((v\ e_1))\ e_2)] = (\textbf{let}\ ((v\ \mathcal{T}_v[e_1]))\ \mathcal{T}_v[e_2])\\
R_{\mathcal{T}_v6} : \mathcal{T}_v[(\textbf{letrec}\ ((v\ e_1))\ e_2)] = (\textbf{letrec}\ ((v\ \mathcal{T}_v[e_1]))\ \mathcal{T}_v[e_2])\\
R_{\mathcal{T}_v7} : \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\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{lambda}\ (v_1\ ...\ v_n)\ \mathcal{T}_t[\mathcal{T}_v[e_0]]))\\
R_{\mathcal{T}_v8} : \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])\\
\\
R_{\mathcal{T}_t1} : \mathcal{T}_t[v] = T_{var}\\
R_{\mathcal{T}_t2} : \mathcal{T}_t[(c\ e_1\ ...\ e_n)] = (+_t\ T_c\ \mathcal{T}_t[e_1]\ ...\ \mathcal{T}_t[e_n])\\
R_{\mathcal{T}_t3} : \mathcal{T}_t[(p\ e_1\ ...\ e_n)] = (+_t\ T_p\ \mathcal{T}_t[e_1]\ ...\ \mathcal{T}_t[e_n])\\
R_{\mathcal{T}_t4} : \mathcal{T}_t[(\textbf{if}\ e_1\ e_2\ e_3)] = (\textbf{if}\ e_1\ (+_t\ T_{if}\ \mathcal{T}_t[e_1]\ \mathcal{T}_t[e_2])\ (+_t\ T_{if}\ \mathcal{T}_t[e_1]\ \mathcal{T}_t[e_3]))\\
R_{\mathcal{T}_t5} : \mathcal{T}_t[(\textbf{let}\ ((v\ e_1))\ e_2)] = (\textbf{let}\ ((v\ e_1))\ (+_t\ T_{let}\ \mathcal{T}_t[e_1]\ \mathcal{T}_t[e_2]))\\
R_{\mathcal{T}_t6} : \mathcal{T}_t[(\textbf{letrec}\ ((v\ e_1))\ e_2)] = (\textbf{letrec}\ ((v\ e_1))\ (+_t\ T_{letrec}\ \mathcal{T}_t[e_1]\ \mathcal{T}_t[e_2]))\\
R_{\mathcal{T}_t7} : \mathcal{T}_t[(\textit{lambda-pair}\ e_1\ e_2)] = T_{lambda}\\
R_{\mathcal{T}_t8} : \mathcal{T}_t[((\textit{value}\ e_0)\ e_1\ ...\ e_n)] = (+_t\ T_{call}\ \mathcal{T}_t[e_0]\ \mathcal{T}_t[e_1]\ ...\ \mathcal{T}_t[e_n]\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ((\textit{time}\ e_0)\ e_1\ ...\ e_n))
\end{array}
$$

FIGURE 3. Rules for *time transformation* $\mathcal{T}$.

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 in Figure 3. 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 not cause any problem since the old meanings of theses identifiers are not used in the transformed program.

Rules $R_{\mathcal{T}_v1}$ to $R_{\mathcal{T}_v6}$ handle expressions other than lambda expressions or function calls, so they transform subexpressions recursively. Rule $R_{\mathcal{T}_v7}$ 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 $R_{\mathcal{T}_v8}$ 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 $R_{\mathcal{T}_t1}$ transforms a variable reference to the time of a variable reference T_{var}. Rule $R_{\mathcal{T}_t2}$ (respectively $R_{\mathcal{T}_t3}$) sums the times of evaluating the arguments and the time of the primitive (respectively constructor). Rule $R_{\mathcal{T}_t4}$ 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 $R_{\mathcal{T}_t5}$ and $R_{\mathcal{T}_t6}$ 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 $R_{\mathcal{T}_t7}$ 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 $R_{\mathcal{T}_t8}$ 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 4 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_k for their times.

```
(define index-cps
  (lambda-pair
    (lambda (item ls k)
      (if (null? ls)
          −1
          (if (= item (car ls))
              ((value k) 0)
              ((value index-cps) item (cdr ls) lambda_1))))
    (lambda (item ls k)
      (if (null? ls)
          (+_t T_if (+_t T_null? T_varref) T_k)
          (+_t T_if (+_t T_null? T_varref)
             (if (= item (car ls))
                 (+_t T_if (+_t T_= T_varref (+_t T_car T_varref))
                   (+_t T_call ((time k) 0) T_var T_k))
                 (+_t T_if (+_t T_= T_varref (+_t T_car T_varref))
                   (+_t T_call T_var T_var
                     ((time index-cps) item (cdr ls) lambda_1)
                     T_closure (+_t T_cdr T_varref)))))))))
;; where lambda_1 is
   (lambda-pair
     (lambda (v) ((value k) (+ v 1)))
     (lambda (v) (+_t T_call ((time k) (+ v 1)) T_var
                          (+_t T_+ T_k T_varref))))
```

FIGURE 4. Function *index-cps* after transformation $\mathcal{T}$.

This transformation is similar to the local cost assignment [**100**], step-counting function [**84**], cost function [**88**], 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. 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.

2.2. Constructing time-bound functions. To characterize the program input we use again partially known input structures with *unknown* values. We also define a new primitive function f_p for each primitive function p and a new *least upper bound* function *lub* as in Chapter 2.

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 mentioned in this Chapter, this is not needed, i.e., the naturally given partially known input contains enough information to decide all lambdas at analysis time.

```
(define value_apply                     (define time_apply
  (lambda (v0 v1 ... vn)                  (lambda (v0 v1 ... vn)
    (if (unknown? v0)                       (if (unknown? v0)
        'unknown                                'infinite
        ((value v0) v1 ... vn))))               ((time v0) v1 ... vn))))
```

The *time-bound transformation* $\mathcal{T}_b$ given in Figure 5 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.

$$R_{\mathcal{T}_b 0} : \mathcal{T}_b \left[\!\!\left[\begin{array}{l}(\textbf{define}\ v_1\ e_1)\\ \quad\vdots\\ (\textbf{define}\ v_n\ e_n)\end{array}\right]\!\!\right] = \begin{array}{l}(\textbf{define}\ v_1\ \mathcal{T}_{vb}[\![e_1]\!])\\ \quad\vdots\\ (\textbf{define}\ v_n\ \mathcal{T}_{vb}[\![e_n]\!])\end{array}$$

$$\begin{array}{l}
vb_1 : \mathcal{T}_{vb}[\![v]\!] = v\\
vb_2 : \mathcal{T}_{vb}[\![(c\ e_1\ ...\ e_n)]\!] = (c\ \mathcal{T}_{vb}[\![e_1]\!]\ ...\ \mathcal{T}_{vb}[\![e_n]\!])\\
vb_3 : \mathcal{T}_{vb}[\![(p\ e_1\ ...\ e_n)]\!] = (f_p\ \mathcal{T}_{vb}[\![e_1]\!]\ ...\ \mathcal{T}_{vb}[\![e_n]\!])\\
vb_4 : \mathcal{T}_{vb}[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] = (\textbf{let}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\\
\qquad\qquad\qquad\qquad\qquad\quad (\textbf{if}\ (\textit{unknown?}\ v)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\textit{lub}\ \mathcal{T}_{vb}[\![e_2]\!]\ \mathcal{T}_{vb}[\![e_3]\!])\\
\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{if}\ v\ \mathcal{T}_{vb}[\![e_2]\!]\ \mathcal{T}_{vb}[\![e_3]\!])))\\
vb_5 : \mathcal{T}_{vb}[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] = (\textbf{let}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\ \mathcal{T}_{vb}[\![e_2]\!])\\
vb_6 : \mathcal{T}_{vb}[\![(\textbf{letrec}\ ((v\ e_1))\ e_2)]\!] = (\textbf{letrec}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\ \mathcal{T}_{vb}[\![e_2]\!])\\
vb_7 : \mathcal{T}_{vb}\left[\!\!\left[\begin{array}{l}(\textit{lambda-pair}\\ \quad(\textbf{lambda}\ (v_1\ ...\ v_n)\ e_1)\\ \quad(\textbf{lambda}\ (v_1\ ...\ v_n)\ e_2))\end{array}\right]\!\!\right] = \begin{array}{l}(\textit{lambda-pair}\\ \quad(\textbf{lambda}\ (v_1\ ...\ v_n)\ \mathcal{T}_{vb}[\![e_1]\!])\\ \quad(\textbf{lambda}\ (v_1\ ...\ v_n)\ \mathcal{T}_{tb}[\![e_2]\!]))\end{array}\\
vb_8 : \mathcal{T}_{vb}[\![((\textit{value}\ e_0)\ e_1\ ...\ e_n)]\!] = (\textit{value_apply}\ \mathcal{T}_{vb}[\![e_0]\!]\ \mathcal{T}_{vb}[\![e_1]\!]\ ...\ \mathcal{T}_{vb}[\![e_n]\!])
\end{array}$$

$$\begin{array}{l}
tb_1 : \mathcal{T}_{tb}[\![T]\!] = T\\
tb_2 : \mathcal{T}_{tb}[\![(+_t\ e_1\ ...\ e_n)]\!] = (+_t\ \mathcal{T}_{tb}[\![e_1]\!]\ ...\ \mathcal{T}_{tb}[\![e_n]\!])\\
tb_3 : \mathcal{T}_{tb}[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] = (\textbf{let}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\\
\qquad\qquad\qquad\qquad\qquad\quad (\textbf{if}\ (\textit{unknown?}\ v)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\textit{max}\ \mathcal{T}_{vb}[\![e_2]\!]\ \mathcal{T}_{vb}[\![e_3]\!])\\
\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{if}\ v\ \mathcal{T}_{vb}[\![e_2]\!]\ \mathcal{T}_{vb}[\![e_3]\!])))\\
tb_4 : \mathcal{T}_{tb}[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] = (\textbf{let}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\ \mathcal{T}_{tb}[\![e_2]\!])\\
tb_5 : \mathcal{T}_{tb}[\![(\textbf{letrec}\ ((v\ e_1))\ e_2)]\!] = (\textbf{letrec}\ ((v\ \mathcal{T}_{vb}[\![e_1]\!]))\ \mathcal{T}_{tb}[\![e_2]\!])\\
tb_6 : \mathcal{T}_{tb}[\![((\textit{time}\ e_0)\ e_1\ ...\ e_n)]\!] = (\textit{time_apply}\ \mathcal{T}_{vb}[\![e_0]\!]\ \mathcal{T}_{vb}[\![e_1]\!]\ ...\ \mathcal{T}_{vb}[\![e_n]\!])
\end{array}$$

FIGURE 5. Rules for *time-bound* transformation $\mathcal{T}_b$.

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 4 yields function *index-cps* in Figure 6. Again, shared code is presented with *where* clauses.

The transformed time-bound function is guaranteed to terminate, provided the original program terminates. In practice, we impose an upper bound on the analysis time, and, if the analysis does not terminate within this time, we report this together with the time-bound calculated till this time

3. Optimizing time-bound function

Time-bound functions may be extremely inefficient to evaluate given values for their parameters. In fact, even when it terminates, 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.

```
(define index-cps
  (lambda-pair
    (lambda (item ls k)
      (let ((v_1 (f_null? ls)))
        (if (unknown? v_1)
            (lub -1 exp_1)
            (if v_1 -1 exp_1))))
    (lambda (item ls k)
      (let ((v_2 (f_null? ls)))
        (if (unknown? v_2)
            (max (+_t T_if (+_t T_null? T_varref) T_k) time_1)
            (if v_2 (+_t T_if (+_t T_null? T_varref) T_k) time_1))))))
  where exp_1 is
      (let ((v_3 (f_= item (f_car ls))))
        (if (unknown? v_3)
            (lub ((value k) 0) ((value index-cps) item (f_cdr ls) lambda_1))
            (if v_3 ((value k) 0) ((value index-cps) item (f_cdr ls) lambda_1))))
  and time_1 is
      (+_t T_if (+_t T_null? T_varref)
        (let ((v_4 (f_= item (f_car ls))))
          (if (unknown? v_4)
              (max time_2 time_3)
              (if v_4 time_2 time_3))))
      where time_2 is
            (+_t T_if (+_t T_= T_varref (+_t T_car T_varref))
              (+_t T_call ((time k) 0) T_varref T_k))
      and time_3 is
            (+_t T_if (+_t T_= T_varref (+_t T_car T_varref))
              (+_t T_call ((time index-cps) item (f_cdr ls)lambda_1)
              T_varref T_varref (+_t T_cdr T_varref) T_lambda))
            where lambda_1 is (lambda-pair
                                (lambda (v) ((value k) (f_+ v 1)))
                                (lambda (v) (+_t T_call ((time k) (f_+ v 1))
                                                 T_varref, (+_t T_+ T_k T_varref))))
```

FIGURE 6. Function *index-cps* after time-bound transformation $\mathcal{T}_b$.

3.1. 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 in Figure 7 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, i.e., it sums the counts respectively for primitive parameters, and max_s is a function that symbolically takes the maximum of its arguments.

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

FIGURE 7. Rules for symbolic evaluation of programs

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:

$\mathcal{E}\,[\![index(unknown, list(100))]\!]\,\emptyset =$

$$101 * T_k + 802 * T_{var} + 201 * T_{if}$$
$$+ 201 * T_{call} + 101 * T_{lambda} + 100 * T_{car}$$
$$+ 100 * T_{cdr} + 101 * T_{null?} + 99 * T_{+} + 100 * T_{=}$$

3.2. 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 $\mathcal{S}$ defined in Figure 8 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.

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

FIGURE 8. Transformation $\mathcal{S}$ to optimize repeated summations.

Applying this optimization to the time-bound version of function *index-cps* in Figure 6 yields the definition in Figure 9.

```
(define index-cps
  (lambda-pair
    (lambda (item ls k)
      (let ((v1 (f_null? ls)))
        (if (unknown? v1)
            (lub −1 exp1)
            (if v1 −1 exp1))))
    (lambda (item ls k)
      (let ((v2 (f_null? ls)))
        (if (unknown? v2)
            (max_sv ⟨0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0⟩ time1)
            (if v2 ⟨0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0⟩ time1))))))
   where exp1 is
     (let ((v3 (f_= item (f_car ls))))
       (if (unknown? v3)
           (lub ((value k) 0) ((value index-cps) item (f_cdr ls) lambda1))
           (if v3 ((value k) 0) ((value index-cps) item (f_cdr ls) lambda1))))
   and time1 is
     (let ((v4 (f_= item (f_car ls))))
       (if (unknown? v4)
           (max_sv (add_sv ⟨1 0 0 1 1 0 0 0 0 0 0 1 4 2 0 0 1 0⟩ ((time k) 0))
             (add_sv ⟨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) lambda1)))
           (if v4
               (add_sv ⟨1 0 0 1 1 0 0 0 0 0 0 1 4 2 0 0 1 0⟩ ((time k) 0))
               (add_sv ⟨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) lambda1)))))
       where lambda1 is
         (lambda-pair
           (lambda (v) ((value k) (f_+ v 1)))
           (lambda (v) (add_sv ⟨0 0 0 0 0 1 0 0 0 0 0 1 2 0 0 0 1 0⟩
                          ((time k) (f_+ v 1)))))
```

FIGURE 9. Function *index-cps* after optimization for avoiding repeated summations, where the tuples are for $\langle T_{car}, T_{cdr}, T_{cons}, T_{null?}, T_{eq?}, T_{+}, T_{-}, T_{*}, T_{>}, T_{<}, T_{=}, T_{const}, T_{varref}, T_{if}, T_{let}, T_{letrec}, T_{funcall}, T_{closure}\rangle$.

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. Note that profiling will not produce a time bound for all inputs described by the partially known input structures; if enumeration is used, then it will not be faster than our analysis, which is essentially doing a smart form of enumeration.

Time-bound functions can further be made more accurate by lifting conditions, simplifying conditionals, and inlining non-recursive functions, as done previously in [**66**].

4. Implementation and experimentation

We have implemented the analysis approach in our prototype system ALPA. 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 [**24**]. The input is a program as defined in Section 1, 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 implementations consists of 500 lines of scheme code, nearly twice the size of the implementation for the functional language described in Chapter 2.

The computer used to take the measurements is a Sun Enterprise 450 Model 4400 with four 400MHz CPUs, 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 [**35**]; *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 1. Results of symbolic evaluation of time-bound functions for a higher-order language.

program	size	var ref	constant	list ops	+/-	compare	if	let(rec)	lambda	call
ack	⟨3,1⟩	472	328	0	153	164	164	0	0	106
	⟨3,5⟩	190848	127560	0	63533	63780	63780	0	0	42438
	⟨3,7⟩	3122332	2082904	0	1040439	1041452	1041452	0	0	693964
	⟨3,9⟩	50237624	33497192	0	16744513	16748596	16748596	0	0	11164370
ack curried	⟨3,1⟩	277	171	0	98	62	62	6	4	111
	⟨3,5⟩	105989	63787	0	42194	21346	21346	6	4	42443
	⟨3,7⟩	1734421	1041459	0	692954	347492	347492	6	4	693969
	⟨3,9⟩	27908901	16748603	0	11160290	5584230	5584230	6	4	11164375
tak-cps	⟨19,8,1⟩	16121904	1560183	0	1560183	2080245	2080245	1	1560185	3640430
	⟨19,9,1⟩	46538205	4503696	0	4503696	6004929	6004929	1	4503698	10508627
	⟨19,9,3⟩	2582251	249894	0	249894	333193	333193	1	249896	583089
	⟨19,10,1⟩	122680095	11872266	0	11872266	15829689	15829689	1	11872268	27701957
rev	10	299	10	231	0	0	66	0	0	66
	20	1094	20	861	0	0	231	0	0	231
	50	6479	50	5151	0	0	1326	0	0	1326
	100	25454	100	20301	0	0	5151	0	0	5151
	200	100904	200	80601	0	0	20301	0	0	20301
	500	627254	500	501501	0	0	125751	0	0	125751
	1000	2504504	1000	2003001	0	0	501501	0	0	501501
	2000	10009004	2000	8006001	0	0	2003001	0	0	2003001
rev-cps	10	422	11	231	0	0	66	0	56	123
	20	1537	21	861	0	0	231	0	211	443
	50	9082	51	5151	0	0	1326	0	1276	2603
	100	35657	101	20301	0	0	5151	0	5051	10203
	200	141307	201	80601	0	0	20301	0	20101	40403
	500	878257	501	501501	0	0	125751	0	125251	251003
	1000	3506507	1001	2003001	0	0	501501	0	500501	1002003
	2000	14013007	2001	8006001	0	0	2003001	0	2001001	4004003
split	10	128	33	53	0	20	41	0	10	32
	20	248	63	103	0	40	81	0	20	62
	50	608	153	253	0	100	201	0	50	152
	100	1208	303	503	0	200	401	0	100	302
	200	2408	603	1003	0	400	801	0	200	602
	500	6008	1503	2503	0	1000	2001	0	500	1502
	1000	12008	3003	5003	0	2000	4001	0	1000	3002
	2000	24008	6003	10003	0	4000	8001	0	2000	6002
fix	10	275	22	0	20	11	11	0	212	233
	20	545	42	0	40	21	21	0	422	463
	50	1355	102	0	100	51	51	0	1052	1153
	100	2705	202	0	200	101	101	0	2102	2303
	200	5405	402	0	400	201	201	0	4202	4603
	500	13505	1002	0	1000	501	501	0	10502	11503
	1000	27005	2002	0	2000	1001	1001	0	21002	23003
	2000	54005	4002	0	4000	2001	2001	0	42002	46003
map	10	84	2	41	10	0	11	0	1	22
	20	164	2	81	20	0	21	0	1	42
	50	404	2	201	50	0	51	0	1	102
	100	804	2	401	100	0	101	0	1	202
	200	1604	2	801	200	0	201	0	1	402
	500	4004	2	2001	500	0	501	0	1	1002
	1000	8004	2	4001	1000	0	1001	0	1	2002
	2000	16004	2	8001	2000	0	2001	0	1	4002
union	10	705	10	361	0	100	231	10	0	121
	20	2605	20	1321	0	400	861	20	0	441
	50	15505	50	7801	0	2500	5151	50	0	2601
	100	61005	100	30601	0	10000	20301	100	0	10201
	200	242005	200	121201	0	40000	80601	200	0	40401
	500	1505005	500	753001	0	250000	501501	500	0	251001
	1000	6010005	1000	3006001	0	1000000	2003001	1000	0	1002001
	2000	24020005	2000	12012001	0	4000000	8006001	2000	0	4004001
index	10	72	11	31	9	10	21	1	12	21
	20	142	21	61	19	20	41	1	22	41
	50	352	51	151	49	50	101	1	52	101
	100	702	101	301	99	100	201	1	102	201
	200	1402	201	601	199	200	401	1	202	401
	500	3502	501	1501	499	500	1001	1	502	1001
	1000	7002	1001	3001	999	1000	2001	1	1002	2001
	2000	14002	2001	6001	1999	2000	4001	1	2002	4001

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

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

	ackermann			ackermann (curried)		
size	calculated	measured	me/ca	calculated	measured	me/ca
⟨3,1⟩	0.03207	0.02861	89.2002	0.02059	0.01503	72.9892
⟨3,5⟩	12.8462	10.6540	82.9348	7.89957	5.33000	67.4719
⟨3,7⟩	210.051	174.943	83.2863	129.241	89.1780	69.0009
⟨3,9⟩	3379.19	2888.33	85.4739	2079.53	1517.27	72.9621

	takeuchi (CPS)		
size	calculated	measured	me/ca
⟨19,8,1⟩	576.058	509.402	88.4289
⟨19,9,1⟩	1662.87	1473.49	88.6111
⟨19,9,3⟩	92.2675	81.6250	88.4655
⟨19,10,1⟩	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		
size	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
20	0.04087	0.03879	94.9079	0.01133	0.00900	79.4816	0.16680	0.13401	80.3414
50	0.10169	0.09605	94.4445	0.02798	0.02169	77.5121	0.99204	0.80972	81.6212
100	0.20308	0.19183	94.4597	0.05572	0.04360	78.2375	3.90155	3.08000	78.9429
200	0.40584	0.38599	95.1080	0.11121	0.08781	78.9532	15.4734	12.1280	78.3794
500	1.01413	0.97661	96.3001	0.27768	0.22843	82.2614	96.2121	75.1470	78.1055
1000	2.02794	1.93700	95.5154	0.55513	0.48007	86.4776	384.186	315.918	82.2304
2000	4.05556	4.00700	98.8024	1.11003	0.95652	86.1700	1535.42	1260.83	82.1163

	index		
size	calculated	measured	me/ca
10	0.00476	0.00344	72.26890
20	0.00894	0.00647	72.37136
50	0.02148	0.01561	72.67225
100	0.04273	0.03073	71.91668
200	0.08575	0.06166	71.90670
500	0.21951	0.15412	70.21092
1000	0.43402	0.31197	71.87917
2000	1.05827	0.77977	73.68346

CHAPTER 5

Production of a Worst-Case Input

There has been much work in analyzing worst-case execution time as well as bounds for other cost measures, making use of program annotations and various approximations and no profiling, but these analyses do not produce actual worst-case inputs. Because of the use of annotations and approximations, such analyzed bounds may be loose and unreliable. Therefore, it is extremely important to check whether the analyzed bounds are realizable by actual worst-case inputs. Once actual worst-case inputs are constructed, one can analyze worst-case behaviors most precisely, by doing profiling and tracing on the worst-case inputs.

This chapter describes a method for automatic production of worst-case inputs. Given a measure of interest and a set of possible inputs, *worst-case input analysis* constructs an input that has the worst-case cost for the given measure among the given set of inputs.

The method consists of five steps: (1) construct a cost function of the program based on the cost model for the given measure, (2) construct a cost-bound function of the program using abstractions based on the cost function and the given partially known input structure, (3) optimize the cost-bound function drastically to avoid heavily repeated cost computations, (4) symbolically evaluate the optimized cost-bound function to collect a set of constraints on the worst-case inputs, and (5) generate a worst-case input to satisfy the collected constraints.

The center of the method is the cost-bound function, which exploits both the idea of abstraction in program analysis and the idea of enumeration in model checking.

Therefore, we call the method *model analysis*. The cost model is the basis of correctness; the constraint satisfaction validates the accuracy; and the drastic optimization makes the analysis feasible in terms of efficiency. Given a cost-bound analysis based on enumeration and abstraction, our method provides a framework for validating the accuracy of the cost bounds computed, by trying to construct actual worst-case inputs.

To the best of our knowledge, this is the first method for automatic analysis of worst-case input. It meets the challenge by exploiting and combining many methods and techniques for cost modeling, input capture, abstraction, enumeration, optimization, symbolic evaluation, constraint construction, and constraint satisfaction.

This chapter presents the precise formulation of the method for a simple functional language, but the framework underlying the method is general and applies to imperative languages as well. There are still two caveats: the cost-bound function could be too expensive to compute and the constraints could not unsatisfiable, meaning that an accurate bound could not be computed and the bound is too loose to be realized by an actual input. However, for common challenging examples used in real-time and embedded applications, such as various sorting and queuing methods, the method succeeded easily in making the cost-bound functions efficient and constructing the actual worst-case inputs. This is shown through an implementation of the analysis in ALPA (Automatic Language-based Performance Analyzer) and our experiments with a number of programs for sorting and other tasks.

The rest of the chapter is organized as follows. Section 1 describes the programming language used. Section 2 describes the method to construct the cost functions. Section 3 describes the method to construct the cost-bound functions. Section 4 describe the method to obtain the constraints that a worst-case input should satisfy. Section 5 describes optimization methods to speed up the analysis. Section 6 describes the method to obtain an actual input out of the set of constraints. Section 7 discusses the power and limitations of the method. Section 8 describes the implementation and experiments.

1. Language

We use the same language used in Chapter 2: a first-order, call-by-value functional language that has structured data, primitive arithmetic, Boolean, and comparison operations, conditionals, bindings, and mutually recursive function calls. A program is a set of mutually recursive function definitions. Its syntax is given by the grammar in Figure 1.

$$
\begin{array}{lcll}
\mathit{program} & ::= & (\textbf{define}\ (f_1\ v_{1_1}\ \ldots\ v_{1_n})\ e_1) & \\
 & & \qquad \cdots & \\
 & & (\textbf{define}\ (f_m\ v_{m_1}\ \ldots\ v_{m_n})\ e_m) & \\
e & ::= & v & \text{variable reference} \\
 & \mid & (c\ e_1\ \ldots\ e_n) & \text{data construction} \\
 & \mid & (p\ e_1\ \ldots\ e_n) & \text{primitive operation} \\
 & \mid & (\textbf{if}\ e_1\ e_2\ e_3) & \text{conditional expression} \\
 & \mid & (\textbf{let}\ ((v\ e_1))\ e_2) & \text{binding expression} \\
 & \mid & (f\ e_1\ \ldots\ e_n) & \text{function application}
\end{array}
$$

FIGURE 1. Definition of the functional language.

For example, the program in Figure 2 computes the set union of two sets.

```
(define (union set1 set2)
  (if (null? set1)
      set2
      (let ((rr (union (cdr set1) set2)))
        (if (member? (car set1) set2)
            rr
            (cons (car set1) rr)))))
 (define (member? item ls)
   (if (null? ls)
       #f
       (if (eq? item (car ls))
           #t
           (member? item (cdr ls)))))
```

FIGURE 2. Program *union*, which computes the set union of two sets.

2. Constructing cost functions

To construct a cost function we transform the original program to return a tuple with two values, instead of only one. The tuple returned contains the original returned value, and the cost of computing that value.

We use parameters to represent the cost of each primitive operation. For example, C_+ is the cost of the addition operation, C_{call} is the cost of a function call, and C_{if} is the cost of a conditional branch.

The program transformation is defined by $\mathcal{T}_c$ in Figure 3.

Rule R_{c0} defines a function f^* for every function f in the original program. The body of this function will be the original body transformed with expression transformer $\mathcal{T}_{c_e}$.

Rule R_{c1} transforms a variable reference v into the tuple composed by the variable v, and the constant C_{var} which represents the cost of computing a variable reference.

Rule R_{c2} transform the constant c into the tuple composed by the original constant c, and the constant C_c which represents the cost of computing the constant c.

Rule R_{c3} transforms a conditional expression. The transformed program will first compute the tuple value/cost for the condition expression, and it will bind those values to fresh variables v_1 and t_1. If the value v_1 is $true$ then the program will compute the tuple value/cost of the then branch and bind the values to fresh variables v_2 and c_2, and return a tuple with the value v_2, and the addition of the costs of the conditional expression and the then branch plus the cost of the jump, represented by C_{if}. If the value v_1 is $false$, the value and cost taken are those from the else branch.

The expression (**let** (($\langle v\ \ c\rangle$ *exp*)) *body*) is not part of the language, but it is presented for clarity, instead of the expression (**let** ((*tmp exp*)) (**let** ((*v* (*car tmp*)) (*c* (*cdr tmp*))) *body*)) where *tmp*, v, and c are fresh variables. Similarly, the expression $\langle exp_1\ exp_2\rangle$ is not part of the language, and it is presented instead of $(cons\ exp_1\ exp_2)$.

$$
Rc_0 : \mathcal{T}_c \left[\!\!\left[\begin{array}{l} (\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad e_1) \\ (\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad e_2) \\ \qquad \vdots \\ (\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad e_n) \end{array} \right]\!\!\right] = \begin{array}{l} (\textbf{define}\ (f_1^*\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad \mathcal{T}_{c_e} [\![e_1]\!]) \\ (\textbf{define}\ (f_2^*\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad \mathcal{T}_{c_e} [\![e_2]\!]) \\ \qquad \vdots \\ (\textbf{define}\ (f_n^*\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad \mathcal{T}_{c_e} [\![e_n]\!]) \end{array}
$$

$$
\begin{array}{lll}
Rc_1 : \mathcal{T}_{c_e} [\![v]\!] & = & \langle v\ C_{var} \rangle \\
Rc_2 : \mathcal{T}_{c_e} [\![c]\!] & = & \langle c\ C_c \rangle \\
Rc_3 : \mathcal{T}_{c_e} [\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1 \rangle\ \mathcal{T}_{c_e} [\![e_1]\!])) \\
 & & \quad (\textbf{if}\ v_1 \\
 & & \qquad (\textbf{let}\ ((\langle v_2\ c_2 \rangle\ \mathcal{T}_{c_e} [\![e_2]\!])) \\
 & & \qquad\quad \langle v_2\ (+\ C_{if}\ c_1\ c_2) \rangle) \\
 & & \qquad (\textbf{let}\ ((\langle v_3\ c_3 \rangle\ \mathcal{T}_{c_e} [\![e_3]\!])) \\
 & & \qquad\quad \langle v_3\ (+\ C_{if}\ c_1\ c_3) \rangle))) \\
Rc_4 : \mathcal{T}_{c_e} [\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] & = & (\textbf{let}\ ((\langle v\ c_1 \rangle\ \mathcal{T}_{c_e} [\![e_1]\!])) \\
 & & \quad (\textbf{let}\ ((\langle v_2\ c_2 \rangle\ \mathcal{T}_{c_e} [\![e_2]\!])) \\
 & & \qquad \langle v_2\ (+\ C_{let}\ c_1\ c_2) \rangle)) \\
Rc_5 : \mathcal{T}_{c_e} [\![(cons\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1 \rangle\ \mathcal{T}_{c_e} [\![e_1]\!]) \\
 & & \qquad\quad \vdots \\
 & & \qquad (\langle v_n\ c_n \rangle\ \mathcal{T}_{c_e} [\![e_n]\!])) \\
 & & \quad \langle (cons\ v_1\ \ldots\ v_n)\ (+\ C_{cons}\ c_1\ \ldots\ c_n) \rangle) \\
Rc_6 : \mathcal{T}_{c_e} [\![(prim\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1 \rangle\ \mathcal{T}_{c_e} [\![e_1]\!]) \\
 & & \qquad\quad \vdots \\
 & & \qquad (\langle v_n\ c_n \rangle\ \mathcal{T}_{c_e} [\![e_n]\!])) \\
 & & \quad \langle (prim\ v_1\ \ldots\ v_n)\ (+\ C_{prim}\ c_1\ \ldots\ c_n) \rangle) \\
Rc_7 : \mathcal{T}_{c_e} [\![(f\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1 \rangle\ \mathcal{T}_{c_e} [\![e_1]\!]) \\
 & & \qquad\quad \vdots \\
 & & \qquad (\langle v_n\ c_n \rangle\ \mathcal{T}_{c_e} [\![e_n]\!])) \\
 & & \quad (\textbf{let}\ ((\langle v_0\ c_0 \rangle\ (f^*\ v_1\ \ldots\ v_n))) \\
 & & \qquad \langle v_0\ (+\ C_{call}\ c_0\ c_1\ \ldots\ c_n) \rangle))
\end{array}
$$

FIGURE 3. Rules for *transformation* $\mathcal{T}_c$.

Rule R_{c4} transforms a binding expression, where the tuple returned by the transformation of exp_1 is bound to variable v, and fresh variable t_1. The resulting tuple will contain the value of the second expression, and the sum of C_{let} and the costs for exp_1 and exp_2.

Rule R_{c5} transform a constructor application. The transformation of the n arguments to the constructor are bound to n fresh tuples. The resulting tuple consists of the application of the constructor to the values, and the sum of all the costs for the arguments plus the cost associated with the constructor (C_{prim}).

Rule R_{c6} applies to primitives other than constructors, and it is very similar to rule R_{c5}.

Rule R_{c7} binds the transformed arguments to fresh tuples, and then it binds the tuple resulting from the call to the transformed function f^* to fresh variables v_0, t_0, p_0. The resulting tuple will be composed of value v_0, and the sum of all the costs of the arguments plus the cost of the function plus C_{call} which represents the cost associated with a function call.

Figure 4 Shows the function *member*? after transformation $\mathcal{T}_c$.

```
(define (member?* item ls)
  (let ((⟨v1 c1⟩ (let ((⟨v2 c2⟩ ⟨ls C_var⟩))
                   ⟨(null? v2) (+ C_null? c2)⟩)))
    (if v1
        (let ((⟨v3 c3⟩ ⟨#f C_c⟩))
          ⟨v3 (+ C_if c1 c3)⟩)
        (let ((⟨v4 c4⟩ (let ((⟨v5 c5⟩
                              (let ((⟨v6 c6⟩ ⟨item C_var⟩)
                                    (⟨v7 c7⟩ (let ((⟨v8 c8⟩ ⟨ls C_var⟩))
                                               ⟨(car v8) (+ C_car c8)⟩)))
                                ⟨(eq? v6 c7) (+ C_eq? c6 c7)⟩)))
                         (if v5
                             (let ((⟨v9 c9⟩ ⟨#t C_c⟩))
                               ⟨v9 (+ C_if c5 c9)⟩)
                             (let ((⟨v10 c10⟩
                                    (let ((⟨v11 c11⟩ ⟨item C_var⟩)
                                          (⟨v12 c12⟩
                                           (let ((⟨v13 c13⟩ ⟨ls C_var⟩))
                                             ⟨(cdr v13) (+ C_cdr c13)⟩)))
                                      (let ((⟨v14 c14⟩ (member?* v11 v12)))
                                        ⟨v14 (+ C_call c11 c12 c14)⟩))))
                               ⟨v10 (+ C_if c5 c10)⟩)))))
          ⟨v4 (+ C_if c1 c4)⟩))))
```

FIGURE 4. Program *member*, after transformation $\mathcal{T}_c$.

3. Constructing cost-bound functions

Characterizing program inputs and capturing them in the timing function are difficult to automate. However, partially known input structures provide a natural mean. Special values *unknown* represents unknown values. For example, to capture all input lists of length n, the following partially known input structure can be used.

$$\langle unknown_1, unknown_2, unknown_3, unknown_4, ..., unknown_n \rangle$$

The reason to have unique unknowns is to be able to define constrains in using the unknowns. For example, for the list $\langle unknown_1, unknown_2, unknown_3, unknown_4 \rangle$, the worst-case input for the insert sort algorithm should satisfy the following constraints:

$$unknown_1 < unknown_2$$
$$unknown_2 < unknown_3$$
$$unknown_3 < unknown_4$$

which means the list should be in decreasing order.

To create partially known input structures the following procedure can be used.

```
(define (list n)
  (if (= n 0)
      '()
      (cons (make-unknown) (list (- n 1)))))
```

where *make-unknown* is a procedure with no arguments that returns a unique value *unknown*. 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 values *unknown*. In particular, for each primitive function p, we define a new 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 function *lub* that takes two values and returns the most precise partially known structure that both values conform with. These definitions are shown in Figure 5.

```
(define (f_prim v1 ... vn)
  (if (or (unknown? v1) ... (unknown? vn))
      (make-unknown)
      (prim v1 ... vn)))

(define (lub a b)
  (if (equal? a b)
      a
      (if (unknown? a)
          a
          (if (unknown? b)
              b
              (if (and v1 is (c1 x1 ... xi)
                       v2 is (c2 y1 ... yj)
                       c1 = c2
                       i = j)
                  (c1 (lub x1 y1) ... (lub xi yi))
                  (make-unknown))))))
```

FIGURE 5. Redefinition of primitives and definition of the new *lub* function.

The program transformation is defined by transformation $\mathcal{T}_b$ shown in Figure 6. This transformation is very similar to $\mathcal{T}_c$. The only difference is in the treatment of if expressions, which should now take *unknown* into account, and in the treatment of a primitive function, which now calls the new f_{prim} function.

Rule R_{b3} transforms a conditional expression. The transformed program will first compute the tuple value/cost for the condition expression, and it will bind those values to fresh variables v_1 and c_1. If the value of the condition expression is *unknown* then the program will compute the tuple for both branches and return a tuple with the least upper bound of the values, the cost of the jump plus the cost of the condition expression plus the maximum cost of the two branches. If the value of the condition expression is not *unknown* then the program will take the appropriate branch and return the tuple with the value of the then branch if the condition is *true* or the value

$$R_{\mathcal{T}_b 0} : \mathcal{T}_b \left[\!\!\left[\begin{array}{l} (\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad exp_1) \\ (\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad exp_2) \\ \quad \vdots \\ (\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad exp_n) \end{array} \right]\!\!\right] = \begin{array}{l} (\textbf{define}\ (f^*_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad \mathcal{T}_{b_e}[\![exp_1]\!]) \\ (\textbf{define}\ (f^*_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad \mathcal{T}_{b_e}[\![exp_2]\!]) \\ \quad \vdots \\ (\textbf{define}\ (f^*_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad \mathcal{T}_{b_e}[\![exp_n]\!]) \end{array}$$

$$\begin{array}{lll}
R_{\mathcal{T}_{b_e}1} : \mathcal{T}_{b_e}[\![v]\!] & = & \langle v\ C_{var}\rangle \\
R_{\mathcal{T}_{b_e}2} : \mathcal{T}_{b_e}[\![c]\!] & = & \langle v\ C_c\rangle \\
R_{\mathcal{T}_{b_e}3} : \mathcal{T}_{b_e}[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1\rangle\ \mathcal{T}_{b_e}[\![e_1]\!])) \\
 & & \quad (\textbf{if}\ (unknown?\ v_1) \\
 & & \quad\quad (\textbf{let}\ ((\langle v_2\ c_2\rangle\ \mathcal{T}_{b_e}[\![e_2]\!]) \\
 & & \quad\quad\quad\quad (\langle v_3\ c_3\rangle\ \mathcal{T}_{b_e}[\![e_3]\!])) \\
 & & \quad\quad\quad (\textbf{if}\ (>\ c_2\ c_3) \\
 & & \quad\quad\quad\quad \langle (lub\ v_2\ v_3)\ (+\ C_{if}\ c_1\ c_2)\rangle \\
 & & \quad\quad\quad\quad \langle (lub\ v_3\ v_2)\ (+\ C_{if}\ c_1\ c_3)\rangle)) \\
 & & \quad\quad (\textbf{if}\ v_1 \\
 & & \quad\quad\quad (\textbf{let}\ ((\langle v_2\ c_2\rangle\ \mathcal{T}_{b_e}[\![e_2]\!])) \\
 & & \quad\quad\quad\quad \langle v_2\ (+\ C_{if}\ c_1\ c_2)\rangle) \\
 & & \quad\quad\quad (\textbf{let}\ ((\langle v_3\ c_3\rangle\ \mathcal{T}_{b_e}[\![e_3]\!])) \\
 & & \quad\quad\quad\quad \langle (lub\ v_3\ v_2)\ (+\ C_{if}\ c_1\ c_3)\rangle))))) \\
R_{\mathcal{T}_{b_e}4} : \mathcal{T}_{b_e}[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] & = & (\textbf{let}\ ((\langle v\ c_1\rangle\ \mathcal{T}_{b_e}[\![e_1]\!])) \\
 & & \quad (\textbf{let}\ ((\langle v_2\ c_2\rangle\ \mathcal{T}_{b_e}[\![e_2]\!])) \\
 & & \quad\quad \langle v_2\ (+\ C_{let}\ c_1\ c_2)\rangle)) \\
R_{\mathcal{T}_{b_e}5} : \mathcal{T}_{b_e}[\![(cons\ e_1\ ...\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1\rangle\ \mathcal{T}_{b_e}[\![e_1]\!]) \\
 & & \quad\quad \vdots \\
 & & \quad\quad (\langle v_n\ c_n\rangle\ \mathcal{T}_{b_e}[\![e_n]\!])) \\
 & & \quad \langle (cons\ v_1\ ...\ v_n)\ (+\ C_{cons}\ c_1\ ...\ c_n)\rangle) \\
R_{\mathcal{T}_{b_e}6} : \mathcal{T}_{b_e}[\![(prim\ e_1\ ...\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1\rangle\ \mathcal{T}_{b_e}[\![e_1]\!]) \\
 & & \quad\quad \vdots \\
 & & \quad\quad (\langle v_n\ c_n\rangle\ \mathcal{T}_{b_e}[\![e_n]\!])) \\
 & & \quad \langle (f_{prim}\ v_1\ ...\ v_n)\ (+\ C_{prim}\ c_1\ ...\ c_n)\rangle) \\
R_{\mathcal{T}_{b_e}7} : \mathcal{T}_{b_e}[\![(f\ e_1\ ...\ e_n)]\!] & = & (\textbf{let}\ ((\langle v_1\ c_1\rangle\ \mathcal{T}_{b_e}[\![e_1]\!]) \\
 & & \quad\quad \vdots \\
 & & \quad\quad (\langle v_n\ c_n\rangle\ \mathcal{T}_{b_e}[\![e_n]\!])) \\
 & & \quad (\textbf{let}\ ((\langle v_0\ c_0\ (f^*\ v_1\ ...\ v_n)))) \\
 & & \quad\quad \langle v_0\ (+\ C_{call}\ c_0\ c_1\ ...\ c_n)\rangle)))
\end{array}$$

FIGURE 6. Rules for Time-Bound transformation $\mathcal{T}_b$

of the else branch if the condition is $false$, and the sum of the cost C_{if}, the cost of the condition expression and the cost of the branch taken.

Rule R_{b6} applies to primitives other than constructors, and it is very similar to rule R_{c5} from transformation $\mathcal{T}_c$. The difference is the value part of the resulting tuple which calls function f_{prim} instead of primitive $prim$.

The other rules are the same as in transformation $\mathcal{T}_c$, and so are not described here.

Example. Figure 7 show function *member*? after transformation $\mathcal{T}_b$.

```
(define (member?* item ls)
  (let ((⟨v_1 c_1⟩ (let ((⟨v_2 c_2⟩ ⟨ls C_var⟩))
                         ⟨(f_null? v_2) (+ C_null? c_2)⟩))))
    (if (unknown? v_1)
        (let ((⟨v_3 c_3⟩ ⟨#f C_c⟩)
              (⟨v_4 c_4⟩ expr_1))
          (if (> c_3 c_4)
              ⟨(lub v_3 v_4) (+ C_if c_1 c_3)⟩
              ⟨(lub v_4 v_3) (+ C_if c_1 c_4)⟩))
        (if v_1
            (let ((⟨v_3 c_3⟩ ⟨#f c_c⟩))
              ⟨v_3 (+ C_if c_1 c_3)⟩)
            (let ((⟨v_4 c_4⟩ expr1))
              ⟨v_4 (+ C_if c_1 c_4)⟩))))))
;; where expr_1 is
  (let ((⟨v_5 c_5⟩ (let ((⟨v_6 c_6⟩ ⟨item C_var⟩)
                        (⟨v_7 c_7⟩ (let ((⟨v_8 c_8⟩ ⟨ls C_var⟩))
                                         ⟨(f_car v_8) (+ C_car c_8)⟩))))
                     ⟨(f_eq? v_6 v_7) (+ C_eq? c_6 c_7)⟩))))
    (if (unknown? v_5)
        (let ((⟨v_9 c_9⟩ ⟨#t C_c⟩)
              (⟨v_10 c_10⟩ expr_2))
          (if (> c_9 c_10)
              ⟨(lub v_9 v_10) (+ C_if c_5 c_9)⟩
              ⟨(lub v_10 v_9) (+ C_if c_5 c_10)⟩))
        (if v_5
            (let ((⟨v_9 c_9⟩ ⟨#t C_c⟩))
              ⟨v_9 (+ C_if c_5 c_9)⟩)
            (let ((⟨v_10 c_10⟩ expr_2))
              ⟨v_10 (+ C_if c_5 c_10)⟩)))))
;; where expr_2 is
  (let ((⟨v_11 c_11⟩ ⟨item C_var⟩)
        (⟨v_12 c_12⟩ (let ((⟨v_13 c_13⟩ ⟨ls C_var⟩))
                           ⟨(f_cdr c_13) (+ C_car c_13)⟩))))
    (let ((⟨v_14 c_14⟩ (member?* v_11 v_12)))
      ⟨v_14 (+ C_call c_11 c_12 c_14)⟩)))
```

FIGURE 7. Function *member*? after transformation $\mathcal{T}_b$

4. Collecting worst case constraints

In order to collect the constraints that will give us the worst-case input, we need to modify the program to return the set of constraints along with the original value and the bound cost of the expressions. The program transformation is defined by $\mathcal{T}_p$ shown in Figure 8. This transformation is very similar to $\mathcal{T}_b$, but now every transformed expression returns a triple instead of a tuple. This triple contains the original value, the cost, and the set of constraints. For example, in the transformation of a function call, we collect the triple for all the arguments and for the actual call, and the resulting triple will have the value and cost as in $\mathcal{T}_b$, and the union of all the constraints returned by the arguments and the call.

Rule R_{p0} defines a function f^* for every function f in the original program. The body of this function will be the original body transformed with expression transformer $\mathcal{T}_{p_e}$.

Rule R_{p1} transforms a variable reference v into the triple composed by the variable v, the constant C_{var} which represents the cost of computing a variable reference, and the empty set which means there are no constraints to satisfy to get that cost.

Rule R_{p2} transform the constant c into the triple composed by the original constant c, the constant C_c which represents the cost of computing the constant c, and the empty set.

Rule R_{p3} transforms a conditional expression. This rule is very similar to rule R_{b3} from transformation $\mathcal{T}_b$. The constraints part of the triple is computed as follows. If the value of the condition expression is not *unknown*, then the constraints for the transformed expression will be the union of the constraints in the condition expression and the constraints of the appropriate branch. If the value of the condition expression is *unknown* then the constraints will include the condition expression if the then

$$R_{\mathcal{T}_p 0} : \mathcal{T}_p \left[\!\!\left[\begin{array}{l} (\textbf{define}\ (f_1\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad exp_1) \\ (\textbf{define}\ (f_2\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad exp_2) \\ \quad\vdots \\ (\textbf{define}\ (f_n\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad exp_n) \end{array} \right]\!\!\right] = \begin{array}{l} (\textbf{define}\ (f_1^*\ v_{1_1}\ v_{1_2}\ \ldots\ v_{1_k}) \\ \quad \mathcal{T}_{p_e}[\![exp_1]\!]) \\ (\textbf{define}\ (f_2^*\ v_{2_1}\ v_{2_2}\ \ldots\ v_{2_k}) \\ \quad \mathcal{T}_{p_e}[\![exp_2]\!]) \\ \quad\vdots \\ (\textbf{define}\ (f_n^*\ v_{n_1}\ v_{n_2}\ \ldots\ v_{n_k}) \\ \quad \mathcal{T}_{p_e}[\![exp_n]\!]) \end{array}$$

$$\begin{array}{lll}
R_{\mathcal{T}_{p_e}1} : \mathcal{T}_{p_e}[\![v]\!] & = & \langle v\ C_{var}\ \emptyset\rangle \\
R_{\mathcal{T}_{p_e}2} : \mathcal{T}_{p_e}[\![c]\!] & = & \langle v\ C_{c}\ \emptyset\rangle \\
R_{\mathcal{T}_{p_e}3} : \mathcal{T}_{p_e}[\![(\textbf{if}\ e_1\ e_2\ e_3)]\!] & = & (\textbf{let}\ (((\langle v_1\ c_1\ p_1\rangle\ \mathcal{T}_{p_e}[\![e_1]\!])) \\
& & \quad (\textbf{if}\ (unknown?\ v_1) \\
& & \quad\quad (\textbf{let}\ ((\langle v_2\ c_2\ p_2\rangle\ \mathcal{T}_{p_e}[\![e_2]\!]) \\
& & \quad\quad\quad\quad (\langle v_3\ c_3\ p_3\rangle\ \mathcal{T}_{p_e}[\![e_3]\!])) \\
& & \quad\quad\quad (\textbf{if}\ (>\ c_2\ c_3) \\
& & \quad\quad\quad\quad \langle (lub\ v_2\ v_3)\ (+\ C_{if}\ c_1\ c_2)\ (\cup\ p_1\ p_2\ \{\text{'e}_1\})\rangle \\
& & \quad\quad\quad\quad \langle (lub\ v_3\ v_2)\ (+\ C_{if}\ c_1\ c_3)\ (\cup\ p_1\ p_3\ \{\text{'(not e}_1)\})\rangle))) \\
& & \quad\quad (\textbf{if}\ v_1 \\
& & \quad\quad\quad (\textbf{let}\ (((\langle v_2\ c_2\ p_2\rangle\ \mathcal{T}_{p_e}[\![e_2]\!])) \\
& & \quad\quad\quad\quad \langle v_2\ (+\ C_{if}\ c_1\ c_2)\ (\cup\ p_1\ p_2)\rangle) \\
& & \quad\quad\quad (\textbf{let}\ (((\langle v_3\ c_3\ p_3\rangle\ \mathcal{T}_{p_e}[\![e_3]\!])) \\
& & \quad\quad\quad\quad \langle v_3\ (+\ C_{if}\ c_1\ c_3)\ (\cup\ p_1\ p_3)\rangle))))) \\
R_{\mathcal{T}_{p_e}4} : \mathcal{T}_{p_e}[\![(\textbf{let}\ ((v\ e_1))\ e_2)]\!] & = & (\textbf{let}\ (((\langle v\ c_1\ p_1\rangle\ \mathcal{T}_{p_e}[\![e_1]\!])) \\
& & \quad (\textbf{let}\ (((\langle v_2\ c_2\ p_2\rangle\ \mathcal{T}_{p_e}[\![e_2]\!])) \\
& & \quad\quad \langle v_2\ (+\ C_{let}\ c_1\ c_2)\ (\cup\ p_1\ p_2)\rangle))) \\
R_{\mathcal{T}_{p_e}5} : \mathcal{T}_{p_e}[\![(cons\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ (((\langle v_1\ c_1\ p_1\rangle\ \mathcal{T}_{p_e}[\![e_1]\!]) \\
& & \quad\quad\quad \vdots \\
& & \quad\quad ((\langle v_n\ c_n\ p_n\rangle\ \mathcal{T}_{p_e}[\![e_n]\!])) \\
& & \quad \langle (cons\ v_1\ \ldots\ v_n)\ (+\ C_{cons}\ c_1\ \ldots\ c_n)\ (\cup\ p_1\ \ldots\ p_n)\rangle) \\
R_{\mathcal{T}_{p_e}6} : \mathcal{T}_{p_e}[\![(prim\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ (((\langle v_1\ c_1\ p_1\rangle\ \mathcal{T}_{p_e}[\![e_1]\!]) \\
& & \quad\quad\quad \vdots \\
& & \quad\quad ((\langle v_n\ c_1\ p_n\rangle\ \mathcal{T}_{p_e}[\![e_n]\!])) \\
& & \quad \langle (f_{prim}\ v_1\ \ldots\ v_n)\ (+\ C_{prim}\ c_1\ \ldots\ c_n)\ (\cup\ p_1\ \ldots\ p_n)\rangle) \\
R_{\mathcal{T}_{p_e}7} : \mathcal{T}_{p_e}[\![(f\ e_1\ \ldots\ e_n)]\!] & = & (\textbf{let}\ (((\langle v_1\ c_1\ p_1\rangle\ \mathcal{T}_{p_e}[\![e_1]\!]) \\
& & \quad\quad\quad \vdots \\
& & \quad\quad ((\langle v_n\ c_n\ p_n\rangle\ \mathcal{T}_{p_e}[\![e_n]\!])) \\
& & \quad (\textbf{let}\ (((\langle v_0\ c_0\ p_0\ (f^*\ v_1\ \ldots\ v_n))) \\
& & \quad\quad \langle v_0\ (+\ C_{call}\ c_0\ c_1\ \ldots\ c_n)\ (\cup\ p_0\ p_1\ \ldots\ p_n)\rangle)))
\end{array}$$

FIGURE 8. Rules for Time-Bound transformation $\mathcal{T}_p$

branch has a higher cost than the else branch, or it will include the negation of the condition expression otherwise.

Rules R_{p4}, R_{p5}, R_{p6}, and R_{p7} are similar to their corresponding rules in transformation $\mathcal{T}_b$, while also they collect the constraints of the subexpressions.

Example. Figure 9 show function *member*? after transformation $\mathcal{T}_p$.

$$
\begin{array}{l}
(\textbf{define}\ (\mathit{member?^*}\ \mathit{item}\ \mathit{ls})\\
\quad(\textbf{let}\ (((\langle v_1\ c_1\ p_1\rangle\ (\textbf{let}\ (((\langle v_2\ c_2\ p_2\rangle\ \langle \mathit{ls}\ C_{var}\ \emptyset\rangle)))\\
\qquad\qquad\qquad\qquad\qquad\qquad \langle(f_{null?}\ v_2)\ (+\ C_{null?}\ c_2)\ p_2\rangle))))\\
\qquad(\textbf{if}\ (\mathit{unknown?}\ v_1)\\
\qquad\quad(\textbf{let}\ (((\langle v_3\ c_3\ p_3\rangle\ \langle\#\mathsf{f}\ C_c\ \emptyset\rangle)\\
\qquad\qquad\quad(\langle v_4\ c_4\ p_4\rangle\ \mathit{expr}_1))\\
\qquad\qquad(\textbf{if}\ (>\ c_3\ c_4)\\
\qquad\qquad\quad\langle(\mathit{lub}\ v_3\ v_4)\ (+\ C_{if}\ c_1\ c_3)\ (\cup\ p_1\ p_3\ \{\text{'(null? ls)}\})\rangle\\
\qquad\qquad\quad\langle(\mathit{lub}\ v_4\ v_3)\ (+\ C_{if}\ c_1\ c_4)\ (\cup\ p_1\ p_4\ \{\text{'(not (null? ls))}\})\rangle)))\\
\qquad\quad(\textbf{if}\ v_1\\
\qquad\qquad(\textbf{let}\ (((\langle v_3\ c_3\ p_3\rangle\ \langle\#\mathsf{f}\ C_c\ \emptyset\rangle))\\
\qquad\qquad\quad\langle v_3\ (+\ C_{if}\ c_1\ c_3)\ (\cup\ p_1\ p_3)\rangle)\\
\qquad\qquad(\textbf{let}\ (((\langle v_4\ c_4\ p_4\rangle\ \mathit{expr}_1))\\
\qquad\qquad\quad\langle v_4\ (+\ C_{if}\ c_1\ c_4)\ (\cup\ p_1\ p_4)\rangle))))))
\end{array}
$$

;; where expr_1 is

$$
\begin{array}{l}
\quad(\textbf{let}\ (((\langle v_5\ c_5\ p_5\rangle\ (\textbf{let}\ (((\langle v_6\ c_6\ p_6\rangle\ \langle \mathit{item}\ C_{var}\ \emptyset\rangle)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\langle v_7\ c_7\ p_7\rangle\ (\textbf{let}\ (((\langle v_8\ c_8\ p_8\rangle\ \langle \mathit{ls}\ C_{var}\ \emptyset\rangle)))\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle(f_{car}\ v_8)\ (+\ C_{car}\ c_8)\ p_8\rangle))))\\
\qquad\qquad\qquad\qquad\qquad \langle(f_{eq?}\ v_6\ v_7)\ (+\ C_{eq?}\ c_6\ c_7)\ (\cup\ p_6\ p_7)\rangle))))\\
\qquad(\textbf{if}\ (\mathit{unknown?}\ v_5)\\
\qquad\quad(\textbf{let}\ (((\langle v_9\ c_9\ p_9\rangle\ \langle\#\mathsf{t}\ C_c\ \emptyset\rangle)\\
\qquad\qquad\quad(\langle v_{10}\ c_{10}\ p_{10}\rangle\ \mathit{expr}_2))\\
\qquad\qquad(\textbf{if}\ (>\ c_9\ c_{10})\\
\qquad\qquad\quad\langle(\mathit{lub}\ v_9\ v_{10})\ (+\ C_{if}\ c_5\ c_9)\ (\cup\ p_5\ p_9\ \{\text{'(eq? item (car ls))}\})\rangle\\
\qquad\qquad\quad\langle(\mathit{lub}\ v_{10}\ v_9)\ (+\ C_{if}\ c_5\ c_{10})\ (\cup\ p_5\ p_{10}\ \{\text{'(not (eq? item (car ls)))}\})\rangle)))\\
\qquad\quad(\textbf{if}\ v_5\\
\qquad\qquad(\textbf{let}\ (((\langle v_9\ c_9\ p_9\rangle\ \langle\#\mathsf{t}\ C_c\ \emptyset\rangle))\\
\qquad\qquad\quad\langle v_9\ (+\ c_{if}\ c_5\ c_9)\ (\cup\ p_5\ p_9)\rangle)\\
\qquad\qquad(\textbf{let}\ (((\langle v_{10}\ c_{10}\ p_{10}\rangle\ \mathit{expr}_2))\\
\qquad\qquad\quad\langle v_{10}\ (+\ C_{if}\ c_5\ c_{10})\ (\cup\ p_5\ p_{10})\rangle)))))
\end{array}
$$

;; where expr_2 is

$$
\begin{array}{l}
\quad(\textbf{let}\ (((\langle v_{11}\ c_{11}\ p_{11}\rangle\ \langle \mathit{item}\ C_{var}\ \emptyset\rangle)\\
\qquad\quad(\langle v_{12}\ c_{12}\ p_{12}\rangle\ (\textbf{let}\ (((\langle v_{13}\ c_{13}\ p_{13}\rangle\ \langle \mathit{ls}\ C_{var}\ \emptyset\rangle)))\\
\qquad\qquad\qquad\qquad\qquad \langle(f_{cdr}\ c_{13})\ (+\ C_{car}\ c_{13})\ p_{13}\rangle))))\\
\qquad(\textbf{let}\ (((\langle v_{14}\ c_{14}\ p_{14}\rangle\ (\mathit{member?^*}\ v_{11}\ v_{12})))\\
\qquad\quad\langle v_{14}\ (+\ C_{call}\ c_{11}\ c_{12}\ c_{14})\ (\cup\ p_{11}\ p_{12}\ p14)\rangle)))
\end{array}
$$

FIGURE 9. Function *member?* after transformation $\mathcal{T}_p$

It is important to instantiate the variables inside a constraint to be added. This means, instead of "(*null? ls*)" we should have (for example) "(*null? unknown*$_3$)". The constraints in the listing above are presented as the former for brevity, but they actually represents the latter.

5. Optimization

As a result of the simple mechanical transformation, the resulting program has several inefficiencies. For example, in the definition of *member?**, the value of variable c_5 is always $(+\ C_{eq?}\ C_{var}\ C_{car}\ C_{var})$, and at every iteration in the loop the program performs those three additions (along with the corresponding variable bindings). Also, the value of variable c_9 is always C_c, and the value of c_{10} is always $(+\ C_{call}\ C_{var}\ C_{cdr}\ C_c\ c_{14})$, which means c_9 is never greater than c_{10} so we can avoid the test (**if** $(>\ c_9\ c_{10})$ ...) and execute only the else branch.

Another inefficiency comes from the triple construction and destruction. For every subexpression there is a construction of a triple, and an immediate destruction. Instead of

$$
\begin{array}{l}
(\textbf{let}\ ((\mathit{tmp}\ \langle a\ \ b\ \ c\rangle))\\
\quad(\textbf{let}\ ((v\ (\mathit{car}\ \mathit{tmp}))\\
\qquad\qquad(c\ (\mathit{cadr}\ \mathit{tmp}))\\
\qquad\qquad(p\ (\mathit{caddr}\ \mathit{tmp})))\\
\qquad\mathit{body}))
\end{array}
$$

we can have

$$
\begin{array}{l}
(\textbf{let}\ ((v\ \ a)\\
\qquad\quad(c\ \ b)\\
\qquad\quad(p\ \ c))\\
\quad\mathit{body})
\end{array}
$$

The only triples we must create now are the triples in tail position in the body of a function, and conversely the only triples to destruct are the resulting from function calls.

The optimizations are implemented using a modified transformer and an abstract interpreter. The modified transformer implements transformation $\mathcal{T}_p$, but it avoids creating triples when possible. It uses a technique similar to Destination Driven Code Generation [**26**]. In this case, the subexpression transformer receives the variable names where the values of the triple should be stored, and the transformed program will bind them directly if possible, or through triple construction/deconstruction if not (for example, when there is a function call).

The abstract interpreter receives a program and returns an optimized program. The domain of the interpreter is the set of tagged expressions, where the tag is the type of the expression (if known). The special forms are not tagged. For example, the expression (+ 3 5) is fed to the interpreter as (⟨*procedure* +⟩ ⟨*number* 3⟩ ⟨*number* 5⟩) and the more complicated expression (**if** (*null?* (*cons* 1 '())) 5 4) is fed as (**if** (⟨*procedure null?*⟩ (⟨*procedure cons*⟩ ⟨*number* 1⟩ ⟨*null* '()⟩)) ⟨*number* 4⟩ ⟨*number* 8⟩). The primitive procedures are clever enough to do constant folding, so the interpreter returns ⟨*number* 8⟩ for both previous expressions. The interpreter is also capable of doing copy propagation. To do this, a variable reference is transformed into its value, if it is a constant. For example, the in the expression (**let** ((x 5)) x) the reference x is transformed into ⟨*number* 5⟩. The interpreter also does useless binding elimination, so the previous expression is evaluated finally to ⟨*number* 5⟩.

Example. Figure 10 show function *member*? after the optimizations.

Notice that $eq?(item, car(ls))$ cannot be a constraint anymore, and the only triple constructions are in tail position, and the only triple destructions are at function call sites.

6. Constructing worst-case inputs

The output of the transformed program is a set of constraints that the worst case input must satisfy. For example, with two lists of size three, the system says the worst input case for the program union

$$union([unknown_0, unknown_1, unknown_2], [unknown_3, unknown_4, unknown_5])$$

should satisfy the following constrains:

```
(define (member?* item ls)
  (let ((v1 (f_null? ls)))
    (if (unknown? v1)
      (let ((v7 (f_car ls)))
        (let ((v5 (f_eq? item v7)))
          (let (((⟨v4 c4 p4⟩
                  (if (unknown? v5)
                    (let ((v6 (f_cdr ls)))
                      (let (((⟨v14 c14 p14⟩ (member?* item v6)))
                        ⟨(lub v14 #t) (+ c14 302.325) (∪ p14 {'(not (eq? item (car ls)))})⟩)))
                    (if v5
                      ⟨#t 162.25 ∅⟩
                      (let ((v6 (f_cdr ls)))
                        (let (((⟨v14 c14 p14⟩ (member?* item v6)))
                          ⟨v14 (+ c14 302.325) p14⟩)))))))
            (if (> 0.25 c4)
              ⟨(lub #f v4) 79.35 (∪ p4 {'(null? ls)})⟩
              ⟨(lub v4 #f) (+ c4 79.1) (∪ p4 {'(not (null? ls))})⟩)))))
      (if v1
        ⟨#f 79.35 ∅⟩
        (let ((v7 (f_car ls)))
          (let ((v5 (f_eq? item v7)))
            (let (((⟨v4 c4 p4⟩
                    (if (unknown? v5)
                      (let ((v6 (f_cdr ls)))
                        (let (((⟨v14 c14 p14⟩ (member?* item v6)))
                          ⟨(lub v14 #t) (+ c14 302.325) (∪ p14 {'(not (eq? item (car ls)))})⟩)))
                      (if v5
                        ⟨#t 162.25 ∅⟩
                        (let ((v6 (f_cdr ls)))
                          (let (((⟨v14 c14 p14⟩ (member?* item v6)))
                            ⟨v14  (+ c14 302.325) p14⟩)))))))
              ⟨v4 (+ c4 79.1) p4⟩)))))))
```

FIGURE 10. Function *member*? after transformation $\mathcal{T}_b$

$$unknown_3 \neq unknown_0 \wedge unknown_3 \neq unknown_1 \wedge unknown_3 \neq unknown_2 \wedge$$
$$unknown_4 \neq unknown_0 \wedge unknown_4 \neq unknown_1 \wedge unknown_4 \neq unknown_2 \wedge$$
$$unknown_5 \neq unknown_0 \wedge unknown_5 \neq unknown_1 \wedge unknown_5 \neq unknown_2$$

Once we have the constraints we use the Omega Calculator to obtain an actual input. The Omega Calculator (OC) is a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets. We feed the constraints to the OC and it will simplify the constraints. For example, for the constraint

$$(unknown_0 < unknown_1) \wedge (unknown_0 < unknown_2) \wedge (unknown_1 < unknown_2)$$

OC will reply with

$$unknown_0 < unknown_1 < unknown_2$$

.

The process to obtain an actual input is iterative. At each iteration we look at the constraints and apply the following rules.

(1) For every constraint that involves only one variable, a constant and a $\leq$ (or $\geq$) operator, we equate the variable to that constant. For example, if the constraint is $unknown_2 \leq 4$ we define $unknown_2 = 4$.

(2) If no constraint satisfies the first rule, then for every constraint that involves only one variable, a constant and a $<$ (or $>$) operator, we equate the variable to that constant minus (or plus) 1. For example, if the constraint is $unknown_1 < 4$ then we define $unknown_1 = 3$.

(3) If no constraint satisfies the first 2 rules, then we choose any variable and equate the variable to any constant.

The iterative process ends when there are no more variables to define.

Example. After following this rules to the constraint for the *union* program with sets of size 3, OC says that the worst case input are the lists $[0, 0, 0]$ and $[1, 1, 1]$. Notice that those lists are not really sets, but nowhere in the definition of union implies that the arguments are sets. Since we want a valid input for the worst case, we may want to introduce those constraints manually($unknown_0 \neq unknown_1$, $unknown_0 \neq unknown_2$ and so on). After the introduction of these constraints, the OC comes up with the input $[0, -1, -3]$ and $[-2, 2, 1]$.

Following this method, the worst input cases for insert sort, select sort and merge sort, on a list of size 16 are respectively a list with all zeroes, a list in decreasing order, and the list $[0, -3, -2, -3, -1, -4, -3, -4, 0, -6, -5, -6, -4, -7, -6, -7]$. For these simple cases it is easy to verify that those are indeed examples of worst case inputs for each algorithm.

7. Discussion

We have described a method to automatically construct an input that exhibits the worst case behavior for a program. This method is completely automatic and based in high-level language constructs. Since the method is automatic, there is no need for the programmer to annotate the program, and there is no danger of the annotation mismatching the program and other problems described by De Millo et. al.[**18**].

Under certain conditions, this method generates constraints that are not satisfiable. This means that the bound found is not realizable. Consider the expression

$$(+\ (\textbf{if}\ (>\ x\ 0)\ (*\ x\ 2)\ 2)\ (\textbf{if}\ (>\ x\ 0)\ 2\ (*\ x\ 2)))$$

The set of constraints for the first part is $\{x > 0\}$, and the set of constraint for the second part is $\{x \not> 0\}$. The set of constraints for the whole expression is then $\{x > 0, x \not> 0\}$ which of course is unsatisfiable. The solution is to lift the conditions, as is done in Chapter 2.

8. Implementation and experimentation

We have implemented this technique 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 functional subset of Scheme. The prototype is implemented using Chez Scheme v6.9 compiler [**25**] and the Omega Calculator. The input to the system is a program as defined in Section 1, but with Scheme syntax. The output from the system is an actual input with the worst case execution cost if the system was able to find one. The implementations consists of 1000 lines of scheme code, not counting comments or blank lines.

The computer used to take the measurements is a Sun Enterprise 450 Model 4400 with four 400MHz CPUs, 2 GB of RAM, and 6.6 GB virtual memory.

8.1. Experiments. The implementation was tested with a small set of functional programs. The program *index* takes an item and a list, and return the index of such item in the list, if it is there, or -1 if it is not. The program *union* computes the union of two sets. The program *selectsort* sorts a list of numbers in increasing order using the selection sort algorithm. The program *insertsort* sorts a list of numbers in increasing order using the insertion sort algorithm. We used three different implementations of merge sort: *traditional* merge sort (split in first half and second half), *odd–even* merge sort (split in even-indexed and odd-indexed items) and *bottom–up* merge sort (split in lists of one element). The program pqueue maintains a priority queue. The program *bigindex* computes the index of an item in a list (the same as the program *index*), but it is 700 lines long. It is there as a quick attempt to test the scalability of this technique. It is that long because it has 100 functions, where f_0 calls f_1 if there is need for recursion, f_1 calls f_2 and so on, where f_{100} calls f_1.

The translation times for some of the programs are shown in Table 1. The transformations appear to run in linear time, with respect to the size of the program, as expected. It is easy to see from structural induction that the transformation $\mathcal{T}_p$ is applied only once to each subexpression in a program. From the table we can also see that the optimization pass is the most expensive in the transformation, taking up to 80% of the total transformation time.

TABLE 1. Translation times with and without optimizations enabled. Lines of code of each program. Times in milliseconds.

Program:	index	insert	union	select	odd-even	bottom-up	traditional	bigindex
Without:	4.02	5.70	5.94	8.76	12.18	15.72	20.34	630.50
With:	14.40	20.24	22.82	30.18	73.10	80.88	93.50	849.90
LOC:	9	11	14	18	23	31	34	703

Table 2 shows the execution times of the constraint generator programs for *insert–sort* and *union* for input lists of various sizes. For an input of size 10, the optimized version of *insert–sort** executes in 68% of the time of the non optimized version. For an input of size 1000, the optimized version executes in 12% of the time of the non optimized version.

TABLE 2. Execution times for the constraint generator for *insert–sort* and *union* procedures. Times for optimized (o) and not optimized (d) programs. Times in milliseconds.

Input Size	10	20	100	200	500	1000
Insert Sort (n)	0.56	2.60	200.0	1505.0	23512.4	199637.0
Insert Sort (o)	0.38	1.97	76.0	428.1	3923.0	24192.0
Union (n)	0.90	4.00	150.0	737.3	7858.0	55710.0
Union (o)	0.35	1.16	42.5	237.0	2112.6	11302.0

CHAPTER 6

Conclusion

An overview of comparison with related work in time analysis appears in Chapter 1, Section 1. 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 [**61, 88, 87, 104**], this work counts symbolic primitive parameters precisely, 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 [**33**], which has a different goal than worst-case bounds. Compared to work in systems [**91, 76, 75, 62**], 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 [**86, 85**], 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 [**81**] 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 [**53**], 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 [**53, 52, 17**] have been proposed for reasoning about space and time bounds, and some of them include implementations of type checkers [**53, 17**]. 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 [**53, 52**].

A number of techniques have been studied for obtaining loop bounds or execution paths for analyzing time bound [**75, 2, 29, 44, 47, 11**]. Manual annotations [**75, 62**] 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 [**29**], separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [**70**]. Second, approximations for merging paths from loops, or recursions, very often lead to nontermination of the time analysis, not just looser bounds [**29, 44, 70**]. Some new methods, while powerful, apply only to certain classes of programs [**47**]. 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 [**31**], 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 [**70**] 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 [**84**]. 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 [**8, 57**], incremental computation [**68, 67**], 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 [**76**]. 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 [**62, 31**]. In particular, we have started applying our general approach to analyze space consumption [**95**] 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.

APPENDIX A

Tables

TABLE 1. Calculated and measured worst-case times (in milliseconds), without garbage collection.

	insertion sort			selection sort			merge sort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.06751	0.06500	96.3	0.13517	0.12551	92.9	0.11584	0.11013	95.1
20	0.25653	0.25726	100.3	0.52945	0.47750	90.2	0.29186	0.27546	94.4
50	1.55379	1.48250	95.4	3.26815	3.01125	92.1	0.92702	0.85700	92.4
100	6.14990	5.86500	95.4	13.0187	11.9650	91.9	2.15224	1.98812	92.4
200	24.4696	24.3187	99.4	51.9678	47.4750	91.4	4.90017	4.57200	93.3
300	54.9593	53.8714	98.0	116.847	107.250	91.8	7.86231	7.55600	96.1
500	152.448	147.562	96.8	324.398	304.250	93.8	14.1198	12.9800	91.9
1000	609.146	606.000	99.5	1297.06	1177.50	90.8	31.2153	28.5781	91.6
2000	2435.29	3081.25	126.5	5187.17	5482.75	105.7	68.3816	65.3750	95.6

	set union			list reversal			reversal w/app.		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.10302	0.09812	95.2	0.00918	0.00908	98.8	0.05232	0.04779	91.3
20	0.38196	0.36156	94.7	0.01798	0.01661	92.4	0.19240	0.17250	89.7
50	2.27555	2.11500	92.9	0.04436	0.04193	94.5	1.14035	1.01050	88.6
100	8.95400	8.33250	93.1	0.08834	0.08106	91.8	4.47924	3.93600	87.9
200	35.5201	33.4330	94.1	0.17629	0.16368	92.9	17.7531	15.8458	89.3
300	79.6987	75.1000	94.2	0.26424	0.24437	92.5	39.8220	35.6328	89.5
500	220.892	208.305	94.3	0.44013	0.40720	92.5	110.344	102.775	93.1
1000	882.094	839.780	95.2	0.87988	0.82280	93.5	440.561	399.700	90.7
2000	3525.42	3385.31	96.0	1.75937	1.65700	94.2	1760.61	2235.75	127.0

Table 2. Calculated and measured worst-case times (in milliseconds), with garbage collection.

	insertion sort			selection sort			merge sort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.06844	0.06698	97.9	0.13610	0.12778	93.9	0.11701	0.11273	96.3
20	0.26008	0.26476	101.8	0.53301	0.48645	91.3	0.29486	0.28216	95.7
50	1.57539	1.53062	97.2	3.28974	3.06625	93.2	0.93673	0.88150	94.1
100	6.23544	6.06750	97.3	13.1042	12.1850	93.0	2.17502	2.03875	93.7
200	24.8100	25.1187	101.2	52.3083	49.3375	94.3	4.95249	4.70100	94.9
300	55.7240	55.8428	100.2	117.612	115.718	98.4	7.94661	7.75000	97.5
500	154.570	153.125	99.1	326.519	320.833	98.3	14.2718	13.3200	93.3
1000	617.623	630.750	102.1	1305.53	1585.50	121.4	31.5533	29.5937	93.8
2000	2469.18	3318.50	134.3	5221.06	8376.25	160.4	69.1252	68.7000	99.4

	set union			list reversal			reversal w/app.		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.10318	0.09875	95.7	0.00935	0.00960	102.7	0.05325	0.04996	93.8
20	0.38230	0.36242	94.8	0.01832	0.01740	95.0	0.19596	0.18077	92.2
50	2.27639	2.12062	93.2	0.04521	0.04375	96.8	1.16194	1.06250	91.4
100	8.95569	8.3650	93.4	0.09003	0.08531	94.8	4.56477	4.1840	91.7
200	35.5235	33.5167	94.4	0.17967	0.17131	95.3	18.0936	16.6416	92.0
300	79.7037	75.3800	94.6	0.26932	0.25625	95.1	40.5867	37.4921	92.4
500	220.901	208.355	94.3	0.44860	0.42530	94.8	112.465	108.325	96.3
1000	882.111	839.96	95.2	0.89682	0.86580	96.5	449.038	421.8	93.9
2000	3525.45	3385.93	96.0	1.79324	1.74350	97.2	1794.50	2473.5	137.8

Table 3. Calculated and measured worst-case times (in milliseconds) for the imperative language, using SPS.

	insertsort			list-mergesort			mergesort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.067637	0.057228	84.61026	0.064566	0.049438	76.56949	0.154043	0.123428	80.12580
20	0.261368	0.212860	81.44076	0.166667	0.128768	77.26105	0.371773	0.299041	80.43660
50	1.596142	1.305419	81.78590	0.548978	0.429809	78.29264	1.129826	0.927734	82.11295
100	6.332710	5.114257	80.75938	1.287664	1.018432	79.09148	2.556371	2.114746	82.72453
200	25.22562	20.76171	82.30407	2.955253	2.417968	81.81934	5.700747	4.778808	83.82775
300	56.67825	45.20312	79.75390	4.681983	3.841308	82.04446	9.054851	7.610351	84.04722
500	157.2626	124.9062	79.42524	8.385997	6.981445	83.25122	16.08930	13.43945	83.53036
1000	628.5185	500.6250	79.65158	18.67365	16.21875	86.85363	35.09691	29.51171	84.08636
2000	2513.008	1990.750	79.21778	41.15115	33.88281	82.33745	76.02501	64.23437	84.49110

	reverse!			selectsort			vector-sum		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.007180	0.006777	94.39671	0.057102	0.053535	93.75354	0.013775	0.009872	71.66783
20	0.013994	0.013376	95.58393	0.208539	0.186370	89.36968	0.025902	0.015682	60.54226
50	0.034436	0.032634	94.76739	1.224741	1.087646	88.80617	0.062285	0.045570	73.16320
100	0.068507	0.064651	94.37153	4.791386	4.218750	88.04863	0.122924	0.087127	70.87917
200	0.136648	0.126800	92.79296	18.94832	16.47460	86.94493	0.244200	0.170776	69.93274
300	0.204790	0.188507	92.04881	42.47012	37.35937	87.96623	0.365477	0.262512	71.82718
500	0.341073	0.304565	89.29618	117.6083	101.6562	86.43626	0.608030	0.432556	71.14050
1000	0.681780	0.573608	84.13386	469.3390	404.5625	86.19835	1.214413	0.736328	60.63238
2000	1.363195	1.081420	79.32985	1875.165	1613.750	86.05907	2.427180	1.754638	72.29124

TABLE 4. Calculated and measured worst-case times (in milliseconds) for the imperative language, using the direct approach.

	insertsort			list-mergesort			mergesort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.064447	0.057228	88.79754	0.060180	0.049438	82.14974	0.128231	0.123428	96.25400
20	0.249517	0.212860	85.30865	0.155863	0.128768	82.61660	0.310681	0.299041	96.25348
50	1.524904	1.305419	85.60666	0.514573	0.429809	83.52728	0.947944	0.927734	97.86796
100	6.051141	5.114257	84.51724	1.209050	1.018432	84.23405	2.149917	2.114746	98.36404
200	24.10583	20.76171	86.12736	2.778507	2.417968	87.02401	4.803626	4.778808	99.48335
300	54.16348	45.20312	83.45682	4.408732	3.841308	87.12954	7.637164	7.610351	99.64891
500	150.2876	124.9062	83.11144	7.907639	6.981445	88.28734	13.58397	13.43945	98.93608
1000	600.6499	500.6250	83.34721	17.61970	16.21875	92.04893	29.66984	29.51171	99.46705
2000	2401.596	1990.750	82.89278	38.84885	33.88281	87.21701	64.33920	64.23437	99.83706

	reverse!			selectsort			vector-sum		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.006808	0.006777	99.54794	0.053656	0.053535	99.77370	0.009919	0.009872	99.52056
20	0.013381	0.013376	99.96277	0.196160	0.186370	95.00957	0.018833	0.015682	83.26668
50	0.033099	0.032634	98.59662	1.152156	1.087646	94.40093	0.045574	0.045570	99.99003
100	0.065962	0.064651	98.01231	4.507106	4.218750	93.60217	0.090143	0.087127	96.65435
200	0.131689	0.126800	96.28760	17.82309	16.47460	92.43406	0.179280	0.170776	95.25632
300	0.197416	0.188507	95.48717	39.94719	37.35937	93.52190	0.268418	0.262512	97.79971
500	0.328869	0.304565	92.60977	110.6197	101.6562	91.89703	0.446692	0.432556	96.83527
1000	0.657503	0.573608	87.24037	441.4430	404.5625	91.64545	0.892379	0.736328	82.51291
2000	1.314770	1.081420	82.25166	1763.698	1613.750	91.49807	1.783752	1.754638	98.36784

TABLE 5. Calculated and measured worst-case times (in milliseconds) for the imperative language on program `cruft`, using the direct approach.

	cruft		
size	calculated	measured	me/ca
10	0.49460	0.5699919	86.8
20	0.50297	0.5800838	86.7
50	0.53088	0.6103595	87.0
100	0.57823	0.6608190	87.5
200	0.66760	0.7617380	87.6
300	0.75428	0.8626570	87.4
500	0.93486	1.0644950	87.8
1000	1.38272	1.5690900	88.1
2000	2.27100	2.5782800	88.1

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 2nd edition, July 1996.

[2] P. Altenbernd. On the false path problem in hard real-time programs. In *ECRTS '96: Proceedings of the Eighth EuroMicro Workshop on Real-Time Systems*, pages 102–107. IEEE Computer Society, Los Alamitos, CA, June 1996.

[3] P. Altenbernd, L. Burchard, and F. Stappert. Worst-case execution times analysis of MPEG-2 decoding. In *ECRTS '00: Proceedings of the Twelfth Euromicro Conference on Real-Time Systems*, pages 73–80. IEEE Computer Society, Los Alamitos, CA, 2000.

[4] R. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding worst-case instruction cache performance. In *RTSS '94: Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*. IEEE Computer Society, Los Alamitos, CA, 1994.

[5] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A calculus of bounded capacities. In *ASIAN '03: Proceedings of the Eighth Asian Computing Science Conference*, number 2896 in Lecture Notes in Computer Science, pages 205–223. Springer, Berlin, December 2003.

[6] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[7] G. Bernat, A. Colin, and S. M. Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Technical Report YCS-2003-353, Real-Time Systems Research Group, University of York, England, January 2003.

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

[9] L.-O. Burchard and P. Altenbernd. Estimating decoding times of MPEG-2 video streams. In *ICIP '00: Proceedings of the International Conference on Image Processing*, volume 3, pages 560–563. IEEE Computer Society, Los Alamitos, CA, September 2000.

[10] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310. ACM Press, New York, June 1990.

[11] W.-N. Chin and S.-C. Khoo. Calculating sized types. In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–72. ACM Press, New York, January 2000.

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

[13] M. L. Cooper. Cruft: a replacement for the 'crypt' utility. http://freshmeat.net/projects/-cruft/, January 2000. (Accessed January 10, 2006).

[14] M. Corti, R. Brega, and T. Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *LCTES '00: Proceedings of the 2000 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1985 of *Lecture Notes in Computer Science*, pages 178–198. Springer, London, 2001.

[15] M. Corti and T. Gross. Approximation of the worst-case execution time using structural analysis. In *EMSOFT '04: Proceedings of the Fourth ACM International Conference on Embedded Software*, pages 269–277. ACM Press, New York, September 2004.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, 1977.

[17] K. Crary and S. Weirich. Resource bound certification. In *POPL '00: Proceedings of the Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, January 2000.

[18] R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[19] M. Dipperstein. Arithmetic code discussion and implementation. http://michael.dipperstein.-com/arithmetic/index.html, November 2005. (Accessed January 10, 2006).

[20] M. Ditze, P. Altenbernd, and C. Loeser. Improving resource utilization for MPEG decoding in embedded end-devices. In *ACSC '04: Proceedings of the Twenty-Seventh Conference on*

Australasian Computer Science, pages 133–142. Australian Computer Society, Darlinghurst, Australia, 2004.

[21] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the Eighteenth ACM SIGACT Symposium on Theory of Computing*, pages 109–121. ACM Press, New York, May 1986.

[22] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, February 1989.

[23] D. Dussart and P. Thiemann. Partial evaluation for higher-order languages with state. Technical Report WSI–96–01, Berichte des Wilhelm-Schickard-Instituts, Universit at Tubingen, November 1996.

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

[25] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, Englewood Cliffs, N.J., 3rd edition, 2003.

[26] R. K. Dybvig, R. Hieb, and T. Butler. Destination-driven code generation. Technical Report 302, Computer Science Department, Indiana University, February 1992.

[27] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *RTAS '03: Proceedings of the Ninth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 152–159. IEEE Computer Society, Washington, May 2003.

[28] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *ECRTS '98: Proceedings of the Tenth EuroMicro Workshop on Real-Time Systems*. IEEE Computer Society, Berlin, June 1998.

[29] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *EuroPar '97: Proceedings of the Third International European Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 1298–1307. Springer, London, August 1997.

[30] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 51–62. ACM Press, New York, 2003.

[31] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *LCTES '97: Proceedings of the 1997 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46. ACM Press, New York, 1997.

[32] P. Flajolet, B. Salvy, and P. Zimmermann. *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, chapter Lambda-Upsilon-Omega: an assistant algorithms analyzer, pages 201–212. Springer, London, July 1989.

[33] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theor. Comput. Sci.*, 79(1):37–109, February 1991.

[34] Y. Futamura and K. Nogi. *Partial Evaluation and Mixed Computation*, chapter Generalized partial evaluation, pages 133–151. Elsevier Science, New York, 1988.

[35] R. P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press Series in Computer Systems. MIT Press, Cambridge, 1985.

[36] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behavior*. PhD thesis, Department of Electrical Engineering. Princeton University, Princeton, NJ, November 1999.

[37] G. Gomez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume 37 of *SIGPLAN Notices*, pages 75–86. ACM Press, New York, March 2002.

[38] M. J. C. Gordon. *The Denotational Description of Programming Languages: an Introduction*. Springer, New York, 1979.

[39] B. Grobauer. Cost recurrences for DML programs. In *ICFP '01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 253–264. ACM Press, New York, September 2001.

[40] J. Gustafsson. *Analysing execution time of object oriented programs using abstract interpretation*. PhD thesis, Uppsala University, Uppsala, Sweden, May 2000.

[41] J. Gustafsson. Eliminating annotations by automatic flow analysis of real-time programs. In *RTCSA '00: Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society, Washington, December 2000.

[42] J. Gustafsson. Worst case execution time analysis of object-oriented programs. In *WORDS '02: Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 71–76. IEEE Computer Society, Washington, January 2002.

[43] J. Gustafsson, N. Bermudo, and L. Sjöberg. Flow analysis for WCET calculation. Technical report, Department of Computer Engineering, Mälardalen University, Västerâs, Sweden, April 2002.

[44] 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.

[45] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. Tool for automatic flow analysis of C-programs for WCET calculation. In *WORDS '03: Proceedings of the Eighth IEEE International Workshop on Object-Oriented Real-time Dependable Systems*, pages 106–112. IEEE Computer Society, Los Alamitos, CA, January 2003.

[46] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *RTSS '92: Proceedings of the Eleventh IEEE Real-Time Systems Symposium*, pages 68–77. IEEE Computer Society, Los Alamitos, CA, December 1992.

[47] C. A. Healy, M. Sjödin, V. Rustagi, and D. B. Whalley. Bounding loop iterations for timing analysis. In *RTAS '98: Proceedings of the IEEE Real-Time Applications Symposium*, page 12. IEEE Computer Society, Washington, June 1998.

[48] C. A. Healy, M. Sjödin, V. Rustagi, D. B. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *The Journal of Real-Time Systems*, 18(2/3):129–156, 2000.

[49] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *PLDI '92: Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–260. ACM Press, New York, June 1992.

[50] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197. ACM Press, New York, January 2003.

[51] E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Deriving java machine timing models for portable worst-case execution time analysis. In *JTRES '03: Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, volume 2889 of *Lecture Notes in Computer Science*, pages 411–424. Springer, London, 2003.

[52] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In *ICFP '99: Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 70–81. ACM Press, New York, September 1999.

[53] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the Twenty-Third ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423. ACM Press, New York, January 1996.

[54] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

[55] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *POPL '88: Proceedings of the Fifteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 158–168. ACM Press, New York, January 1988.

[56] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *FPLCA '85: Proceedings of the 1985 ACM SIGPLAN-SIGARCH Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203. Springer, New York, September 1985.

[57] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Upper Saddle River, NJ,, June 1993.

[58] R. Kelsey, W. Clinger, and J. Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, February 1998.

[59] R. Kirner and P. Puschner. Timing analysis of optimised code. In *WORDS '03: Proceedings of the Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 100–105. IEEE Computer Society, Los Alamitos, CA, January 2003.

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

[61] D. Le Métayer. ACE: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, April 1988.

[62] 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.

[63] M. Lindgren, H. Hansson, and H. Thane. Using measurements to derive the worst-case execution time. In *RTCSA '00: Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications*, pages 15–22. IEEE Computer Society, Washington, December 2000.

[64] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In J. Gustafsson, editor, *WCET '03: Proceedings of the Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80. Department of Computer Engineering, Mälardalen University, Västerâs, Sweden, July 2003.

[65] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In F. Mueller and A. Bestavros, editors, *LCTES '98: Proceedings of the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer, London, June 1998.

[66] Y. A. Liu and G. Gomez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, December 2001.

[67] 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.

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

[69] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *DATE '03: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 10196–10203. IEEE Computer Society, Washington, March 2003.

[70] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. Technical Report 98-3, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1998.

[71] J. Marshall. Evil: a C++ to Scheme Compiler. http://home.comcast.net/˜ prunesquallor/-Evil_Part_II.zip, January 2006. (Accessed January 20, 2006).

[72] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990.

[73] S. Mohan, F. Mueller, D. B. Whalley, and C. A. Healy. Timing analysis for sensor network nodes of the Atmega processor family. In *RTAS '05: Proceedings of the Eleventh IEEE Real Time and Embedded Technology and Applications Symposium*, pages 405–414. IEEE Computer Society, Washington, March 2005.

[74] J. G. Morrisett. Refining first-class stores. In *SIPL '93: Proceedings of the 1993 ACM SIGPLAN Workshop on State in Programming Languages*, pages 73–87. ACM Press, New York, June 1993.

[75] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5(1):31–62, March 1993.

[76] 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.

[77] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *RTCSA '99: Proceedings of the Sixth IEEE International Conference on Real-Time Computing Systems and Applications*, pages 442–449. IEEE Computer Society, Washington, December 1999.

[78] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York, 1987.

[79] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05: Proceedings of the Eleventh IEEE Real Time and Embedded Technology and Applications Symposium*, pages 148–157. IEEE Computer Society, Washington, March 2005.

[80] Á. J. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. B. Vasconcelos. Cost analysis using automatic size and time inference. In *IFL '02: Proceedings of the Fourteenth International Workshop on the Implementation of Functional Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 232–247. Springer, Berlin, 2002.

[81] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 65–78. ACM Press, New York, June 1994.

[82] S. Rele, V. Jain, S. Pande, and J. Ramanujam. Compact and efficient code generation through program restructuring on limited memory embedded DSPs. *IEEE Transactions on Computer-Aided Design of Computer Circuits and Systems*, 20(4):477–494, April 2001.

[83] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer, New York, 1988.

[84] M. Rosendahl. Automatic complexity analysis. In *FPLCA '89: Proceedings of the Fourth ACM SIGPLAN-SIGARCH International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM Press, New York, September 1989.

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

[86] 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, December 1989. Special issue on Performance Evaluation.

[87] D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, London, September 1990.

[88] D. Sands. Complexity analysis for a lazy higher-order language. In *ESOP '90: Proceedings of the Third European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer, London, May 1990.

[89] W. L. Scherlis. Program improvement by internal specialization. In *POPL '81: Proceedings of the Eighth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 41–49. ACM Press, New York, January 1981.

[90] H. W. Schmidt and W. Zimmermann. A complexity calculus for object-oriented programs. *Journal of Object-Oriented Systems*, 1(2):117–147, 1994.

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

[92] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 132–140. ACM Press, New York, 2001.

[93] T. Tammet. Lambda-lifting as an optimization for compiling Scheme to C. Available as ftp://www.cs.chalmers.edu/pub/users/tammet/www/hobbit.ps, 1996.

[94] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, July 1986.

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

[96] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *LCTES '01: Proceedings of the 2001 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 102–111. ACM Press, New York, June 2001.

[97] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In L. Zuck et al., editors, *VMCAI '03: Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 70–85. Springer, London, 2003.

[98] E. Vivancos, C. A. Healy, F. Mueller, and D. B. Whalley. Parametric timing analysis. In *LCTES '01: Proceedings of the 2001 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 88–93. ACM Press, New York, June 2001.

[99] P. Wadler. Strictness analysis aids time analysis. In *POPL '88: Proceedings of the Fifteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, New York, January 1988.

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

[101] B. Wegbreit. Goal-directed program transformation. *IEEE Trans. Softw. Eng.*, SE-2(2):69–80, June 1976.

[102] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the Twenty-First ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, pages 297–310. ACM Press, New York, January 1994.

[103] W. Zhao, W. Kreahling, D. B. Whalley, C. A. Healy, and F. Mueller. Improving WCET by optimizing worst-case paths. In *RTAS '05: Proceedings of the Eleventh IEEE Real Time and Embedded Technology and Applications Symposium*, pages 138–147. IEEE Computer Society, Washington, March 2005.

[104] P. Zimmermann and W. Zimmermann. Automatic complexity analysis of divide-and-conquer algorithms. In *ISCIS '91: Proceedings of the Sixth International Symposium on Computer and Information Sciences*, pages 395–404. Elsevier Science, New York, 1991.

Curriculum Vitae

Gustavo Gómez received a Bachelor of Science Degree in Computer Science from the Instituto Tecnológico y de Estudios Superiores de Monterrey, Mexico, in 1991. From 1990 through 1994 he was a Research Assistant at the Instituto Tecnológico y de Estudios Superiores de Monterrey, and he received a Master of Science from the Instituto Tecnológico y de Estudios Superiores de Monterrey in 1994. From 1995 through 1997 he was an Assistant Instructor in the Computer Science Department at Indiana University. From 1998 through 2001 he was a Research Assistant in the Computer Science Department at Indiana University. From 2001 through 2003 he was an Assistant Instructor in the Computer Science Department at Indiana University.