Levin, Blum and the Time Optimality of Programs

Niels H. Christensen

April 26, 1999

Preface

This report constitutes a Master's Thesis forming part of the credit towards the Master's Degree (cand. scient.) in Computer Science at DIKU (Dept. of Computer Science, University of Copenhagen). It reports work done at DIKU between the summer of 1998 and April 1999.

The report has two parts, the first one of which is on the topic of complexity theory. This part requires the reader to have some insight into the results and methods of this field, for example as presented in [6]. The second part is on the topic of programming languages and requires the reader to be literate in functional languages and operational semantics.

The first part is purely theoretical, while the second part mixes some theory with practical experiments.

Acknowledgements I would like to thank Neil D. Jones for supervising my studies through several years, Stephen Cook and the CS Dept. at the University of Toronto for hosting me during my earlier work on these topics, Amir Ben-Amram for inspiring discussions on Levin's Theorem, Stefan Kahrs for explaining some details about his work to me, Kristian de Lichtenberg for proof-reading a large part of this thesis, and my darling Lísbita for her support and, well, everything.

Contents

1 General Introduction **4**

I The Levin and Blum Theorems **5**

2 Introduction **6**
- 2.1 Notation . . . 6
- 2.2 The Existence of Optimal Programs . . . 7
- 2.3 The Speedup Theorems . . . 8
- 2.4 Levin's Theorem . . . 8
- 2.5 Undecidability of Optimality . . . 10
- 2.6 Undecidable Properties in Restricted Program Classes . . . 11
- 2.7 Program Optimizers . . . 13
- 2.8 Natural Problems . . . 14
- 2.9 Overview of this Part . . . 14

3 Variants of Optimality **15**
- 3.1 The Classical Theorem . . . 15
- 3.2 Limiting the Enumerated Programs . . . 18
- 3.3 A Levin Theorem for Program Transformation . . . 19

4 Two Speedup Theorems **21**
- 4.1 Speedups and Budgets . . . 21
- 4.2 Time Analysis . . . 22
  - 4.2.1 The Algorithm . . . 22
  - 4.2.2 Running time . . . 22
  - 4.2.3 The Theorem . . . 25
- 4.3 A Low-Complexity Result . . . 25
- 4.4 The Proof . . . 27
  - 4.4.1 Definitions . . . 27
  - 4.4.2 Lower Bound . . . 27
  - 4.4.3 A First Upper Bound . . . 28
  - 4.4.4 A Sequence of Upper Bounds . . . 30

5 Conclusion and Future Work **34**

II Memoizing Both Cons And User-Defined Functions **35**

6 Introduction **36**
- 6.1 Motivation . . . 36
- 6.2 Overview of this Part . . . 36

7 Uniqueness as a Leitmotiv for Implementation **37**
7.1 The F Language 37
7.2 On Heaps and Memoization 37
7.2.1 Cons Memoization 40
7.2.2 Call Memoization 40
7.2.3 Combined Memoization 40
7.3 Defining the Semantics of F 41
7.4 Measures 41
7.5 Interpretations of the Semantics 43
7.5.1 Standard Semantics 43
7.5.2 DAG Semantics 43
7.5.3 Memoizing Cons Semantics for F 44
7.5.4 Memoizing Call Semantics for F 44
7.5.5 Combined Memoizing Semantics 44
7.6 Some Background 45

8 Unlimp Experiments **46**
8.1 Implementation 46
8.1.1 Overall Implementation of Semantics 46
8.1.2 DAG Heaps and Call Memoization 47
8.1.3 Memoization for Uniqueness 47
8.1.4 Test Programs 47
8.2 Comparing Different Search Structures 48
8.3 Cons Memoization 48
8.4 Call Memoization 49
8.5 Unlimp for Enumerations 49

9 Conclusion and Future Work **50**

A Test Programs **51**

B Unlimp Experiments: Tries vs. Double Hashing **60**

C Unlimp Experiments: Cons Memoization **62**

D Unlimp Experiments: Call Memoization **67**

E Unlimp Experiments: Enumeration **71**

Chapter 1

General Introduction

The general topic of this thesis is optimality of running times of programs. In the first, and longest, part we focus directly on these issues. We provide a general discussion of the kind of computational problems for which some program solving a problem can be said to have optimal running time among all programs solving the problem. Classical results by Leonid A. Levin and Manuel Blum demonstrate that for some computational problems such a program does exist, and for others such a program does not exist. We present the two results and prove some new variations of these. We also consider the possibility of deciding whether a problem is solved by an optimal program and of obtaining optimal programs by use of automatic optimizers.

The second part is a spin-off of our work with Levin's Optimal Search Theorem. In [4] we performed some experiments with an implementation of the algorithm associated with this theorem. A serious problem was the amount of space used by a particular part of the algorithm which was supposed to enumerate programs in some language efficiently. This problem made us look into ways of improving the space-usage of such algorithms. One particularly interesting construction was Stefan Kahr's so-called "Unlimp"-interpreters. An "Unlimp" interpreter manages heap usage in a clever way that minimizes the number of cells allocated by the constructors of the interpreted language. Moreover, it allows very easy implementation of function memoization. We found that this idea was so interesting in itself that we dedicated the second part of this thesis to describing the idea more carefully and investigating an implementation of the scheme by a number of experiments.

Part I

The Levin and Blum Theorems

Chapter 2

Introduction

In this chapter we sum up some results regarding time optimality of programs.

2.1 Notation

We will begin by defining some notation that will be used throughout this thesis.

We base our presentation on definitions from [6]. Unless otherwise specified, programs are of the I language of this book. This is a very simple, imperative language in which the set, $\mathbb{D}$, of values is a restriction of that of LISP: The set $\mathbb{D}$ is the smallest set containing

- one atomic value `nil`, and
- the pairing $(d_1.d_2)$ of every two values $d_1, d_2 \in \mathbb{D}$.

Following standard conventions, we say that $[\![p]\!](d) = \texttt{true}$ if and only if $[\![p]\!](d) \in \mathbb{D} \setminus \texttt{nil}$. Note that this implies that the computation of p on input d terminates; otherwise we say that $[\![p]\!](d) = \bot \notin \mathbb{D}$. For a program p, we denote the set $\{d \in \mathbb{D} \mid [\![p]\!](d) = \texttt{true}\}$ by *accept*(p). Some notation regarding computable, binary predicates will be introduced in Section 2.2.

Almost Everywhere If P is a predicate defined on a set S, then by $\overset{\infty}{\forall} x \in S : P(x)$ we mean that the set $\{x \in S \mid \neg P(x)\}$ is finite. We also say that P holds *almost everywhere* in S.

Infinitely Often If P is a predicate defined on a set S, then by $\overset{\infty}{\exists} x \in S : P(x)$ we mean that the set $\{x \in S \mid P(x)\}$ is infinite. We also say that P holds *infinitely often* in S.

Time Constructible As in [6], we say that a computable function $f : \mathbb{N} \to \mathbb{N}$ is *time constructible* if and only if there is a program p which computes $f(n)$ (using some reasonable representation of numbers) in time $\mathcal{O}(f(n))$. Schnorr, in [10], terms such functions "linearly honest".

Constant Slowdown Timed Interpreter An *interpreter* is a program `int` such that given a program p and a $d \in \mathbb{D}$ we have $[\![\texttt{int}]\!](p.d) = [\![p]\!](d)$. A *timed interpreter* is a program `tint` takes as input a program p, a value $d \in \mathbb{D}$ and a time bound $t \in \mathbb{N}$ (in some reasonable representation), and for which $[\![\texttt{tint}]\!]((p.d).t) = [\![p]\!](d)$ if $\text{time}_p(d) \leq t$. If $\text{time}_p(d) > t$ then $[\![\texttt{tint}]\!]((p.d).t) = \texttt{nil}$. Finally, a *constant slowdown timed interpreter* is a timed interpreter `tint`, for which $\text{time}_{\texttt{tint}}((p.d).t) = \mathcal{O}(t)$.

Constant-Time Enumeration of Sets An enumeration of a set S is a function which returns an new element of S each time it is called and which will eventually produce any element of the set. Formally, we may define *an enumeration of* $S \subseteq \mathbb{D}$ to be a program `enum` such that $[\![\texttt{enum}]\!](\texttt{nil}) = (d_1.s_1)$, $[\![\texttt{enum}]\!](d_1.s_1) = (d_2.s_2)$, more generally $[\![\texttt{enum}]\!]^i(\texttt{nil}) = (d_i.s_i)$, and the set S is $S = \{p_i \mid i \in \mathbb{N}_+\}$. We call such an enumeration *constant-time* if and only if there is a constant $c \in \mathbb{N}$ such that for all $i \in \mathbb{N}_+$: $\text{time}_{\texttt{enum}}(p_i.s_i) \leq c$. The existence of constant-time enumerations of the set of all programs is established in [6].

Tables In some constructions we will use tables for which we adopt the following notation. For a table T, $T[]$ denotes the whole table, $T[i]$ denotes the ith entry of the table. Notation $T[i :]$ means a table with $i - 1$ entries fewer than T, and for which $T[i :][j] = T[i + j]$ for all j such that this is defined. Two-dimensional arrays are indexed $T[i, j]$.

2.2 The Existence of Optimal Programs

In the following we would like to consider a very essential question of the theory of programs: *Which computable problems have optimal programs ?* That is, which of the formal tasks of computation (that may be solved by a computer) can be solved by a program *that is essentially no slower* than any other program solving the task. There are two points in the above formulation that should be formalized: What is a "formal task of computation" and what does it mean for a program to be "essentially no slower" than another ?

The classical definitions of computational tasks are semideciding (or enumerating) a set and deciding a set. Program p semidecides the set $S \subseteq \mathbb{D}$ if and only if $[\![p]\!](d) = \texttt{true} \Leftrightarrow d \in S$. The program decides S if the above holds and $[\![p]\!]$ is total. In this chapter, we will focus on decision as well as two other kinds of tasks: *checking* and *searching* a relation.

Let $\rho \subseteq \mathbb{D} \times \mathbb{D}$ be a binary relation (sometimes called a binary predicate) on the set of values. The *set of witnesses* for $x \in \mathbb{D}$ is $\rho(x) = \{y \mid \rho(x, y)\}$ (abusing notation a bit). The *set defined by* ρ is $S_\rho = \{x \in \mathbb{D} \mid \rho(x) \neq \emptyset\}$.

A program p is said to be a checker for ρ if and only if $[\![p]\!]$ is total and $[\![p]\!](d_1.d_2) = \texttt{true} \Leftrightarrow \rho(d_1, d_2)$ for all $d_1, d_2 \in \mathbb{D}$. We also say that p solves the *check problem* for ρ. A program w solves the *search problem* for ρ if and only if $x \in S_\rho \Rightarrow [\![w]\!](d) \in \rho(x)$. If p is a checker for ρ and w solves the search problem for ρ we say that w is a *witness program* for p.

As for "essentially no slower", we will focus on the classical big-O definition: Program p is no slower than program p' if and only if $\text{time}_p(d) = \mathcal{O}(\text{time}_{p'}(d))$. If p solves some problem and the above criterion holds for all programs p' solving this problem, we say that p is an optimal program for the problem. We also say that the problem has the "optimality property".

Returning to our question, we may wonder if it has a trivial answer – no problem has an optimal program, or perhaps all problems have optimal programs. The former suggestion is rather easily ruled out; many well-known problems are solved by programs that have running time linear in the input size, and for many of these problems this is clearly optimal. But could it not be that *all* problems have optimal programs ? A problem which clearly does not have an optimal program does not spring to mind a priori.

2.3 The Speedup Theorems

The answer is no. This is the rather startling consequence of Blum's Speedup Theorem. For some (decision) problems, *any* program that solves the problem is *essentially slower* than some other program that solves the problem. We cannot even bound the speedup proved by the theorem: there are problems for which any program that solves the problem is *much* slower than some other program solving the problem, for any computable definition of "much". And unlike the well-known Gap Theorem ([11],[3]), the result does not rely upon rather unnatural time-functions which are not time constructible. Furthermore, a new speedup theorem proved in this thesis also demonstrates that such problems also exist among the "feasible" ones – problems that are computable in polynomial time. Thus, it is not a feature of complexity classes of "purely theoretical interest".

Let us state these two theorems, which shall be proved in Chapter 4. The first one is classical:

Theorem 1 (Blum's Speedup Theorem). *For any time constructible, total recursive function h, there is a computable function* $\Phi : \mathbb{N} \to \{0,1\}$ *such that*

$$\forall \mathtt{p} : [\![\mathtt{p}]\!] = \Phi \text{ implies } \exists \mathtt{p}' : [\![\mathtt{p}']\!] = \Phi \wedge \overset{\infty}{\forall} \mathtt{n} : \mathit{time}_{\mathtt{p}}(\mathtt{n}) \geq \mathtt{h}(\mathit{time}_{\mathtt{p}'}(\mathtt{n}))$$

Stating the new speedup theorem in its full generality would be a bit too technical at this point. Instead we simply state two special cases:

Theorem 2. *There is a computable function* $\Phi : \mathbb{D} \to \mathbb{D}$ *such that* $\Phi(x)$ *may be computed in time* $\mathcal{O}(2^{|x|})$ *but*

$$\begin{gathered}\forall p : [\![p]\!] = \Phi \text{ implies} \\ \exists c \in \mathbb{N} \exists p' : [\![p']\!] = \Phi \ \wedge \overset{\infty}{\forall} x \in \mathbb{D} : \mathit{time}_{p'}(x) < c \times \frac{\mathit{time}_p(x)}{|x|}\end{gathered}$$

Theorem 3. *There is a computable function* $\Phi : \mathbb{D} \to \mathbb{D}$ *such that* $\Phi(x)$ *may be computed in time* $\mathcal{O}(|x|^2)$ *but*

$$\begin{gathered}\forall p : [\![p]\!] = \Phi \text{ implies} \\ \exists c \in \mathbb{N} \exists p' : [\![p']\!] = \Phi \ \wedge \overset{\infty}{\forall} x \in \mathbb{D} : \mathit{time}_{p'}(x) < c \times \frac{\mathit{time}_p(x)}{\log |x|}\end{gathered}$$

2.4 Levin's Theorem

Having ruled out a trivial answer to our question, we may now try to identify the class of problems which have the "optimality property" that some solving program is optimal. We know that many concrete problems from algorithmics have optimal programs. For a given problem, this can often be established in two steps: First one proves that an explicitly given algorithm solves the problem. Then one proves that the running time of this algorithm is actually a lower bound of any algorithm solving the problem.

But these proofs are often very problem-specific and so do not identify larger classes of problems with the optimality property. In contrast, Levin's Theorem, which we shall prove in Chapter 3, asserts that *a wide range* of search problems have optimal programs. Assume p is a checker for a predicate ρ.

Theorem 4 (Levin's Optimal Search Theorem). *Let p be a program. There is a witness program,* `opt`*, for p such that for any witness program w for p,*

$$\mathit{time}_{\mathtt{opt}}(d) = \mathcal{O}(\mathit{time}_w(d) + \mathit{time}_p(d.[\![w]\!](d)))$$

where $d \in \mathbb{D}$.

Levin's Theorem has the below corollary.

Definition 1. We say that a program p is *swift* if and only if for every witness program w for p there is a constant $k_w \in \mathbb{N}$ such that for all $d \in \mathbb{D}$:

$$\text{time}_p(d) \leq k_w \times \text{time}_w(d)$$

Corollary 1. *If p is swift then there is an optimal witness program for p.*

So now we have a simple and general criterion which is sufficient for a search problem to have an optimal program. Very interestingly we may observe that if $P \neq NP$ then all problems in $NP \setminus P$ have swift checkers. Recall that predicate $SAT(x, y)$ is true if and only if x is a satisfiable boolean expression and y is a satisfying assignment of the variables in x, and that S_{SAT} is *NP*-complete. So if $P \neq NP$ then there is a time-optimal program for e.g. computing a satisfying assignment for a satisfiable boolean expression.

Can we prove something along the lines of Levin's Theorem for decision problems ? One approach to this question is to try to establish close connections between the time bounds for the search problem and the decision problem of a predicate.

Let us demonstrate part of this idea in the case of SAT. It is well-known that the search problem for SAT can be solved almost as fast as the corresponding decision problem. The idea is this: Given a decision algorithm, p_{SAT} for SAT, we may solve the search problem by the following algorithm:

1. Given a boolean expression E.
2. If E is not satisfiable then stop with output `nil`. Else set A to the empty assignment (of 0 variables) and continue with step 3.
3. If E has no variables then stop with output A. Else continue with step 4.
4. Pick some variable x of E, and let $E_{\texttt{true}}$ be E with all instances of x substituted by the value `true`. Accordingly for $E_{\texttt{false}}$. If $E_{\texttt{true}}$ is satisfiable the extend A with the assignment $x :=$ `true`, set $E := E_{\texttt{true}}$ and go to step 3. Else extend A with the assignment $x :=$ `false`, set $E := E_{\texttt{false}}$ and go to step 3.

It is not hard to see that this algorithm solves the search problem for SAT in time $\mathcal{O}(|E| \times \text{time}_{p_{SAT}}(E))$.

Schnorr, in [10], formalizes the above idea and applies it to other problems. In the rest of this section we shall review some of his results. Schnorr's notation is very different from ours, so to avoid complicated technicalities we will restrict our definitions and theorems to important special cases of his results. We shall still need some extra notation, though.

In [10], values are bit strings and we shall need a similar concept here. As is standard, we will encode boolean values in $\mathbb{D}$ by letting `nil` represent `false` and letting `(nil.nil)` represent `true`. We say that $d \in \mathbb{D}$ is a list of booleans if d is empty (i.e. `nil`) or a pair $(b.d')$ where b is a boolean value and (recursively) d' is a list of booleans.

Definition 2. Let ρ be a decidable predicate, and assume that $\rho(x, y)$ implies that y is a list of booleans and $|y| = \mathcal{O}(|x|)$. We call ρ *self-reducible* if and only if there is a polynomial-time computable function $\mathit{fix} : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ such that

- $\forall x, y_0, ys \in \mathbb{D} : \rho(x, (y_0.ys)) \Rightarrow \rho(\mathit{fix}(x, y_0), ys)$
- $\forall x, y \in \mathbb{D} : |\mathit{fix}(x, y)| \leq |x|$

Clearly, SAT is a self-reducible predicate. Schnorr proves the following result (Theorem 2.4 in [10]):

Theorem 5. *Let ρ be self-reducible. There are constants $c, r \in \mathbb{N}$ such that if program p decides S_ρ there is a program w solving the search problem for ρ such that*

$$time_w(x) \leq c \times |x|^r \times time_p(x)$$

He then makes a claim (Theorem 2.7 in [10]) which implies the following:

> Let Φ be as in Theorem 2, and let $S = \{d \mid \Phi(d) = \texttt{true}\}$. If ρ is a predicate such that $S_\rho = S$ then ρ cannot be self-reducible.

This is not true, and we shall give a simple counter-example. Let Φ and S be as defined in the claim. The set S is (trivially) defined by the predicate

$$\rho(x, y) \Rightarrow x \in S \wedge \texttt{y} = \texttt{nil}$$

which is trvially self-reducible (let $\mathit{fix}(x, y) = x$). This predicate has a curious property: Checking, searching and deciding it are equally hard, and none of these problems have optimal programs.

The reader should be aware that this is the simplest but not the only way to construct a counter-example to the claim. The flaw in the claim is not the definition of self-reducibility but that we must also require ρ to have a swift checker. This is also reflected in the proof, in which the only flaw is in the first equation where Levin's Theorem is used. In this equation, the time to run the checker has been left out. This is correct if the checker is swift, and the whole proof holds under this assumption. So we *may* conclude

Theorem 6. *Let Φ be as in Theorem 2, and let $S = \{d \mid \Phi(d) = \texttt{true}\}$. If ρ is a predicate such that $S_\rho = S$ then ρ cannot both be self-reducible and have a swift checker.*

Note that this result does not ensure us that for any self-reducible predicate which has a swift checker there is an optimal program solving its decision problem (it could have a logarithmic speedup as in Theorem 3). Nonetheless, we *do* know that for such decision problems *some* program solving it cannot be sped up by a factor linear in the size of the input.

2.5 Undecidability of Optimality

We have now identified some classes of problems which have (respectively don't have) the "optimality property" that some solving program is optimal. It seems natural to wonder how hard it is to tell whether a give problem has the propoerty – in particular, whether the property is decidable.

In this discussion, we will expect any reasonable "notation" for decision problems to be directly translatable into a program semideciding the described set. Thus, the "optimality property" can be thought of as a property of programs: Program p has the property if and only if $[\![p]\!]$ is total and there is an optimal program for deciding the set $\{d \in \mathbb{D} \mid [\![p]\!](d) = \texttt{true}\}$.

This property is certainly a non-trivial, extensional property of programs, so by Rice's Theorem (see [6]), it is undecidable.

But could it be that the undecidability of this property is mainly caused by fact that it requires program p to be total ? What if we restricted our attention to a class of simpler programs that all computed total functions, e.g. all programs running in polynomial time ?

It turns out that in many such cases, the optimality property is *still* undecidable. This will follow from the results below.

2.6 Undecidable Properties in Restricted Program Classes

In this section, we shall establish a result regarding the undecidability of certain problems connected to restricted classes of programs. An example is the following theorem, which is a consequence of Theorem 8 below:

Theorem 7. *Consider the hierarchy* $C_1 \subseteq \cdots \subseteq C_6$ *of classes*

$$\text{Regular} \subseteq \text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE}$$

For any proper inclusion $C_i \subset C_j$ *among these, the class* $C_i \setminus C_j$ *is undecidable.*

Definition 3. A set $C \subseteq \mathcal{P}(\mathbb{D})$ is called an *enumerable class of sets* if and only if there is a program e_C such that $[\![e_C]\!]$ is total and $C = \{\mathit{accept}(p) \mid p \in [\![e_C]\!](\mathbb{D})\}$. If the enumerated class is clear from the context we will denote $[\![e_C]\!](d)$ by p_d.

A subset $C' \subseteq C$ is called decidable if and only if the set $\{d \in \mathbb{D} \mid \mathit{accept}(p_d) \in C'\}$ is decidable.

Definition 4. Let $S, S' \subseteq \mathcal{P}(\mathbb{D})$. The *concatenation* of S and S' is defined by

$$S\#S' = \{(x.y) \mid x \in S \wedge y \in S'\}$$

A class C as above is said to be *computably closed under concatenation* if and only if $S, S' \in C \Rightarrow S\#S' \in C$ and there is a concatenation program, *concat*, such that $\forall d, d' \in \mathbb{D} : \mathit{accept}(p_{[\![concat]\!](d.d')}) = \mathit{accept}(p_d)\#\mathit{accept}(p_{d'})$.

Theorem 8. *Let* C *be an enumerable class of sets such that* $\emptyset \in C$ *and let* C *be computably closed under concatenation.*

If the class $\{\emptyset\}$ *is undecidable then for all non-trivial* $C' \subset C$ *such that* $\emptyset \in C'$ *and* $S\#S' \in C' \Rightarrow S = \emptyset \vee S' \in C'$*, the class* C' *is undecidable.*

The latter condition on C' may look peculiar at a first glance but is actually quite natural. For if $S\#S' \in C'$ and $x \in S$ it would be natural to expect that also $\{x\}\#S' \in C'$ and then $S' \in C'$. This line of reasoning will be valid in all our applications of this theorem.

Proof. (Of Theorem 8). We prove the theorem by reducing the problem of deciding $\emptyset$ to the problem of deciding C'.

Since C' is a non-trivial subset of C we may choose $S_{outside} \in C \setminus C'$. Let $d_{outside} \in \mathbb{D}$ be such that $\mathit{accept}(p_{d_{outside}}) = S_{outside}$. Now the computable function $R : \mathbb{D} \to \mathbb{D}$ given by

$$\lambda d.[\![concat]\!](d.d_{outside})$$

is a reduction as proposed.

For if $\mathit{accept}(p_d) = \emptyset$ for a given $d \in \mathbb{D}$, then we must have $\mathit{accept}(p_{R(d)}) = \emptyset \in C'$. And if $\mathit{accept}(p_d) \neq \emptyset$ then $\mathit{accept}(p_{R(d)}) = \mathit{accept}(p_d)\#S_{outside} \notin C'$ by the choice of $S_{outside}$ and the condition on C'. □

This theorem is certainly related to Rice's Theorem. The latter does not require anything of C' (except that it should be a non-trivial subset of C). Neither does it (specifically) require the class $\{\emptyset\}$ to be undecidable. Instead it requires C to contain all semidecidable sets.

Theorem 8 establishes a reduction from the problem of deciding $\{\emptyset\}$ to the problem of deciding C' for any C' as described. Note that $\{\emptyset\}$ satisfies the requirements for C'. Thus, in a sense, $\{\emptyset\}$ is the *opposite* of being complete among the classes of sets C' as described.

We may in many cases strengthen the theorem. Since the reduction function is computed simply by applying a specialized version of the concatenation program to the input it is reasonable to suspect that it can be computed using very few resources. More precisely:

Theorem 9. *Let C be an enumerable class of sets such that $\emptyset \in C$ and let C be computably closed under concatenation.*

If $[\![concat]\!]$ is computable in LOGSPACE then there is a LOGSPACE-reduction from the problem of deciding $\{\emptyset\}$ to the problem of deciding C' for any non-trivial $C' \subset C$ such that $\emptyset \in C'$ and $S\#S' \in C' \Rightarrow S = \emptyset \vee S' \in C'$.

Theorem 10. *Let C be an enumerable class of sets such that $\emptyset \in C$ and let C be computably closed under concatenation.*

If $[\![concat]\!]$ is computable in linear time then there is a linear time reduction from the problem of deciding $\{\emptyset\}$ to the problem of deciding C' for any non-trivial $C' \subset C$ such that $\emptyset \in C'$ and $S\#S' \in C' \Rightarrow S = \emptyset \vee S' \in C'$.

Our first application of Theorem 8 is the one we gave in the beginning of the section:

Corollary 2. *Consider the hierarchy $C_1 \subseteq \cdots \subseteq C_6$ of classes*

$$\text{Regular} \subseteq \text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE}$$

For any proper inclusion $C_i \subset C_j$ among these, the class $C_i \setminus C_j$ is undecidable.

We do not know if all the inclusions in the above theorem are proper. What we *do* know is that the class of regular languages is properly contained in LOGSPACE, and that NLOGSPACE is properly contained in PSPACE. For a discussion of the open questions regarding this hierarchy see e.g. [6].

There are other interesting applications of the theorem the above. The Time Hierarchy Theorem(see [6]) provides us with a wealth of classes that are properly contained in each other.

Corollary 3. *Let $f(n)$ and $g(n)$ be time-constructible, $f(n) \geq n$, $g(n) \geq n$. If $\lim_{n\to\infty} g(n)/f(n) = 0$ then the class $\text{TIME}^I(\mathcal{O}(f(n))) \setminus \text{TIME}^I(\mathcal{O}(g(n)))$ is undecidable.*

Proof. The Time Hierarchy Theorem asserts that for f and g as described the class $\text{TIME}^I(\mathcal{O}(f(n)))\setminus\text{TIME}^I(\mathcal{O}(g(n)))$ is not empty. The corollary follows immediately from Theorem 8 and the two lemmata below. □

Lemma 1. *Let $f(n)$ be time-constructible, $f(n) \geq n$. The class $\text{TIME}^I(\mathcal{O}(f(n)))$ is semidecidable.*

Proof. Given integer i, map this to a pair of integers (i_1, i_2) using some enumeration of $\mathbb{N}^2$. Now return the constant slowdown timed interpreter specialized to the i_1th program with time bound $i_2 f(n)$.

For any i, the constructed program will clearly run in time $\mathcal{O}(f(n))$. Furthermore, if $p \in \text{TIME}^I(\mathcal{O}(f(n)))$ then there is a $c \in \mathbb{N}$ such that $\text{time}_p(x) \leq c \times f(n)$. Now for any $i \in \mathbb{N}$ such that i maps to $(i_1, i_2) \in \mathbb{N}^2$, $p_{i_1} = p$ and $i_2 \geq c$, the returned program must compute the same function as p. □

Lemma 2. *Let $f(n) \geq n$. The class $\text{TIME}^I(\mathcal{O}(f(n))) \setminus \{\emptyset\}$ is undecidable.*

Proof. We will demonstrate a sequence of programs for which it is undecidable whether a given one of them accepts the empty set. As each of these has linear running time (and thus runs in time $\mathcal{O}(f(n))$), the theorem follows.

Given i, let the ith program accept an input if and only if it is a correct computation string of the ith Turing machine with the empty string as input. This can be done in linear time. Deciding whether the ith program accepts the empty set is equivalent to deciding whether the ith Turing machine terminates on the empty string. $\square$

The Space Hierarchy Theorem allows us to establish the same kind of result for space complexity.

We finally note the connection to Speedup:

Corollary 4. *For any $i \in \mathbb{N}$, the class $\mathrm{TIME}^I(\mathcal{O}(\frac{n^2}{(\log n)^i})) \setminus \mathrm{TIME}^I(\mathcal{O}(\frac{n^2}{(\log n)^{i+1}}))$ is undecidable.*

This follows immediately from Corollary 3. A similar statement will hold for all cases of the new Speedup Theorem.

2.7 Program Optimizers

In connection to the questions considered above we may wonder about the following: Even though we cannot decide whether a given program has the optimality property, could it be that some automatic optimizer would always yield such a program if it existed ? This would not imply decidability of the optimality property – the optimizer could construct an optimal program without realising it had done so.

Alton, in [1], answers this in the negative. He proves three very general theorems, the first one of which implies Theorem 11 below.

Definition 5. Let $h : \mathbb{N} \to \mathbb{N}$ be a strictly growing function (i.e. $\forall n \in \mathbb{N} : h(n) < h(n+1)$), and let $\Phi : \mathbb{D} \to \mathbb{D}$ be computable. We say that p is *h-optimal almost everywhere* for Φ if and only if

$$[\![p]\!] = \Phi \wedge \forall p' : [\![p']\!] = \Phi \Rightarrow \overset{\infty}{\forall} d \in \mathbb{D} : \mathrm{time}_p(d) \leq h(\mathrm{time}_{p'}(d))$$

We say that p is *h-optimal infinitely often* for Φ if and only if

$$[\![p]\!] = \Phi \wedge \forall p' : [\![p']\!] = \Phi \Rightarrow \overset{\infty}{\exists} d \in \mathbb{D} : \mathrm{time}_p(d) \leq h(\mathrm{time}_{p'}(d))$$

Note that these are very weak definitions of optimality: An $h - optimal$ program should not too often be too much slower than any other program computing the same function.

Theorem 11 (Alton). *There is a strictly growing but polynomially bounded $\tilde{h} : \mathbb{N} \to \mathbb{N}$ such that for any $h : \mathbb{N} \to \mathbb{N}$ for which $\forall n : h(n) \geq \tilde{h}(n)$ there does not exist a program* `tr` *such that the following condition holds:*

$$\forall p : \left(\exists p' : [\![p']\!] = [\![p]\!] \wedge p' \textit{ is } \tilde{h}\textit{-optimal almost everywhere}\right) \Rightarrow$$
$$([\![\mathtt{tr}]\!](p) = p'' \wedge [\![p'']\!] = [\![p]\!] \wedge p'' \textit{ is } h\textit{-optimal infinitely often})$$

So we cannot expect an optimizer to produce optimal programs in general, even if we only require infinitely often h-optimality for some large h. However, in Chapter 3 we shall prove the following result:

Theorem 12. *There is a program $\mathtt{tr}_{opt}$ such that*

- $[\![\mathtt{tr}_{opt}]\!]$ *is total,*
- $\forall p : [\![[\![\mathtt{tr}_{opt}]\!](p)]\!] = [\![p]\!]$, *and*

- *If p and p' are programs and there is a proof that*

$$\forall d, d' \in \mathbb{D} : [\![p']\!](d) = d' \Rightarrow [\![p]\!](d) = d'$$

then $time_{[\![\mathtt{tr}_{opt}]\!](p)}(d) = \mathcal{O}(time_{p'}(d))$.

2.8 Natural Problems

Finding optimal algorithms for given problems is the key goal of algorithmics. Behind this goal there seems to be an expectation that such optimal algorithms *do* exist for the considered problem. This expectation has been supported not only by the results of Section 2.4, but also by the tremendous succes of the field – for a very large number of practical problems, optimal algorithms *have* been found.

So it might seem that all "natural problems" *do* have optimal programs. At the current, we have *no* evidence to counter this claim. Gill and Blum, in [5], say that

> An informal but useful criterion for a recursive function to be natural is that its definition does not involve the concept of time of computation.

Whether there are problems that fit this informal definition of naturality but which do not have an optimal program is an open question at the present time – certainly none of the problems produced by the constructions of Chapter 4 can be considered natural. But if such a problem was discovered it could change our view on algorithmics significantly. We will refrain from making any conjectures regarding the existence of such problems.

2.9 Overview of this Part

Having finished our general discssion, we move on to the more technical side. In Chapter 3 we present and prove Levin's Theorem. We also provide a technical note about the program enumeration involved in the proof of the result. In the last section of the chapter we establish a result about program transformations that has Theorem 12 (above) as a consequence.

In Chapter 4 we present Blum's Speedup Theorem in detail and give a proof of it that is intended to give the reader some intuition about the result as a preparation for the second half of that chapter. In this second half we state and prove our new speedup theorem that has Theorems 2 and 3 as consequences.

Finally, in Chapter 5 we sum up and point to future work.

Chapter 3

Variants of Optimality

In this chapter we state Levin's Optimal Search Theorem and derive some variants of this result.

Levin's Optimal Search Theorem was originally proved in [9] (in Russian, [8] is an English version). A version of it was formulated and proved by us in [4]. For completeness, that proof is quoted here (with a few modifications) as Section 3.1. The proof was inspired by Amir Ben-Amram's exposition in [6].

3.1 The Classical Theorem

Definition 6. Let p be a program. A program w is called a *witness program* for program p, if for all $d \in \mathbb{D}$

$$(\exists d' : [\![p]\!](d.d') = \texttt{true}) \text{ implies } [\![p]\!](d.[\![w]\!](d)) = \texttt{true}$$

Theorem 13 (Levin). *Let p be a program. There is a witness program,* `opt`*, for p such that for any witness program w for p,*

$$time_{\texttt{opt}}(d) = \mathcal{O}(time_w(d) + time_p(d.[\![w]\!](d)))$$

where $d \in \mathbb{D}$.

Proof. We shall construct `opt` explicitly from p. Program `opt` uses the technique of *dovetailing*. For this, we will need a constant-time enumeration of programs. We assume that this is implemented by letting `opt` have access to an auxiliary variable, `Prog`, and two associated subroutines, `init-Prog` and `next-Prog`. Calling `init-Prog` will initialise `Prog` to be the first program in the enumeration. Calling `next-Prog` will at any time cause `Prog` to assume the value of the next program in the enumeration. Both operations take constant time to perform.

Let $p_0, p_1, \ldots$ be the programs as enumerated. Now, given d, `opt` will work as follows:

Program	$i = 1$	$i = 2$	$i = 3$	$\cdots$	i
p_0	1	2	4	$\cdots$	2^{i-1}
p_1	–	1	2	$\cdots$	2^{i-2}
p_2	–	–	1	$\cdots$	2^{i-3}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
p_k	–	–	–	$\cdots$	2^{i-k-1}

Figure 3.1: The number of steps simulated in the ith loop of the algorithm.

```
1. Given $d$.
2. init-Prog.
3. List := nil.
4. REPEAT
   (a) Derive from Prog the program Prog′, such that for any input $x$:
           $[\![\texttt{Prog}']\!](x) = (y.[\![p]\!](x.y))$  where  $y = [\![\texttt{Prog}]\!](x)$
   (b) List := cons Prog′ List.
   (c) Current := List.
   (d) Time := 1.
   (e) REPEAT
       i. Simulate the program in hd Current (applied to input $d$) by Time
          steps.
       ii. If the simulation terminates answering ($d'$.true), output $d'$ and halt.
           If the simulation terminates with an answer not of this form, remove
           the program from List. Otherwise continue.
       iii. Current := tl Current.
       iv. Time := 2 * Time.
       UNTIL Current = nil.
   (f) next-Prog.
   UNTIL FALSE.
```

Figure 3.2: The algorithm for program opt.

- In the first step, the computation of p_0 on d is simulated for one unit of time.
- In the second step, the computation of p_0 on d is simulated for *two* units of time. Then p_1 (as applied to d) is simulated for one unit of time.
- In the third step, p_0 is simulated for four units of time. Then p_1 is simulated for two time units. Finally p_2 is simulated for one time unit.

And so on. In each step, a new program is introduced and its computation on d is simulated for one unit of time. For each new step, the time spent simulating it will be doubled until (if) it terminates (see figure 3.1). The computed value, d', will then be supplied to the program p along with d. If $[\![p]\!](d, d') = \texttt{true}$, we answer d'. Otherwise we just ignore both d' and the program in question.

An explicit algorithm for opt is shown in figure 3.2. In the ith round of the outer loop, a new program (which we shall call p_i') is derived from p_i. The derived program is basically a composition of p_i and p. It is added to a list, which is then traversed by the inner loop. In this loop, the derived programs are simulated in the manner described above.

To prove the theorem, we need to establish two facts: That opt *is* a witness program, and that the claimed running time property holds.

We will begin by proving that `opt` is a witness program for p. Let d be given, and choose d' such that $[\![p]\!](d.d')=$ `true` (if there is no such d', there is nothing to prove). By looking at the loop in line 4(e), we see that `opt` yields only values that programs constructed in line 4(a) have paired with the value `true`. By line 4(a) it is now clear that only correct answers are given. Furthermore, for some i, p_i must be a program of the form "Output d'". As p'_i *does* terminate with a correct answer for this i, so will `opt`, provided it simulates this program for enough time units. But as p'_i moves further down `List`, it will eventually be simulated long enough for it to answer. Thus, `opt` *is* a witness program for p.

Now for the time analysis. An important assumption here is that the simulation done in line 4(e)(i) has constant slowdown, i.e. there is a universal constant α, such that simulating any program p on any input d takes time less than $\alpha \times \text{time}_p(d)$. As explained in [6], such an interpreter can be implemented.

Assume w is a witness program for p. Since the `Prog`-routine enumerates *all* programs, there is a k such that $w = p_k$. Notice that from the construction in line 4(a), there must be a constant c such that for all $d \in \mathbb{D}$ that have a witness:

$$\text{time}_{p'_k}(d) \leq c(\text{time}_w(d) + \text{time}_p(d.[\![w]\!](d)) \qquad (3.1)$$

Now, given d, assume there *is* an answer to find. Run `opt` on d. Now observe the following facts:

1. Since the simulation done in line 4(e)(i) has constant slowdown, the running time of line 4(e) is at most $a(1 + 2 + 4 + \cdots + 2^{|\texttt{List}|-1}) < a(2^{|\texttt{List}|})$ for some constant a.

2. In each loop of line 4, the length of `List` is increased by 1.

3. Thus, if `opt` yields an answer during the ith loop of line 4, the total running time of `opt` is at most $b(2a+4a+8a+\cdots+2^i a) < ab2^{i+1}$ for some constant b.

Furthermore, we have:

4. After k loops of line 4, the program p'_k derived from $p_k= w$ will be inserted into `List`.

5. Thus, when $i \geq k$, `opt` simulates the first 2^{i-k-1} steps of p'_k during the ith loop of line 4.

6. By equation 3.1, `opt` must yield an answer in the first loop i of line 4 such that $2^{i-k-1} \leq c(\text{time}_w(d) + \text{time}_p(d.[\![w]\!](d)))$ i.e. the first i such that $2^i \leq c2^{k+1}(\text{time}_w(d) + \text{time}_p(d.[\![w]\!](d)))$.

By observations 3 and 6, `opt` answers in time at most

$$abc2^{k+2} \times (\text{time}_w(d) + \text{time}_p(d.[\![w]\!](d)))$$

This completes the proof. □

Remark 1. The reader should note the immediate implications of the above theorem. Assume for a given problem that checking a witness is *always* faster than producing it. An example would be the *SAT* problem on a RAM: Checking that an assignment is satisfying can be done in linear time, and surely no program could ever produce these assignments in sublinear time. This means that `opt` is an optimal witness program. Formally, this can be stated as Corollary 6 below.

Corollary 5. *Let p be a program, and let w be a witness program for p such that*

$$time_p(d.[\![w]\!](d)) = \mathcal{O}(time_w(d))$$

Then $time_{\texttt{opt}}(d) = \mathcal{O}(time_w(d))$.

Definition 7. We say that p is *swift* if and only if for every witness program w for p there is a constant $k_w \in \mathbb{N}$ such that for all $d \in \mathbb{D}$:

$$\mathrm{time}_p(d) \leq k_w \times \mathrm{time}_w(d)$$

Corollary 6. *If p is swift then* `opt` *is a time-optimal witness program (up to a constant factor).*

3.2 Limiting the Enumerated Programs

In [4] we worked with implementation of the described algorithm. In this connection, an interesting question was proposed to us:

> Would it work if the range of the program enumeration is the set of all *total* programs ? (Forget about this set's undecidability for now). The question is whether optimal solutions would be found.

We have since found some answers to this. As a first answer: In "nice" situations, it will work (rather trivially). Please recall the definitions regarding predicates given in Section 2.2.

In the following, let p be a checker for a binary predicate ρ, and assume that p is swift. Furthermore, let w be a witness program for p.

Theorem 14. *Let $t : \mathbb{D} \to \mathbb{N}$ be a total time constructible function such that $x \in S_\rho \Rightarrow \mathrm{time}_w(x) \leq t(x)$. Then there is a total witness program w' for p that runs in time $\mathcal{O}(t(x))$ on all inputs.*

Proof. The total witness program w' works like this: Given x, it simulates w with time bound $t(x)$ on a timed interpreter. If w terminates within the time bound, w' outputs the resulting value. Otherwise it outputs a dummy value (in this case we must have $x \notin S_\rho$.

□

Thus if the running times involved are standard, "nice" time functions, we may assume the witness programs to be total.

But even with very simple programs we may get results. In the following constructions

- We do not assume any one program in the enumeration to solve the search problem for ρ.
- All enumerated programs are not only total but *constant-time*. That is, each program has a running time that is bounded by a constant independent of the size of the input. Thus, each program corresponds to a single boolean expression or circuit.

Theorem 15. *Given a witness program w, there is an enumeration producing only constant-time programs that makes algorithm* `opt` *run in time $\mathcal{O}((\mathrm{time}_w(x))^2)$.*

Note that this enumeration works only for the given witness program w.

Proof. Given x, the ith program simulates 2^i steps of the computation of w on x. If w terminated, its answer is returned. Otherwise a dummy value is returned. This program is obviously constant-time as i is independent of x.

Now, given an $x \in S_\rho$, program w will compute a witness in $\text{time}_w(x)$ steps. Thus, program number $i = \lceil \log \text{time}_w(x) \rceil$ will provide the same witness in time $\mathcal{O}(\text{time}_w(x))$. This means that the `opt` algorithm will run in time $\mathcal{O}(2^i \text{time}_w(x))$, i.e. in time $\mathcal{O}((\text{time}_w(x))^2)$.

□

This is not a very interesting result as the construction works for only one w. It turns out that we do have a remedy for this:

Theorem 16. *There is an enumeration producing only constant-time programs that makes algorithm* `opt` *run in time* $\mathcal{O}((\text{time}_w(x))^c)$ *for any witness program w, where c is a constant depending only on w (and not on x).*

Proof. This enumeration simply interleaves all sequences like the one constructed in the theorem above.

Define the bijection

$$\text{split} : \mathbb{N}_+ \to \mathbb{N}_+ \times \mathbb{N}_+$$

by $i \mapsto (l+1, \frac{u+1}{2})$ where $i = 2^l u$ is the unique decomposition of i into a power of twos and an odd number.

Now we may prove the theorem by letting the ith program simulate the jth program of the standard enumeration for 2^k steps, where $\text{split}(i) = (j, k)$. Again, if the simulation terminates the result is returned. Otherwise a dummy value is returned. Again, these programs are constant-time.

Let x be given. If $w = p_m$ computes the witness y for x in time $\text{time}_w(x)$ then the ith program of the new enumeration will return y in time $\mathcal{O}(\text{time}_w(x))$ where $i = 2^{m-1}(2\lceil \log \text{time}_w(x) \rceil - 1)$, Thus the `opt` algorithm will return a witness in time at most $\mathcal{O}(2^i \text{time}_w(x)) = \mathcal{O}((\text{time}_w(x))^{2^m+1})$.

□

3.3 A Levin Theorem for Program Transformation

Finally, we wish to consider a slightly different situation. This has to do with programs output by "program transformations" or "program optimizers".

Definition 8. We call a program `tr` a *transformation*, if for any program p and all values $d \in \mathbb{D}$

$$[\![\texttt{tr}]\!](p) = p' \wedge [\![p']\!](d) = d' \Rightarrow [\![p]\!](d) = d'$$

This condition is very weak: Even a program, $p_\perp$, computing the function $\lambda p.\perp$ is a transformation. If, however, `tr` *always terminates*, we call it a *total transformation.* This is still a somewhat weak condition: A program computing the function $\lambda p.p_\perp$ (with $p_\perp$ as above) is a total transformation. If `tr` is total and biimplication holds in the above condition, we say that `tr` is a *stable transformation.*

Theorem 17. *Let* $\texttt{tr}_1, \texttt{tr}_2, \ldots$ *be a recursively enumerable sequence of transformations. There is a total transformation,* `tr`*, such that for all i and all p:*

$$[\![\texttt{tr}_i]\!](p) = p' \Rightarrow \textit{time}_{[\![\texttt{tr}]\!](p)}(d) = \mathcal{O}(\textit{time}_{p'}(d))$$

If there is an i such that $\texttt{tr}_i$ *is stable, then* `tr` *is stable too.*

Note that the enumerated transformations do *not* have to be total, and yet `tr` will be. Also, if just one of the enumerated transformations satisfies the strong condition of being stable then so does `tr`. Actually, the condition that one of the enumerated transformations be stable could be relaxed. All that we will need to make `tr` stable is that

$$\forall p \forall d, d' \in \mathbb{D} : \; [\![p]\!](d) = d' \Rightarrow \exists i : [\![\mathtt{tr}_i]\!](p) = p' \wedge [\![p']\!](d) = d'$$

Proof. (Of Theorem 17). Below we will sketch an interpreter `int`. Program `tr` simply specializes this interpreter to the input program. The specialization can be trivial and thus `tr` will surely terminate. Proof of the other claims follow the lines of the proof of Levin's Theorem and is omitted here.

The interpreter `int` takes as input a program p and a value d. Like the `opt` algorithm of Section 3.1, `int` simulates a sequence of programs under a time budget that distributes half the time to the first program, one fourth of the time to the second program, one eight of the time to the third program and so on. In the `opt` algorithm, the ith program is a composition of a checker and the ith program of a reference enumeration of all programs. In `int`, the ith program first applies $\mathtt{tr}_i$ to the input program p, then (if $\mathtt{tr}_i$ stopped on p) applies the resulting program to the input d. There is no checker – if one of the programs answers, this answer is output by `int`. The checker is not needed since all the $\mathtt{tr}_i$s are transformations.

□

We now fix our attention to the sequence of all programs that provably are transformations. By standard methods, this sequence is recursively enumerable. We denote by $\mathtt{tr}_{opt}$ the total transformation formed from this sequence in the manner of the Theorem 17.

Remark 2. Since the stable transformation "READ d; OUTPUT d" is in this sequence, $\mathtt{tr}_{opt}$ is stable, and for any program p:

$$\text{time}_{[\![\mathtt{tr}_{opt}]\!](p)}(d) = \mathcal{O}(\text{time}_p(d))$$

Note that the constant factor in this relation need not be large. If the simple "identity program" is the first in the sequence, $\mathtt{tr}_{opt}(p)$ will in the worst case be about a factor of two slower than simulating p using our "standard" timed interpreter.

Corollary 7. *Let p and p' be programs. If there is a proof that*

$$\forall d, d' \in \mathbb{D} : \; [\![p']\!](d) = d' \Rightarrow [\![p]\!](d) = d'$$

then $time_{[\![\mathtt{tr}_{opt}]\!](p)}(d) = \mathcal{O}(time_{p'}(d))$

Proof. If there is a proof of the condition, then the program

READ p_0; IF $(p_0 = p)$ THEN OUTPUT p' ELSE OUTPUT p_0

is provably a transformation.

□

Remark 3. If `tr` is provably a transformation, and p is any program, $[\![\mathtt{tr}_{opt}]\!](p)$ will always be asymptotically at least as fast as $[\![\mathtt{tr}]\!](p)$. Hence the *opt*-tag. There are transformations which for some input programs will yield asymptotically faster result programs than $\mathtt{tr}_{opt}$ (we omit the proof of this), but it cannot be proved that these *actually are* transformations. Corollary 7 remains interesting as people tend to prefer program transformations that provably do not change the semantics of the input programs.

Chapter 4

Two Speedup Theorems

We first explain and present the classical *Blum's Speedup Theorem*. It is stated in a context where program inputs are natural numbers. After this presentation, we move on to a similar, low-complexity result, which is presented for our standard data values from $\mathbb{D}$.

4.1 Speedups and Budgets

Blum proved his famous Theorem in [2]. Let us first state the theorem:

Theorem 18 (Blum's Speedup Theorem). *For any time constructible, total recursive function h, there is a computable function $\Phi : \mathbb{N} \to \{0,1\}$ such that*

$$\forall \mathtt{p} : [\![\mathtt{p}]\!] = \Phi \text{ implies } \exists \mathtt{p}' : [\![\mathtt{p}']\!] = \Phi \wedge \overset{\infty}{\forall} \mathtt{n} : \mathit{time}_{\mathtt{p}}(\mathtt{n}) \geq \mathtt{h}(\mathit{time}_{\mathtt{p}'}(\mathtt{n}))$$

In this section we will discuss the correspondance between the speedup, h, mentioned in Blum's Speedup Theorem and the time budgets used in the proof (which will be given later).

The speedup h is a "parameter" of the theorem – given such a function, we prove the existence of a set such that for any program p accepting this set, there is another program accepting the set in time $h^{-1}(\mathit{time}_p(x))$.

What the proof actually constructs given a function h, is a sequence of programs $\mathtt{blum}_0$,$\mathtt{blum}_1, \ldots$ that all decide the same set (call it B). The first interesting property of the programs in this sequence is that *each program is significantly faster than its predecessor*. The second interesting property is that *any program deciding the set B is provably slower than some program in the sequence* (on all but finitely many input values).

To put it more formally, given h we define the kth budget on n to be

$$b_k(n) = h^{(n-k)}(1)$$

Under the assumption that these budgets are time constructible and $h(n) \geq 2n$ for all n, the proof demonstrates that

- $\mathit{time}_{\mathtt{blum}_k}(n) \leq b_{k+1}(n)$ for any k and all but finitely many values of n
- For any program p deciding B, there exists a k such that $\mathit{time}_p(n) > b_k(n)$ for all but finitely many values of n.

Thus, the time bounds of the sequence of programs "leap by h in each step" (since $h(b_{k+1}(n)) = b_k(n)$), and every program deciding the same set as the Blum programs is outrun by all the programs in the sequence with index larger than some k.

The assmuption that h has to be time constructible is not really needed to prove the theorem (and is not required in the formulation of the theorem in [6]). We use it to simplify our exposition of the proof.

The minimal h treated in the proof is $h(n) = 2n$, asserting the existence of a problem for which all programs solving it may be sped up by a factor of two. As can be seen from the above, the programs $\texttt{blum}_k$ solving this problem will all run in time $\Theta(2^n)$. Larger speedups will result in even higher complexity problems, e.g. for quadratic speedup we define $h(n) = \max(4, n^2)$ and get programs running in time $\Theta(2^{2^n})$. Assuming that the representation d of a number n has size $\Theta(\log n)$ (as is reasonable), we get running times $\Theta(2^{2^{|d|}})$ and $\Theta(2^{2^{2^{|d|}}})$ respectively.

4.2 Time Analysis

4.2.1 The Algorithm

We now give the algorithm for $\texttt{blum}_0$ on input n.

Definition 9. Let the budgets $b_k(n)$ be given. Define $Q(i,j)$ to be true if $\mathit{time}_{p_i}(j) < b_i(n)$, false otherwise. If $Q(i,j)$ is true we say that "program p_i is quick on input j". The algorithm for $\texttt{blum}_0$ on input n is sketched in Figure 4.1. We denote by B the set $\{n \in \mathbb{N} \mid [\![\texttt{blum}_0]\!](n) = \texttt{true}\}$.

Comments We will use the terminology that if $T[i,n] = \texttt{true}$ after this algorithm finishes we say that program p_i has been *killed* on input n. By the definition of $[\![\texttt{blum}_0]\!]$, this function cannot be computed by a program that is killed on some input.

The simple, but central observation necessary to obtain the speedup theorem is that *programs live only once.* If program p_i is killed on input n, it will not be chosen for killing on any input $n' > n$. This can easily be seen from Figure 4.3. So every program is killed at most once. Thus, for any k there is a $J_k \in \mathbb{N}$ such that none among $p_1, p_2, \ldots, p_k$ are killed on any input $j \geq J_k$. From this observation we make two others:

- If p_{k+1} is quick on some input greater than or equal to J_k then it will be killed. Thus, any program that is infinitely often quick cannot decide B.
- As the final value of $T[i,j]$ (after running the algorithm) is given for $i \leq k$ and $j > J_k$ (it is $\texttt{false}$), it is not essentially necessary to compute $Q(i,j)$ for these values. That is, if our program initialized T with enough information to answer on all inputs smaller than J_k, we would not need to simulate the first k programs in computations on inputs larger than J_k.

The latter point motivates the definition of $\texttt{blum}_k$:

Definition 10. For all $k > 0$, the algorithm $\texttt{blum}_k$ is as given in Figure 4.4. By the above observations, $\texttt{blum}_k$ decides B for all k.

4.2.2 Running time

From the above description it should be clear that the running time of each algorithm is of the order of the time needed to compute the necessary values of Q. By the existence of a constant slowdown timed interpreter (as shown in [6]), the value

1. Read n.
2. Compute $Q(i,j)$ for $j = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, j$ and initialize table T as sketched in Figure 4.2. The upper right triangle of T is not used.
3. Update T such that every row has at most one `true` using the algorithm in Figure 4.3.
4. By definition

$$[\![\texttt{blum}_0]\!](n) = \begin{cases} \neg[\![p_i]\!](n) & \text{if } T[i,n] = \texttt{true} \text{ for some } i \\ \texttt{false} & \text{otherwise} \end{cases}$$

Figure 4.1: Definition of $\texttt{blum}_0$ on input n.

Row	$i = 1$	$i = 2$	$i = 3$	$\ldots$	$i = n$
$T[i,1]$	$Q(1,1)$				
$T[i,2]$	$Q(1,2)$	$Q(2,2)$			
$T[i,3]$	$Q(1,3)$	$Q(2,3)$	$Q(3,3)$		
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	
$T[i,n]$	$Q(1,n)$	$Q(2,n)$	$Q(3,n)$	$\ldots$	$Q(n,n)$

Figure 4.2: Initial value of the table T used by the $\texttt{blum}_0$ algorithm.

```
FOR $i = 1, 2, \ldots, n$ DO
    1. If there is a $n'$ such that $T[i,n']$ is true, pick the least such
       $n'$. (Else continue with next $i$.)
    2. Set $T[K,n']$ to false for all $K > i$.
    3. Set $T[i,m]$ to false for all $m > n'$.
```

Figure 4.3: Updating T/Picking quick program to kill.

1. Read n.
2. If $n \leq J_k$ then output $[\![\texttt{blum}_0]\!]$ using the algorithm in Figure 4.1. Else continue to step 3.
3. Compute $Q(i, j)$ for $j = 1, 2, \ldots, n$ and $i = k+1, k+2, \ldots, j$ and initialize table T as sketched in Figure 4.5. The upper right triangle of T is not used.
4. Set $T[i, j] = \texttt{true}$ if $i \leq k$, $j \leq J_k$ and the $\texttt{blum}_0$ algorithm kills p_i on input j. (This finite set of updates of T is hard-coded into the algorithm.)
5. Update T such that every row has at most one `true` using the algorithm in Figure 4.3.
6. By definition

$$[\![\texttt{blum}_k]\!](n) = \begin{cases} \neg[\![p_i]\!](n) & \text{if } T[i,n] = \texttt{true} \text{ for some } i \\ \texttt{false} & \text{otherwise} \end{cases}$$

Figure 4.4: Definition of $\texttt{blum}_k$ on input n.

Row	$i = 1$	$\ldots$	$i = k$	$i = k+1$	$\ldots$	$i = n$
$T[i, 1]$	`false`					
$\ldots$	$\ldots$	$\ldots$				
$T[i, k]$	`false`	$\ldots$	`false`			
$T[i, k+1]$	`false`	$\ldots$	`false`	$Q(k+1, k+1)$		
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	
$T[i, n]$	`false`	$\ldots$	`false`	$Q(k+1, n)$	$\ldots$	$Q(n, n)$

Figure 4.5: Initial value of the table T used by the $\texttt{blum}_k$ algorithm.

of $Q(i, j)$ can be computed in time $\mathcal{O}(b_i(j))$. Thus, the total running time of the $\texttt{blum}_0$ algorithm on input n is less than

$$c \times \sum_{j=1}^{n} \sum_{i=1}^{j} b_i(j)$$

for some constant c, assuming all budgets are time constructible and $b_i(j) \geq 1$ for all $i < j$.

More generally, for any k, program $\texttt{blum}_k$ runs in time

$$c \times \sum_{j=1}^{n} \sum_{i=k+1}^{j} b_i(j)$$

on input n (with the same constant and assumptions as above).

4.2.3 The Theorem

We are now ready to sum up:

Theorem 19. *Given time budgets $b_k(n)$ that are time constructible and for which $b_i(j) \geq 1$ for all $i < j$, there is a set $B = \{n \in \mathbb{N} \mid \texttt{blum}_0(n) = \texttt{true}\}$ such that*

- *For every program p accepting B there is a k such that $time_p(n) > b_k(n)$ for all but finitely many values of n.*
- *There is a sequence $\texttt{blum}_0, \texttt{blum}_1, \ldots$ of programs accepting B for which $time_{\texttt{blum}_k}(n) \leq c \times \sum_{j=1}^{n} \sum_{i=k+1}^{j} b_i(j)$ for some constant c.*

As a consequence, we get *Blum's Speedup Theorem*:

Corollary 8. *For any time constructible, total recursive function h, there is a computable function $\Phi : \mathbb{N} \rightarrow \{0, 1\}$ such that*

$$\forall \mathtt{p} : [\![\mathtt{p}]\!] = \Phi \text{ implies } \exists \mathtt{p}' : [\![\mathtt{p}']\!] = \Phi \wedge \forall_0^\infty \mathtt{n} : time_{\mathtt{p}}(\mathtt{n}) \geq \mathtt{h}(time_{\mathtt{p}'}(\mathtt{n}))$$

4.3 A Low-Complexity Result

In the rest of this chapter we use data values from a set $\mathbb{D}$ as defined in [6]. We assume to have constant-time enumerations of all programs $p_j \in \texttt{Programs}$ and all values $d_i \in \mathbb{D}$. Of the latter we assume for convenience that indices are positive, i.e. $i \in \mathbb{N}_+$ and that

$$|d_i| < 2i \text{ for all } i \in \mathbb{N}_+$$

This is reasonable – in $\mathbb{D}$ there is one element of size 1, one element of size 3, two elements of size 5, and for s odd and larger than 5, the number of values of size s is more than double the number of elements of size $s - 2$.

We also use two functions $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ and assume for all $n \in \mathbb{N}$: $g(n) \geq 2$. We define a sequence of *budget functions* $b_j : \mathbb{N} \rightarrow \mathbb{N}$ by

$$b_j(n) = \frac{f(n)}{(g(n))^j}$$

for all $j, n \in \mathbb{N}$ (rounding down if necessary). In particular $b_0 = f$. Of these budget functions we assume

(4.1) The function $b : \mathbb{N}^2 \to \mathbb{N}$ given by $b(j,n) = b_j(n)$ is time constructible

$$\forall j \in \mathbb{N} \forall n \in \mathbb{N} : b_j(n) \geq n \tag{4.2}$$

$$\forall j \in \mathbb{N} : b_j \text{ is non-decreasing} \tag{4.3}$$

$$\forall j \in \mathbb{N} \forall n \in \mathbb{N} : (\log n) \times b_j(2 \log n) < b_j(n) \tag{4.4}$$

Equation 4.4 may be thought of as a kind of uniformity condition on the growth of the budget functions.

Examples The reader may check that the above assumptions hold when

$$f(n) = 2^n \text{ and } g(n) = \max(n, 2)$$

Likewise, the assumptions hold when

$$f(n) = n^2 \text{ and } g(n) = \max(\log n, 2)$$

Actually, the assumptions hold for almost any superlinear, time constructible f and $g(n) = 2$, but for our results to be non-trivial, g must be non-constant. Still, f can be very close to linear. A simple example is

$$f(n) = n \times \log n \text{ and } g(n) = \max(\log \log n, 2)$$

Result In the following we prove the existence of a computable function $\Phi : \mathbb{D} \to \mathbb{D}$ such that

Theorem 20.

$$\forall p \in \texttt{Programs} : [\![p]\!] = \Phi \Rightarrow \exists k \in \mathbb{N} \forall x \in \mathbb{D} : \mathit{time}_p(x) > \frac{f(|x|)}{g(|x|)^k}$$

Theorem 21.

$$\forall k' \in \mathbb{N} \exists c \in \mathbb{N} \exists p' \in \texttt{Programs} : [\![p']\!] = \Phi \wedge \forall x \in \mathbb{D} : \mathit{time}_{p'}(x) \leq c \times \frac{f(|x|)}{g(|x|)^{k'}}$$

Although this result is certainly of the same flavour as Theorem 19 neither one implies the other. This theorem does not assert the existence of as large speedups as Blum's Speedup Theorem does. However, it does assert the existence of smaller speedups *within very limited complexity classes*:

Corollary 9. *Given f and g as described above there is a computable function $\Phi : \mathbb{D} \to \mathbb{D}$ such that*

$$\exists C \in \mathbb{N} \exists p : [\![p]\!] = \Phi \wedge \forall x \in \mathbb{D} : \mathit{time}_p(x) \leq C \times f(|x|) \tag{4.5}$$

$$\begin{array}{c} \forall p : [\![p]\!] = \Phi \text{ implies} \\ \exists c \in \mathbb{N} \exists p' : [\![p']\!] = \Phi \ \wedge \forall x \in \mathbb{D} : \mathit{time}_{p'}(x) < c \times \frac{\mathit{time}_p(x)}{g(|x|)} \end{array} \tag{4.6}$$

Proof. (Of Corollary). Equation 4.5 follows from Theorem 21 with $k' = 0$ and $C = c$.

For Equation 4.6, let p computing Φ be given. Choose k as in Theorem 20. Now use Theorem 21 with $k' = k + 1$, and the claim follows.

□

Result Proof Outline The above result is proved along the same lines as Theorem 19: We define Φ and prove Theorem 20 by diagonalization (this will be Theorem 22). We then construct an algorithm that computes Φ and runs in time $\mathcal{O}(f(n))$. Finally, we generalise this result and obtain Theorem 24 which proves Theorem 21.

4.4 The Proof

4.4.1 Definitions

For $x \in \mathbb{D}$ we define the set Quick_x by

$$\boxed{p_j \in \mathit{Quick}_x \Leftrightarrow \mathrm{time}_{p_j}(x) < b_j(|x|)}$$

and let $\mathit{last}_x = \max\{j \mid b_j(|x|) > 1\}$. For convenience, we assume that the running time of any program on any input is non-zero. Note that $\mathit{Quick}_x \subseteq \{p_0, p_1, \ldots, p_{\mathit{last}_x}\}$. Note also that $\mathit{last}_x \leq \lceil \log b_0(|x|) \rceil - 1$ and in general $\mathit{last}_x - k \leq \lceil \log b_k(|x|) \rceil - 1$ when $k \in \mathbb{N}$.

We let Quick_x be represented by a table, $\mathtt{Q}_x[]$, of boolean values, such that $\mathtt{Q}_x[0]$ is true if and only if $p_0 \in \mathit{Quick}_x$, $\mathtt{Q}_x[1]$ is true if and only if $p_1 \in \mathit{Quick}_x$, and so on. The last entry is $\mathtt{Q}_x[\mathit{last}_x]$.

Now define Kill_x by

$$\boxed{p_j \in \mathit{Kill}_x \Leftrightarrow (p_j \in \mathit{Quick}_x) \wedge (\forall j' < j : p_{j'} \notin \mathit{Kill}_x) \wedge (\forall i < \log|x| : p_j \notin \mathit{Kill}_{d_i})}$$

The set is well-defined as $i < \log|x| \Rightarrow |d_i| < 2\log|x| \leq |x|$ by the assumption on the enumeration of $\mathbb{D}$. Also note that $|\mathit{Kill}_x| \leq 1$ and $\mathit{Kill}_x \subseteq \mathit{Quick}_x$. We use a representation like that of Quick_x, that is we let Kill_x be represented by a table, $\mathtt{K}_x[]$, of boolean values, the ith indicating whether $p_i \in \mathit{Kill}_x$.

We are now ready to define the function discussed in the theorem. It is given by

$$\boxed{\Phi(x) = \begin{cases} \mathtt{true} & \text{if } \exists p \in \mathtt{Programs} : \mathit{Kill}_x = \{p\} \wedge [\![p]\!](x) = \mathtt{false} \\ \mathtt{false} & \text{otherwise} \end{cases}}$$

Note the diagonalizing nature of Φ: If $\mathit{Kill}_x = \{p\}$ then $\Phi(x) \Leftrightarrow \neg[\![p]\!](x)$ and so $[\![p]\!] \neq \Phi$.

4.4.2 Lower Bound

Lemma 3. *For any program p, the set p-Killers $= \{d \mid p \in \mathit{Kill}_d\}$ is finite.*

Proof. Assume the set to be non-empty (if it is empty, it is finite) and let $d_i \in$ *p-Killers*. For any x with $|x| > 2^i$ we have $i < \log|x|$ and thus (per definition of Kill_x) $p \notin \mathit{Kill}_x$, i.e. $x \notin$ *p-Killers*. The set *p-Killers* can therefore only contain values of size $\leq 2^i$ and is thus finite.

□

Corollary 10. *Given $k \in \mathbb{N}$ there is an $N_k \in \mathbb{N}$ such that*

$$\forall k' \leq k : \forall x \in \mathbb{D} : p_{k'} \in \mathit{Kill}_x \Rightarrow |x| \leq N_k$$

Theorem 22. *Let $p \in \mathtt{Programs}$. If $[\![p]\!] = \Phi$ there is a $k \in \mathbb{N}$ such that*

$$\overset{\infty}{\forall} x \in \mathbb{D} : \mathit{time}_p(x) > \frac{f(|x|)}{(g(|x|))^k}$$

```
Found:=false;
for $j$:=0 to $last_x$;
  if Found
  then $\mathtt{K}_x[j]$:=false
  else if $\mathtt{Q}_x[j]$
       then Dead:=false;
            for $i$:=1 to $\log|x|$
              if $\mathtt{K}_{d_i}[j]$ then Dead:=true;
            if Dead
            then $\mathtt{K}_x[j]$:=false
            else $\mathtt{K}_x[j]$:=true;
                 Found:=true;
            endif
       else $\mathtt{K}_x[j]$:=false;
       endif;
  endif;
endfor;
```

Figure 4.6: Lemma 4 algorithm.

Proof. Let $p \in \mathtt{Programs}$ be such that $[\![p]\!] = \Phi$ and let $k \in \mathbb{N}$ be such that $p_k = p$. Now choose $N = N_{k-1}$ as in Corollary 10.

If x is such that $|x| > N$ and $\text{time}_p(x) \leq \frac{f(|x|)}{(g(|x|))^k}$ then we must have $p \in Quick_x$. By the definition of $Kill_x$ and the choice of N we may conclude that either $Kill_x = \{p\}$ or there is an $x' \in \mathbb{D}$ with $|x'| < \log|x|$ for which $p \in Kill_{x'}$. Either way we must have $p\text{-}Killers \neq \emptyset$ and so $[\![p]\!] \neq \Phi$ (by the diagonalizing definition of Φ), a contradiction. This proves the theorem.

□

4.4.3 A First Upper Bound

Lemma 4. *Given x, $\mathtt{K}_{d_1}[], \mathtt{K}_{d_2}[], \ldots, \mathtt{K}_{d_{\log|x|}}[]$ and $\mathtt{Q}_x[]$ we may compute $\mathtt{K}_x[]$ in time less than*

$$\alpha \times (\log b_0(|x|))^2$$

where α is a constant independent of x.

Proof. The table is clearly computed by the algorithm in Figure 4.6.

The outer loop has $last_x + 1 \leq \lceil \log b_0(|x|) \rceil$ iterations, and each iteration clearly takes time less than $\alpha' \times \log|x| \leq \alpha' \times \alpha'' \times \log b_0(|x|)$ for some $\alpha', \alpha'' \in \mathbb{N}$, using Equation 4.2. This proves the lemma as α' and α'' are independent of x.

□

Lemma 5. *Given x,$\mathtt{Q}_{d_1}[], \mathtt{Q}_{d_2}[], \ldots, \mathtt{Q}_{d_{\log|x|}}[]$ and $\mathtt{Q}_x[]$ we may compute $\mathtt{K}_x[]$ in time less than*

$$\beta \times (\log b_0(|x|))^3$$

where β is a constant independent of x.

Proof. Incrementally build the tables using the algorithm in Figure 4.7. The algorithm in Figure 4.6 can be applied in the `for`-loop as the computation of $\mathtt{K}_{d_i}[]$ depends only on tables $\mathtt{K}_{d_{i'}}[]$ where $i' < i$ (as noted above). In particular, the value of $\mathtt{K}_{d_1}[]$ does not depend on $\mathtt{K}_{d_{i'}}[]$ for any i'.

```
for $i$:=1 to $\log|x|$
  compute K$_{d_i}$[] from K$_{d_1}$[],K$_{d_2}$[],...,K$_{d_{\log|d_i|}}$[] and Q$_{d_i}$[]
endfor;
compute K$_x$[] from K$_{d_1}$[],K$_{d_2}$[],...,K$_{d_{\log|x|}}$[] and Q$_x$[]
```

Figure 4.7: Lemma 5 algorithm.

Using Lemma 4 we see that the total running time of the algorithm on input x is less than

$$\beta' \times \alpha \times \left((\log b_0(|x|))^2 + \sum_{i=1}^{\log|x|} (\log b_0(|d_i|))^2 \right)$$

for some $\beta' \in \mathbb{N}$. Using the assumption on the enumeration of $\mathbb{D}$ we see that for each i in the sum: $|d_i| < 2i \leq 2\lfloor \log|x| \rfloor \leq |x|$. As b_0 is non-decreasing (by Equation 4.3), the running time must be less than

$$\beta' \times \alpha \times \left((\log b_0(|x|))^2 + \sum_{i=1}^{\log|x|} (\log b_0(|x|))^2 \right)$$

which is equal to

$$\beta' \times \alpha \times (1 + \log|x|) \times (\log b_0(|x|))^2$$

Finally, using the fact that $b_0(n) \geq n$ almost everywhere (Equation 4.2), we see that this is less than

$$\beta' \times \alpha \times \beta'' \times (\log b_0(|x|))^3$$

for some $\beta'' \in \mathbb{N}$.

□

Lemma 6. *Given $x \in \mathbb{D}$, the table* Q$_x$[] *can be computed in time less than*

$$\gamma \times b_0(|x|)$$

where $\gamma \in \mathbb{N}$ is independent of x.

Proof. By the existence of a constant slowdown, timed interpreter and the time constructibility of the budgets (Equation 4.1), the table may be computed in time

$$\gamma' \times \sum_{j=0}^{last_x} b_j(|x|)$$

for some $\gamma' \in \mathbb{N}$. By definition, this is equal to

$$\gamma' \times \sum_{j=0}^{last_x} \frac{b_0(|x|)}{(g(|x|))^j}$$

As $g(|x|) \geq 2$, this must be less than

$$\gamma' \times 2b_0(|x|)$$

and the lemma follows.

□

Lemma 7. *Given $x \in \mathbb{D}$, the tables $\mathtt{Q}_{d_1}[], \mathtt{Q}_{d_2}[], \ldots, \mathtt{Q}_{d_{\log|x|}}[]$ can be computed in time less than*

$$\delta \times b_0(|x|)$$

where $\delta \in \mathbb{N}$ is independent of x.

Proof. By Lemma 6, the tables may be computed in time

$$\gamma \times \sum_{i=1}^{\log|x|} b_0(|d_i|)$$

By the assumption on the enumeration of $\mathbb{D}$ and that b_0 is non-decreasing (Equation 4.3), this is less than

$$\gamma \times \sum_{i=1}^{\log|x|} b_0(2\log|x|)$$

and by the uniformity condition on b_0 (Equation 4.4) we see that this is less than

$$\gamma \times \delta' \times b_0(|x|)$$

for some $\delta' \in \mathbb{N}$. □

As an immediate consequence of the above lemmata, we have a special case ($k' = 0$) of Theorem 21:

Theorem 23. *There is a program p that computes Φ in time*

$$time_p(x) \leq c \times f(|x|)$$

Proof. Use the above lemmata and observe that if $p \in Kill_x$ then we may compute $[\![p]\!](x)$ in time less than $c' \times b_0(|x|) = c' \times f(|x|)$ for some $c' \in \mathbb{N}$.

□

4.4.4 A Sequence of Upper Bounds

Parameter k: In this section, let $k \in \mathbb{N}$ be fixed. The below statements are generalisations of those proved in Section 4.4.3.

Lemma 8. *Given $x \in \mathbb{D}$, $\mathtt{K}_{d_1}[k:], \mathtt{K}_{d_2}[k:], \ldots, \mathtt{K}_{d_{\log|x|}}[k:]$ and $\mathtt{Q}_x[k:]$ we may compute $\mathtt{K}_x[k:]$ in time less than*

$$\alpha \times (\log b_k(|x|))^2$$

where α is a constant independent of x (but not of k).

Proof. Our algorithm works in one of two ways depending on the size of x. For all values x such that $|x| \leq N_{k-1}$, the value of $\mathtt{K}_x[k:]$ is stored in a table that is "hard-coded" into our algorithm. This is possible since there are only finitely many such values.

If $|x| > N_{k-1}$ we know that if there is a p such that $Kill_x = \{p\}$ then $p = p_i$ for some $i \geq k$. Thus we know that $\mathtt{K}_x[j]$ should be `false` for all $j \leq k$. This allows us to compute $\mathtt{K}_x[k:]$ by the algorithm in Figure 4.8

The outer loop has $last_x + 1 - k \leq \lceil \log b_k(|x|) \rceil$ iterations, and each iteration clearly takes time less than $\alpha' \times \log|x| \leq \alpha' \times \alpha'' \times \log b_k(|x|)$ for some $\alpha', \alpha'' \in \mathbb{N}$, using Equation 4.2. This proves the lemma as α' and α'' are independent of x.

□

```
if $|x| \le N_{k-1}$
then K$_x[k:]$:=*(value from a hard-coded table)*
else
  Found:=false;
  for $j$:=k to $last_x$;
    if Found
    then K$_x[j]$:=false
    else if Q$_x[j]$
         then Dead:=false;
              for $i$:=1 to $\log|x|$
                if K$_{d_i}[j]$ then Dead:=true;
              if Dead
              then K$_x[j]$:=false
              else K$_x[j]$:=true;
                   Found:=true;
              endif
         else K$_x[j]$:=false;
         endif;
    endif;
  endfor;
endif;
```

Figure 4.8: Lemma 8 algorithm.

```
for $i$:=1 to $\log|x|$
  compute K$_{d_i}[k:]$ from K$_{d_1}[k:]$,K$_{d_2}[k:]$,...,K$_{d_{\log|d_i|}}[k:]$ and Q$_{d_i}[k:]$
endfor;
compute K$_x[k:]$ from K$_{d_1}[k:]$,K$_{d_2}[k:]$,...,K$_{d_{\log|x|}}[k:]$ and Q$_x[k:]$
```

Figure 4.9: Lemma 9 algorithm.

Lemma 9. *Given $x \in \mathbb{D}$, $\mathtt{Q}_{d_1}[k:], \mathtt{Q}_{d_2}[k:], \ldots, \mathtt{Q}_{d_{\log|x|}}[k:]$ and $\mathtt{Q}_x[k:]$ we may compute $\mathtt{K}_x[k:]$ in time less than*

$$\beta \times (\log b_k(|x|))^3$$

where β is a constant independent of x (but not of k).

Proof. Incrementally build the tables using the algorithm in Figure 4.9. The algorithm in Figure 4.8 can be applied in the `for`-loop as the computation of $\mathtt{K}_{d_i}[k:]$ depends only on tables $\mathtt{K}_{d_{i'}}[k:]$ where $i' < i$ (as noted above). In particular, the value of $\mathtt{K}_{d_1}[k:]$ does not depend on $\mathtt{K}_{d_{i'}}[k:]$ for any i'.

Using Lemma 8 we see that the total running time of the algorithm on input x is less than

$$\beta' \times \alpha \times \left((\log b_k(|x|))^2 + \sum_{i=1}^{\log|x|} (\log b_k(|d_i|))^2 \right)$$

for some $\beta' \in \mathbb{N}$. Using the assumption on the enumeration of $\mathbb{D}$ we see that for each i in the sum: $|d_i| < 2i \le 2\lfloor \log|x| \rfloor \le |x|$. As b_k is non-decreasing (by Equation 4.3), the running time must be less than

$$\beta' \times \alpha \times \left((\log b_k(|x|))^2 + \sum_{i=1}^{\log|x|} (\log b_k(|x|))^2 \right)$$

which is equal to

$$\beta' \times \alpha \times (1 + \log|x|) \times (\log b_k(|x|))^2$$

Finally, using the fact that $b_k(n) \ge n$ almost everywhere (Equation 4.2), we see that this is less than

$$\beta' \times \alpha \times \beta'' \times (\log b_k(|x|))^3$$

for some $\beta'' \in \mathbb{N}$.

□

Lemma 10. *Given $x \in \mathbb{D}$, the table $\mathtt{Q}_x[k:]$ can be computed in time less than*

$$\gamma \times b_k(|x|)$$

where $\gamma \in \mathbb{D}$ is independent of x (but not of k).

Proof. By the existence of a constant slowdown, timed interpreter and the time constructibility of the budgets (Equation 4.1), the table may be computed in time

$$\gamma' \times \sum_{j=k}^{last_x} b_j(|x|)$$

for some $\gamma' \in \mathbb{N}$. By definition, this is equal to

$$\gamma' \times \sum_{j=0}^{last_x - k} \frac{b_k(|x|)}{(g(|x|))^j}$$

As $g(|x|) \ge 2$, this must be less than

$$\gamma' \times 2b_k(|x|)$$

and the lemma follows.

□

Lemma 11. *Given $x \in \mathbb{D}$, the tables $\mathtt{Q}_{d_1}[k:], \mathtt{Q}_{d_2}[k:], \ldots, \mathtt{Q}_{d_{\log|x|}}[k:]$ can be computed in time less than*

$$\delta \times b_k(|x|)$$

where $\delta \in \mathbb{D}$ is independent of x (but not of k).

Proof. By Lemma 10, the tables may be computed in time

$$\gamma \times \sum_{i=1}^{\log|x|} b_k(|d_i|)$$

By the assumption on the enumeration of $\mathbb{D}$ and that b_k is non-decreasing (Equation 4.3), this is less than

$$\gamma \times \sum_{i=1}^{\log|x|} b_k(2\log|x|)$$

and by the uniformity condition on b_k (Equation 4.4) we see that this is less than

$$\gamma \times \delta' \times b_k(|x|)$$

for some $\delta' \in \mathbb{N}$. □

Theorem 24. *Let $k \in \mathbb{N}$. There is a $c \in \mathbb{N}$ and a program p that computes Φ in time*

$$time_p(x) \leq c \times b_k(|x|)$$

Proof. The theorem follows easily from the above lemmata.

□

Chapter 5

Conclusion and Future Work

We started this part of the thesis with a discussion of different aspects of the time optimality of programs and outlined several results that are relevant to the questions considered here. Two of these results were presented in depth in Chapters 3 and 4. We also provided some elaboration on the ideas behind these results.

Our main contributions to the topic were

- A proof of the undecidability of optimisability of programs in many settings (Theorem 8).
- The characterization of a progam optimizer which is as good as any in a given sequence of optimizers (Theorem 17).
- A theorem asserting that even small complexity classes posses problems for which there is no tight lower time bound on the programs solving them (Theorem 9).

There are several open questions related to our topic, but in our opinion there following one is by far the most interesting:

> Are there any “natural” problems that have speedup ?

That is, are there natural problems for which no program solving them are time-optimal up to a constant factor ? As discussed in Section 2.8, an affirmative answer to this question would have very interesting philosophical implications. An analogy: The class of *NP*-complete problems is regarded as more than a theoretical curiousity because many natural problems are in fact *NP*-complete.

Another related question that we would like to investigate is whether Theorem 9 could be improved (e.g. to assert the existence of low-complexity problems with constant factor speedup) or generalised (e.g. to other complexity measures than time).

Part II

Memoizing Both Cons And User-Defined Functions

Chapter 6

Introduction

6.1 Motivation

As the title says, this part of the thesis is about memoization. This is an old technique in the world of programming but we consider a less well-known approach, an approach which combines memoization of function calls and of heap allocations. The motivation behind this combination is to *both* speed up the computation *and* reduce the amount of space used.

Our interest in this idea took its outset in work done by us on implementing the algorithm constructed in the proof of Levin's Theorem (see Chapter 3). During this work (documented in [4]) we found that a major challenge in the implementation was to efficiently enumerate all expressions generated by a context-free grammar. While we were able to describe a solution to this problem that was theoretically satisfactory, it turned out that in practice the vast amount of space used by our algorithm was a serious efficiency problem.

This challenging problem motivated us to search for methods to improve the performance of such space-demanding algorithms. We found Stefan Kahr's so-called "Unlimp"-interpreters to be particularly interesting. An "Unlimp" interpreter manages heap usage in a clever way that minimizes the number of cells allocated by the constructors of the interpreted language. Moreover, it allows very easy implementation of function memoization. We found that this idea was so interesting in itself that we dedicated this second part of the thesis to describing the idea more carefully and investigating its possibilities in a line of experiments.

6.2 Overview of this Part

Chapter 7 presents the "Unlimp"-idea both informally and formally. In the process we define the semantics of a simple, first-order, functional language that we use as a base of our studies. In Chapter 8 we outline our implemetation of the memoizing interpreter and report on our experiments with this. Finally, in Chapter 9 we sum up our results and point to future work.

Chapter 7

Uniqueness as a Leitmotiv for Implementation

In this chapter, we discuss the ideas behind "Unlimp" (shorthand for "Uniqueness as a Leitmotiv for Implementation") program interpretation conceived by Stefan Kahrs ([7]). We also define the semantics of a simple first-order, functional language called F. We shall use this language as a base of our studies in this chapter and Chapter 8.

7.1 The F Language

The F language was used in [6] (we present a variant of the language). It is a simple first-order, functional language featuring LISP-like structured data with two constructors `nil` and `cons`, and two destructors `hd` and `tl`. For control flow it has (recursive) function calls and a conditional expression (again this is like simple LISP). The programmer can only define *one* function which is always called `f`. Likewise, there is only one parameter to this function and this is always called `X`. There is no explicit input – the computation of a program is the computation of function `f` applied to the value `nil`.

The programmer may pretend to have more variables by `cons`ing their values together in a list. To pretend to have more than one function, he can let one of these "virtual variables" represent a tag that tells which "virtual function" to invoke, then branch depending on the tag.

This technique is illustrated in the example program in Figure 7.1. The program uses a well-known algorithm to compute the fourth Fibonacci number. Some syntactic sugar has been added in a form that may be removed by the preprocessor of the C language. More example programs will be given in Section 8.1.4

The syntax of F can be seen in Figure 7.2.

7.2 On Heaps and Memoization

In this section we introduce the ideas behind the "Unlimp" way of interpreting programs. This presentation is kept informal. We refer the reader to Sections 7.3 and 7.5 for the formal definitions of the concepts involved. It may be helpful to consult these definitions while reading the current presentation to get a clearer picture of the concepts used in the discussion.

As with many languages, values in F are terms which may be thought of as trees. More concretely, all values in F are built using the 0-ary constructor `nil` and the

```
/* Numbers: */
#define ZERO   nil
#define ONE   (cons nil ZERO)
#define TWO   (cons nil ONE)
#define THREE (cons nil TWO)
#define FOUR  (cons nil THREE)

/* (hd X) will be the function tag: */
#define TAG   (hd X)

/* We have tags for two functions: plus and fib */
#define PLUS  ZERO
#define FIB   ONE

/* Which Fibonacci number to compute: */
#define INPUT FOUR

/* f(X) =  */
(if X
    (if TAG
/* Implementation of (fib y): */
#define Y (tl X)
        (if Y
            (if (tl Y)
                (f (cons PLUS (cons
                  (f (cons FIB (tl Y)))
                  (f (cons FIB (tl (tl Y))))
                )  )          )
                ONE
            )
            ONE
        )
/* Implementation of (plus n k): */
#define N (hd (tl X))
#define K (tl (tl X))
        (if N
            (cons nil (f (cons PLUS (cons (tl N) K))))
            K
        )
    )
/* Call to evaluate: */
    (f (cons FIB INPUT))
)
```

Figure 7.1: An example F program computing the fourth Fibonacci number.

```
F  →  f(X)=B

B  →  E

E  →  nil
E  →  X
E  →  hd E
E  →  tl E
E  →  cons E E
E  →  if E E E
E  →  f E
```

Figure 7.2: Grammar of the F language.

		n	0	1	2	3	$next = 4$	5	$\cdots$
$v_1 = 3$	$H_1 =$	car		0	0	1	0	0	$\cdots$
		car		0	0	2	0	0	$\cdots$

		n	0	1	2	$next = 3$	4	5	$\cdots$
$v_2 = 2$	$H_2 =$	car		0	1	0	0	0	$\cdots$
		car		0	1	0	0	0	$\cdots$

Figure 7.3: Two value-heap pairs. Value v_1 (from heap H_1) is isomorphic to v_2 (from heap H_2) in the sense that the two values can not be told apart using tests and destructors.

binary constructor `cons`. Values may be taken taken apart using the destructors `hd` and `tl`, and they may be analysed using the `pair?` test (this test is implicit in the `if` conditional expression). Thus, they behave like and may be analysed like binary trees. This is also the view taken by the standard semantics defined in Section 7.5.1.

However, in the DAG semantics of Section 7.5.2, the representation of constructed values is more elaborate. The values are indexes of an array that forms the heap of the program. For each index the array contains a heap cell containing the indices for the `hd` (also called the *car*) of the value and the `tl` (also called the *cdr*) of the value.

In this representation, the heap is a DAG but not necessarily a forest, as some nodes in the DAG have indegree larger than one – they are *shared* by several values on the heap.

An example of this sharing can be seen in Figure 7.3 (for an explanation of the *next* symbol, see the definition in Section 7.5.2). Two different value-heap pairs are given. Both values represent the expression

```
(cons (cons nil nil) (cons nil nil))
```

in their respective heaps. In the first representation, the values of the two subexpressions both denoted by `(cons nil nil)` are associated with heaps cells numbers 1 and 2 respectively. In the second representation, however, the values of the two expressions are *both* associated with heap cell number 1. These expressions may be said to *share* the cell. It is the first pair that will be returned by the DAG semantics, but the second would be a valid result in the sense that the two *cannot* be separated using destructors and tests in the language. We shall call such values *isomorphic*. This means that if an interpretation *did* return the second pair (but otherwise worked like the DAG semantics) it would not change program evalutation in an essential way.

7.2.1 Cons Memoization

This idea of sharing subterms is not at all unusual, it is what we intuitively expect to happen when we write

```
(let ((v (cons nil nil))) (cons v v))
```

in Lisp. Having a `let`-construction is very common as sharing may save both time and space in the evaluation. Time may be saved as the interpreter *avoids re-evaluation* of an expression that is used twice. Space may be saved because we don't create two cells on the heap that represent the same term. It is the programmer who guides the interpreter to these savings by specifying the possibility of sharing using `let`. However, some compilers and interpreters also try automatically extract sharing information from the source program to improve performance. Unlimp semantics may be seen as a way of obtaining maximal sharing on the heap automatically (that is, without user guidance).

So what is *maximal* sharing ? The principle in Uniqueness as a Leitmotiv for Implementation is this:

Every term represented on the heap should be represented *uniquely*.

That is, terms represented by two different cells on the heap should be separable using destructors and tests in the language. This means that we will use a minimal number of cells to represent the constructed values (short of garbage collection which is not supported by our semantics).

The Unlimp principle stated above will be implemented by searching the heap every time a `cons` operation is made. If the the `cons`ed value is isomorphic to an existing value on the heap, the existing value is returned. Otherwise a new heap cell is created. Formally we define this "cons memoizing" semantics as described in Section 7.5.3.

7.2.2 Call Memoization

The cons memoizing semantics addresses the issue of automatically saving space by identifying equivalent terms. However, it does *not* save time in the evaluation by avoiding recomputation of expressions that are used more than once.

In the F semantics there is an obvious way to try to accomplish such improvements: Memoizing function return values. Concentrating on this issue, we will go back to the DAG semantics and add the feature of return value memoization using the *isknown?*, *remember* and *recall* symbols. The resulting *memoizing call semantics* is defined formally in Section 7.5.4.

In this semantics, if `f` is called with some value more than once, the call will not be re-evaluated. Instead the return value is memoized and used again.

7.2.3 Combined Memoization

The memoizing call semantics does not avoid all the re-evaluation we would like it to. An example of the problem is the following: Given the inital heap we evaluate the expression

```
(cons (f (cons nil nil)) (f (cons nil nil)))
```

Sorts: *Value*,*Heap*,*Bool*. We implicitly assume to have equality in *Bool*.

Symbols:

nil:	$\cdot$	$\rightarrow$	*Value*
pair?:	*Value*	$\rightarrow$	*Bool*
hd:	*Value* $\times$ *Heap*	$\rightarrow$	*Value*
tl:	*Value* $\times$ *Heap*	$\rightarrow$	*Value*
cons:	*Value* $\times$ *Value* $\times$ *Heap*	$\rightarrow$	*Value* $\times$ *Heap*
initial:	$\cdot$	$\rightarrow$	*Heap*
true:	$\cdot$	$\rightarrow$	*Bool*
false:	$\cdot$	$\rightarrow$	*Bool*
isknown?:	*Value* $\times$ *Heap*	$\rightarrow$	*Bool*
recall:	*Value* $\times$ *Heap*	$\rightarrow$	*Value*
remember:	*Value* $\times$ *Value* $\times$ *Heap*	$\rightarrow$	*Heap*

Figure 7.4: Signature for the semantics of F-programs.

We would want the subexpression `(f (cons nil nil))` to be evaluated only once. But in the above interpretation, the two isomorphic subexpressions both denoted `(cons nil nil)` will not evaluate to the same value (although they will of course evaluate to isomorphic values) and hence the result of the first evaluation of `(f (cons nil nil))` will not be used to avoid re-evaluating the expression.

The immediate remedy to this shortcoming is to let a stored return value be indexed by an encoding of the structure of the call parameter rather than by the parameter itself, so that isomorphic values would share this information. Thus, for each call `f(`v`)` encountered during evaluation, we must transform the value v into a unique representation of its structure, i.e. a unique representation of all values isomorphic to v. But this is exactly what we got out of the Unlimp semantics ! So by combining the two kinds of memoization, we should be able to save both time and space. The resulting *combined memoizing semantics* is defined in Section 7.5.5.

7.3 Defining the Semantics of F

The signature for our semantics is given in Figure 7.4. An operational semantics for F programs using this signature is given in Figure 7.5.

There are two jugdement forms in the operational semantics. In evaluation of expressions, we write $E : v, H \Rightarrow v', H'$. Here, E is the expression to be evaluated and v is the value of `X`. The H is the heap with which the evaluation is performed, while H' is the new heap obtained from evaluating the expression. Value v' is the result of evaluation.

The other judgement form concerns the evaluation of programs. We denote by $P \Rightarrow v$ the fact that program P evaluates to value v.

Note that we implicitly include the body of `f` (i.e. the code of the program) in the definition of the semantics of an expression,

7.4 Measures

We will use four different measures on computations done by F-programs. The first two of these relate to our implementation of the semantics and different interpretations. We will elaborate on this implemetation in Section 8.1.

Expressions:

$$\frac{}{\texttt{nil} : v, H \Rightarrow \mathit{nil}, H}$$

$$\frac{}{\texttt{x} : v, H \Rightarrow v, H}$$

$$\frac{E : v, H \Rightarrow v', H'}{\texttt{hd}\, E : v, H \Rightarrow \mathit{hd}(v', H'), H'}$$

$$\frac{E : v, H \Rightarrow v', H'}{\texttt{tl}\, E : v, H \Rightarrow \mathit{tl}(v', H'), H'}$$

$$\frac{E_1 : v, H \Rightarrow v_1, H' \qquad E_2 : v, H' \Rightarrow v_2, H''}{\texttt{cons}\, E_1 E_2 : v, H \Rightarrow \mathit{cons}(v_1, v_2, H'')}$$

$$\frac{E_1 : v, H \Rightarrow v', H' \qquad E_2 : v, H' \Rightarrow v'', H''}{\texttt{if}\, E_1 E_2 E_3 : v, H \Rightarrow v'', H''} \; \mathit{pair?}(v') = \mathit{true}$$

$$\frac{E_1 : v, H \Rightarrow v', H' \qquad E_3 : v, H' \Rightarrow v'', H''}{\texttt{if}\, E_1 E_2 E_3 : v, H \Rightarrow v'', H''} \; \mathit{pair?}(v') = \mathit{false}$$

$$\frac{E : v, H \Rightarrow v', H' \qquad B : v', H' \Rightarrow v'', H''}{\texttt{f}\; E : v, H \Rightarrow v'', \mathit{remember}(v', v'', H'')} \; \mathit{isknown?}(v', H') = \mathit{false}$$

$$\frac{E : v, H \Rightarrow v', H'}{\texttt{f}\; E : v, H \Rightarrow \mathit{recall}(v', H'), H'} \; \mathit{isknown?}(v', H') = \mathit{true}$$

Programs:

$$\frac{B : \mathit{nil}, \mathit{initial} \Rightarrow v, H}{\texttt{f(X)} = B \Rightarrow v}$$

Figure 7.5: Operational semantics of F-programs.

Real time The actual time spent evaluating the program in our implementation running on a 233 MHz AMD K6.

Physical Heap Size The actual number of bytes allocated by our implementation. The allocated memory is used for representing the heap and for representing search structures used for implementing cons memoization.

Cons Misses This depends on the interpretation. Intuitively it is the number of values on the heap.

Call Misses The number of invocations of *remember* in the derivation tree of the computation.

7.5 Interpretations of the Semantics

The different interpretations will all agree on a few points. The sort *Bool* along with the symbols *true* and *false* always have the standard interpretation of boolean algebra.

Below, we will give the different interpretations of the remaining sorts and symbols.

7.5.1 Standard Semantics

In this interpretation, *Value* is the set $\mathbb{D}$ which was defined in [6]. This set is defined as the smallest set containing

- one atomic value `nil`, and
- the pairing $(d_1.d_2)$ of every two values $d_1, d_2 \in \mathbb{D}$.

We will have no need for explicit heaps, so *Heap* will be a one-point set, $\mathit{Heap} = \{\mathit{initial}\}$. Symbols *nil*, *hd*, *tl* and *cons* are the standard constructors and destructors; the heap argument is ignored in each case. Function *pair?* is false only on `nil`.

We do not perform any call memoization, and so *isknown?* is false on all input values, and $\mathit{remember}(d, d', \mathit{initial}) = \mathit{initial}$. Function *recall* may be defined to be constant *nil*.

7.5.2 DAG Semantics

In the DAG semantics, *Value* is the set $\mathbb{N}$, while *Heap* is the set $\mathbb{N}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}} \times \mathbb{N}$. For $H \in \mathit{Heap}$, we will write $H = (\mathit{car}, \mathit{cdr}, \mathit{next})$. The idea behind these definitions is as follows: The heap is to be thought of as an array $[0, \ldots, \mathit{next} - 1]$ of pairs of integers. The 0th entry of this array represents the value `nil`. For $i > 0$, let the ith entry of the array be $(j, k) \in \mathbb{N}^2$. This entry represents a value, v_i, such that `hd` v_i is represented by the jth entry of the array and `tl` v_i is represented by kth entry of the array. Function *car* stores the `hd` pointers, and function *cdr* stores the `tl` pointers.

The symbols are in the DAG semantics defined to be

$$\begin{aligned}
\mathit{nil} &= 0 \\
\mathit{pair?}(n) &= \begin{cases} \mathit{false} & \text{if } n = 0 \\ \mathit{true} & \text{if } n > 0 \end{cases} \\
\mathit{hd}(n, (\mathit{car}, \mathit{cdr}, \mathit{next})) &= \mathit{car}(n) \\
\mathit{tl}(n, (\mathit{car}, \mathit{cdr}, \mathit{next})) &= \mathit{cdr}(n)
\end{aligned}$$

$$\begin{aligned}
\mathit{cons}(n_1, n_2, (\mathit{car}, \mathit{cdr}, \mathit{next})) &= (\mathit{next}, (\mathit{car}[\mathit{next} \mapsto n_1], \mathit{cdr}[\mathit{next} \mapsto n_2], \mathit{next} + 1)) \\
\mathit{initial} &= (\lambda n.0, \lambda n.0, 1) \\
\mathit{isknown?}(n, H) &= \mathit{false} \\
\mathit{remember}(n_1, n_2, H) &= H \\
\mathit{recall}(n, H) &= 0
\end{aligned}$$

7.5.3 Memoizing Cons Semantics for F

The memoizing cons semantics differs from the DAG semantics only by the definiton of *cons*. Given n_1, n_2 and a heap $H = (\mathit{car}, \mathit{cdr}, \mathit{next})$ there are two cases. If there is no $n < \mathit{next}$ such that $\mathit{car}(n) = n_1$ and $\mathit{cdr}(n) = n_2$, then the result of *cons* will be

$$(\mathit{next}, (\mathit{car}[\mathit{next} \mapsto n_1], \mathit{cdr}[\mathit{next} \mapsto n_2], \mathit{next} + 1))$$

as in the DAG semantics. However, if there *is* such an n, then *cons* returns

$$(N, H)$$

with N being the least n satisfying the above condition. We may prove that if a heap occurs during the evaluation of a program, then there will be at most one n satisfying the condition.

7.5.4 Memoizing Call Semantics for F

Again, *Value* is the set $\mathbb{N}$. Now, *Heap* is the set $\mathbb{N}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \times (\mathbb{N} \cup \{\mathit{undef}\})^{\mathbb{N}}$. We use the notation $H = (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})$ for heaps.

We define new interpretation by

$$\begin{aligned}
\mathit{nil} &= 0 \\
\mathit{pair?}(n) &= \begin{cases} \mathit{false} & \text{if } n = 0 \\ \mathit{true} & \text{if } n > 0 \end{cases} \\
\mathit{hd}(n, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= \mathit{car}(n) \\
\mathit{tl}(n, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= \mathit{cdr}(n) \\
\mathit{cons}(n_1, n_2, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= (\mathit{next}, (\mathit{car}[\mathit{next} \mapsto n_1], \mathit{cdr}[\mathit{next} \mapsto n_2], \mathit{next} + 1, \mathit{mem})) \\
\mathit{initial} &= (\lambda n.0, \lambda n.0, 1, \lambda n.\mathit{undef}) \\
\mathit{isknown?}(n, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= \begin{cases} \mathit{true} & \text{if } \mathit{mem}(n) \neq \mathit{undef} \\ \mathit{false} & \text{if } \mathit{mem}(n) = \mathit{undef} \end{cases} \\
\mathit{remember}(n_1, n_2, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem}[n_1 \mapsto n_2]) \\
\mathit{recall}(n, (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})) &= \begin{cases} \mathit{mem}(n) & \text{if } \mathit{mem}(n) \neq \mathit{undef} \\ 0 & \text{if } \mathit{mem}(n) = \mathit{undef} \end{cases}
\end{aligned}$$

7.5.5 Combined Memoizing Semantics

This semantics is derived form the memoizing call semantics in the same way the memoizing cons semantics was derived from the DAG semantics.

Again, all we need to change is the definition of *cons*. Given n_1, n_2 and a heap $H = (\mathit{car}, \mathit{cdr}, \mathit{next}, \mathit{mem})$ there are two cases. If there is no $n < \mathit{next}$ such that $\mathit{car}(n) = n_1$ and $\mathit{cdr}(n) = n_2$, then the result of *cons* will be

$$(\mathit{next}, (\mathit{car}[\mathit{next} \mapsto n_1], \mathit{cdr}[\mathit{next} \mapsto n_2], \mathit{next} + 1), \mathit{mem})$$

as in the memoizing call semantics. However, if there *is* such an n, then *cons* returns

$$(N, H)$$

with N being the least n satisfying the above condition. Again, we may prove that if a heap occurs during the evaluation of a program, then there will be at most one n satisfying the condition.

7.6 Some Background

Disadvantages of Unlimp What has not been discussed above are the hidden costs of Unlimp. Searching the heap for isomorphic values requires time (for searching) and space (for search structures), as does the memoization of function calls. These topics will be investigated experimentally in Chapter 8.

History The ideas behind Unlimp were introduced by Stefan Kahrs in [7]. He considers a very general setting which models both higher-order functions and lazy evaluation. We have stuck to a strict, first-order language to keep things as simple as possible and to try to present the basic concepts in a clearer manner.

Motivation Our attention was brought to Unlimp in connection to practical experiments with an implementation of Levin's Algorithm (as described in Chapter 3), see [4].

One problem with this implementation was the vast amount of space used by the code for enumerating programs efficiently. Observing that the enumeration algorithm would start with the smallest programs and then enumerate larger and larger programs (as is natural), it seemed a good idea to consider Unlimp for the implementation. Having learnt about these concepts, we found the Unlimp was worth a study in its own right.

Chapter 8

Unlimp Experiments

In this chapter we sum up a line of experiments and discuss their results. The aim of these experiments is to gain some insight into the advantages and disadvantages of Unlimp interpretation of programs (as presented in 7). After sketching our implementation, we first compare two versions of this. We then evaluate first cons memoization in itself then combined cons and function call memoization. Finally we look at Unlimp for some enumeration problems as was our initial motivation for investigating the idea.

8.1 Implementation

The different interpretations of the semantics of the F language that were described in Section 7.5 have been implemented in C. Below follows a sketch of the implementation. The experiments were performed on a 233 MHz *AMD* K6 running the *Linux* operating system. The programs were compiled using GNU's *gcc* compiler with no special options invoked.

8.1.1 Overall Implementation of Semantics

Depending on the user's choice of interpretation of our semantics, we define

- types implementing the sorts of the semantics, and
- functions implementing the symbols of the semantics.

A parser for F expressions (constructed using the *Bison* lexer and the *Flex* parser generator) puts program into a simple, linked structure. We have one function (called `eval`) for evaluating expressions. This function takes as parameters

- an expression E,
- a value v, and
- a heap H.

The body of function `f` (which is needed for evaluating expressions of the form $E = (\texttt{f}\ E'))$ is stored in a global variable. Function `eval` determines which kind of expression, E is, then applies the according rule of the semantics in Figure 7.5. This may involve using some of the types and functions that were defined for the sorts and symbols of the semantics. If E is neither `nil` or `X` then `eval` will call itself recursively.

When our implementation is executed, the `eval` function is called with the body of function `f`, the value *nil* and the heap *initial*. This call is performed 10 times in a timed loop to get a good measure of the running time of the evaluation. The system timing is in hundreths of a second, and we report the total time of the 10 calls as time in milliseconds. A number of counters keep track of the three other measures defined in Section 7.4. These observed variables are output in a form suitable for handling by automatic scripts.

8.1.2 DAG Heaps and Call Memoization

The heap that the interpretations (except the standard semantics) all share in some form is implemented using an array of structs (heap cells), each containing the value *car* and *cdr* for the particular index of the array. This array is expanded (more memory is allocated for it) if more heap cells is needed than originally assumed. The memoization table for function calls is implemented by adding an extra entry to the heap cell struct. This entry contains for index i the value of function `f` applied to value i if this is known (otherwise it contains -1).

8.1.3 Memoization for Uniqueness

The memoization of `cons` – i.e. the searching for isomorphic values on the heap – is done using an abstract search structure. When two values are to be `cons`ed, they are paired using Polya's "efficient" pairing function (this is a bijection $pair : \mathbb{N}^2 \to \mathbb{N}$ which is polynomially bounded in both arguments, see [6]). This value is then looked up in the search structure. If found, the associated value is returned as the result of `cons`; otherwise the next empty cell on the heap is the result. This result will then be registred in the search structure.

We have implemented two different kinds of search structures: one is a 16-ary search trie, the other is standard double hashing combined with digital search trees (this is very much like Stefan Kahrs's implementation). The combination of hashing and search trees works as follows: We have a binary tree with a hash table at every node. When looking up value, we check if it is in the hash table of the root node. If not, we choose the left or the right subtree depending on the first bit of the value, and (recursively) look up the value in this tree. The tree of hashtables is expanded when a table at one of the leafs has become too full (a condition that has been defined pragmatically).

There may very well be other search structures more suitable for our needs, but we have not been able to come up with any such structures.

8.1.4 Test Programs

All our test programs are written in F. To make programs easier to read, we have used macros to be expanded by the preprocessor for C. The programs can be seen in Appendix A.

Even though F programs in general do not have explicit input, for our test programs we define "the input" to be the first non-`nil` argument that `f` is called with during the evaluation of a program. In all of our programs this value will be very explicit from the program text so we will only present one instance of each algorithm (and not one for each input value we shall use). In the graphs documenting our experiments, numbers on the x-axis will indicate the length of the input as a list.

The programs will all share some macros defining the natural numbers (in unary). The macro `ZERO` is defined to be `nil`, `ONE` is defined as `(cons nil ZERO)`, `TWO` is defined as `(cons nil ONE)` and so on. Likewise, `FALSE` is the same as `ZERO` and `TRUE` is the same as `ONE`. Finally, `TAG` is always defined to be `(hd X)`. These macro definitions have been left out in the program listings.

The simple, recursive definitions of binomial coefficients (*Binom* in Figure A.1) and Fibonacci numbers (*Fib* in Figure A.2) are well-known examples of algorithms that may be sped up by the use of memoization, and we use them to illustrate the possible benefits of Unlimp.

The recursive definitions of product as iterated sum (*Prod* in Figure A.3) and exponentiation as iterated product (*Expo* in Figure A.4) are also well-known algorithms. We use these as examples of programs that will most likely *not* benefit from memoization.

As enumeration for Levin's Algorithm was our initial motivation for investigating Unlimp, we also look at two algorithms of this kind. The first one (*Sl* in Figure A.5) returns for a given list the list of all sublists of this. Our second algorithm enumerates binary trees (*Enum* in Figures A.6, A.6 and A.6), and is based on an algorithm from [4] that enumerated all expressions produced by a context-free grammar in constant time per expression. Our present algorithm works in amortized constant time only (as we have not implemented an efficient simulation of queues).

8.2 Comparing Different Search Structures

Please regard Appendix B.

We have compared the two different search structures by using each for running the *Binom* and *Fib* programs with the cons memoizing semantics.

From the results it is quite clear that the trie structure performs far worse than the hashing scheme, with regard to the space usage as well as with regard to running time. A closer look at the performance (not documented here) revealed that the problem with the trie is that it is much too sparse to be effective.

Based on this experience, we have only used the hashing scheme in the rest of the experiments.

8.3 Cons Memoization

Please regard Appendix C.

We have compared the DAG semantics to the cons memoizing semantics by running the *Binom*, *Fib*, *Prod* and *Expo* programs with each kind of semantics.

For *Binom* we see a substantial, superlinear reduction in the amount of space used and in the number of cons misses. For the *Fib* program we see a somewhat less substantial but still clear (and superlinear) reduction in the use of these resources. That the number of cons misses is substatially reduced was to be expected as it is well-known that both algorithms perform many redundant evaluations. But it is worth noting that the size of the search structure (which is included in the physical heap size) does *not* change the conclusion that we are saving a lot of space by memoizing conses.

The *Prod* and *Expo* programs were not expected to respond well to cons memoization, as they do not perform many repeated expression evaluations. This expectation was confirmed by the experiments: In both cases, when we go from DAG semantics to cons memoizing semantics the amount cons misses is reduced by a very small constant factor and the physical heap size grows superlinearly. With *Expo* we were at first surprised by the fairly small cost of memoizing conses. It was only after some consideration we realised that a significant number of products are calculated more than once in the algorithm. This gives some benefit from memoizing conses that lessens the costs of this approach.

The last graph reveals that going from DAG semantics to cons memoizing semantics slows down the interpretation by more than a constant factor. This is not terribly surprising as much time will be spent searching the heap.

8.4 Call Memoization

Please regard Appendix D.

We have compared the DAG semantics to the combined memoizing semantics by running the *Binom*, *Fib*, *Prod* and *Expo* programs with each kind of semantics. We also ran all of these test programs with the call memoizing semantics, but in all cases no memoized function calls could be utilized due to the problems discussed in the beginning of Section 7.2.3.

In the cases of *Binom* and *Fib* we see that there really is something to be gained by memoizing function calls: The number of call misses is in each case reduced substantially. This is not too surprising as these two programs are standard examples of algorithms that benefit dramatically from the application of memoization.

It is, however, interesting to note that the time spent searching does *not* seem to be a big problem: In both cases the reduction in real running time is also substantial.

In the *Prod* algorithm, the DAG semantics is quicker because there simply are *no* redundant function calls. The slowdown is clear and more than linear (although not as drastic as the speedup in the above cases).

For the *Expo* algorithm, the slowdown is fairly small. As mentioned above this came as a surprise – it was only when reasoning about the results we saw that there are a number of redundant function calls in the computations.

8.5 Unlimp for Enumerations

Please regard Appendix E.

We have compared the DAG semantics to the cons memoizing semantics by running the enumeration algorithms *Sl* and *Enum* with each kind of semantics.

For the *Sl* program, the results are not promising. A small amount of cons misses are avoided by memoization, but the physical heap size is much smaller with the DAG semantics.

This results for *Enum* are more interesting: Memoization reduces the number of cons misses to just above a third of the number with the DAG semantics. At the same time, the space needed for the search structure means that the physical heap size is about a factor of 5 larger for the cons memoizing semantics than for the DAG semantics. But there is a very notable anomaly in the physical heap size of the cons memoizing interpretation: The factor of 5 mentioned above drops to a factor of just above 2 for input $n = 18$.

Some investigations revealed this to be a chance “interference” between the hashing strategy and the concrete computation. Some further investigations confirmed that the size of the search structure is very unstable for this algorithm, and that changes in the hashing strategy could lead to substantially better performance. We have not had succes in developing a hashing strategy for which the physical heap size of the cons memoizing semantics was actually smaller than that of the DAG semantics, but we conjecture that this is possible.

Chapter 9

Conclusion and Future Work

In this part of the thesis we have worked with the Uniqueness as a Leitmotiv for Implementation priciple of interpretation. In Chapter 7 we presented the ideas both informally in a general discussion and formally as different interpretations of the semantics of a very simple, functional language.

In Chapter 8 we then moved on to perform a line of experiments with an implementation of the semantics of the previous chapter. The experiments showed that the advanced interpretation of the semantics performed very differently on different test programs. On benevolent test programs the "Unlimp" idea was extremely beneficial, but on other programs one was clearly better off with standard interpretation. For the particular application of enumerations the results were not decisive – more work with the details of implementing "Unlimp" would be needed to reach a final conclusion on this point.

We feel that the "Unlimp" idea, while not likely to be of much use in ordinary programs, may turn out to be useful when specialised (and fine-tuned) to particular applications (like enumeration) that are very space-demanding and for which it is hard for the programmer to reason about heap usage and possibilities of sharing. Investigations of this could (and probably should) be very practically oriented and involve detailed study of the heap usage of programs performing non-trivial, space-demanding tasks.

Appendix A

Test Programs

```
/* We have tags for three functions: plus, equal and binom */
#define PLUS  ZERO
#define EQUAL ONE
#define BINOM TWO

/* Which binmoial coefficient to compute: */
#define INPUT (cons EIGHT FOUR)

/* f(X) =  */
(if X
    (if TAG
        (if (tl TAG)
/* Implementation of (binom n k): */
#define N (hd (tl X))
#define K (tl (tl X))
            (if K
                (if (f (cons EQUAL (cons N K)))
                    ONE
                    (f (cons PLUS (cons
                      (f (cons BINOM (cons
                        (tl N)
                        (tl K)
                      )  )           )
                      (f (cons BINOM (cons
                        (tl N)
                        K
                      )  )           )
                    )  )         )
                )
                ONE
            )
/* Implementation of (equal n k) */
            (if N
                (if K
                    (f (cons EQUAL (cons (tl N) (tl K))))
                    FALSE
                )
                (if K FALSE TRUE)
            )
        )
/* Implementation of (plus n k): */
        (if N
            (cons nil (f (cons PLUS (cons (tl N) K))))
            K
        )
    )
    (f (cons BINOM INPUT))
)
```

Figure A.1: Binom: Computing n over $\frac{n}{2}$ (here $n = 8$).

```
/* We have tags for two functions: plus and fib */
#define PLUS  ZERO
#define FIB   ONE

/* Which Fibonacci number to compute: */
#define INPUT EIGHTEEN

/* f(X) =  */
(if X
    (if TAG
/* Implementation of (fib y): */
#define Y (tl X)
        (if Y
            (if (tl Y)
                (f (cons PLUS (cons
                  (f (cons FIB (tl Y)))
                  (f (cons FIB (tl (tl Y))))
                )  )          )
                ONE
            )
            ONE
        )
/* Implementation of (plus n k): */
#define N (hd (tl X))
#define K (tl (tl X))
        (if N
            (cons nil (f (cons PLUS (cons (tl N) K))))
            K
        )
    )
/* Call to evaluate: */
    (f (cons FIB INPUT))
)
```

Figure A.2: Fib: Computing the nth Fibonacci number (here $n = 18$).

```
/* We have tags for two functions: plus and prod */
#define PLUS  ZERO
#define PROD ONE

/* Which number to compute the square of: */
#define INPUT (cons TWENTYTWO TWENTYTWO)

/* f(X) =  */
(if X
    (if TAG
/* Implementation of (* n k): */
#define N (hd (tl X))
#define K (tl (tl X))
        (if K
            (f (cons PLUS (cons N (f (cons PROD (cons N (tl K))))))
            )
            ZERO
        )
/* Implementation of (plus n k): */
        (if N
            (cons nil (f (cons PLUS (cons (tl N) K))))
            K
        )
    )
    (f (cons PROD INPUT))
)
```

Figure A.3: Prod: Computing n^2 (here $n = 22$).

```
/* We have tags for three functions: plus, prod and expo */
#define PLUS  ZERO
#define PROD ONE
#define EXPO TWO

/* Which numbers to compute the exponent of: */
#define INPUT (cons TWO FIFTEEN)

/* f(X) =  */
(if X
    (if TAG
        (if (tl TAG)
/* Implementation of n^k : */
            (if K
                (f (cons PROD (cons N (f (cons EXPO (cons N (tl K))))))
                )
                ONE
            )
/* Implementation of (* n k): */
            (if K
                (f (cons PLUS (cons N (f (cons PROD (cons N (tl K))))))
                )
                ZERO
            )
        )
/* Implementation of (plus n k): */
        (if N
            (cons nil (f (cons PLUS (cons (tl N) K))))
            K
        )
    )
    (f (cons EXPO INPUT))
)
```

Figure A.4: Expo: Computing 2^n (here $n = 15$).

```
/* We have tags for two functions: consider and sublists */
#define CONSIDER ZERO
#define SUBLISTS ONE

/* Which list to compute the lists of all sublists of: */
#define INPUT FOURTEEN

/* f(X) =  */
(if X
    (if TAG
/* Implementation of (sublists list): */
#define LIST (tl X)
        (if LIST
            (f (cons CONSIDER (cons (hd LIST)
                                    (f (cons SUBLISTS (tl LIST)))
               )              )
            )
            (cons nil nil)
        )
/* Implementation of (consider elem slxs): */
#define ELEM (hd (tl X))
#define SLXS (tl (tl X))
        (if SLXS
            (cons (hd SLXS)
            (cons (cons ELEM (hd SLXS))
                  (f (cons CONSIDER (cons ELEM (tl SLXS))))
            )
            )
            nil
        )
    )
    (f (cons SUBLISTS INPUT))
)
```

Figure A.5: Sl: Computing the lists of all sublists of L (here $L = \texttt{nil}^{14}$).

```
#define ARGS (tl X)
#define GENBIN FOUR
#define NEXTSTATE THREE
#define RETRIEVE TWO
#define RECREV ONE
#define GETME ZERO
#define INPUT FOURTEEN
#define TOCOME (hd ARGS)
#define BUFFER (hd (tl ARGS))
#define NEXTOLD (hd (tl (tl ARGS)))
#define OLDS (hd (tl (tl (tl ARGS))))
#define NEW (hd (tl (tl (tl (tl ARGS)))))
#define STATUS (hd (tl (tl (tl (tl (tl ARGS))))))
#define LASTGEN (hd TOCOME)
#define INITOC nil
#define INIBUF nil
#define ININEX (cons ZERO nil)
#define INIOLD (cons ZERO nil)
#define ININEW (cons ZERO ZERO)
#define INISTA nil
#define HALFSTATE (cons INIOLD (cons ININEW (cons INISTA nil)))
#define INITIALSTATE (cons INITOC (cons INIBUF (cons ININEX HALFSTATE)))
#define NEWVALS (f (cons RETRIEVE (cons TOCOME BUFFER)))
#define NEWOLDS (cons NEW OLDS)
#define RTTOC (hd ARGS)
#define RTBUF (tl ARGS)
#define RRLIS (hd ARGS)
#define RRSOF (tl ARGS)

/* How many expression to enumerate: */
#define INPUT TWENTY
```

Figure A.6: Definitions for Enum: Computing the nth binary tree in an enumeration of these (here $n = 20$).

```
/* f(X) =  */
(if X
(if TAG
(if (tl TAG)
(if (tl (tl TAG))
(if (tl (tl (tl TAG)))
 /* genbin: */
(cons (hd (hd (f (cons NEXTSTATE ARGS))))
      (f (cons NEXTSTATE ARGS))
)
 /* nextstate: */
(if NEXTOLD
  (if STATUS
    (cons (cons (cons (tl NEXTOLD) NEW) TOCOME)
    (cons BUFFER
    (cons (tl NEXTOLD)
    (cons OLDS
    (cons NEW
    (cons FALSE
    nil))))))
    (cons (cons (cons NEW (hd NEXTOLD)) TOCOME)
    (cons BUFFER
    (cons NEXTOLD
    (cons OLDS
    (cons NEW
    (cons TRUE
    nil))))))
  )
  (cons (cons (cons NEW NEW) (hd NEWVALS))
  (cons (hd (tl NEWVALS))
  (cons NEWOLDS
  (cons NEWOLDS
  (cons (hd (tl (tl NEWVALS)))
  (cons FALSE
  nil))))))
))
```

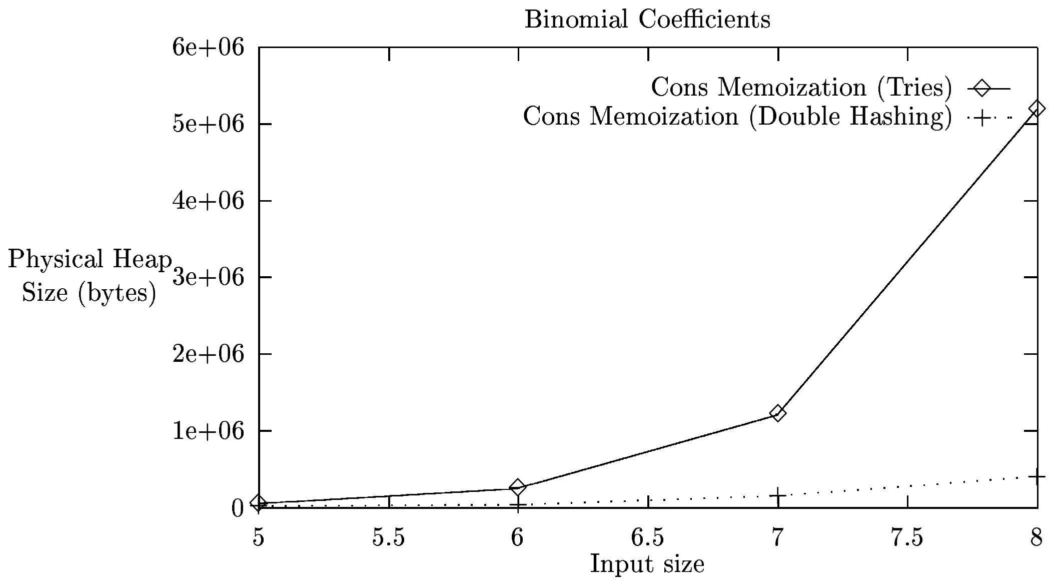
Figure A.7: Enum (continued).

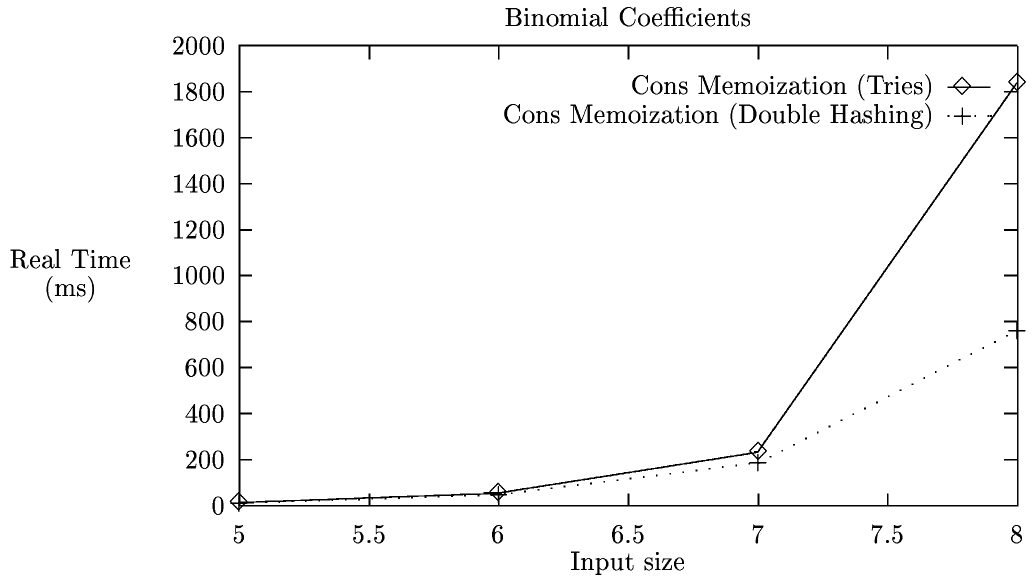
```
 /* retrieve: */
(if RTBUF
    (cons RTTOC (cons (tl RTBUF) (cons (hd RTBUF) nil)))
    (f (cons RETRIEVE (cons nil (f (cons RECREV (cons RTTOC nil))))))
))
 /* recrev: */
(if RRLIS
    (f (cons RECREV (cons (tl RRLIS) (cons (hd RRLIS) RRSOF))))
    RRSOF
))
 /* getme: */
(if ARGS
    (f (cons NEXTSTATE (f (cons GETME (tl ARGS)))))
    INITIALSTATE
))
 /* Run program: */
    (hd (hd (f (cons GETME INPUT))))
)
```

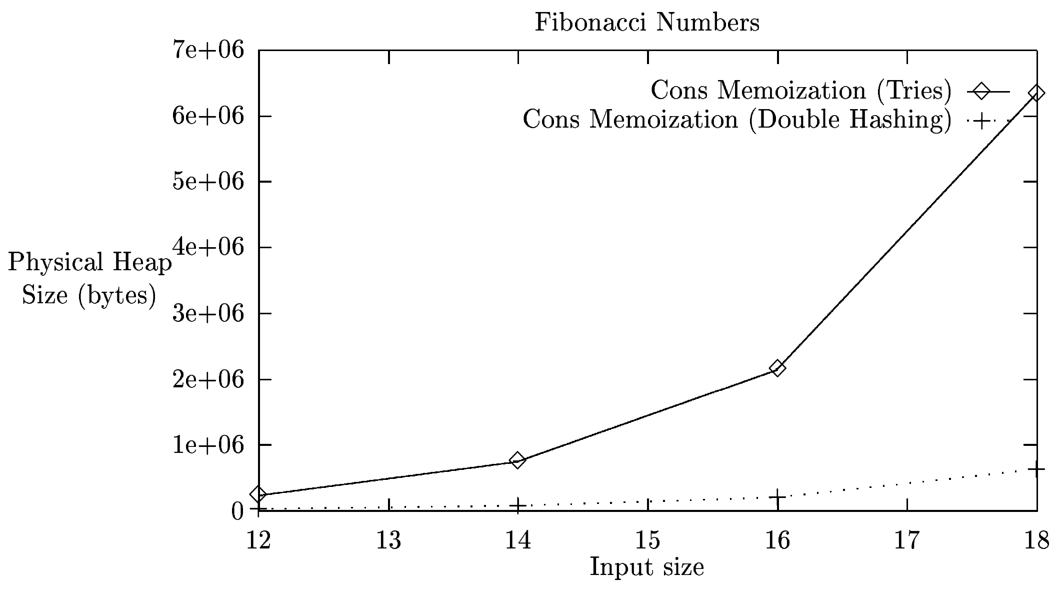
Figure A.8: Enum (continued).

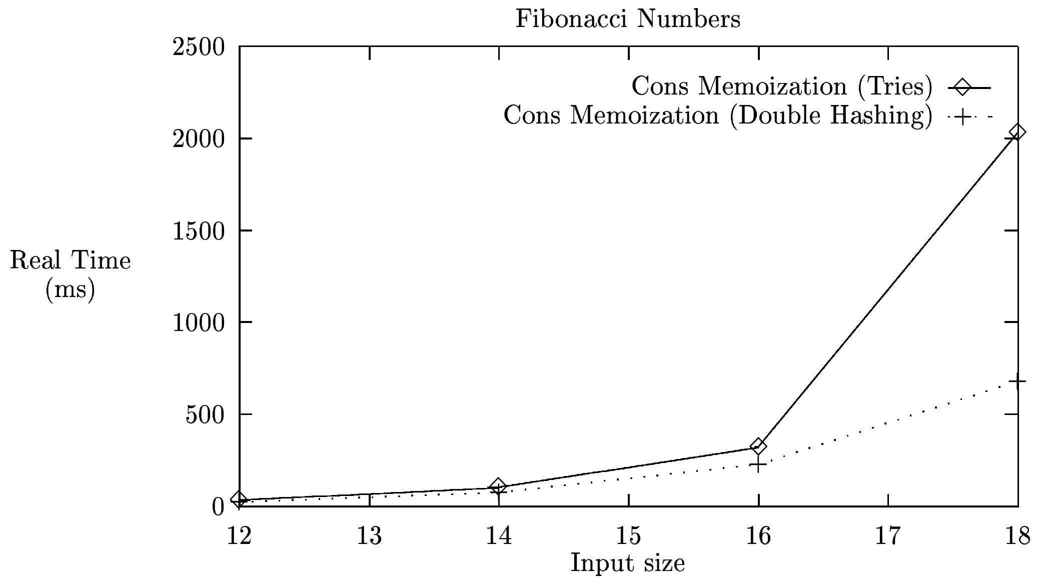
Appendix B

Unlimp Experiments: Tries vs. Double Hashing



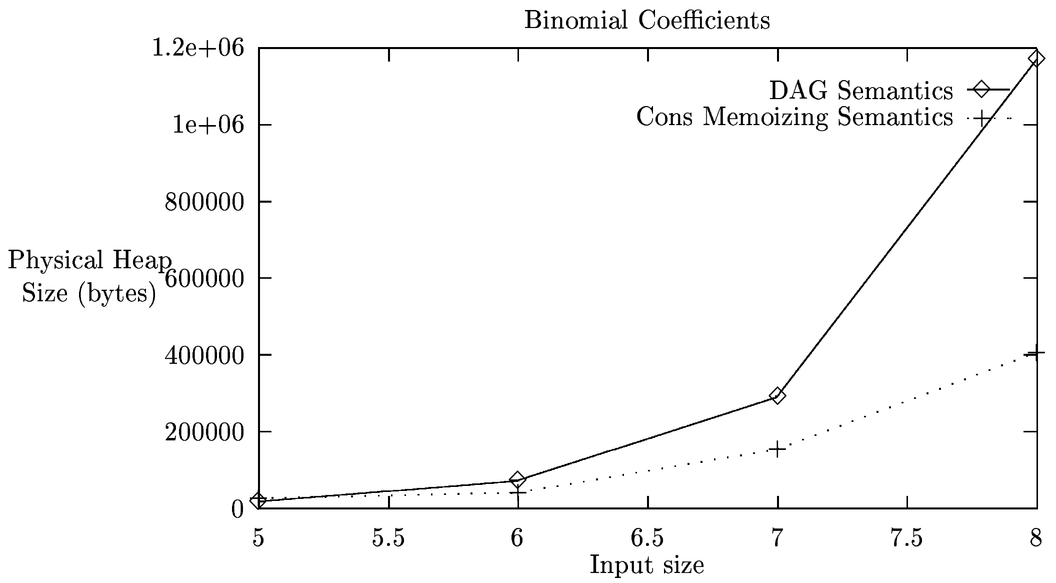


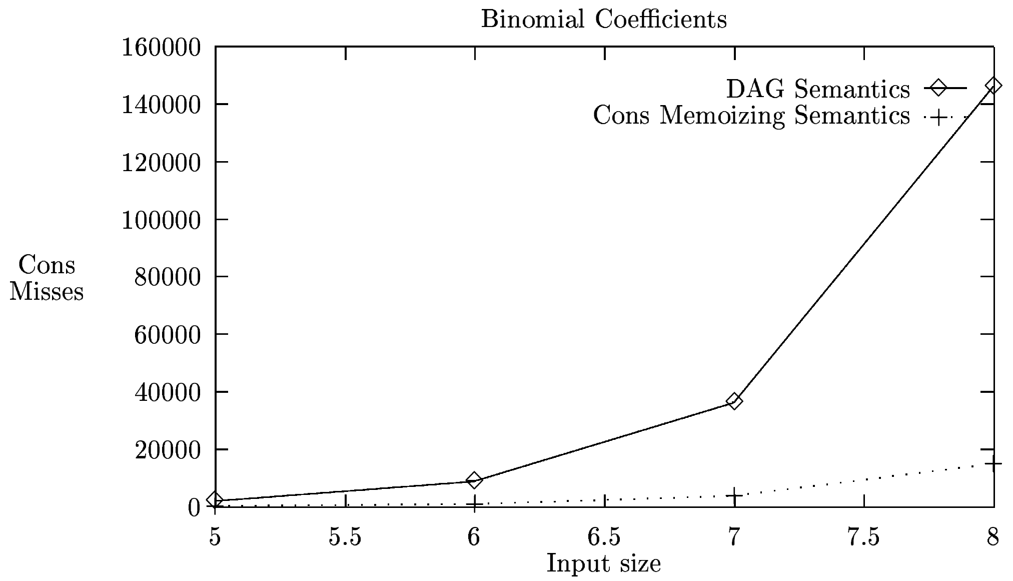

Fibonacci Numbers
Cons Memoization (Tries)
Cons Memoization (Double Hashing)
Physical Heap Size (bytes)
Input size
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
12
13
14
15
16
17
18

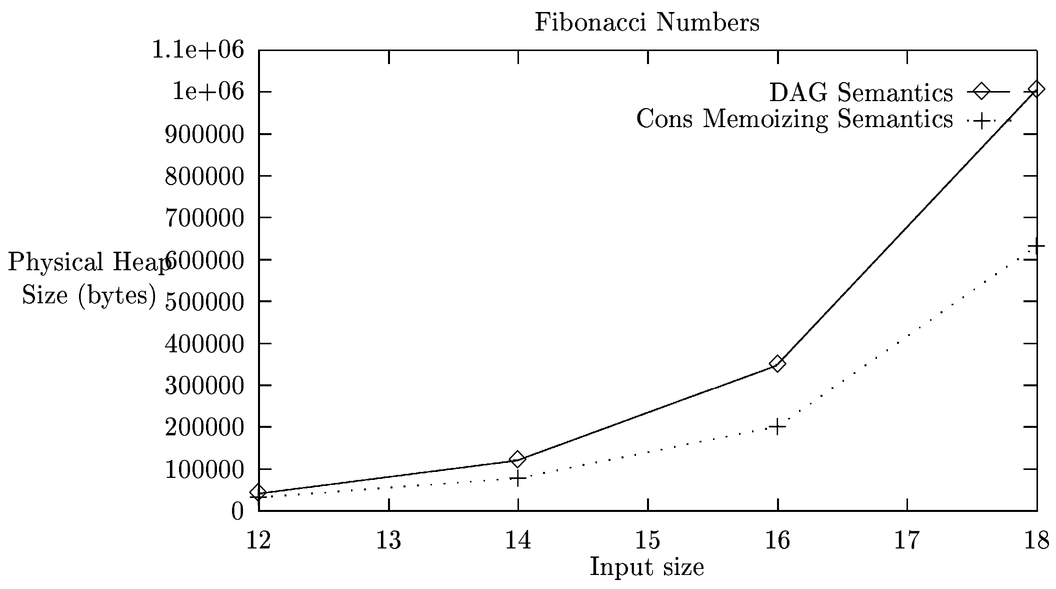

Fibonacci Numbers
Cons Memoization (Tries)
Cons Memoization (Double Hashing)
Real Time (ms)
Input size
2500
2000
1500
1000
500
0
12
13
14
15
16
17
18

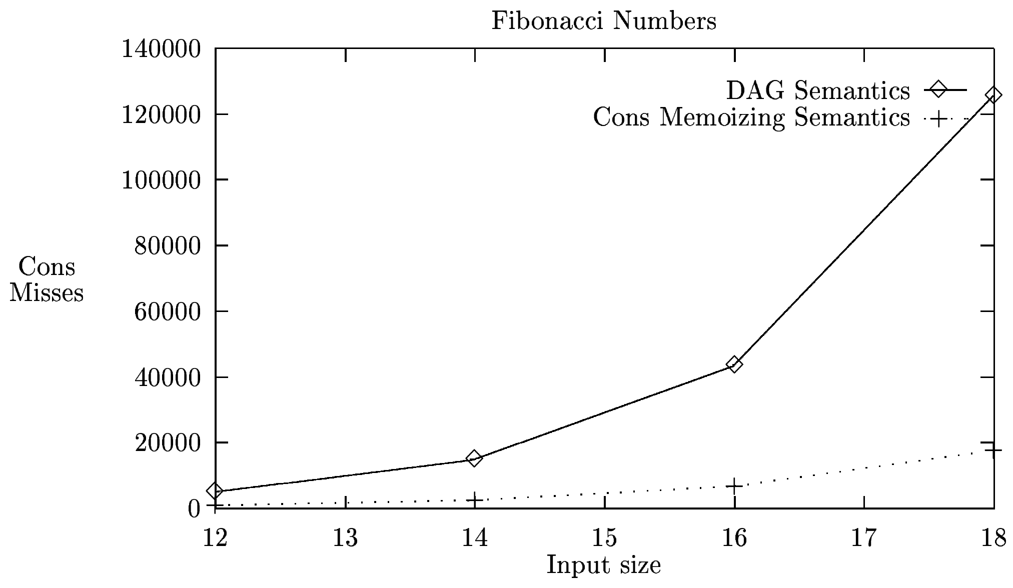
Appendix C

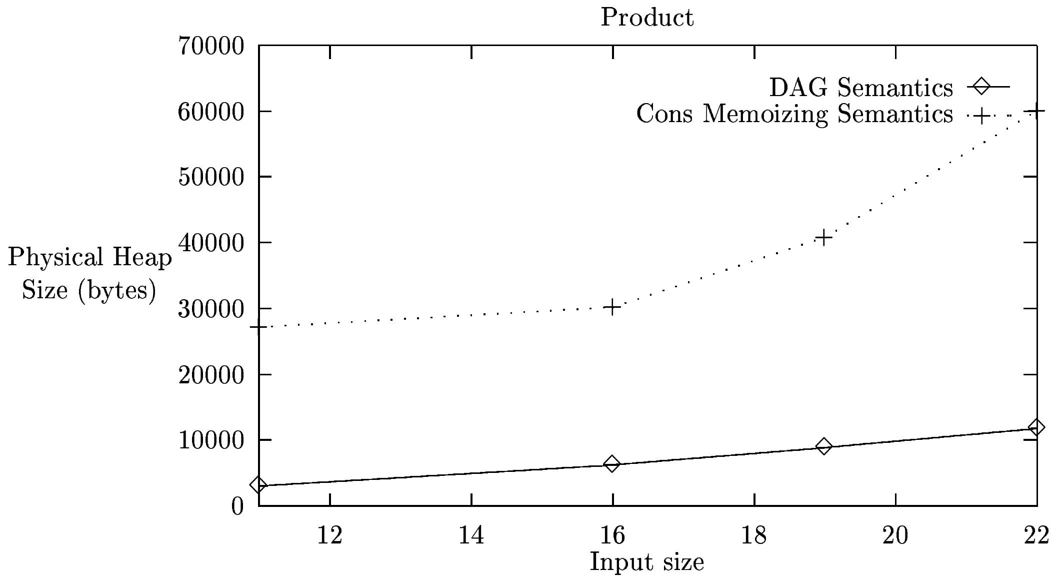
Unlimp Experiments: Cons Memoization

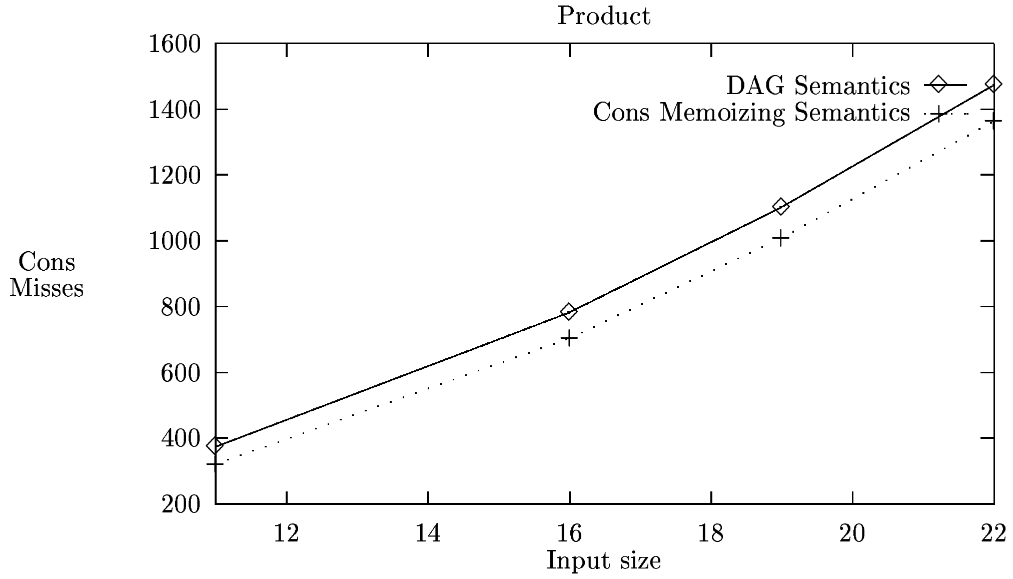


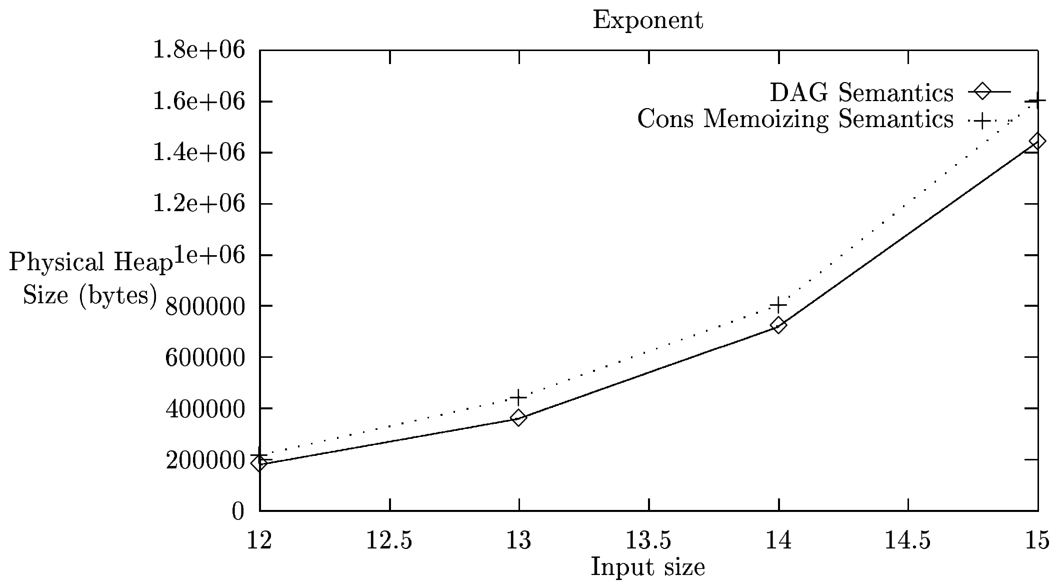


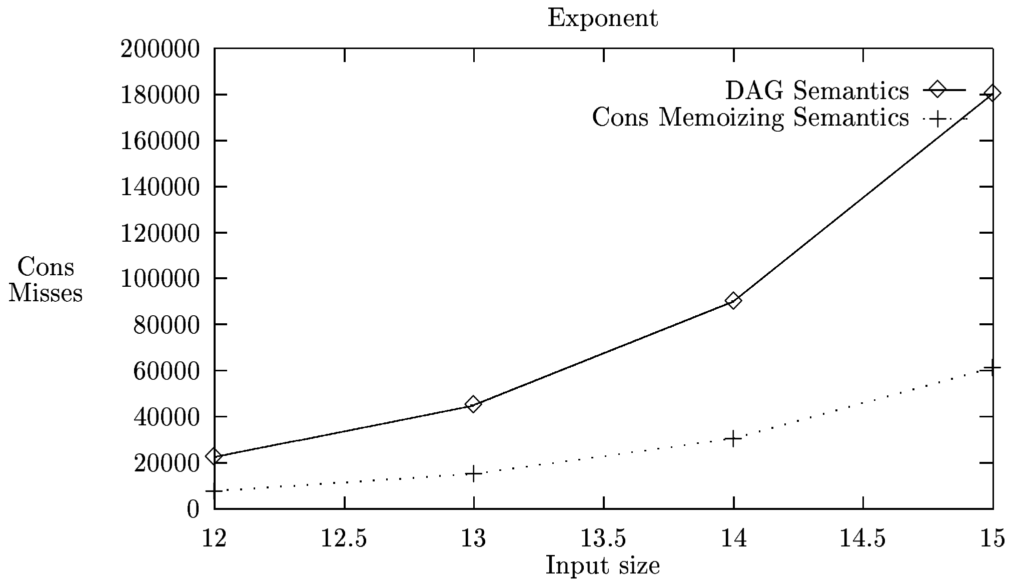

Fibonacci Numbers
1.1e+06
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Physical Heap Size (bytes)
DAG Semantics
Cons Memoizing Semantics
12
13
14
15
16
17
18
Input size

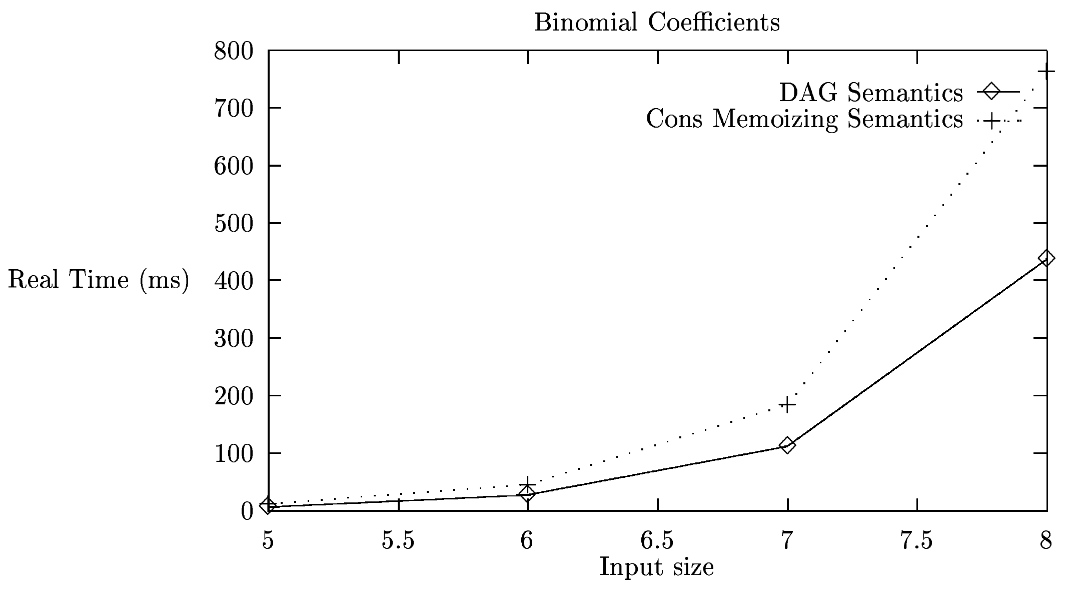

Fibonacci Numbers
140000
120000
100000
80000
60000
40000
20000
0
Cons Misses
DAG Semantics
Cons Memoizing Semantics
12
13
14
15
16
17
18
Input size


Product
DAG Semantics
Cons Memoizing Semantics
Physical Heap Size (bytes)
70000
60000
50000
40000
30000
20000
10000
0
12
14
16
18
20
22
Input size


Product
DAG Semantics
Cons Memoizing Semantics
Cons Misses
1600
1400
1200
1000
800
600
400
200
12
14
16
18
20
22
Input size

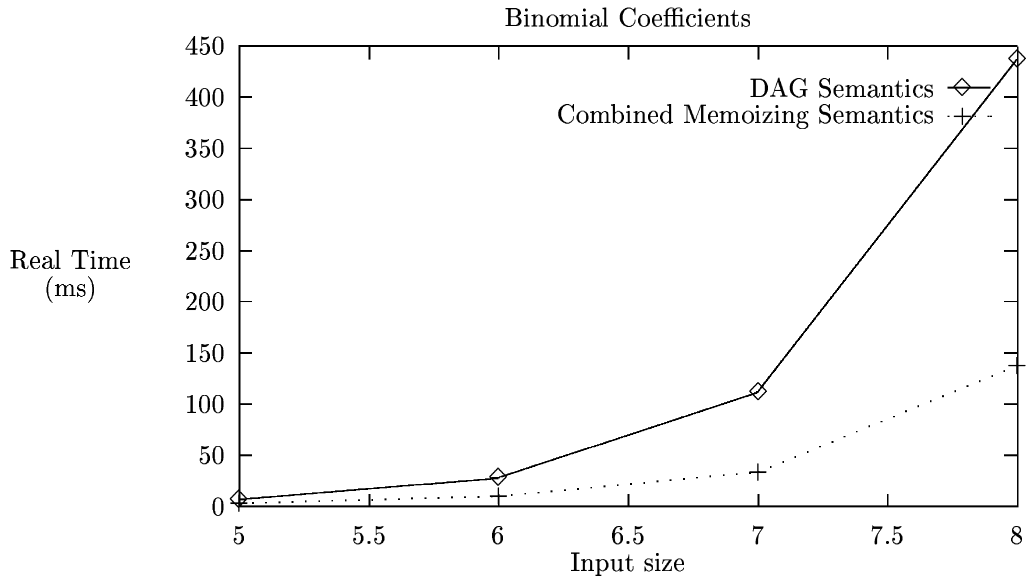

Exponent
DAG Semantics
Cons Memoizing Semantics
Physical Heap Size (bytes)
0
200000
400000
600000
800000
1e+06
1.2e+06
1.4e+06
1.6e+06
1.8e+06
12
12.5
13
13.5
14
14.5
15
Input size

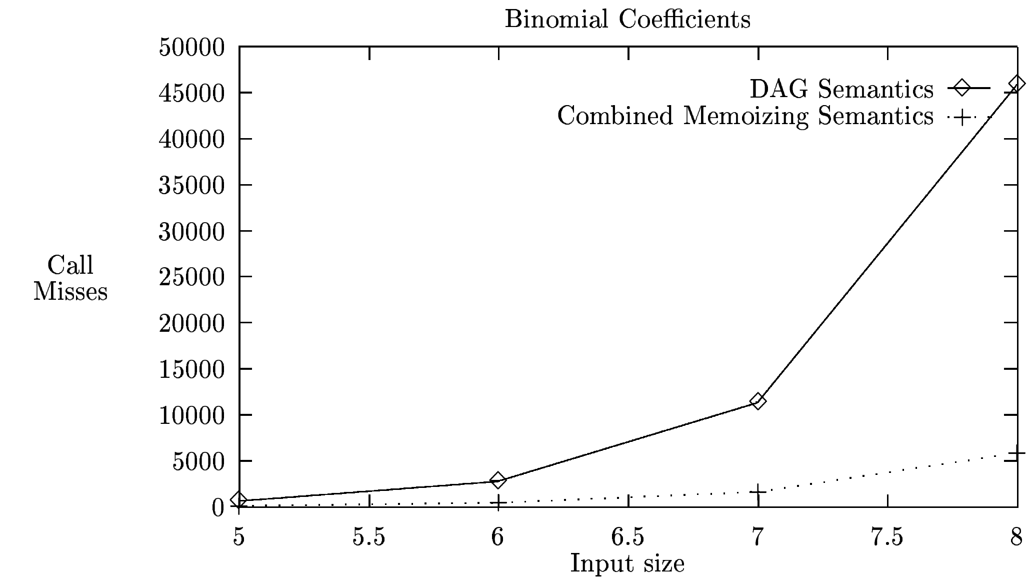

Exponent
DAG Semantics
Cons Memoizing Semantics
Cons Misses
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
12
12.5
13
13.5
14
14.5
15
Input size

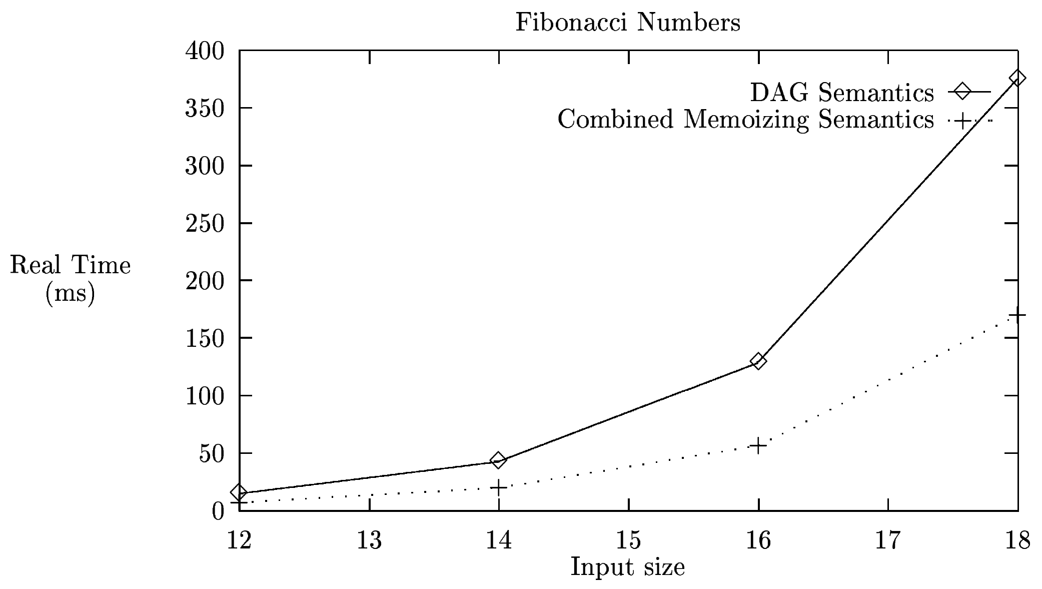

Binomial Coefficients
DAG Semantics
Cons Memoizing Semantics
Real Time (ms)
Input size
0
100
200
300
400
500
600
700
800
5
5.5
6
6.5
7
7.5
8

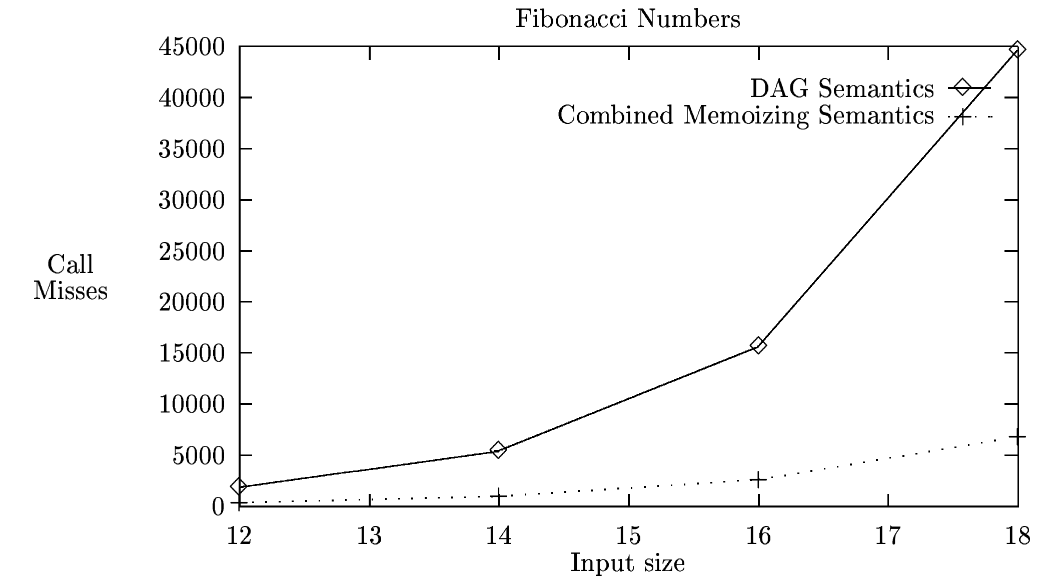
Appendix D

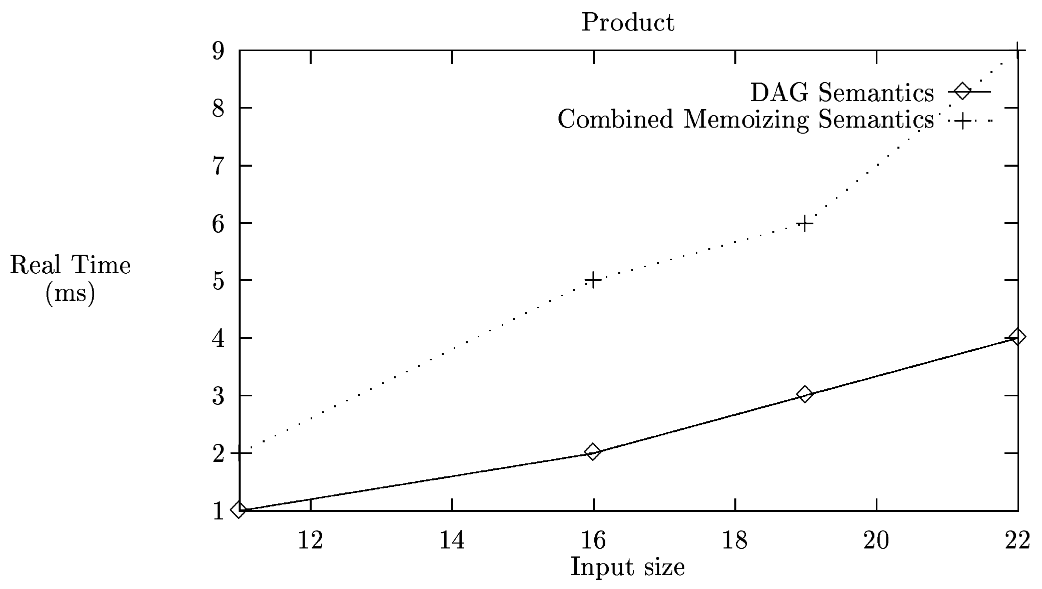
Unlimp Experiments: Call Memoization

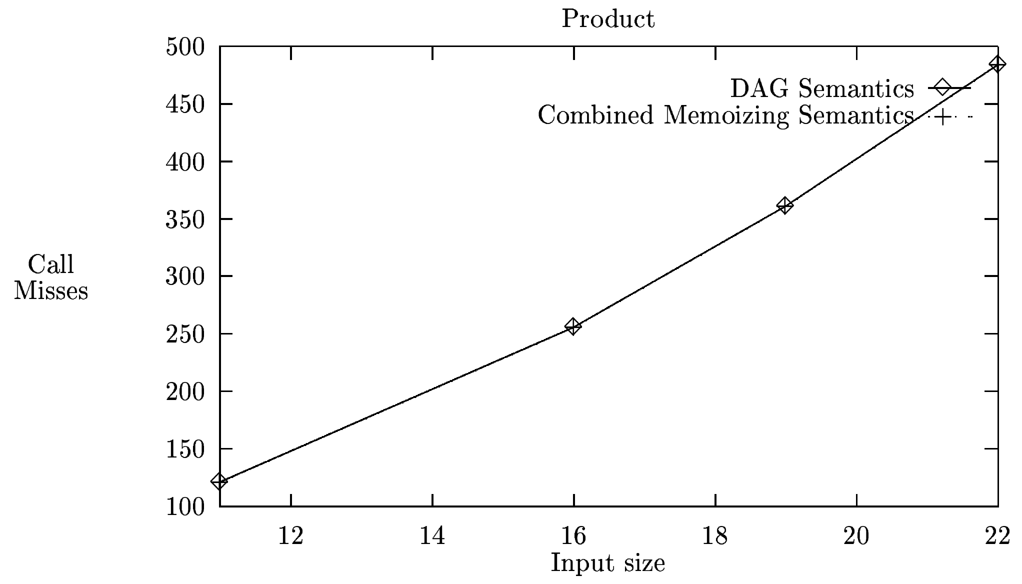


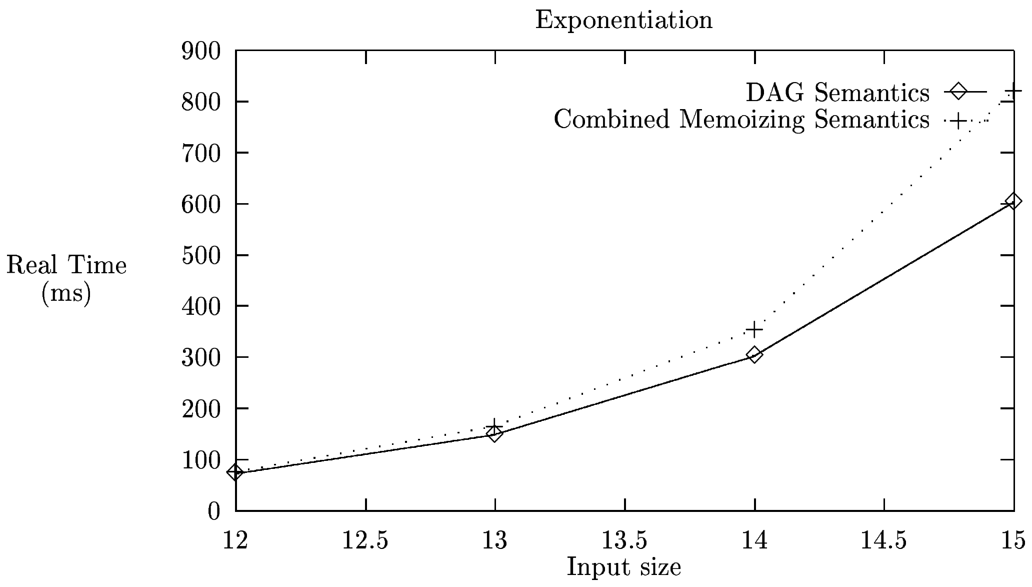


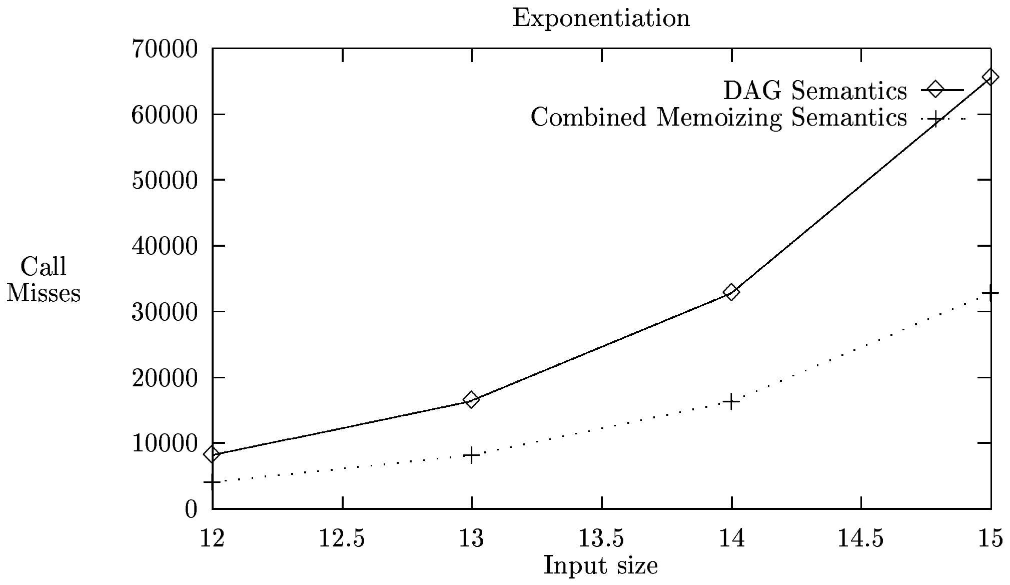

Fibonacci Numbers
Real Time (ms)
Input size
DAG Semantics
Combined Memoizing Semantics
0
50
100
150
200
250
300
350
400
12
13
14
15
16
17
18


Fibonacci Numbers
Call Misses
Input size
DAG Semantics
Combined Memoizing Semantics
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
12
13
14
15
16
17
18


Product
Real Time (ms)
Input size
DAG Semantics
Combined Memoizing Semantics
1
2
3
4
5
6
7
8
9
12
14
16
18
20
22

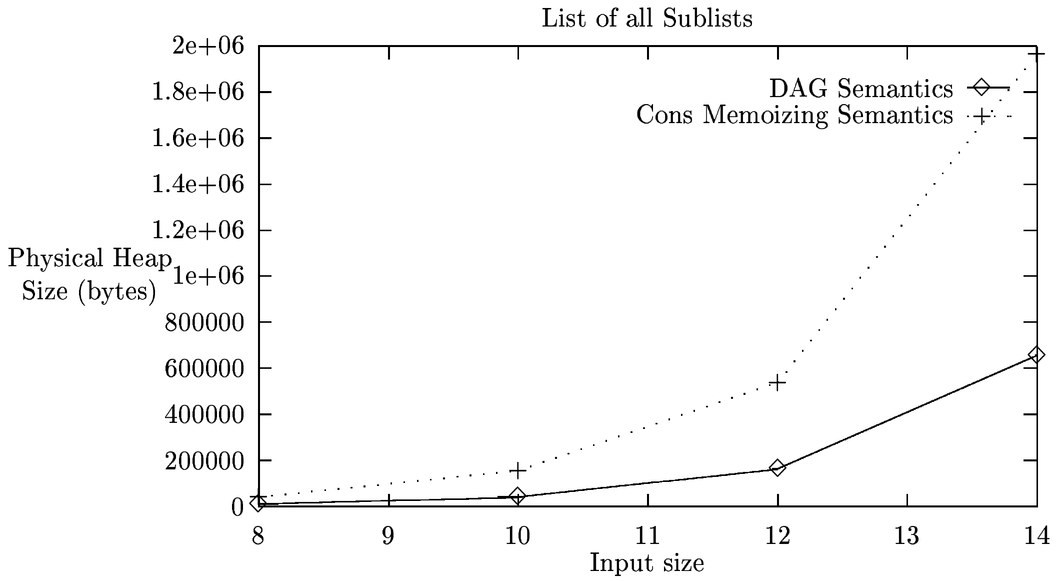

Product
Call Misses
Input size
DAG Semantics
Combined Memoizing Semantics
100
150
200
250
300
350
400
450
500
12
14
16
18
20
22

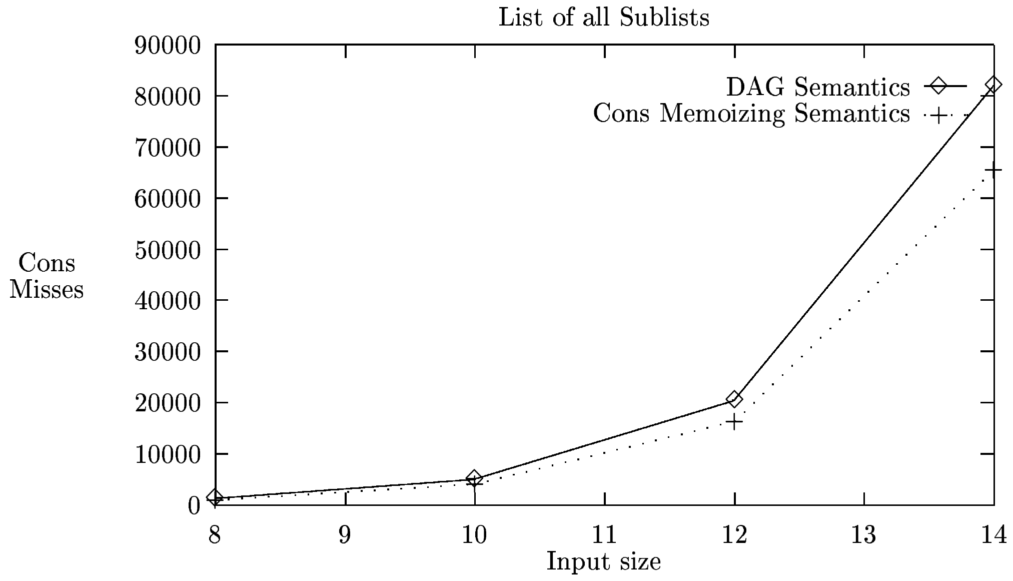

Exponentiation
Real Time (ms)
Input size
DAG Semantics
Combined Memoizing Semantics
0
100
200
300
400
500
600
700
800
900
12
12.5
13
13.5
14
14.5
15

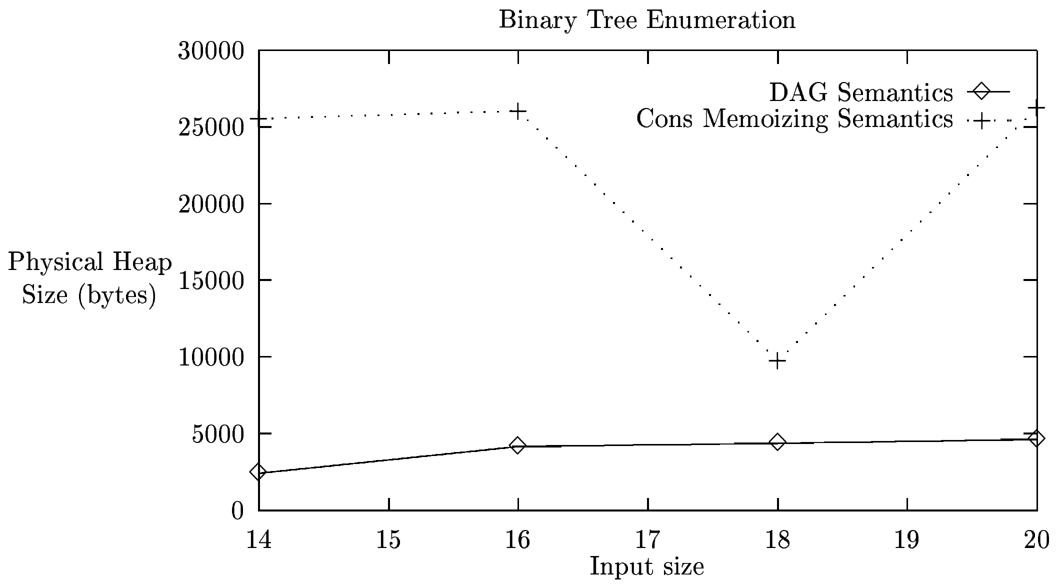

Exponentiation
Call Misses
Input size
DAG Semantics
Combined Memoizing Semantics
0
10000
20000
30000
40000
50000
60000
70000
12
12.5
13
13.5
14
14.5
15

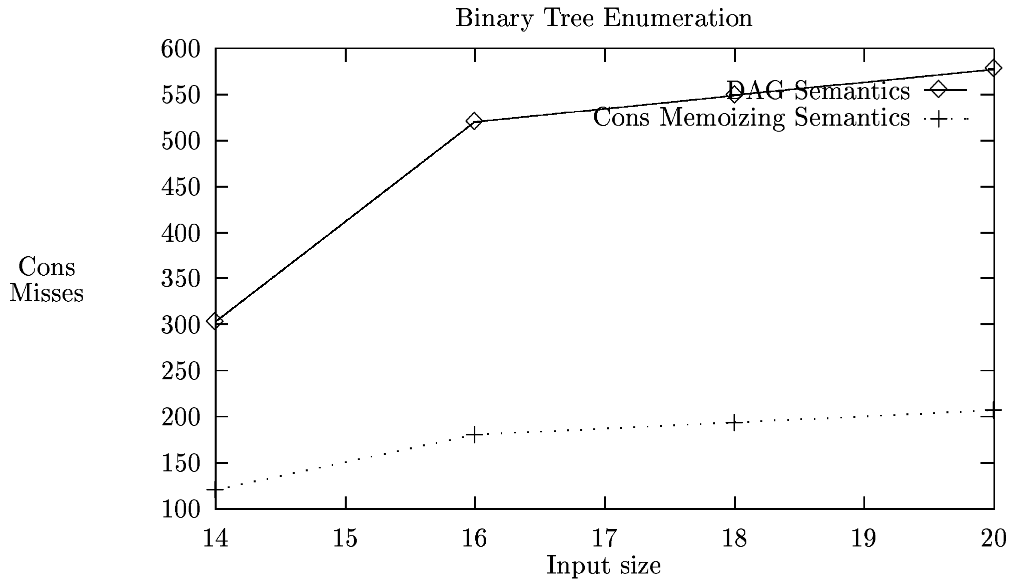
Appendix E

Unlimp Experiments: Enumeration






Binary Tree Enumeration
DAG Semantics
Cons Memoizing Semantics
Physical Heap Size (bytes)
Input size
0
5000
10000
15000
20000
25000
30000
14
15
16
17
18
19
20


Binary Tree Enumeration
DAG Semantics
Cons Memoizing Semantics
Cons Misses
Input size
100
150
200
250
300
350
400
450
500
550
600
14
15
16
17
18
19
20

Bibliography

[1] Donald A. Alton. Nonexistence of Program Optimizers in Several Abstract Settings. *Journal of Computer and System Sciences*, 1976.

[2] Manuel Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, 14(2):322–336, April 1967.

[3] A. Borodin. Computational Complexity and the Existence of Complexity Gaps. *Journal of the ACM*, 19(1):158–174, January 1972.

[4] Niels H. Christensen. Implementation of Levin's Optimal Search Theorem – First Steps. Technical report, DIKU, 1997.

[5] John Gill and Manuel Blum. On Almost Everywhere Complex Recursive Functions. *Journal of the ACM*, 21(3):425–435, July 1974.

[6] Neil D. Jones. *Computability and Complexity from a Programming Perspective.* Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.

[7] S. Kahrs. Unlimp, Uniqueness as a Leitmotiv for Implementation. *Lecture Notes in Computer Science*, 631:115–129, 1992.

[8] L. A. Levin. Computational Complexity of Functions. *Theoretical Computer Science*, 157(2):267–271, May 1996.

[9] Leonid A. Levin. Complexity of Algorithms and Computations. *Mir*, 1974.

[10] C. P. Schnorr. Optimal Algorithms for Self-Reducible Problems. In S. Michaelson and R. Milner, editors, *Third International Colloquium on Automata, Languages and Programming*, pages 322–337, University of Edinburgh, 20–23 July 1976. Edinburgh University Press.

[11] B. A. Trakhtenbrot. Turing Computations with Logarithmic Delay. *Algebra i Logika*, 3(4):33–48, 1964.