Automatic Runtime Analysis for First Order Functional Programs

Carl Christian Frederiksen

September 10, 2002

Contents

1 Abstract **4**

2 Introduction **5**
2.1 Our Approach 6
2.1.1 Call Depth 7
2.1.2 Nested Calls 9
2.1.3 Call Non-linearity 11
2.2 Overview 12

3 The language **13**
3.1 Semantics 14
3.2 Program Running time 17

4 Overview of an Algorithm to Find Running Time Bounds **18**
4.1 Call Graph 18
4.2 Algorithm outline 19

5 Size Analysis **20**
5.1 Size-Measures 23
5.2 Size Bounds 23
5.2.1 Tracking Deconstructions 24
5.2.2 Size Bounds 26
5.3 Size Analysis 28
5.4 Size-Change Graph extraction 31
5.5 Linear bounds on function return values 32

6 Limiting SCCs **36**
6.1 Tuple Descent Systems 37
6.2 Input bounded parameters 38

7 Disjointness **39**

8 Bounding Sub-Function Calls **42**
8.1 Doubling . . . 43
8.2 Definition of Safety for Sub-Calls . . . 45
8.3 Proof of safety . . . 47
8.4 Sub-Function Calls . . . 49
8.5 Implications for the Component . . . 50

9 The Algorithm **51**
9.1 Refinement . . . 51
9.2 Algorithm . . . 52

10 The Bellantoni-Cook Function Class **54**

11 Size-Measures **57**
11.1 Homogeneous size-measures . . . 57
11.2 Structural Size-Measures . . . 58

12 Experiments **61**
12.1 Test Suites . . . 61
12.2 Analysis Output . . . 62
12.3 Positive Examples . . . 64
12.3.1 Greatest Common Divisor . . . 64
12.4 False Negatives . . . 67
12.4.1 Graph Coloring . . . 67
12.4.2 Permutation . . . 69
12.4.3 The "fgh" function . . . 69
12.4.4 Associative String Rewriting . . . 70
12.4.5 Quicksort . . . 71
12.4.6 Mergesort . . . 72
12.4.7 Minsort . . . 72
12.4.8 Contrived 1 . . . 73
12.4.9 The Trick . . . 73
12.4.10 Regular Expression Pattern Matcher . . . 73

13 Conclusion **75**

A Examples **77**
A.1 Jones Examples . . . 77
A.2 Wahlstedt Examples . . . 78
A.3 Size Examples . . . 80
A.4 Ptime Examples . . . 87
A.5 Glenstrup Examples - Algorithms . . . 88
A.6 Glenstrup Examples - Basic . . . 96
A.7 Glenstrup Examples - Interpreters . . . 108
A.8 Glenstrup Examples - Simple . . . 115
A.9 Glenstrup Examples - Sorting . . . 116

B Results **119**
B.1 Jones Results 119
B.2 Wahlstedt Results 121
B.3 Size Results 124
B.4 Ptime Results 130
B.5 Glenstrup Results - Algorithms 132
B.6 Glenstrup Results - Basic 138
B.7 Glenstrup Results - Interpreters 153
B.8 Glenstrup Results - Simple 158
B.9 Glenstrup Results - Sorting 160

Chapter 1

Abstract

The PTIMEF function class is the classical definition of the class of functions which can be computed efficiently, due to the nice closure properties of PTIMEF and the fact that the class is invariant under many different computational models and programming languages.

For this reason, PTIMEF has received much attention and many different characterizations have been proposed.

The present work presents an new program analysis approach [12, 13] to conservatively deciding whether a program terminates in polynomial time based on the size-change termination principle [14]. Our method treats a first order generally recursive functional language. This means that the analyzer does not apriori forbid non-linear recursion, mutual recursion or nested function calls.

We claim that the running time of terminating program is governed by the following three phenomena: call depth, call non-linearity and sub-function calls. If the phenomena can be controlled, then the running time can be controlled as well. Thus we develop program analyses to safely and conservatively identify all programs that execute in super polynomial time.

The analyses are run against a large and diverse test suite to illustrate the capabilities of the developed method.

Chapter 2

Introduction

In the quest for efficient programs and algorithms, the PTIMEF function class is usually considered *the* definition of the class of functions which can be computed efficiently. This is undoubtedly due to the nice closure properties of PTIMEF and the fact that the class is invariant under many different computational models and programming languages.

For this reason, PTIMEF has received much attention and many different characterizations have been proposed.

In [10], Hofmann provides a survey of the recent works of Bellantoni, Goerdt, Leivant, Schwichtenberg et al. The subject of the surveyed works, is to impose restrictions on programming languages to restrict the definable programs to certain complexity classes, including PTIMEF. Clote [4] also provides a survey expressing results in terms of function algebras.

Bellantoni and Cook [2] defined the notion of *safe primitive recursion on notation* which they prove to capture exactly the PTIMEF function class. But at the same time, safe primitive recursion does not capture all *programs* which define functions in PTIMEF – in fact the Bellantoni-Cook criterion has the disadvantage that many of the natural implementations of polynomial time algorithms fall outside the class of programs definable by primitive recursion on notation. Furthermore, Colson has proven that the minimum function requires at least quadratic time to compute, if implemented using safe primitive recursion on notation.

Thus in order to extend the number of PTIMEF programs[1] that can be automatically identified, Caseiro [3] investigated both purely syntactic and semantic criteria. Her criteria extends the number of natural polynomial time algorithms that can be identified, but requires the user to provide input and output families and require the programs to be executed using a form of lazy evaluation to achieve a polynomial running time for the identified programs.

Hofmann presented [11] a type-theoretic alternative, using types to represent resources, i.e. available storage. The programs that are typeable, essentially

[1] i.e. programs that compute a function in PTIMEF.

forbids function that may double the size of an argument to be iterated. This is also the approach taken by Schwichtenberg [21].

Other works have focused more on extracting actual expressions for upper bounds on the running time the subject programs [9, 27, 28, 19, 20, 25, 26]. Naturally, obtaining expressions for the bounds is a much harder problem. Much of the effort has been centered around extracting recurrence equations and solving them to obtain a closed form expression for the running time.

2.1 Our Approach

In the present work we present a fully automatic algorithm for conservatively deciding whether a program terminates in polynomial time. The subject language essentially corresponds to a first order generally recursive subset of Lisp – In particular we do not enforce non-standard syntactic restrictions, e.g. limited primitive recursive calls as in the Bellantoni-Cook framework [2]. This means that the language permits non-linear recursion, mutual recursion and does not rely on "non-standard" execution, e.g. memoization, lazy evaluation, etc. Such restrictions can be employed to restrict the running time of the subject program, but we approach PTIMEF from a program analysis perspective. Since the language is intended to model a first order fragment of Lisp, the language permits constructors with arity greater than one. This also has some interesting implications for the running time, due to sharing.

The work is an extension of the size-change termination principle by Lee, Jones, Ben-Amram [14, 15] to conservatively deciding whether the running time of a program can be bounded by a polynomial. The overall outline of the approach has been presented by Jones [12, 13]. Thus a contribution of the present work is to formalize the ideas and to invent concrete analyses that implement the ideas, demonstrating their credibility.

Automatic termination analysis has been investigated in many different settings. One noteworthy point is that results for one programming language do not necessarily translate easily into results for another language, e.g. termination for imperative programs Cólon and Sipma [6] is of a rather different nature than termination analysis for functional programs. This is due to the fact that imperative languages traditionally do not operate on well-founded data.

The size-change termination principle of Lee et al. is a principle can be stated as: if every infinite sequence would have an infinite decrease of a data value then the (assumed) well-founded ordering on the data domain would be violated. Thus if every infinite sequence has an infinitely many decreases of a data value, then the program must terminate. The basic technique is similar to that of the Termilog [16] termination analyzer for Prolog.

In Lee et al. [14, 15], the treatment of size size analysis was on purpose not treated in detail, as the main objective was the exposition of the new approach to program termination. In the following presentation, the reader is assumed to be familiar with the basics of size-change termination. In particular, understanding the basics of the graph based algorithm would help in understanding

the termination arguments.

One of the contributions of this work is an improved size analysis, which is able to construct simple linear bounds on the function return values. This is an improvement over previous results [24, 7] obtained for functional programs, and borrowing ideas from termination analysis for logic programming languages [5]. Obtaining linear bounds on function return values, plays a key role in our approach to the subsequent extension to handle PTIMEF.

We claim that the running time of a terminating program is governed by the following three phenomena.

- The call depth.
- Nested calls.
- Call non-linearity.

In order to bound running times the phenomena must be controlled. In the following sections, we illustrate the phenomena by several examples.

2.1.1 Call Depth

Append, Addition. Consider the size-change graph for the `append` program, depicted in Figure 2.1.1.

```
append(xs, ys) =
  if eq(xs, Nil)
    then ys
    else Cons(1st(xs), append(2nd(xs), ys))
```

The graph has an in-situ descent in the parameter `xs`, denoted by the solid grey arrow from `xs` to `xs`. The black punctuated arrow from `ys` to `ys` denotes an in-situ non-increase in the parameter `ys`. Since the program only contains the single function call described by the size-change graph, any infinite sequence will have an infinitely many decreases of `xs`. Thus the program size-change terminates. Observe that the size-change graph also provides a *bound* on the length of the longest possible call-chain, i.e. $\|xs\|$.

The situation is the same for the addition program.

```
add(u,v) =
  if eq(u, Z)
    then v
    else S(add(1st(u),v))
```

The size-change graph for addition is given in Figure 2.1.1 and is identical (modulo alpha-renaming) to the size-change graph for the append program.

Iterative Multiplication. Next consider the following iterative definition of multiplication:

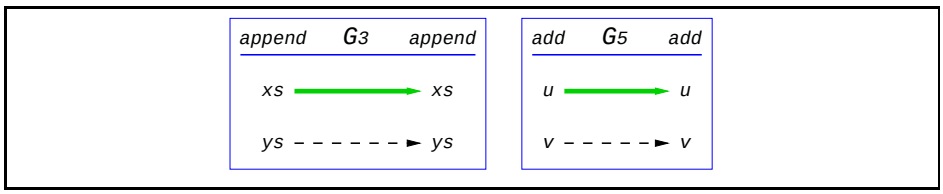

Figure 2.1.1: Size-Change Graphs for Addition and Append.

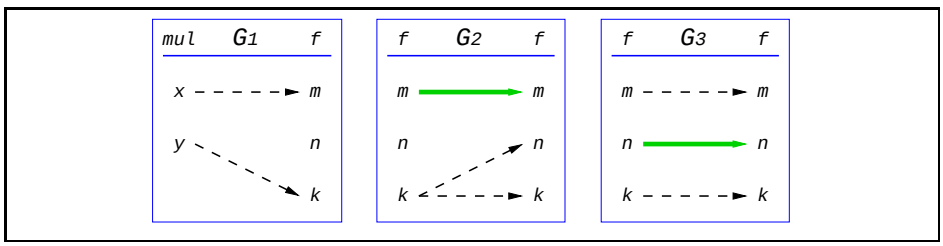

Figure 2.1.2: Size-Change Graphs for Iterative Multiplication.

```
mul(x,y) = f(x,Z,y)
f(m,n,k) =
  if eq(m, Z) then Z
    else if eq(n, Z)
           then f(1st(m), k, k)
           else S(f(m,1st(n),k))
```

The program computes $\mathtt{x} * \mathtt{y}$ and the size-change graphs are given in Figure 2.1.2. Examining the size-change graphs the program is seen to size-change terminate. Note that the sizes of $\mathtt{m}, \mathtt{n}, \mathtt{k}$ are bounded by the size of the input $\mathtt{x}, \mathtt{y}$: $\mathtt{m} \leq \mathtt{x}$, $\mathtt{n} \leq \mathtt{y}$, $\mathtt{k} \leq \mathtt{y}$. We can analyze the call depth by cardinality: If the input tuple (m, n, k) can be repeated in the recursion in f, then f would be non-terminating, so all parameter tuples must be unique. The number of tuples is thus bounded by $(\mathtt{x}+1) * (\mathtt{y}+1) * 1$ since k does not change its value during the recursion.

Iterative Exponentiation. In the next program, an iterative definition of exponentiation, not all parameters are bounded by the input. The program computes the value $2^{\|\mathtt{x}\|}$, by repeatedly doubling the value in a.

```
exp(x) = f(x, Z, x)
f(b,m,a)=
  if eq(b, Z) then a
    else if eq(m, Z)
           then f(1st(b), a, a)
           else f(b, 1st(m), S(a))
```

Observe that in the size-change graphs in Figure 2.1.3 that in the second

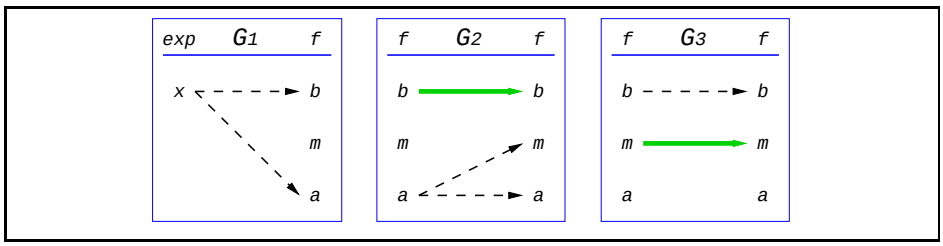

Figure 2.1.3: Size-Change Graphs for Iterative Exponentiation.

recursive call in `f`, no size-change information is available for the parameter `a` due to the increase in size. The program demonstrates that when termination depends on a parameter which is not bounded by the input, then the call depth may be super polynomial.

A principle for bounding call depth. As illustrated in the examples, an upper bound on the call depth can be directly derived from the size-change graphs, provided that all parameters are bounded by the input. In the examples, the derived bound on the call depth also bounds the running time of the entire program, but for general programs, other principles must be applied to check whether the subject program runs in polynomial time.

Principle: The call depth of a size-change terminating function is bounded by a polynomial if the parameters are bounded by the input. This provides a bound for the analyzed function in terms of the input *to the function*, but to handle entire *programs*, bounds are needed on the argument sizes.

2.1.2 Nested Calls

Nested calls cause potential problems for running time estimation.

Multiplication. For the "natural" implementation of multiplication, the situation is similar:

```
mul(x,y) =
  if eq(x, Z)
    then Z
    else add(mul(1st(x), y), y)
```

The program computes a polynomially large value because `add` does not "grow too fast". In particular, `add` does not double the result size relative to the first parameter, so when `add` is iterated n times, the "context transformation" $\lambda x.x + \|\mathtt{y}\|$ is iterated n times to yield the size $n\|\mathtt{y}\|$.

Reverse. Consider the following naïve implementation of the reverse function:

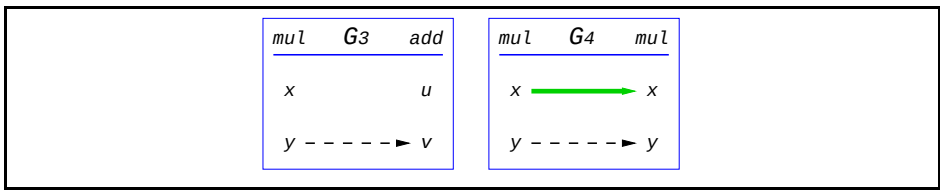

Figure 2.1.4: Size-Change Graphs for Multiplication.

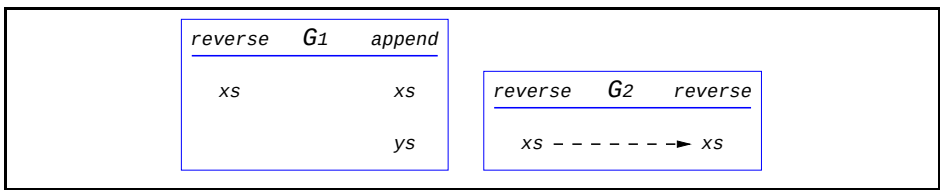

Figure 2.1.5: Size-Change Graphs for Reverse.

```
reverse(xs) =
   if eq(xs, Nil)
     then Nil
     else append(reverse(2nd(xs)), Cons(1st(xs), Nil))
```

By examining the size-change graphs for reverse, the function is seen to size-change terminate. The parameters are even input bounded, but the running time depends on the result size of the `append` function. The problem is that during the recursion, the `append` function is iterated on its own result. If append doubled its size, the function would compute an exponential value. But fortunately, `append` grows the size of the first parameter `xs` with a value that does not depend on `xs`, so reverse cannot compute exponentially large values.

Natural Exponentiation. To illustrate what happens when values grow "too fast", consider the "natural" implementation of the exponentiation function:

```
exp(a,b) =
  if eq(a, Z)
    then Z
    else mul(exp(1st(a), b), b)
```

Clearly, the size of `mul` is the product of the two parameters. For a sequence of n iterations the call context $\lambda x.x * \|\mathtt{b}\|$ is iterated to yield the result size $\|\mathtt{b}\|^n$. The problem is that `mul` can double the size of the result value relative to the first parameter. So when `exp` iterates over the `mul` function, the result value is multiplied with `b` in each iteration, leading to an exponentially large value.

Principle for handling nested calls. Running time bounds can be derived by iterating argument or value transformation. If the argument transformation

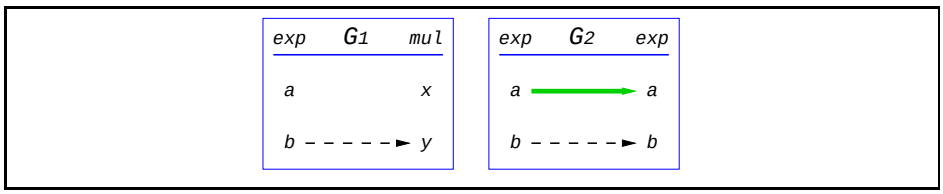

Figure 2.1.6: Size-Change Graphs for Natural Exponentiation.

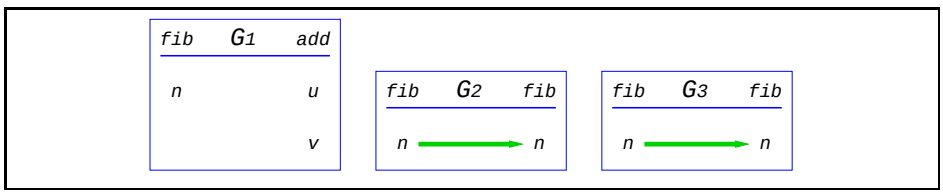

Figure 2.1.7: Size-Change Graphs for Fibonacci.

can double the resulting value relative to the input `x` then iteration over `x` might lead to exponentially large values. To obtain running time bounds, it is thus necessary to find upper bounds on the function result sizes.

2.1.3 Call Non-linearity

Since we treat a general recursive language, the programs may have non-linear calls. Non-linear call can easily give super polynomial running times, a famous example is the Fibonacci function:

Fibonacci.

```
fib(n) =
  if or(eq(n, Z), eq(1st(n), Z))
    then S(Z)
    else add(fib(1st(n)), fib(1st(1st(n))))
```

Examining the size-change graphs, the program is seen to terminate. The only parameter `n` is even input bounded, so the call depth is bounded by a polynomial. But the program does not terminate in polynomial time due to the non-linear calls. The problem is that this implementation of the Fibonacci function can solve the same sub-problem several times. This is due to the fact that the two calls to the `fib` function can share parts of the input. More specificly, the substructure `1st(1st(n))` is passed on both calls. This leads to a super polynomially branching call tree, i.e a call tree where the number of branches cannot be bounded by a polynomial in the size of the input.

Principle for handing non-linearity. This principle we will apply to ensure that we stay in polynomial time is that if arguments are disjoint (i.e. a

given substructure of the input must only be passed to a single recursive function), then the program is polynomially branching. Principle: identify disjoint arguments.

2.2 Overview

The subject language is introduced in Chapter 3, followed by a brief outline of the analysis algorithm in Chapter 4. Chapter 5 present the size analysis, which forms the basis of the analysis. The reader is advised that the chapter requires a basic knowledge of the size-change termination principle [14]. Bounding the call depth is treated in Chapter 6. In chapter 7, the problem of handling non-linear calls is addressed. The final step is provided in Chapter 8 which describes the criterion for bounding sub-function calls. With all parts of analysis properly defined, the algorithm is presented in Chapter 9. In chapter 10 it is proven that programs defined from safe primitive recursion on notation is automatically handled by our methods. Chapter 11 elaborates on some alternative notions of size, which can be important in proving program termination. In chapter 12 we report on results obtained by an implementation of the algorithm and finally in Chapter 13 we summarize the results obtained and review future work.

Chapter 3

The language

In this section we present the subject language used in the analysis. It essentially corresponds to a first order generally recursive subset of Lisp – In particular we do not enforce non-standard syntactic restrictions, e.g. limited primitive recursive calls as in the Bellantoni-Cook framework [2] (but called functions must be defined, variables must be in scope, etc.). However in order not to imply the existence of certain features of Lisp, a new syntax is introduced. The main difference from Lisp is the value domain we compute functions over. The concept of a list constructor is extended to constructors of arbitrary (but fixed) arity. In short, a data value can be regarded as a tree with a identifier attached to each node and leaf. In particular no data type exists for numbers and Lisp symbols (but they can be encoded using the constructors). This gives a reasonable balance between ease of encoding values and complexity of the language. The domain of values in the language is denoted *Value*, and is defined by

$$Value ::= \texttt{con} \mid \texttt{con}(Value_1, \ldots, Value_n).$$

Note that the same names are used for the constructor operators and the constructed values. In order to avoid needless confusion in the absence of a type system, a constructor may not be used with different arities in a program (this also extends to the input to a program).

Constructors of arity zero can be regarded as Lisp atoms, and the binary constructor `Cons` can be regarded as the usual Lisp list constructor. Numbers, however, must be encoded. The base-1 representation of numbers, using e.g. $\texttt{Z}, \texttt{S}(\texttt{Z}), \texttt{S}(\texttt{S}(\texttt{Z})), \ldots$, goes quite well with the termination analysis, which will be defined subsequently.

The inclusion of constructors of arity greater than one gives rise to some interesting problems when considering the running time of a program.

As in Lisp, a set of *destructors* or "field selector operators" are defined to extract different parts of a data value. Inspired by [17] the names `1st`, `2nd`, `3rd`, etc. have been chosen as destructor names. The destructor `1st` correspond to the predecessor operator when applied to base-1 numbers, and `1st` and `2nd` corresponds to *car* and *cdr* in Lisp, when operating on lists.

The syntax for the subject language.

$\pi \in$ *Prog*	::=	$\texttt{def}_1 \ldots \texttt{def}_n$
$\texttt{def} \in$ *Def*	::=	$\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{e}^{\texttt{f}}$
$\texttt{e} \in$ *Expr*	::=	$\texttt{x}$
	|	$\texttt{con}$
	|	$\texttt{con}(\texttt{e}_1, \ldots, \texttt{e}_n)$
	|	$\texttt{des}(\texttt{e})$
	|	$\texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3$
	|	$\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_n)$
	|	$\texttt{op}(\texttt{e}_1, \ldots, \texttt{e}_n)$
$\texttt{x} \in$ *Variable*	::=	identifier beginning with lower case
$\texttt{f} \in$ *FcnName*	::=	identifier beginning with lower case, not in *Op*
$\texttt{con} \in$ *Constructor*	::=	capitalized identifier
$\texttt{des} \in$ *Destructor*	::=	$\texttt{1st} \mid \texttt{2nd} \mid \texttt{3rd} \mid \ldots$
$\texttt{op} \in$ *Op*	::=	primitive operator

Program. A program definition is a sequence of function definitions, and has the form $\texttt{def}_1 \ldots \texttt{def}_k$ for some $k \geq 1$. The *entry function* is the function defined in the first function definition of a program, and is denoted $\texttt{f}_{in}$. The semantics of a program is defined to be the semantics of the entry function. A function definition has the form $\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{e}^{\texttt{f}}$, where $\texttt{e}^{\texttt{f}}$ is the *function body* of f. The number of parameters n is called the *arity* of the function, denoted *arity*(f). The *parameters* of a function are written: $\mathit{Param}(\texttt{f}) = \{\texttt{f}^{(1)}, \ldots, \texttt{f}^{(\mathit{arity}(\texttt{f}))}\}$. Variables are assumed to be in scope when evaluated.

Call Site. For ease of reference each function call will be associated with a unique number, denoting its *call site*. The call sites are numbered by the order they are encountered in the program text by scanning the program left-right, top-down. The set of call sites in a program π is denoted C_π. A given call from function f to function g occurring in a program π at call site c is denoted $\texttt{f} \xrightarrow{c} \texttt{g}$.

Call Sequence. A *call sequence* is a finite or infinite sequence $cs = c_1 c_2 \ldots \in C_\pi^{*\omega}$ of call sites in a given program π. A call sequence is said to be *well formed* if there exists a sequence of functions $\texttt{f}_1\texttt{f}_2 \ldots$ in π such that $\texttt{f}_1 \xrightarrow{c_1} \texttt{f}_2 \xrightarrow{c_2} \texttt{f}_3 \ldots$ is a sequence of function calls.

3.1 Semantics

The semantics of the language is a standard call-by-value semantics, and is given in Figure 3.1.1. Let constructs are regarded as syntactic sugar for a function definition, and are assumed to have been removed in a preprocessing phase for the purpose of program analysis.

$v \in \mathit{Value} = \mathit{Constructor} + \mathit{Constructor} \times \mathit{Value}^*$ (a flat domain).
$u, w \in \mathit{Value}^\sharp = \mathit{Value} \cup \{\mathit{Error}, \bot\}$.
$\mathit{Env} \in \mathit{Variable} \to \mathit{Value}^\sharp$.

$$
\begin{array}{lcl}
\sigma & : & \mathit{Env} \\
[\![\cdot]\!] \cdot & : & \mathit{Prog} \to \mathit{Value}^* \to \mathit{Value}^\sharp \\
\mathcal{E} & : & \mathit{Expr} \to \mathit{Env} \to \mathit{Value}^\sharp \\
\mathcal{B} & : & (\mathit{Expr}, \mathit{Variable}^*) \to \mathit{Env} \to \mathit{Value}^\sharp \\
\mathcal{O} & : & \mathit{Op} \to \mathit{Value}^* \to \mathit{Value}^\sharp \\
\mathcal{C} & : & \mathit{Constructor} \to \mathit{Value}^* \to \mathit{Value}^\sharp \\
\mathcal{D} & : & \mathit{Destructor} \to \mathit{Value} \to \mathit{Value}^\sharp \\
\mathit{lift} & : & \mathit{Value} \to \mathit{Value}^\sharp \qquad \text{(the natural injection)} \\
\mathit{strictapply} & : & (\mathit{Value}^* \to \mathit{Value}^\sharp) \to (\mathit{Value}^\sharp)^* \to \mathit{Value}^\sharp
\end{array}
$$

$$[\![\pi]\!]\vec{v} = \mathcal{E}[\![\mathtt{e}^{\mathtt{f}_{in}}]\!][\mathtt{f}_{in}{}^{(i)} \mapsto v_i] \ \forall i \in \{1, \ldots, \mathit{arity}(\mathtt{f}_{in})\}$$

$$
\begin{array}{ll}
\mathcal{E}[\![\mathtt{x}]\!]\sigma & = \sigma(\mathtt{x}) \\
\mathcal{E}[\![\mathtt{con}]\!]\sigma & = \mathit{lift}(\mathtt{con}) \\
\mathcal{E}[\![\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma & = \mathit{strictapply}(\mathcal{C}[\![\mathtt{con}]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma) \\
\mathcal{E}[\![\mathtt{des}(\mathtt{e})]\!]\sigma & = \mathit{strictapply}(\mathcal{D}[\![\mathtt{des}]\!])(\mathcal{E}[\![\mathtt{e}]\!]\sigma) \\
\mathcal{E}[\![\mathtt{if\ e}_1\ \mathtt{then\ e}_2\ \mathtt{else\ e}_3]\!]\sigma & = \begin{cases} \mathcal{E}[\![\mathtt{e}_1]\!]\sigma & \text{if } \mathcal{E}[\![\mathtt{e}_1]\!]\sigma \in \{\mathtt{Error}, \bot\}, \\ \mathcal{E}[\![\mathtt{e}_2]\!]\sigma & \text{if } \mathcal{E}[\![\mathtt{e}_1]\!]\sigma = \mathtt{True}, \\ \mathcal{E}[\![\mathtt{e}_2]\!]\sigma & \text{otherwise.} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma = \\
\qquad \mathit{strictapply}(\mathcal{B}[\![\mathtt{e}^{\mathtt{f}}, (\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(n)})]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma) \\
\mathcal{E}[\![\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma = \\
\qquad \mathit{strictapply}(\mathcal{O}[\![\mathtt{op}]\!])(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_1]\!]\sigma)
\end{array}
$$

$$\mathcal{C}[\![\mathtt{con}]\!](v_1, \ldots, v_n) = \mathtt{con}(v_1, \ldots, v_n)$$

$$\mathcal{D}[\![\mathtt{des}]\!]v = \begin{cases} v_{\mathit{arity}(\mathtt{des})} & \text{if } v = \mathtt{con}(v_1, \ldots, v_n) \wedge \mathit{idx}(\mathtt{des}) \leq n \\ \mathtt{Error} & \text{otherwise.} \end{cases}$$

$$\mathcal{B}[\![\mathtt{e}, (\mathtt{x}_1, \ldots, \mathtt{x}_n)]\!](v_1, \ldots, v_n) = \mathcal{E}[\![\mathtt{e}]\!][\mathtt{x}_i \mapsto v_i] \ \forall i \in \{1, \ldots, n\}$$

$$\mathit{strictapply}\ \psi\ w = \begin{cases} \psi(v_1, \ldots, v_n) & \text{if } w_i \notin \{\mathtt{Error}, \bot\} \text{ for } i \in \{1, \ldots, n\}, \\ & \text{where } w_i = \mathit{lift}\ v_i, \text{ for } i \in \{1, \ldots, n\}. \\ w_i & \text{otherwise, where } i = \text{least index} \\ & \text{such that } w_i \in \{\mathtt{Error}, \bot\}. \end{cases}$$

$$\mathcal{O}[\![\mathtt{op}]\!]\vec{v} \in \{\mathit{True}, \mathit{False}\}$$

Figure 3.1.1: Semantics. `True`, `False` *and* `Error` *are distinguished elements of Value.*

Note that in the context of a program, `con` is a constructor *operator*, so `Cons(Nil,Nil)` is a expression consisting of a constructor operators that evaluates to a constant, and not a constant. Since this is a minor technicality, the constructed values are denoted using the same names as the constructor operators – thus the expression `Cons(Nil,Nil)` $\in$ *Expr* evaluates to the value `Cons(Nil,Nil)` $\in$ *Value*. The number of sub-terms in a constant c is called the *a*rity of the constant, denoted $arity(c)$. It should be clear from the context whether *arity* refers to the arity of a constant or a function. Constructor names are unique and must be used with the same fixed arity throughout the program. This can easily be verified by a syntactic check.

For a destructor operator `des` the *index* of the destructor indicates the number of the subtree to be extracted from the top-level constructor. Example: $idx(\texttt{3rd}) = 3$. Runtime errors are modeled by the constant `Error`. If a subcomputation evaluates to `Error`, then the result of the entire computation is `Error`. Note that the 0-ary constructor `Error` evaluates to the constant `Error`. The $\mathcal{C}$ and $\mathcal{D}$ operators implement constructors and destructors respectively.

The language contains a set of "primitive operators" which include "real" primitive operators for equality testing as well as "library functions" like boolean and. The operator `eq`, tests whether the topmost constructors of each of the two arguments have the same constructor name. Since a given constructor name can only be used with a fixed arity for the entire program, name equality also implies arity equality. Other primitive functions can be added by extending the $\mathcal{O}$ operator, essentially extending the default function library. This allows us to avoid specifying trivial functions the boolean predicates `and`, `or` and `not` with each and every example. The only requirement is that a primitive function never fails and always terminate in *constant* time with either `True` or `False`, so primitive operators are essentially restricted to simple predicates.

The semantics of the subject language is given in Figure 3.1.1. The $\|\cdot\|$ operator is the *meaning function* – given a subject program and the input in the form of a vector of data values, the meaning function computes the function implemented by the program.

The meaning function uses several other operators to assign meaning to a program. The most important one is the expression evaluation function $\mathcal{E}$ which evaluates an expression given an environment that maps to free variables in the expression to data values. Since programs can perform illegal operations, like taking the head of an empty list or taking the predecessor of zero, the expression evaluation function must account for erroneous return values. Thus the values computed are either ordinary data values or a data value indicating that an exception has occurred: this is captured by the $Value^\sharp$ domain, where `Error` represents "exceptions" and $\perp$ indicates non-termination. The operator *list* is provided to translate from data values in *Value* to the extended domain $Value^\sharp$.

Other operators include the bind operator, which is used to bind a list of parameters to a list of data values to form a new environment. The operator then evaluates an expression using the new environment. The operator is used on function calls to create the new environment.

Finally the operator *strictapply* is used to enforce a left-to-right evaluation order. The operator is given a list of expressions to be evaluated; If any expression returns an "erroneous" value (i.e. `Error` or $\perp$) then the return value of the first expression that returned the erroneous value is returned as the result.

State Transition. A *state* is a pair in $State = FcnName \times Env$. A *state transition* is a relation $(\mathtt{f}, \sigma) \xrightarrow{c} (\mathtt{g}, \sigma')$ where the call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ occurs in the body of `f` at call site c such that $\sigma' = [\mathtt{g}^{(i)} \mapsto v_i] \ \forall i = 1 \ldots n$ and $\mathcal{E}[\![\mathtt{e}_i]\!]\sigma = v_i \neq \perp$ for all $i \in \{1, \ldots, n\}$.

State transition sequence. A *State transition sequence* is a sequence of state transitions $(\mathtt{f}_i, \vec{v_i}) \xrightarrow{c_i} (\mathtt{f}_{i+1}, \vec{v}_{i+1})$. Such a sequence is denoted

$$(\mathtt{f}_1, \vec{v_1}) \xrightarrow{c_1} (\mathtt{f}_2, \vec{v_2}) \xrightarrow{c_1} (\mathtt{f}_3, \vec{v_3}) \ldots$$

The calls occurring in the state transition sequence cs is denoted $calls(sts)$.

3.2 Program Running time

Since our goal is to classify programs by their asymptotic running time, the notion of running time must be specified. One distinctive point of this work is to analyze a program with respect to its natural running time – the runtime of the program when executed under the standard execution model – i.e. without relying on memoization, lazy evaluation or other forms of "non-standard" execution.

A classical way of measuring the running time of a program is to assume that each operation takes unit time to execute and simply count the number of instructions executed. However, the *precise* running time is not important, since we will be classifying programs by their asymptotic running time. Thus the number of function calls will be used as a measure for the running time. This is acceptable, since the time to execute the operations in a function body (minus the sub-function calls) can be bounded by a constant.

This definition has some interesting implications. Firstly, since the subject language permits the use of constructors with arity greater than one, it is possible to double the "perceived size" of a value: e.g. suppose `f` traverses the parameter `x`, exploring all subterms of the value that `x` is bound to. The call `f(Cons(x,x))` doubles the number of function calls used in recursive traversal of the term, but can be *implemented* (using pointers) to use a fixed number of memory cells more than the space needed to store `x`. This complicates the running time analysis since a value constructed in polynomial time might require *exponentially many* function calls during recursive traversal of the term. For this reason it is vital that the running time of a sub-function cannot depend on parameters that might be bound to a value of super-polynomial "perceived size".

Chapter 4

Overview of an Algorithm to Find Running Time Bounds

In this section we give an overview of the algorithm. The purpose of the analysis is to conservatively decide whether a functional (Lisp) style program will terminate in polynomial time in the size of the program input, i.e. computes a function in PTIMEF. A natural way to break down the problem is to analyze each recursive function in turn, starting with functions that only call themselves recursively. This is easily done by computing the strongly connected components in the call graph and processing the components in reverse topological (bottom-up) order.

4.1 Call Graph

Definition 4.1.1. (Call Relation)*: The call relation is defined in terms of the direct call relation as defined in Section 3. Suppose* $\mathtt{f}$ *is a function definition, then the* direct call relation *is defined by* $\mathtt{f} \rightarrow \mathtt{g}$ *iff* $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ *occurs in the body* $\mathtt{e}^{\mathtt{f}}$ *of* $\mathtt{f}$*. The* call relation $\rightarrow^* \subseteq \mathit{FcnName} \times \mathit{FcnName}$ *is the transitive closure of the direct call relation and it defines the* call graph *for a program:* $(\mathit{FcnName}, \rightarrow^*)$*.*

Definition 4.1.2. (Strongly Connected Components)*: The call relation induces an equivalence relation on the functions in the program:* $\mathtt{f} \equiv \mathtt{g}$ *iff* $\mathtt{f} \rightarrow^* \mathtt{g}$ *and* $\mathtt{g} \rightarrow^* \mathtt{f}$*. A strongly connected component in the call graph (henceforth: "a component"), is an equivalence class induced by the call relation.*

The call relation also induces a partial ordering on the strongly connected components in the call graph. This ordering is called the topological *ordering and is denoted:* $\succ$*.*

Suppose C is a component such that $C \succ C'$ for any $C' \neq C$, then the functions $\mathtt{f}' \in C'$ is a sub-function *of any $\mathtt{f} \in C$.*

4.2 Algorithm outline

Following is a brief overview of the algorithm being developed. For each component, the algorithm's goal is to show that the running time is bounded by a polynomial in the size of the input where possible. This is broken down into several steps:

- Show that the program size-change terminates. If this step fails, no bound can be deduced. Extraction of size-change graphs is described in Chapter 5.
- Prove that the program terminates when restricted to a parameter subset which behaves in a restricted way (input bounded). If termination can be proven using only such parameters, then the call depth can be shown to be bounded by a polynomial.
- Next it must be ensured that only polynomially many recursive calls can be made on any input. In particular we do not wish to restrict the algorithm to linear functions, i.e. functions with one recursive call in each branch. This will enable the algorithm to analyze "divide and conquer" algorithms, but at the same time it must be ensured that non-linearity cannot give rise to polynomially many branches in the call tree.
- The last step consists of proving that sub-function calls can only give rise to polynomially many calls in the called sub-function *in each recursive evaluation of the function body of the calling function.* This is ensured by requiring that the arguments which can be passed to the sub-functions cannot grow "too fast", but the arguments are allowed to increase in size as the recursion in the calling function progresses.

Chapter 5

Size Analysis

The purpose of the size analysis is to provide sufficient information to extract safe size-change graphs from the subject program, i.e. information relating the sizes of the parameters to a function to the sizes of the arguments for a given call site. In Lee et al. [14, 15], the treatment of size-change graph extraction was on purpose not treated in detail, as the main objective was the exposition of the new approach to program termination.

Different approaches to size-change graph extraction have been considered. In Wahlsted's thesis [24], size-change graphs are extracted by straight-forward syntax analysis. The subject language has explicit deconstruction on base-1 numbers, so the analysis is implemented by matching parameters to the variables representing the deconstructed values. The analysis does not handle nested functions calls and unfortunately sizes of returned values often play an important role in size-change termination proofs.

The size-change termination principle itself does not deal with the behavior over "sub-computations" – it only deals with the size-change behavior over infinite *non-terminating* call sequences. Thus it is up to the size-change graph extraction phase to describe the size-change behavior over terminating call sequences, i.e. to describe the size of return values.

A slight improvement was formulated [7] as a simple abstract interpretation using the same abstraction as used in the size-change graphs. With this approach one can obtain a description of the return values as decreases or non-increases with respect to the parameters. The draw-back is that the method only is capable of capturing *material size-relations* – relations can only be inferred from expressions that copy or deconstruct the input of the function. While it greatly increases the number of programs that can be proven to terminate, it is not possible to deduce any size relations for functions with immaterial dependencies. The `append` function is an example of a function with both material and immaterial dependencies:

```
append(xs, ys) =
  if eq(xs, Nil)
```

```
then ys
else Cons(1st(xs), append(2nd(xs), ys))
```

In the base case (the first branch of the conditional), `ys` is returned. This introduces a material dependency between the return value of `f` and `ys`. This is denoted: $\texttt{f} \overset{\texttt{ys}}{\leadsto}_{mat}$. In general, an expression `e` is *materially dependent* on the parameter `x` iff a substructure of `x` might be used in the construction of the value that `e` evaluates to. Such parameters are easily identified by traversing the parse tree of the expression while collecting all variables encountered.

Furthermore, the return value of `f` depends immaterially on `xs`, denoted: $\texttt{f} \overset{\texttt{xs}}{\leadsto}_{imm}$: the return value is *reconstructed* by recursively inspecting the input. The dependency arises because the size of `xs` controls the size of the return value, since the size of `xs` controls the number of times the constructor is applied. More specificly, there exists a (potentially) decreasing self dependency on `x` – there exists a call sequence

$$(\texttt{append}, \sigma) \rightarrow \ldots \rightarrow (\texttt{append}, \sigma')$$

such that the size of $\sigma'(\texttt{xs})$ is smaller than $\sigma(\texttt{xs})$. If a parameter in any way can control the recursion, then there must exists a decreasing self dependency on that parameter. Thus the number of constructors added to any term is bounded by the parameters with decreasing self dependencies. At the same time, the return value of `append` has an increasing self dependency in a loop in the program. Thus the `xs` can control the increasing self dependency on `append`: $\texttt{append} \overset{\texttt{xs}}{\leadsto}_{imm}$. Finally the return value of a function `f` *depends* on a parameter `x`, denoted $\texttt{f} \leadsto \texttt{x}$, if it depends either materially or immaterially on `x`. It is assumed that a size dependency analysis similar to the one in [8] is available.

Note that in the subsequent examples, we sometimes use syntax in place of the corresponding semantic value, e.g. $\|\texttt{e}\| \leq \|\texttt{x}\|$ is not quite proper for programs, though common in practice in mathematics. For brevity we use the improper notation informally when the meaning is clear from the context. The informal notation in the above example should be interpreted as: for any environment σ it holds that, the size of the value that `e` evaluates to is smaller than the size of the value the `x` evaluates to:

$$\|\mathcal{E}[\![\texttt{e}]\!]\sigma\| \leq \|\mathcal{E}[\![\texttt{x}]\!]\sigma\|.$$

In the Mercury termination analyzer [5] for Prolog programs, the size-analysis deduces linear relations between the input-moded parameters and the output-moded parameters. These constraints can then be solved by linear programming to obtain safe bounds on the sizes of the predicates. This information can then be used to check whether evaluation of a predicate will terminate.

Inspired by the Mercury analyzer's approach, we present a size analysis which extracts bounds on the return values of functions in a similar manner. For each function in the program, a set of inequalities is established. The key idea: if all the constraints extracted from a strongly connected component in the call graph can be satisfied, then the size of the return values of the functions in the

component are bounded by a linear function of some of their arguments sizes. The constraints are used to specify a linear programming (LP) problem which then can be solved by a standard linear programming package. The implication: if the LP problem has a bounded solution, then a safe upper bounds on the size of the return values has been found. This information can be used to improve the size analysis. As an example, consider the `reverse` function:

```
reverse(xs, r) =
  if eq(xs, Nil)
    then r
    else reverse(2nd(xs), Cons(1st(xs), r))
```

The goal is to attempt to establish that the size of the return value of `append` is bounded by a linear function – i.e. there exists a constant γ such that for all input `xs`, `r`:

$$\|\texttt{xs}\| + \|\texttt{r}\| + \gamma \geq \|\texttt{reverse}(\texttt{xs}, \texttt{r})\|.$$

Considering each exit from the reverse code, a set of inequalities is established:

$$\begin{array}{lcl}
\|\texttt{xs}\| + \|\texttt{r}\| + \gamma & \geq & \|\texttt{r}\| \\
\|\texttt{xs}\| + \|\texttt{r}\| + \gamma & \geq & \|\texttt{xs}\| + \|\texttt{r}\| + \gamma \\
& \geq & (1 + \|\texttt{1st}(\texttt{xs})\| + \|\texttt{2nd}(\texttt{xs})\|) + \|\texttt{r}\| + \gamma \\
& = & \|\texttt{2nd}(\texttt{xs})\| + (1 + \|\texttt{1st}(\texttt{xs})\| + \|\texttt{r}\|) + \gamma \\
& = & \|\texttt{2nd}(\texttt{xs})\| + \|\texttt{Cons}(\texttt{1st}(\texttt{xs}), \texttt{r})\| + \gamma \\
& \geq & \|\texttt{reverse}(\texttt{2nd}(\texttt{xs}), \texttt{Cons}(\texttt{1st}(\texttt{xs}), \texttt{r}))\|
\end{array}$$

The derivation for the second constraint should be read bottom-up: we wish to ensure that the size of the value that the else-branch evaluates to is bounded by the proposed bound. This is implied (though the derivations) if the bound is equal to itself (top equation), which is vacuously true.

The LP problem is then solved, and $\gamma = 0$ is found as a solution, so $\|\texttt{xs}\| + \|\texttt{ys}\|$ is obtained as a safe upper bound on the size of the return value.

A second point of the size analysis presented is that more precise size information can be obtained by accounting for deconstructions as precisely as possible. This is necessary in order to obtain the inequalities in the above example. The approach is allow the size bounds to keep track of deconstructions, i.e. the bounds can be described in terms of *sub-terms* of the parameters. If the bounds only can be expressed in terms of the entire size of the parameters, then the second inequality for `reverse` becomes:

$$\begin{array}{lcl}
\|\texttt{xs}\| + \|\texttt{r}\| + \gamma & \geq & 2\|\texttt{xs}\| + \|\texttt{r}\| + \gamma - 1 \\
& = & (1 + (\|\texttt{xs}\| - 1) + (\|\texttt{xs}\| - 1)) + \|\texttt{r}\| + \gamma \\
& \geq & (1 + \|\texttt{1st}(\texttt{xs})\| + \|\texttt{2nd}(\texttt{xs})\|) + \|\texttt{r}\| + \gamma
\end{array}$$

But with this constraint, we cannot find a constant γ such that the constraints are satisfied for all input, since $\|\texttt{xs}\|$ can be unboundedly large.

Outline. This chapter covers four different topics: First, the concept of a size-measure is introduced. The size-measure induces a well-founded ordering on the data value domain, which is used to prove termination. Second, a size-analysis is described. The size-analysis is used for extracting the constraints previously mentioned and to extract size-change graphs. Third, the size-change graph extraction is defined and proven to be safe, as defined in [14]. Finally, the method for extracting linear bounds on function return values is described.

5.1 Size-Measures

In order to prove size-change termination, a well-founded ordering on the terms (i.e. on the data value domain) must exist. This can be done for our value domain by first defining a *size-measure* on the terms.

Definition 5.1.1. (Size-measure)*: A size-measure on Value is a mapping* $\|\cdot\| : \mathit{Value} \to \mathbb{N}$ *such that for some* $\gamma > 0$, $I \subseteq \{1, \ldots, n\}$ *and for all* $\mathtt{v}_1, \ldots, \mathtt{v}_n \in \mathit{Value}$*:*

$$\|\mathtt{con}(\mathtt{v}_1, \ldots, \mathtt{v}_n)\| \leq \gamma + \sum_{i \in I} \|\mathtt{v}_i\|.$$

Note that the ordering induced by the ordering of the natural numbers is a well-founded ordering of the terms in *Value*, which will form the basis for the size-change termination proofs.

Since terms from the data value domain *Value* are trees, the size-measures do *not* account for sharing. Therefore, for the term $\mathtt{Cons}(\mathtt{x}, \mathtt{x})$ the left sub-term $\mathtt{x}$ and the right sub-term $\mathtt{x}$ are considered as being two different values.

Naturally, there exist many different size-measures and in general no single "best size-measure" exists. In order to keep the complexity of the size-analysis at a reasonable level, we will use the node size-measure. Other possible size-measures are discussed in Chapter 11.

Definition 5.1.2. (Node size-measure)*: The node size-measure is given by:*

$$\begin{array}{lcl} \|\mathtt{con}\| & = & 0 \\ \|\mathtt{con}(\mathtt{v}_1, \ldots, \mathtt{v}_n)\| & = & 1 + \sum_{i=1}^{n} \|\mathtt{v}_i\|, n \geq 1. \end{array}$$

The node size-measure measures the amount of storage consumed by the constructors of arity greater than zero (the "node" constructors) under the assumption that the sub-terms are disjoint – i.e. using the "perceived size" of the term. One distinct advantage of the node size-measure is that when deconstructing a term, it is not necessary to know the arity of the term in order to obtain a good bound on the size (more on this in Chapter 11).

5.2 Size Bounds

This purpose of this section is to develop a size analysis $\mathcal{S}[\![\mathtt{e}]\!]\delta$ that computes an upper bound on the expression, for which the following holds:

Theorem 5.2.1. (Safety of the Size-Analysis): *If* `e` *is an expression in the body of the function* `f` *in program* π*, and if and any safe function bound environment* δ *the size analysis is safe with respect to the node size-measure:*

$$\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq \|\mathcal{S}[\![\mathtt{e}]\!]\delta\|_{\mathcal{S}}\sigma,$$

for any value environment σ*.*

The analysis processes each strongly connected component in the call graph in turn, starting with the topologically smallest component (the component at the bottom of the call graph). For each component it is assumed that all functions that can be reached have been analyzed and have been found to terminate.

The size analysis takes any expression `e` and computes an upper bound b on the size of `e` such that $\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq \|b\|_{\mathcal{S}}\sigma$ if $\mathcal{E}[\![\mathtt{e}]\!]\sigma \neq \bot$. The upper bound b is a linear expression $b \in \mathit{sizeBd}$ of the form

$$b = \gamma + \mathtt{x}_1 : t_1 + \ldots + \mathtt{x}_n : t_n,$$

where γ is a constant and t_i describes the size with respect to the parameter $\mathtt{x}_i$. The size bound language, *sizeBd*, is augmented with a semantic operator: $\|\cdot\|_{\mathcal{S}}\cdot$ that evaluates the size of a size bound using a data value environment.

5.2.1 Tracking Deconstructions

Rather than simply abstracting size information relative to a parameter, the way the parameters were deconstructed will be described in the size bound as well. As we demonstrated in the introduction with the `reverse` program, it does matter how size information is abstracted relative to a parameter.

Thus we define the notion of a *deconstruction tree*, which captures size information relative to a specific parameter. This serves two purposes: we are able to distinguish substructures of a function's parameters and it helps us construct *solvable* constraints which are used to derive bounds on function return values.

Definition 5.2.1. (Deconstruction Tree)*: A* deconstruction tree *describes how a given size bound relates to a given parameter* `x` *written* `x`*: DeconTree, where DeconTree is a tree with a weight (a natural number) attached to each node and leaf.*

$$t \in \mathit{DeconTree} ::= I\!N \mid I\!N \triangleright [\mathit{DeconTree}_1, \ldots, \mathit{DeconTree}_n].$$

The "t_i"'s in the size bounds describe which substructures of the i'th parameter the size bound depend upon. This is captured by a *deconstruction tree*. A deconstruction tree is a tree with a natural number attached to each node and leaf. Suppose $b = \mathtt{x} : t$ is a size bound for some expression `e`. A non-zero value at any location in the deconstruction tree t, asserts that the size of the expression may depend on the corresponding sub tree in the data value. e.g. suppose $\mathtt{e} = \mathtt{2nd}(\mathtt{x})$, then the corresponding deconstruction tree would be $b = \mathtt{x} : 0 \triangleright [0, 1]$, capturing that the size of `e` depends on the second child of the data value bound to `x`, but not on first child or the root.

An example: the size of the parameter `x` relative to the expression `Cons(x, 2nd(x))` is described by the deconstruction tree: $\mathtt{x} : 1 \triangleright [0,1]$, indicating that the root and second child have been accessed, but not the first child. Furthermore, addition is defined in order to describe the combined size of two terms; e.g. if $\mathtt{x} : 1$ and $\mathtt{x} : 0 \triangleright [0,1]$ are deconstruction trees for the expressions `x` and `2nd(x)` then $\mathtt{x} : 1 + 0 \triangleright [0,1] = \mathtt{x} : 1 \triangleright [0,1]$ is a deconstruction tree for the combined expression `Cons(x, 2nd(x))`.

Semantics of Deconstruction Trees. To assign meaning to the size bounds, we are working towards a formal description of the $\|\cdot\|_{\mathcal{S}}\cdot$ operator on size bounds. A sub-problem is how to translate deconstruction trees into actual bounds expressed purely in terms in the size of the data values that the parameters are bound to.

Suppose $\mathtt{x}_i : t_i$ is a deconstruction tree such that $t_i \in I\!N$, then we can think of the ":" notation as multiplication, e.g. for $\mathtt{x} : 5$, the bound being described is simply: $5\|\mathtt{x}\|$. Deconstruction trees on this form are called *normalized* deconstruction trees. In general we need a method for translating from deconstruction trees to linear size bounds expressed purely in terms of the size of the entire data value bound to the corresponding parameter. This is captured by the *normalize* operator on deconstruction trees.

Definition 5.2.2. (Normalization)*: The normalization operator is defined by:*

$$
\begin{array}{lll}
\mathit{normalize}: & \mathit{Variable} \times \mathit{DeconTree} \to \mathit{sizeBd} & \\
\mathit{ncons}: & \mathit{DeconTree} \to I\!N & \\
\mathit{nrep}: & \mathit{DeconTree} \to I\!N & \\
\mathit{normalize}(\mathtt{x}, t) & = & -\mathit{ncons}(t) + \mathtt{x} : \mathit{nrep}(t) \\
\mathit{ncons}(n) & = & 0 \\
\mathit{ncons}(n \triangleright [t_1, \ldots, t_n]) & = & max_{i=1\ldots n}\mathit{nrep}(t_i) + \sum_{i=1\ldots n} \mathit{ncons}(t_i) \\
\mathit{nrep}(n) & = & n \\
\mathit{nrep}(n \triangleright [t_1, \ldots, t_n]) & = & n + max_{i=1\ldots n}\mathit{nrep}(t_i)
\end{array}
$$

Intuitively, the normalized size comprises two parts – the number of occurrences of the variable in the term (computed by *nrep*) *and* a constant adjusting for the extra constructors used in the normalized term (computed by *ncons*). Consider the expression: `1st(x)`, with the corresponding deconstruction tree $0 \triangleright [0,1]$. Normalization gives $(-1 + \mathtt{x} : 1)$ – the reading: at the cost of $\mathit{ncons}(0 \triangleright [0,1]) = 1$ extra constructor, the size of `1st(x)` can be bounded by $\mathit{nrep}(0 \triangleright [0,1])\|\mathtt{x}\| = 1\|x\|$. Thus `1st(x)` can be bounded by $\|\mathtt{x}\| - 1$.

Definition 5.2.3. (Operators on Deconstruction trees)*: For each deconstructor, a corresponding operator is defined on the deconstruction trees.*

$$
\begin{array}{lll}
\mathcal{D}_{\mathtt{1st}}(t) & = & 0 \triangleright [t] \\
\mathcal{D}_{\mathtt{2nd}}(t) & = & 0 \triangleright [0, t] \\
\mathcal{D}_{\mathtt{3rd}}(t) & = & 0 \triangleright [0, 0, t] \\
\ldots & &
\end{array}
$$

Addition of deconstruction trees is defined recursively by simply adding the weights.

$$\begin{array}{lcl} |\cdot| : & \mathit{DeconTree} \to \mathbb{N} & \\ |n| & = & n \\ |n \triangleright [t_1, \ldots, t_n]| & = & n + \sum_{i=1}^{n} |t_i| \end{array}$$

The $|\cdot|$ operator counts the number of unique sub-terms that the deconstruction tree describes.

Example. Consider the term sequence $[\texttt{2nd(x)}, \texttt{3rd(x)}, \texttt{2nd(x)}, \texttt{1st(2nd(x))}, \texttt{3rd(2nd(x))}]$. The sum of the deconstruction trees with respect to. `x` for the terms, yields the deconstruction tree:

$$\texttt{x}: \ 0 \triangleright [0, 2 \triangleright [1, 0, 1], 1].$$

Normalization gives $-4+\texttt{x} : 3$ since there are 3 copies of the sub-term `2nd(x)` – the sub-term occurs directly twice, and once from the two disjoint sub-terms `1st(2nd(x))` and `3rd(2nd(x))`. Furthermore 4 extra constructors are used in the term $3\|x\|$: one to collapse `1st(2nd(x))` and `3rd(2nd(x))` to `2nd(x)` and three constructors for collapsing the three `2nd(x)` terms.

5.2.2 Size Bounds

The size relations needed are linear bounds on the sizes of program expressions, thus the bounds are simply a sum of deconstruction trees plus a constant. This means that termination arguments relying on anything beyond linear size bounds cannot be handled – e.g. it is not possible to show that `halve(double(x))` preserves the size of `x`. This decision is made to keep the (time) complexity of the analysis at a "practical" level. The complexity of LP solvers is more acceptable than solvers for e.g. Pressburger arithmetics. Furthermore, linear bounds on the sizes are sufficient for the level of abstraction used in the size-change graphs: decreases or non-increases.

Definition 5.2.4. (Size Bound)*: A* Size Bound *is a sum of deconstruction trees and a constant,*

$$\mathit{sizeBd} ::= \top \mid \gamma + \sum_{i=1}^{n} \texttt{x}_i : t_i,$$

where the value $\top$ *represents a possibly unbounded size. The normalization, addition and (deconstruction tree) deconstruction operators are extended in the*

natural way to work on size bounds:

$$\begin{array}{ll} \mathit{normalize}: & \mathit{sizeBd} \to \mathit{sizeBd} \\ (+): & \mathit{sizeBd} \times \mathit{sizeBd} \to \mathit{sizeBd} \\ \mathcal{D}_{\mathtt{des}}: & \mathit{sizeBd} \to \mathit{sizeBd} \end{array}$$

$$\begin{array}{lcl} \mathit{normalize}(\top) & = & \top \\ \mathit{normalize}(\gamma + \sum_{i=1}^{n} \mathtt{x}_i : t_i) & = & \gamma + \sum_{i=1}^{n} \mathit{normalize}(\mathtt{x}_i, t_i) \\ b + \top & = & \top \\ \top + b & = & \top \\ (\gamma + \sum_{i=1}^{n} \mathtt{x}_i : t_i) + (\gamma' + \sum_{i=1}^{n} \mathtt{x}_i : t'_i) & = & (\gamma + \gamma') + \sum_{i=1}^{n} \mathtt{x}_i : (t_i + t'_i) \\ \mathcal{D}_{\mathtt{des}}(\top) & = & \top \\ \mathcal{D}_{\mathtt{des}}(0 + \mathtt{x} : t) & = & 0 + \mathtt{x} : (\mathcal{D}_{\mathtt{des}}(t)) \ \textit{if} \ |t| = 1 \\ \mathcal{D}_{\mathtt{des}}(\gamma + \sum_{i=1}^{n} \mathtt{x}_i : t_i) & = & \gamma - 1 + \sum_{i=1}^{n} \mathtt{x}_i : t_i \end{array}$$

Finally, a means of getting from size bounds to actual sizes must be defined.

Definition 5.2.5. (Size Values from Size Bounds)*: Suppose* $b \neq \top$ *and* $\mathit{normalize}(b) = \gamma + \sum_{i=1}^{n} \mathtt{x}_i : c_i$*, then define the semantic operator for size bounds* $\| \cdot \|_{\mathcal{S}\cdot} : \mathit{sizeBd} \to \mathit{Env} \to I\!N$ *by:*

$$\|b\|_{\mathcal{S}}\sigma = \gamma + \sum_{i=1}^{n} c_i \ \|\sigma(\mathtt{x}_i)\|$$

A size bound b *is* safe *for an expression* e *and the environment* σ *if* $b \neq \top$ *implies* $\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq \|b\|_{\mathcal{S}}\sigma$*.*

Lemma 5.2.1. *Suppose* $\gamma + \sum_{i=1}^{n} \mathtt{x}_i : t_i$ *is a size bound then for all* σ*:*

$$\|\gamma + \sum_{i=1}^{n} \mathtt{x}_i : t_i\|_{\mathcal{S}}\sigma = \gamma - (\sum_{i=1}^{n} \mathit{ncons}(t_i)) + (\sum_{i=1}^{n} \mathit{nrep}(t_i) \ \|\sigma(\mathtt{x}_i)\|).$$

Proof. Follows from Definition 5.2.5 and Definition 5.2.3. □

Bounds on the size to the return value of a function are maintained in the *function bound environment.*

Definition 5.2.6. (Function Bound Environment)*: A* function bound environment δ *is a mapping from function names to bounds on the size of the return value. The size bounds (if they exist) are linear upper bounds of the form:*

$$\gamma_{\mathtt{f}} + \sum_{i \in I_{\mathtt{f}}} \|\mathtt{x}_i\|.$$

A function bound environment δ *is defined by*

$$\delta(f) = \begin{cases} \mathit{Lin}(I_f, \gamma_f) & \textit{if } f \textit{ is bounded by } \gamma_f + \sum_{i \in I_f} \|x_i\| \\ \top & \textit{otherwise.} \end{cases}$$

A function bound environment is safe *iff it satisfies:* $\mathcal{E}[\![\mathtt{e}^{\mathtt{f}}]\!]\sigma \neq \bot$ *implies*

$$\|\mathcal{E}[\![\mathtt{e}^{\mathtt{f}}]\!]\sigma\| \leq \|\gamma_{\mathtt{f}} + \sum_{i \in I_{\mathtt{f}}} \mathtt{x}_i : 1\|_{\mathcal{S}}\sigma,$$

for all $v_1, \ldots, v_n \in$ *Value, where* $\sigma = [\mathtt{x}_i \mapsto v_i]\ \forall i = 1 \ldots n$.

Assumption. When performing the size-analysis on a component, it is assumed that that all functions in or below the current component in the call tree have been classified in a function bound environment δ. This gives the information needed to handle function calls in the component.

5.3 Size Analysis

The size analysis is defined in Figure 5.3.1. Due to nested conditionals, it is possible that more than one bound (arising from the different conditional branches) may exist for a given expression. Thus a *set of bounds*, one for each unique path through the conditionals must be collected. This means that there might exist many call sites within a given function call. Consider the following function call.

```
f(if eq(x, Z) then Z else g(1st(x)),
  if eq(y, Z) then Z else g(1st(y)))
```

This single function call contains four different call sites.

```
f(        Z,         Z)
f(g(1st(x)),         Z)
f(        Z, g(1st(y)))
f(g(1st(x)), g(1st(y)))
```

If the function being analyzed is viewed as a set of recurrence equations, then each each equation gives rise to exactly one size bound. This is done because any safe combinator for the bounds would throw away information when combining two bounds. Thus we risk losing precision when combining bounds, so instead the different bounds are viewed as bounds for different recurrence equations. This allows the size-change graph extraction phase to extract a set of more precise size-change graphs from the expression, rather than one less precise combined description. One way to think of this approach is that the size analysis analyzes the program where the conditionals have been pulled out of the context (i.e. "the trick" as known from partiel evaluation has been applied where possible).

To avoid needlessly complicated notation, the subject program is assumed to have been normalized such that all conditionals have been pulled out of context

$$
\begin{array}{ll}
\mathcal{S}[\![\mathtt{x}]\!]\delta & = \mathtt{x} : 1 \\
\mathcal{S}[\![\mathtt{con}]\!]\delta & = 0 \\
\mathcal{S}[\![\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\delta & = \mathit{normalize}(1 + \sum_{i=1}^{n} \mathcal{S}[\![\mathtt{e}_i]\!]\delta) \\
\mathcal{S}[\![\mathtt{des}(\mathtt{e})]\!]\delta & = \mathcal{D}_{\mathtt{des}}(\mathcal{S}[\![\mathtt{e}]\!]\delta) \\
\mathcal{S}[\![\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\delta = & \\
\qquad \begin{cases} \mathit{normalize}(\gamma_f + \sum_{i \in I_f} \mathcal{S}[\![\mathtt{e}_i]\!]\delta) & \text{if } \delta(f) = \mathit{lin}(I_f, \gamma_f) \\ \top & \text{otherwise} \end{cases} & \\
\mathcal{S}[\![\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\delta = 0 &
\end{array}
$$

Figure 5.3.1: Size analysis wrt. the node size-measure.

to form a "decision tree", i.e. function definitions are of the form:

$$
\begin{array}{lcl}
\mathtt{def} \in \mathit{Def} & ::= & \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}_{cond} \\
\mathtt{e}_{cond} \in \mathit{Expr} & ::= & \mathtt{if}\ \mathtt{e}_{cond}\ \mathtt{then}\ \mathtt{e}'_{cond}\ \mathtt{else}\ \mathtt{e}''_{cond} \\
 & | & \mathtt{e} \\
\mathtt{e} \in \mathit{Expr} & ::= & \mathtt{x} \\
 & | & \mathtt{con} \\
 & | & \mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
 & | & \mathtt{des}(\mathtt{e}) \\
 & | & \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
 & | & \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)
\end{array}
$$

With the definition of size analysis in place, it can be proven to satisfy the safety condition.

Proof. of Theorem 5.2.1 The proof is by structural induction over the semantic definitions.

Induction step: Suppose that the theorem holds for any sub expression in the current expression and suppose that $\mathcal{S}[\![\mathtt{e}]\!]\delta \neq \top$

Variables.

$$
\begin{array}{lcl}
\|\mathcal{E}[\![\mathtt{x}]\!]\sigma\| & = & \|\sigma(\mathtt{x})\| \\
 & = & \|\mathtt{x} : 1\|_{\mathcal{S}}\sigma \\
 & = & \|\mathcal{S}[\![\mathtt{x}]\!]\delta\|_{\mathcal{S}}\sigma
\end{array}
$$

Constructors. The safety of zero-ary constructors follow directly from the definitions, so consider a constructor with positive arity.

$$
\begin{array}{lcll}
\|\mathcal{E}[\![\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\sigma\| & & & \\
 & = & \|\mathtt{con}(\mathcal{E}[\![\mathtt{e}_1]\!]\sigma, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\sigma)\| & \text{(by the program semantics)} \\
 & = & 1 + \sum_{i=1}^{n} \|\mathcal{E}[\![\mathtt{e}_1]\!]\sigma\| & \text{(by the definition of the node size-measure)} \\
 & \leq & 1 + \sum_{i=1}^{n} \|\mathcal{S}[\![\mathtt{e}_i]\!]\delta\|_{\mathcal{S}}\sigma & \text{(induction assumption)} \\
 & = & \|1 + \sum_{i=1}^{n} \mathcal{S}[\![\mathtt{e}_i]\!]\delta\|_{\mathcal{S}}\sigma & \text{(addition distributes over normalization)} \\
 & = & \|\mathcal{S}[\![\mathtt{con}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\delta\|_{\mathcal{S}}\sigma & \text{(by the definition of the size analysis)}
\end{array}
$$

Destructors. Starting from the right-hand side:

$$\mathcal{S}[\![\texttt{des(e)}]\!]\delta \quad = \quad \mathcal{D}_{\texttt{des}}(\mathcal{S}[\![\texttt{e}]\!]\delta)$$

Now, two possibilities for $\mathcal{S}[\![\texttt{e}]\!]\delta$ exist, either

1. $\mathcal{S}[\![\texttt{e}]\!]\delta = \gamma + \sum_{i=1}^{n} \texttt{x}_i : t_i$ or
2. $\mathcal{S}[\![\texttt{e}]\!]\delta = \texttt{x} : t$ and $|t| = 1$

Ad 1:

$$\begin{array}{lll}
\|\mathcal{E}[\![\texttt{des(e)}]\!]\sigma\| & & \\
\quad = & \|\mathit{strictapply}(\mathcal{D}[\![\texttt{des}]\!])(\mathcal{E}[\![\texttt{e}]\!]\sigma)\| & \\
\quad = & \|v_{\texttt{des}}\| & (\text{where } \mathcal{E}[\![\texttt{e}]\!]\sigma = \texttt{con}(v_1, \ldots, v_m)) \\
\quad \leq & \sum_{i=1}^{m} \|v_i\| & (\text{since } v_{\texttt{des}} = v_i \text{ for some } i) \\
\quad = & \|\mathit{con}(v_1, \ldots, v_m)\| - 1 & (\text{definition of node size-measure}) \\
\quad = & \|\mathcal{E}[\![\texttt{e}]\!]\sigma\| - 1 & \\
\quad \leq & \|\mathcal{S}[\![\texttt{e}]\!]\delta\|_{\mathcal{S}}\sigma - 1 & (\text{induction assumption}) \\
\quad = & \|\gamma + \sum_{i=1}^{n} x_i : t_i\|_{\mathcal{S}}\sigma - 1 & (\text{assumption for case 1}) \\
\quad = & \|\gamma - 1 + \sum_{i=1}^{n} x_i : t_i\|_{\mathcal{S}}\sigma & (\text{property of substitution and } \mathit{normalize}) \\
\quad = & \|\mathcal{D}_{\texttt{des}}(\gamma + \sum_{i=1}^{n} x_i : t_i)\|_{\mathcal{S}}\sigma & (\text{definition of } \mathcal{D}_{\texttt{des}}) \\
\quad = & \|\mathcal{D}_{\texttt{des}}(\mathcal{S}[\![\texttt{e}]\!]\delta)\|_{\mathcal{S}}\sigma & (\text{assumption for case 1}) \\
\quad = & \|\mathcal{S}[\![\texttt{des(e)}]\!]\delta\|_{\mathcal{S}}\sigma & (\text{by the definition of the size analysis})
\end{array}$$

Ad 2:

$$\begin{array}{lll}
\|\mathcal{E}[\![\texttt{des(e)}]\!]\sigma\| & & \\
\quad = & \|\mathit{strictapply}(\mathcal{D}[\![\texttt{des}]\!])(\mathcal{E}[\![\texttt{e}]\!]\sigma)\| & \\
\quad = & \|v_{\texttt{des}}\| & (\text{where } \mathcal{E}[\![\texttt{e}]\!]\sigma = \texttt{con}(v_1, \ldots, v_m)) \\
\quad \leq & \sum_{i=1}^{m} \|v_i\| & (\text{since } v_{\texttt{des}} = v_i \text{ for some } i) \\
\quad = & \|\mathit{con}(v_1, \ldots, v_m)\| - 1 & (\text{definition of node size-measure}) \\
\quad = & \|\mathcal{E}[\![\texttt{e}]\!]\sigma\| - 1 & \\
\quad \leq & \|\mathcal{S}[\![\texttt{e}]\!]\delta\|_{\mathcal{S}}\sigma - 1 & (\text{induction assumption}) \\
\quad = & \|\texttt{x} : t\|_{\mathcal{S}}\sigma - 1 & (\text{assumption for case 2}) \\
\quad = & \|\texttt{x} : t\|_{\mathcal{S}}\sigma - \mathit{nrep}(t) & (\text{since } |t| = 1 \Rightarrow \mathit{nrep}(t) = 1) \\
\quad = & \mathit{nrep}(t)\|\sigma(\texttt{x})\| - (\mathit{nrep}(t) + \mathit{ncons}(t)) & (\text{Lemma 5.2.1}) \\
\quad = & \mathit{nrep}(0 \triangleright \overbrace{[0, \ldots, 0, t]}^{\mathit{idx}(\texttt{des})})\|\sigma(\texttt{x})\| & \\
 & \quad -\mathit{ncons}(0 \triangleright [0, \ldots, 0, t]) & (\text{definition of } \mathit{ncons}) \\
\quad = & \|x : 0 \triangleright [0, \ldots, 0, t]\|_{\mathcal{S}}\sigma & (\text{Lemma 5.2.1}) \\
\quad = & \|\mathcal{D}_{\texttt{des}}(x : t)))\|_{\mathcal{S}}\sigma & (\text{definition of } \mathcal{D}_{\texttt{des}}) \\
\quad = & \|\mathcal{D}_{\texttt{des}}(\mathcal{S}[\![\texttt{e}]\!]\delta)\|_{\mathcal{S}}\sigma & (\text{assumption for case 2}) \\
\quad = & \|\mathcal{S}[\![\texttt{des(e)}]\!]\delta\|_{\mathcal{S}}\sigma & (\text{by the definition of the size analysis})
\end{array}$$

Function calls. Given by the assumption for the function bound environment.

Primitive Operators. The primitive operators in the laguage are all predicates over terms: $\mathit{Value} \times \mathit{Value} \to \{\texttt{True}, \texttt{False}\}$, thus for all $\forall v_1, v_2 \in \mathit{Value}$ the size of the return value is $\|\texttt{True}\| = \|\texttt{False}\| = 0$.
Thus by induction, the size analysis is safe. □

5.4 Size-Change Graph extraction

Based on the definition of size analysis, it is easy to define an extraction procedure for size-change graphs. To obtain the size-change graphs, the sizes of the arguments in a call must be related to the sizes of the calling function's input. This is done by analyzing the arguments.

Definition 5.4.1. (Size-Change Graph Extraction)*: Suppose* $\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$ *is a function definition in the subject program, and that* δ *is a safe function bound environment. The set of size-change graphs* $\mathcal{G}_f$ *for function* f *is given as follows. For each function call* $n : \mathtt{g}(\mathtt{e}_1,\ldots,\mathtt{e}_m)$ *in* $\mathtt{e}^{\mathtt{f}}$ *let* $s_j = normalize(\mathcal{S}[\![\mathtt{e}_j]\!]\delta)$ *for all* j *and define* $G_n : \mathtt{f} \to \mathtt{g}$ *by*

$$\begin{array}{ll} \mathtt{f}^{(i)} \stackrel{\mp}{\to} \mathtt{g}^{(j)} & \text{if } s_j = \mathtt{x}_i : 1 \\ \mathtt{f}^{(i)} \stackrel{\downarrow}{\to} \mathtt{g}^{(j)} & \text{if } s_j = \mathtt{x}_i : 1-\gamma, \text{ where } \gamma > 0 \end{array}$$

Next we prove that size-change graph extraction is *safe* is defined in [14]. This follows from the safety of the size analysis.

Theorem 5.4.1. (Safety of Size-Change Graph Extraction): *Size-change graph extraction as defined in Definition 5.4.1 is safe.*

Proof. To prove safety of size-change graph extraction, each size-change graph must be proven safe: Suppose $\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$ is a function definition in the subject program, and that δ is a safe function bound environment. Let $n : \mathtt{g}(\mathtt{e}_1,\ldots,\mathtt{e}_m)$ be a function call in $\mathtt{e}^{\mathtt{f}}$ and let the initial environment $\sigma_0 = [\mathtt{x}_i \mapsto v_i^{\forall i=1\ldots arity(\mathtt{f})}]$ be defined by the input $(v_1,\ldots,v_{arity(\mathtt{f})}) \in Value^{arity(\mathtt{f})}$. Then for each $\mathtt{f}^{(i)} \stackrel{r}{\to} \mathtt{g}^{(j)}$ arc in $G_n : \mathtt{f} \to \mathtt{g}$. it must be shown that

1. $r = \mp$ implies $\|\mathcal{E}[\![\mathtt{e}_j]\!]\sigma_0\| \leq \|v_i\|$ and
2. $r = \downarrow$ implies $\|\mathcal{E}[\![\mathtt{e}_j]\!]\sigma_0\| < \|v_i\|$.

Consider the size of the j'th argument in the call $n : f \to g$.

$$\begin{array}{lcll} \|\mathcal{E}[\![\mathtt{e}]\!]\sigma_0\| & \leq & \|\mathcal{S}[\![e_j]\!]\delta\|_{\mathcal{S}}\sigma_0 & & \text{(by theorem 5.2.1)} \\ & = & \begin{cases} \|\mathtt{x}_i : 1\|_{\mathcal{S}}\sigma_0 & \text{if } r = \mp \\ \|-\gamma+\mathtt{x}_i : 1\|_{\mathcal{S}}\sigma_0 & \text{if } r = \downarrow, \text{ where } \gamma > 0 \end{cases} & \text{(by definition 5.4.1)} \\ & = & \begin{cases} \|\sigma_0(\mathtt{x}_i)\| & \text{if } r = \mp \\ -\gamma + \|\sigma_0(\mathtt{x}_i)\| & \text{if } r = \downarrow, \text{ where } \gamma > 0 \end{cases} & \text{(by definition 5.2.5)} \\ & = & \begin{cases} \|v_i\| & \text{if } r = \mp \\ \|v_i\| - \gamma & \text{if } r = \downarrow, \text{ where } \gamma > 0 \end{cases} & \end{array}$$

This clearly satisfies the safety criterion, so by the size-change graph extraction is safe as defined in [14]. □

5.5 Linear bounds on function return values

The problem of finding the linear bounds on a function's return values still remains. One solution is to assume that there exist linear bounds on all functions is the current strongly connected component in the call graph. That is, for each function `f` there exist a constant $\gamma_{\mathtt{f}}$ such that

$$\|\mathcal{E}[\![\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_m)]\!]\sigma\| \leq \|\gamma_{\mathtt{f}} + \sum_{i\in I_{\mathtt{f}}} \mathtt{x}_i : 1\|_{\mathcal{S}}\sigma,$$

for any environment σ where $I_{\mathtt{f}}$ is the set of parameters that the size of the return value of f depends upon. The assumption gives rise to a set of constraints: If the size of the right-hand side (inductively using the assumption) can be bounded by the proposed bound for each recurrence equation, then the bound is a valid bound. Thus any choice of $I_{\mathtt{f}}$ would give either correct bounds or no bounds at all, so any choice is safe. It is possible to search for minimal dependencies – simply generate all subsets of the parameter set and try to satisfy the resulting constraints. A dependency analysis can be used to give a "maximal first guess": if a linear bound exists, then it by definition cannot depend on variables outside of the set computed by the dependency analysis.

The present method uses a dependency analysis to decide, but does not search for other candidates if the constraints cannot be satisfied. This is done for efficiency reasons – clearly the power-set of the parameter set is exponential in the size of the program, so if searching is performed, then the complexity of the analysis would be at least exponential in the size of the input. As a result of this, the analysis might be overly conservative when selecting the set $I_{\mathtt{f}}$: The `minimum` function is size dependent on both its input (it's bounded by $min : I\!N \times I\!N \to I\!N$), but for linear bounds either input suffice. But since any safe dependency analysis returns both input, the bounding function selected by the analysis is the *sum* of the inputs.

Definition 5.5.1. (Extraction of Constraint Systems)*: Suppose δ is a safe function return environment for all functions below the current component $C = \{f_1,\ldots,f_n\}$, $f \in C$ is a function and that $I_{\mathtt{f}_i} \subseteq \{1,\ldots,arity(\mathtt{f}_i)\}$ for all $1 \leq i \leq n$.*

The right-hand sides are extracted by inserting entries in the function bound environment for the functions in the current component, using place-holder variables $\gamma_{\mathtt{f}_i}$ for the unknown constants. Define the initial function return environment δ_0 as:

$$\delta_0 = \delta \cup \bigcup_{i=1}^{n} [\mathtt{f}_i \mapsto lin(I_{\mathtt{f}_i}, \gamma_{\mathtt{f}_i})].$$

The the right-hand sides can be extracted using the size analysis. Suppose R_i is the set of right-hand sides for $\mathtt{f}_i$. Each right-hand side $rhs \in R_i$ defines a constraint:

$$\|\gamma_{f_i} + \sum_{k\in I_{f_i}} f_i^{(k)} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{S}[\![rhs]\!]\delta_{init}\|_{\mathcal{S}}\sigma$$

Now the constraint is normalized – *all terms are isolated on the left-hand side. If all resulting constraints are of the form:*

$$\begin{array}{l}\|(\gamma_{f_i} + \sum_{k \in I_{f_i}} f_i^{(k)} : 1) - (\mathcal{S}[\![rhs]\!]\delta_{init})\|_{\mathcal{S}}\sigma \\ = \gamma_{f_i} + \sum_{k \in I_{f_i}} c_k \ \|f_i^{(k)} : 1\|_{\mathcal{S}}\sigma \geq 0\end{array}$$

where $c_k \geq 0$, then the constraint system S is well defined.

The rationale for the constraints is: before normalization, each left-hand side safely describes the size return value of the function (under the assumption that it is linear) and each right-hand side safely describes the size of the function definition. The normalized constraint form, is restricted to prevent unbounded (program) variables – Suppose $c_k \mathtt{f}_i^{(k)} : 1$ is a term in the constraint such that $c_k < 0$. Since no assumptions are made about the input, the size of the value in the parameter $\mathtt{f}_i^{(k)}$ can be unboundedly large which implies that the corresponding $\gamma_{\mathtt{f}_i}$ can be made arbitrarily large as well.

Example. Consider the inequalities extracted from the `append` program.

```
append(xs, ys) =
  if eq(xs, Nil)
    then ys
    else Cons(1st(xs), append(2nd(xs), ys)
```

Now assume that the return values of `append` is bounded by the size bound $\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1$ for some constant $\gamma_{\mathtt{append}}$:

$$\begin{array}{l}\mathcal{E}[\![\mathtt{append(xs, ys)}]\!]\sigma \neq \bot \quad \text{implies} \\ \|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{E}[\![\mathtt{append(xs, ys)}]\!]\sigma\|\end{array}$$

Evaluation of `append(xs, ys)` must pick either the "then"-branch or the "else"-branch, so the proposed bound must also be a bound for either branch:

$$\begin{array}{l}\|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{E}[\![\mathtt{ys}]\!]\sigma\| \quad \text{and} \\ \|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{E}[\![\mathtt{Cons(1st(xs), append(2nd(xs), ys)}]\!]\sigma\|\end{array}$$

The above inequalities can be satisfied if the bound satisfies:

$$\begin{array}{l}\|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{S}[\![\mathtt{ys}]\!]\delta\|_{\mathcal{S}}\sigma \quad \text{and} \\ \|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{S}[\![\mathtt{Cons(1st(xs), append(2nd(xs), ys)}]\!]\delta\|_{\mathcal{S}}\sigma\end{array}$$

Using the definition of size analysis, the following constraints are obtained:

$$\begin{array}{lcl}\|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma & \geq & \|\mathtt{ys} : 1\|_{\mathcal{S}}\sigma \\ \|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma & \geq & \|1 + (\mathtt{xs} : 0 \triangleright [0, 1]) \\ & & +(\gamma_{\mathtt{append}} + \mathtt{xs} : 0 \triangleright [1] + \mathtt{ys} : 1)\|_{\mathcal{S}}\sigma \\ & = & \|\gamma_{\mathtt{append}} + \mathtt{xs} : 1 + \mathtt{ys} : 1\|_{\mathcal{S}}\sigma\end{array}$$

The normalized constraints are thus:

$$\begin{array}{lcl} \|\gamma_{\texttt{append}} + \texttt{xs} : 1 + \texttt{ys} : 1 - \texttt{ys} : 1\|_{\mathcal{S}}\sigma & \geq & 0 \\ \|\gamma_{\texttt{append}} + \texttt{xs} : 1 + \texttt{ys} : 1 - (\gamma_{\texttt{append}} + \texttt{xs} : 1 + \texttt{ys} : 1)\|_{\mathcal{S}}\sigma & \geq & 0 \end{array}$$

which reduces to the constraint: $\gamma_{\texttt{append}} + \|\texttt{xs} : 1\|_{\mathcal{S}}\sigma \geq 0$. The value of `xs` is unbounded from above, so it is replace by the smallest value it can assume (which makes it harder to satisfy the constraint): $\gamma_{\texttt{append}} \geq 0$.

Goal Function. Now it remains to find a solution satisfying the constraint system. Clearly we wish to minimize the associated constants to obtain the tightest bounds. A simple way to do this is to minimize the sum of the constants. This might not be the optimal minimization objective – in some cases it might be more important to get a tight bound on a particular function, rather than a "a good bound" on all the functions.

Note that the variables associated with the input to the functions are unbounded from above. That implies that if we're unable to cancel all such variables out on the right-hand side, then it would be impossible to find a bound that would hold for all input. The lower bound is given by the node size-measure: 0. To obtain the normalized constraint form for the LP solver, all gamma-variables are isolated on the left-hand side and the remaining terms are collected on the right-hand side. If all terms involving variables associated with program variables are bounded by 0 (have been canceled out, or appear with a negative sign), then they can be substituted for the lower bound on the node size-measure.

Continuing the example. We obtained the constraint $\gamma_{\texttt{append}} \geq 0$ using the size analysis, so finally the bound can be solved for: minimize the place-holder variable $\gamma_{\texttt{append}}$ for the constant subject to the constraint $\gamma_{\texttt{append}} \geq 0$, which gives the solution: $\gamma_{\texttt{append}} = 0$. Thus the size of the return value of append is bounded by the size bound: $\texttt{xs} : 1 + \texttt{ys} : 1$:

$$\begin{array}{l} \mathcal{E}[\![\texttt{append}(\texttt{xs}, \texttt{ys})]\!]\sigma \neq \bot \qquad \text{implies} \\ \|\texttt{xs} : 1 + \texttt{ys} : 1\|_{\mathcal{S}}\sigma \geq \|\mathcal{E}[\![\texttt{append}(\texttt{xs}, \texttt{ys})]\!]\sigma\| \end{array}$$

Theorem 5.5.1. *Suppose the current component is* $\{\texttt{f}_1, \ldots \texttt{f}_n\}$ *and that* $S = \{c_1, \ldots, c_k\}$ *is the corresponding constraint system as defined in Definition 5.5.1. If* $\Gamma = \{\gamma_{\texttt{f}_1}, \ldots, \gamma_{\texttt{f}_n}\}$ *is the integer solution to the LP problem:*

$$\begin{array}{ll} \textit{minimize} & \gamma_{\texttt{f}_1} + \ldots + \gamma_{\texttt{f}_n} \\ \textit{subject to} & c_1, \ldots, c_k \end{array}$$

then for all i, $\gamma_{\texttt{f}_i} + \sum_{i \in I_{\texttt{f}_i}} \texttt{f}^{(i)} : 1$ *is a safe bound for the return value of* $\texttt{f}_i$*:*

$$\|\mathcal{E}[\![\texttt{e}^{\texttt{f}_i}]\!]\sigma\| \leq \gamma_{\texttt{f}_i} + \sum_{k \in I_{\texttt{f}_i}} \|v_k\|$$

for all $v_1, \ldots, v_{arity(\texttt{f}_i)} \in$ *Value, where* $[f_i^{(k)} \mapsto v_k]$ $\forall k = 1 \ldots arity(\texttt{f}_i)$.

Proof. Proof by induction over the computation — Suppose given the computation tree for a particular input. Correctness of the size analysis provides that the condition is satisfied for the base cases – i.e. the left-hand sides that do not call functions in the component. Now consider the parent node in the computation tree; Again, since the size analysis is safe, the proposed linear bound must also bound the size of return value of the call in the parent node. Thus by induction, the bound must also bound the root node, i.e. the bound bounds the return value of the computation. □

Chapter 6

Limiting SCCs

In this work the main concern is to decide whether there exists a polynomial bound on the running time; If a program satisfies SCT then there exists a bound on the running time for a given input, but further restrictions (analyses) are needed conservatively to decide whether a program computes a function in PTIMEF, or is not analyzable.

An essential observation is that not all parameters have an influence on the runtime bound of a program.

Definition 6.0.2. (Critical Parameters)*: Suppose $\mathcal{G} = \{G_1, \ldots, G_n\}$ is a set of size-change graphs. The restriction of $\mathcal{G}$ to the parameters $\mathcal{P} = \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$, denoted $\mathcal{G}|_{\mathcal{P}}$, is the set of size-change graphs that results from removing all parameters not in $\mathcal{P}$ from the graphs in $\mathcal{G}$.*

A critical set of parameters *$\mathcal{P}$ for a size-change terminating program π with the size-change graphs $\mathcal{G}$, is a set of parameters such that $\mathcal{G}|_{\mathcal{P}}$ is size-change terminating, i.e. π is size-change terminating when only parameters in $\mathcal{P}$ are considered.*

A critical parameter is a parameter that occurs in a critical set and a critical argument position is an expression which is passed to a critical parameter.

Lemma 6.0.1. *If π is a size-change terminating program with size-change graphs $\mathcal{G}$, there exists a set $\mathcal{P}$ of critical parameters.*

Proof. Simply select $\mathcal{P}$ to be *all* parameters. □

The following shows, however, that there might not exist a unique *smallest* set of critical parameters. Consider the minimum function:

```
min(x, y) = if eq(x, Z) then y
              else if eq(y, Z) then x
                else S(min(1st(x), 1st(y)))
```

Clearly the sets $\{\mathtt{x}, \mathtt{y}\}$, $\{\mathtt{x}\}$ and $\{\mathtt{y}\}$ are all critical sets of parameters, since size-change termination can be proven for each of the sets. Intuitively, the critical parameters enforce a bound on the maximal depth of a call sequence in that component.

If the critical parameters can be controlled, then the call depth can be controlled, so smaller sets of critical parameters generally will require fewer restrictions on the program behavior. But smaller sets also means that some precision on the bound on the call depth is lost, which is the case for the `min` function; Clearly $min(x, y)$ is a better bound on the call depth of `min` than x or y, but since the interest is in *polynomial bounds*, it is not an issue to use smaller sets.

The first step to restricting program behaviors, is inspired from a result from *tuple descent systems.*

6.1 Tuple Descent Systems

Definition 6.1.1. (Tuple Descent System*): A* tuple descent system *is a finite set of rewrite rules R of the form $(\mathtt{x}_1, \ldots, \mathtt{x}_n) \to (\mathtt{y}_1, \ldots \mathtt{y}_m)$ where $j = 1 \ldots m$, there is an $i = 1 \ldots n$ such that $\mathtt{y}_j = \mathtt{x}_i$ or $\mathtt{y}_j = \mathtt{x}_i - 1$.*

A rule $(\mathtt{x}_1, \ldots, \mathtt{x}_n) \to (\mathtt{y}_1, \ldots \mathtt{y}_m)$ induces the relation $\Rightarrow \subseteq I\!N^n \times I\!N^m$ where $(u_1, \ldots, u_n) \Rightarrow (v_1, \ldots v_m)$ iff for $i = 1, \ldots, m$ either

- $\mathtt{y}_j = \mathtt{x}_i$ *and* $v_j = u_i$ *or*
- $\mathtt{y}_j = \mathtt{x}_i - 1$, $v_i > 0$ *and* $v_j = u_i - 1$.

Suppose $u_0 = (v_{0,1}, \ldots, v_{0,n}) \in I\!N^n$, then a tuple descent system R generates *the sequence of tuples $u_0, \ldots, u_k$ if for each $0 \leq l < k$: $u_i \Rightarrow u_{i+1}$ as defined by R and $v_{k,j} = 0$ for some j.*

A tuple descent system is terminating *iff it does not generate infinite tuple sequences.*

Lemma 6.1.1. *Suppose R is a terminating tuple descent system, then there exists a polynomial Π such that if R generates the sequence $(u_{0,1}, \ldots, u_{0,n}), \ldots, (u_{k,1}, \ldots, u_{k,n})$, then the length of the sequence is bounded by: $k \leq \Pi(u_{0,1}, \ldots, u_{0,n})$.*

Proof. Let the tuple sequence $u_0, \ldots, u_k$ be the sequence generated by R.

Suppose that $u_i = u_j$ for some i and j, then R can generate the infinite sequence $u_i \Rightarrow \ldots \Rightarrow u_{j-1} \Rightarrow u_i \Rightarrow \ldots \Rightarrow \ldots u_{j-1} \Rightarrow u_i \Rightarrow \ldots$, which violates the assumption that R is terminating. Thus $u_i \neq u_j$ for all $i \neq j$. Any given element $u_{i,k}$ in a tuple is bounded by $max\{u_{0,1}, \ldots, u_{0,n}\}$, since the elements by definition only can be copied or decreased from values in the previous tuple. Thus the number of different tuples is bounded by $(max\{u_{0,1}, \ldots, u_{0,n}\})^n = \Pi(u_{0,1}, \ldots, u_{0,n})$. □

6.2 Input bounded parameters

If a terminating set of size-change graphs for a component C behaves like a terminating tuple descent system, then by Lemma 6.1.1 there will exist a polynomial bound on the length of any state transition sequence that only uses functions $\mathtt{f} \in C$ – i.e. the call depth is bounded by a polynomial.

Definition 6.2.1. (Input bounded parameters)*: Suppose $\mathcal{G}$ is a set of size-change graphs. A parameter $\mathtt{g}^{(j)}$ is* input bounded *iff it is* backwards thread preserving*:*

$$\forall (G : \mathtt{f} \to \mathtt{g}) \in \mathcal{G} : \ \exists i : \ \mathtt{f}^{(i)} \stackrel{\downarrow}{\to} \mathtt{g}^{(j)} \vee \mathtt{f}^{(i)} \stackrel{\mp}{\to} \mathtt{g}^{(j)}$$

A set of parameters is input bounded for $\mathcal{G}$ if each parameter in the set is input bounded. The set of all input bounded $\mathtt{f}$-parameters for $\mathcal{G}$ is denoted $ibd(\mathtt{f})$ [1].

The set of size-graphs $\mathcal{G}|_{\mathcal{P}}$ corresponds to a tuple descent system when restricted to the input bounded parameters $\mathcal{P}$ in the sense that there exists a tuple descent system R such that the length of any state transition sequence given input $\vec{v}$ can be bounded by the length of of any tuple sequence generated by R and $\vec{v}$.

Lemma 6.2.1. *Suppose π is a size-change terminating program with size-change graphs $\mathcal{G}$ describing the functions in the component C. Suppose $\mathcal{P}$ is a set of input bounded parameters for the size-change graphs $\mathcal{G}$ and let n be a function for the length of longest possible state transition sequence in C.*

Suppose the maximal arity of the functions is k; Define the tuple descent system R as follows. For each $(G : \mathtt{f} \to \mathtt{g}) \in \mathcal{G}$: $(x_1, \ldots, x_k) \to (y_1, \ldots, y_k) \in R$ where

$$y_j = \begin{cases} x_i & \text{if } (\mathtt{f}^{(i)} \stackrel{\mp}{\to} \mathtt{g}^{(j)}) \in G \\ x_i - 1 & \text{if } (\mathtt{f}^{(i)} \stackrel{\downarrow}{\to} \mathtt{g}^{(j)}) \in G \end{cases}$$

Then for all input $v_1, \ldots, v_k$, the length of any tuple sequence generated by R, $(v_1, \ldots, v_k) \Rightarrow \ldots$, is an upper bound for $n(v_1, \ldots, v_k)$.

Proof. Follows by a simple induction over the length of the sequence generated by R using Definition 5.4.1 and Definition 6.2.1. □

Corollary 6.2.1. *Suppose $\mathcal{P}$ is a critical input bounded set of parameters for the size-change graphs $\mathcal{G}$. Then the length of any call sequence admissible by $\mathcal{G}$ is bounded by a polynomial*

Proof. Lemmas 6.1.1 and 6.2.1 give a polynomial bound on the call depth of $\mathcal{G}|_{\mathcal{P}}$ which implies a polynomial bound on the call depth of $\mathcal{G}$. □

Definition 6.2.2. (Critical Input Bounded Environment)*: A* critical input bounded environment *$\rho : FcnName \to I\!N^*$ is a mapping from function names to an index set for parameters that are both critical and input bounded, i.e. a parameter $i \in \rho(\mathtt{f})$ implies $\mathtt{f}^{(i)}$ is critical and $\mathtt{f}^{(i)}$ is input bounded.*

[1] Indices make it easier to go from a parameter to the corresponding argument position.

Chapter 7

Disjointness

It is possible that the call depth can be bounded by a polynomial, but the call tree of the recursive calls might *branch exponentially*. An example of this is the Fibonacci function.

```
fib(n) = if or(eq(n, Z), eq(1st(n), Z)
           then S(Z)
           else add(f(1st(n)), f(1st(1st(n))))
add(x, y) = if eq(x, Z) then y else S(add(1st(x), y)))
```

One way to restrict the recursive calls is to require that all recursive calls are *linear*, i.e. that each recurrence equation has at most one recursive call. This is the approach taken by the Bellantoni-Cook syntactic criterion for PTIMEF. However, this restriction excludes the entire "divide and conquer" programming paradigm, so a less restrictive formulation is desired.

The linearity requirement can be extended to handle non-linear recursion provided that the same value is not "inspected more than once" – The guiding principle is that any two recursive calls in the same recurrence equation must not share the same data structure in a critical parameter. This is precisely what occurs in the Fibonacci function: the two recursive calls both use the value `1st(1st(n))`, so there is a danger of exponential branching in the call tree. On the other hand, we would like to accept function that invoke the recursive calls with "disjoint" parameters, e.g. recursive calls like `f(1st(x))` and `f(2nd(x))` would be permitted since `1st(x)` and `2nd(x)` are *different parts of the input*. This in turn enables less conservative handing of "divide and conquer" algorithms.

This type of information is already being traced in the size analysis in the form of deconstruction trees. Our approach to disjointness is essentially to use size analysis to track precise size information where possible, and to fall back on a dependency analysis for functions with no exact size description. The modified dependency analysis is given in Figure 7.0.1. The analysis is quite similar to the size analysis – the differences are: the way function calls are handled, and that constants are not important. The analysis essentially a size analysis that

$edep[\![\mathtt{x}]\!]dep$	$= \mathtt{x} : 1$
$edep[\![\mathtt{con}]\!]dep$	$= 0$
$edep[\![\mathtt{con}(\mathtt{e}_1,\ldots,\mathtt{e}_n)]\!]dep$	$= normalize(\sum_{i=1}^{n} edep[\![\mathtt{e}_i]\!]dep)$
$edep[\![\mathtt{des}(\mathtt{e})]\!]dep$	$= \mathcal{D}_{\mathtt{des}}(edep[\![\mathtt{e}]\!]dep)$
$edep[\![\mathtt{f}(\mathtt{e}_1,\ldots,\mathtt{e}_n)]\!]dep$	$= normalize(\sum_{i\in dep(f)} edep[\![\mathtt{e}_i]\!]dep)$
$edep[\![\mathtt{op}(\mathtt{e}_1,\ldots,\mathtt{e}_n)]\!]dep$	$= 0$

Figure 7.0.1: Extended dependency analysis.

falls back on dependency information as captured by the ordinary dependency analysis when functions are called. The analysis can in fact be implemented by running the size-analysis with a initial function bound environment where all functions are "linear" in the parameters they depend upon:

$$\delta_0 = [\mathtt{f} \mapsto lin(dep(\mathtt{f}), 0)^{\forall \mathtt{f}}].$$

The key property for *edep* is that is any expression $\mathtt{e} \rightsquigarrow \mathtt{f}^{(i)}$ then $\mathtt{f}^{(i)}$ is represented in the "size bound" returned by *edep*: $dep(\mathtt{e}) = I$ implies $edep[\![\mathtt{e}]\!]dep = \gamma + \sum_{i\in I} \mathtt{f}^{(i)} : t_i$, where $normalize(t_i) \leq 1$.

Next we define disjointness for the dependencies returned by *edep* – two expressions are disjoint if they do not depend on the same substructure of a value:

Definition 7.0.3. (Dependency Disjointness*): Two size bounds* $\gamma + \sum_{i=1}^{n} x_i : t_i$, $\gamma' + \sum_{i=1}^{n} x_i : t'_i$ *are* disjoint*:*

$$\gamma + \sum_{i=1}^{n} x_i : t_i \lll \gamma' + \sum_{i=1}^{n} x_i : t'_i$$

if for all i *the trees* t_i, t'_i *do not overlap. The deconstruction tree* t *overlaps with* t', $t \gg t'$, *if* $t = n$ *or* $t = n \triangleright \{\ldots\}$ *for* $n > 0$ *implies* $t' = 0$ *and* $t = 0 \triangleright \{t_1, \ldots, t_n\}$ *and* $t' = 0 \triangleright \{t'_1, \ldots, t'_m\}$ *implies* $t_i \gg t'_i$ *for all* i*. Two deconstruction trees* t *and* t' *do not overlap, provided that* $t \gg t'$ *and* $t' \gg t$*.*

Disjointness is employed to ensure that non-linear functions completely split the input between the different recursive calls. For any two recursive calls in the same recurrence equation, the argument positions must be disjoint.

Definition 7.0.4. (Proper split*): Suppose* $\mathtt{e}$, $\mathtt{e}'$ *are critical input bounded arguments such that* $edep[\![\mathtt{e}]\!]dep = \gamma + \mathtt{x} : t$, $edep[\![\mathtt{e}']\!]dep = \gamma' + \mathtt{x} : t'$*, The arguments* $\mathtt{e}$, $\mathtt{e}'$ properly split $\mathtt{x}$ *if:*

$$\begin{array}{rcl} \|edep[\![\mathtt{e}]\!]dep + edep[\![\mathtt{e}']\!]dep\|_{\mathcal{S}}\sigma & = & \|\gamma + \gamma' + \mathtt{x} : (t + t')\|_{\mathcal{S}}\sigma \\ & \leq & \|\gamma + \gamma' + \mathtt{x} : 1\|_{\mathcal{S}}\sigma \end{array}$$

Lemma 7.0.2. *Suppose* $\mathtt{e}$, $\mathtt{e}'$ *are critical input bounded arguments such that* $edep[\![\mathtt{e}]\!]dep = \gamma + \mathtt{x} : t$ *and* $edep[\![\mathtt{e}']\!]dep = \gamma' + \mathtt{x}' : t'$ *are disjoint:* $\gamma + \mathtt{x} : t \lll \gamma' + \mathtt{x} : t'$ *then either* $\mathtt{x} \neq \mathtt{x}'$ *or* $\mathtt{e}$, $\mathtt{e}'$ *split* $\mathtt{x}$ *properly.*

Finally we can use the definition of dependency disjointness to define the call disjointness criterion: the recursive calls are disjoint if all arguments passed in the critical arguments positions are disjoint for any two critical argument positions in two different recursive function calls.

Definition 7.0.5. (Call Disjointness*): Suppose C is a component in the call graph and* $\mathtt{f} \in C$ *is a function where the recursive calls* $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, $\mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_n)$ *where* $\mathtt{g}, \mathtt{h} \in C$*. The recursive calls in a component C are disjoint iff any two critical input bounded argument positions,* $\mathtt{e}_i$, $\mathtt{e}'_j$ *are disjoint:* $edep[\![\mathtt{e}_i]\!]dep \ggg edep[\![\mathtt{e}'_j]\!]dep$

Theorem 7.0.1. *Suppose C is a component such that the recursive calls in C are disjoint, then any function* $\mathtt{f} \in C$ *is* polynomially branching*, i.e. for any input the number of recursive calls to functions in C is bounded by a polynomial.*

Proof. The recursive calls are disjoint, so let $\mathtt{e}_1, \ldots, \mathtt{e}_n$ be any critical arguments for different function calls $\mathtt{g}_1, \ldots, \mathtt{g}_n \in C$, such that $edep[\![\mathtt{e}_i]\!]dep = \gamma_i + \mathtt{x} : t_i$. All arguments are pairwise disjoint, so by Lemma 7.0.2, it can be shown that

$$\begin{array}{rcl} \| \sum_{i=1}^{n} edep[\![\mathtt{e}_i]\!]dep \|_{\mathcal{S}}\sigma & \leq & \gamma' + \|\mathtt{x} : t_i\|_{\mathcal{S}}\sigma \\ & = & \gamma' + \|\sigma(\mathtt{x})\| \end{array}$$

for some constant γ'. Thus each call to the branch, will spilt the value in $\mathtt{x}$ between the recursive calls $\mathtt{g}_1, \ldots, \mathtt{g}_n$, with no increase in the sum of the size of all the $\mathtt{x}$'s. The value in $\mathtt{x}$ can then be split at most $\|\sigma(\mathtt{x})\|$ times over any call tree, since splitting cannot increase the sum of the $\mathtt{x}$-arguments . Thus the splitting recurrence equation can only be called polynomially many times, before a new value is copied into the $\mathtt{x}$-parameter (reset), so the number of branches that can be created is linear in the size of $\mathtt{x}$.

The component is terminating when restricted to the input bounded parameters, so the calls depth is bounded by a polynomial by Corollary 6.2.1. Therefore, any value can only be "reset" polynomially many times, so the *total* number of splits in the call tree, is bounded by a polynomial in the original input.

As the call depth of each branch in the call tree is bounded by a polynomial and the number of branches is polynomial, the total number of function calls must by bounded by a polynomial as well – i.e. the running time is bounded by a polynomial. □

Chapter 8

Bounding Sub-Function Calls

Using the criteria defined in the previous sections, it can be determined whether a component terminates or not and whether the number of recursive calls in the component is bounded by a polynomial or not. The remaining step to prove that the running time is bounded by a polynomial, is to ensure that calls to the sub-functions for the component can at most give rise to polynomially many recursive calls in the sub-function.

Simply restricting the number of recursive calls in a component is not enough to ensure PTIMEF. This is illustrated by the following examples.

```
exp1(x, y) = if eq(n, Z)
               then y
               else exp1(1st(x), double(y))

exp2(n) = if eq(n, Z)
            then S(Z)
            else double(exp2(1st(n)))

double(x) = if eq(x, Z) then Z else S(S(double(1st(x))))
```

While both `exp1` and `exp2` only execute polynomially many recursive calls they still compute an exponentially large value. The problem is that polynomial iteration of a call to a sub-function (`double`) can lead to exponentially many recursive calls in the called function.

The key to controlling the number of calls – i.e. the running time – is to restrict the values that can be passed as an argument in a critical argument position. Caseiro [3], used "DDC" systems to prevent critical argument positions from doubling their size in the recursion. We will use a conceptually similar approach using the following guiding principle: any inside or outside call will

terminate in time polynomial in the size of the input for the *component* if the arguments passed to its *critical* argument positions are bounded by a polynomial in the size of the input.

Our approach consists of two phases – first the parameters in the current component are classified by whether they *might* "double" their value over a call sequence or whether they exhibit "constant growth" ; Parameters of "constant growth" are guaranteed to be bounded by a polynomial in the (original) input to the program so they can safely be passed in a critical argument position without causing exponential blowup. The set of "constant growth" parameters is a relaxation in comparison to input bounded parameters – the idea is to allow certain types of controlled growth.

The second phase checks all parameters passed to *any* sub-function (including indirect function calls), are of "constant growth". If that is not the case, the called sub-function could potentially give rise to exponentially many recursive calls.

8.1 Doubling

A parameter cannot be doubled in a recursion if all argument positions for the parameter can be bounded by a linear bound: $\gamma + \sum_{i=1}^{m} \mathtt{x}_i : 1$. Furthermore, it must be ensured that at most one parameter $\mathtt{x}_i$ can depend on the parameter corresponding to the argument position – this prevents passing the same argument twice, which is illustrated by the program:

```
f(x) = if ... then ... else g(x, x)
g(y, z) = if ... then ... f(Cons(y, z))
```

The parameter `x` can be doubled in the recursion, since both `y` and `z` depend on `x` in the recursive call $\mathtt{g} \to \mathtt{f}$.

Any variable for which the above cannot be established, is considered "dangerous" because it might lead to super-polynomial computation and any parameters that might depend on such dangerous parameters, are considered dangerous themselves.

Definition 8.1.1. (Doubling*): Suppose there exists a function* $\mathtt{f} \in C$ *such that* $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ *is a function call in the body of* `f`*, then define* $dbl : FcnName \to Variable$ *by:*

- *Suppose* $\mathcal{S}[\![\mathtt{e}_i]\!]\delta = \gamma_{\mathtt{e}_i} + \sum_{j=1}^{m} \mathtt{f}^{(j)} : t_j$*, and let* $j_1, \ldots, j_k \in \{1, \ldots, m\}$ *be the indices for which* $\mathtt{f}^{(j_l)} \rightsquigarrow \mathtt{g}^{(i)}$*. If* $k > 1$ *or* $normalize(t_{j_l}) > 1$ *then* $\mathtt{g}^{(i)} \in dbl(\mathtt{g})$*.*
- *If* $\mathtt{g}^{(i)} \in dbl(\mathtt{g})$ *and* $\mathtt{f}^{(j)} \rightsquigarrow \mathtt{g}^{(i)}$ *then* $\mathtt{f}^{(j)} \in dbl(\mathtt{f})$*.*

The main implication for non-dangerous parameters is that they can be bounded by a polynomial in the input.

Lemma 8.1.1. *Suppose* $\mathtt{f}$ *is a function and* $v_1, \ldots, v_m \in$ *Value is the input. Let* $\sigma_0 = [\mathtt{f}^{(j)} \mapsto v_j]\ \forall j = 1 \ldots m$*, and suppose* $(\mathtt{f}, \sigma_0) \to^* (\mathtt{f}, \sigma)$ *then* $\mathtt{g}^{(i)} \notin dbl(\mathtt{g})$ *implies* $\|\sigma(\mathtt{g}^{(i)})\| \leq \Pi(max\{\|v_1\|, \ldots, \|v_m\|\})$ *for all* i*.*

Proof. Proof by induction over the computation sequence. The base case is trivially satisfied. First consider the size-change behavior of the arguments between two g-states

$$(\mathtt{g}, \sigma) = (\mathtt{f}_0, \sigma_0) \to \ldots \to (\mathtt{f}_{k-1}, \sigma_{k-1}) \to (\mathtt{f}_k, \sigma_k) = (\mathtt{g}, \sigma'),$$

where $\mathtt{f}_l \neq \mathtt{g}$ for all $1 < l < k$. We wish to prove that the parameter $\mathtt{g}^{(i)}$ at most can increase by adding a polynomial in the input. Let $v_{max} = max\{\|v_1\|, \ldots, \|v_m\|\}$.

By assumption the state transition sequence satisfies: $\mathtt{f}_l^{(i)} \notin dbl(\mathtt{f}_l)$ implies $\|\sigma(\mathtt{f}_l^{(i)})\| \leq \Pi(max\{\|v_1\|, \ldots, \|v_m\|\})$ for all i.

Suppose that $\mathtt{g}^{(i)} \notin dbl(\mathtt{g})$ for all i and consider the call in $\mathtt{f}_{k-1}$: $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$.

Suppose J be a set of indices such that $i \in J$ iff $\mathtt{e}_i \rightsquigarrow \mathtt{f}_{k-1}^{(j)}$. Observe that

$$\begin{array}{rcl} \|\mathcal{E}[\![\mathtt{e}_i]\!]\sigma_{k-1}\| & \leq & \|\mathcal{S}[\![\mathtt{e}_i]\!]\delta\|_{\mathcal{S}}\sigma_{k-1} \\ & = & \|\gamma_{\mathtt{e}_i} + \sum_{j \in J_{k-1}} \mathtt{f}_{k-1}^{(j)} : 1\|_{\mathcal{S}}\sigma_{k-1} \end{array}$$

Now two possibilities exist: either

- there is no self-dependency on $\mathtt{g}^{(i)}$: $\nexists \alpha_{k-1} \in J_{k-1}$: $\mathtt{f}_{k-1}^{(\alpha_{k-1})} \rightsquigarrow \mathtt{g}^{(i)}$ or
- there is a self-dependency on $\mathtt{g}^{(i)}$: $\exists! \alpha_{k-1} \in J_{k-1}$: $\mathtt{f}_{k-1}^{(\alpha_{k-1})} \rightsquigarrow \mathtt{g}^{(i)}$.

Ad 1.

$$\begin{array}{l} \|\gamma_{\mathtt{e}_i} + \sum_{j \in J_{k-1}} \mathtt{f}_{k-1}^{(j)} : 1\|_{\mathcal{S}}\sigma_{k-1} \\ \leq (\gamma_{\mathtt{e}_i} + \sum_{j \in J_{k-1}} \|\sigma_{k-1}(\mathtt{f}_{k-1}^{(j)})\|) \\ \leq (\gamma_{\mathtt{e}_i} + \sum_{j \in J_{k-1}} \Pi(v_{max})) \\ = \Pi(v_{max}) \end{array}$$

In which case the claim follows directly by induction over the state transition sequence.

Ad 2. Let the index $\alpha_{k-1} \in J_{k-1}$: $\mathtt{f}_{k-1}^{(\alpha_{k-1})} \rightsquigarrow \mathtt{g}^{(i)}$ and let $\overline{J_{k-1}} = J_{k-1} \setminus \{\alpha_{k-1}\}$.

$$\begin{array}{l} \|\gamma_{\mathtt{e}_i} + \mathtt{f}_{k-1}^{(\alpha_{k-1})} : 1 + \sum_{j \in \overline{J_{k-1}}} \mathtt{f}_{k-1}^{(j)} : 1\|_{\mathcal{S}}\sigma_{k-1} \\ \leq \|\sigma_{k-1}(\mathtt{f}_{k-1}^{(\alpha_{k-1})})\| + (\gamma_{\mathtt{e}_i} + \sum_{j \in \overline{J_{k-1}}} \|\sigma_{k-1}(\mathtt{f}_{k-1}^{(j)})\|) \\ \leq \|\sigma_{k-1}(\mathtt{f}_{k-1}^{(\alpha_{k-1})})\| + (\gamma_{\mathtt{e}_i} + \sum_{j \in \overline{J_{k-1}}} \Pi(v_{max})) \\ = \|\sigma_{k-1}(\mathtt{f}_{k-1}^{(\alpha_{k-1})})\| + \Pi(v_{max}) \\ = \|\mathcal{E}[\![\mathtt{e}_{\alpha_{k-2}}]\!]\sigma_{k-2}\| + \Pi(v_{max}) \\ \leq \|\mathcal{S}[\![\mathtt{e}_{\alpha_{k-2}}]\!]\delta\|_{\mathcal{S}}\sigma_{k-2} + \Pi(v_{max}) \\ = \|\gamma_{\mathtt{e}_{\alpha_{k-2}}} + \sum_{j \in J_{k-2}} \mathtt{f}_{k-2}^{(j)} : 1\|_{\mathcal{S}}\sigma_{k-2} + \Pi(v_{max}) \end{array}$$

where $\mathtt{f}_{k-1}(\mathtt{e}'_1, \ldots, \mathtt{e}'_n)$ is the previous call in $\mathtt{f}_{k-2}$. Note that $\mathtt{f}_{k-1}^{(\alpha_{k-1})} \leadsto \mathtt{g}^{(i)}$ implies $\mathtt{f}_{k-2}^{(\alpha_{k-2})} \leadsto \mathtt{g}^{(i)}$, so the fact that $\mathtt{e}_{\alpha_{k-2}}$ is bounded by $\mathtt{f}_{k-2}^{(\alpha_{k-2})} : 1 + \ldots$ implies $\exists! \alpha_{k-2} : \mathtt{f}_{k-2}^{(\alpha_{k-2})} \leadsto \mathtt{g}^{(i)}$.

Thus by induction there exists a polynomial Π such that

$$\begin{aligned} \|\sigma_k(\mathtt{g}^{(i)})\| &= \|\sigma_k(\mathtt{f}_k^{(\alpha_k)})\| \\ &\leq \|\sigma_1(\mathtt{f}_1^{(\alpha_1)})\| + \Pi(v_{max}) \\ &= \|\sigma_1(\mathtt{g}^{(i)})\| + \Pi(v_{max}) \end{aligned}$$

Arbitrary state transition sequences. Now suppose $(\mathtt{f}, \sigma_0) \rightarrow^* (\mathtt{g}, \sigma)$ is a state transition sequence. The sequence can be broken up into subsequences such that: $(\mathtt{f}, \sigma_0) \rightarrow^* (\mathtt{f}'_1, \sigma'_1) \rightarrow^* \ldots \rightarrow^* (\mathtt{f}'_n, \sigma'_n)$ such that $\mathtt{f}'_l = \mathtt{g}$ and this does not hold for any states in between. By a simple induction it can be shown that: $\|\sigma(\mathtt{g}^{(i)})\| \leq \|\sigma'_1(\mathtt{g}^{(i)})\| + \Pi(v_{max})$, so it follows from the inductive assumption that

$$\|\sigma(\mathtt{g}^{(i)})\| \leq \Pi'(v_{max}),$$

for some polynomial Π'. □

8.2 Definition of Safety for Sub-Calls

The second phase restricts how arguments can be passed to critical argument positions in sub-functions. The criterion needed to ensure that the sub-function terminates in polynomial time in the size of the input to the calling, depends on where the call syntacticly occurs relative to recursive calls – Function calls are thus classified with respect to their occurrence in the component relative to the *recursive calls*.

Definition 8.2.1. (Types of Function Calls)*: Suppose C is a strongly connected component in the call graph and let $\mathtt{f} \in C$.*

A function call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ is a recursive call *if $\mathtt{g} \in C$.*

A function call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ is an inside call *relative to C iff there exist a function call $\mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_{n'})$ in the body of $\mathtt{f}$ where $\mathtt{h} \in C$ such that: $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ occurs in $\mathtt{e}'_i$ for some i.*

A function call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ in the body of $\mathtt{f}$ is an outside call *relative to C iff there exist a function call $\mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_{n'})$ in $\mathtt{e}_i$ for some i such that $\mathtt{h} \in C$.* [1]

A sub-function call that is neither an inside nor outside call is called a base call.

By assumption, restricting the critical arguments for all calls to sub-function, the sub-function can be guaranteed to termination in polynomial time. This, however, must also be ensured for all *indirect* sub-function calls, as illustrated by the (slightly contrived) program:

[1] Note that a given function call can be *both* an inside call and an outside call.

```
f(x, y) =
  if eq(x, Z)
    then Nil
    else Cons(ndup(x, y), f(1st(x), Cons(x, x)))

-- List of n Copies of a Mirrored Tree
ndup(n, c) =
  if eq(n, Z)
    then Nil
    else Cons(mirror(c), ndup(1st(n), c))

-- Mirror a Tree
mirror(t) =
  if eq(t, Nil)
    then Nil
    else Cons(mirror(2nd(t)), mirror(1st(t)))
```

The important factor is that only parameter `n` of the function `ndup` is critical and input bounded. In function `f`, only parameters of "constant growth" are passed to `ndup`'s critical argument position. Yet, `f` computes an exponentially large value, since the `y` doubles it's (node size-measure) size in each recursive call. The problem is that the "dangerous" parameter `y` is passed *indirectly* to `mirror`. Thus it is necessary not only to restrict argument positions which are critical for the called function but also any argument positions whose values can be passed *indirectly* to a sub-function. For this reason, we define the notion of a bounding parameter to keep track of parameters that might be passed in critical argument positions.

Definition 8.2.2. (Bounding Parameter*): Suppose* `f` *is a function, then a parameter* $\mathtt{f}^{(i)}$ *is a* bounding parameter *iff*

- $\mathtt{f}^{(i)}$ *is critical or*
- *there exists a sub-function call* $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_m)$ *in the body of* `f` *such that* $\mathtt{g}^{(j)}$ *is a bounding parameter for* `g` *and* $\mathtt{e}_j \rightsquigarrow \mathtt{f}^{(i)}$.

The set of bounding parameters indices *for* `f` *is denoted by* $bnd(\mathtt{f})$*:* $i \in bnd(\mathtt{f})$ *iff* $\mathtt{f}^{(i)}$ *is a bounding parameter. A* bounding argument position *is an argument position which corresponds to a bounding parameter in the called function.*

Safety for sub-function calls is defined as a criterion which implies that the function will only be passed values in the bounding argument position which are polynomially bounded by the input. This in turn implies that for each evaluation of the function body for any function in the component, the inside (or outside) call will terminate in polynomial time in the size of the input to the component.

The criteria is based on a simple check of the dependency graph and the function bound environment:

Definition 8.2.3. (Inside and base safe)*: Suppose C is a component, $\mathtt{g}(\mathtt{e}_1, \ldots \mathtt{e}_m)$ is a function call in the body of $\mathtt{f} \in C$ and that $bnd(\mathtt{g})$ is the set of bounding parameters for* $\mathtt{g}$.

The inside call or base call $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_m)$ is inside safe, base safe *respectively, iff for each bounding argument position $\mathtt{e}_i$, $i \in bnd(\mathtt{g})$ it holds that $\mathtt{e}_i \rightsquigarrow \mathtt{f}^{(j)}$ implies $j \notin dbl(\mathtt{f})$ for all i.*

For outside calls, a slightly stronger criteria is needed. An outside call contains by definition a call to one of the functions in the component – a function that in general has not yet been proven to run in polynomial time. This avoided by requiring all functions called in outside calls to have a well defined return value size bound if the return value can be used to control recursion in sub components.

Definition 8.2.4. (Outside safe)*: An outside function call $\mathtt{g}(\mathtt{e}_1, \ldots \mathtt{e}_m)$ is* outside safe *for a function $\mathtt{f} \in C$ with the size bound environment δ iff*

- *it is inside safe: $\forall i \in bnd(\mathtt{g}) : \ \mathtt{e}_i \rightsquigarrow \mathtt{f}^{(j)} \Rightarrow j \notin dbl(\mathtt{f})$* and
- *for each bounding* g*-argument $\mathtt{e}_i$, $i \in bnd(\mathtt{g})$, which contains a recursive call $\mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_{k'})$ such that $\mathtt{h} \in C$: the size of the return value of* g *is linear in the input: $\delta(\mathtt{g}) = lin(I_\mathtt{g}, \gamma_\mathtt{g})$ for some $\gamma_\mathtt{g}, I_\mathtt{g}$.*

8.3 Proof of safety

In order to prove that safety of inside and outside calls imply that the critical arguments are bounded by a polynomial, we establish the following central lemma.

Lemma 8.3.1. *Suppose* e *is an expression in a function* f *such that all function calls $\mathtt{h}(\mathtt{e}_1, \ldots, \mathtt{e}_k)$ in* e *terminate and return a value bounded by a polynomial in the input: $\|\mathcal{E}[\![\mathtt{h}(x_1, \ldots, x_k)]\!]\sigma\| \leq \Pi_\mathtt{h}(\|\sigma(x_1)\|, \ldots, \|\sigma(x_k)\|)$, for some environment σ containing the input*[2]*. Suppose the initial input to* f *is given by $v_1, \ldots, v_m \in Value$. If $\mathtt{e} \rightsquigarrow \mathtt{f}^{(i)}$ implies $i \in bnd(\mathtt{f})$ for all i then the size of* e *is bounded by a polynomial* in the input for f*, i.e. there exists a polynomial $\Pi_\mathtt{e}$ such that:*

$$\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq \Pi_\mathtt{e}(max\{v_1, \ldots, v_m\}).$$

Proof. Let $v_0, \ldots v_m$ be the initial input for function f and $v_{max} = max\{v_0, \ldots, v_m\}$. Proof by induction over the nesting depth of calls in e.

[2]The input for the *current iteration* for f.

Base case. Suppose no calls exist in $\mathtt{e}$:

$$\mathtt{f}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)}) = \ldots \mathtt{cxt}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)}) \ldots,$$

where $\mathtt{cxt}$ is some well-formed expression context $\mathtt{e} = \mathtt{cxt}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)})$ which does not contain any function calls.

Let $\mathtt{f}^{(1)}_{bnd}, \ldots, \mathtt{f}^{(k)}_{bnd}$ denote the bounding parameters for $\mathtt{f}$:

$$\{\mathtt{f}^{(i)}_{bnd} \mid \exists j \in \mathit{bnd}(\mathtt{f}) : \ \mathtt{f}^{(i)}_{bnd} = \mathtt{f}^{(j)}\}.$$

By assumption, $\forall i : \ \mathtt{e} \rightsquigarrow \mathtt{f}^{(i)} \Rightarrow i \in \mathit{bnd}(\mathtt{f})$, so by Lemma **??** there exists a function $h : I\!N^{|bnd(\mathtt{f})|} \to I\!N$ such that $\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq h(\|\sigma(\mathtt{f}^{(1)}_{bnd})\|, \ldots, \|\sigma(\mathtt{f}^{(k)}_{bnd})\|)$, where σ is the environment mapping the $\mathtt{f}$-parameters to their *current* values.

The function h describes how the size of $\mathtt{e}$ depends on the parameters of $\mathtt{f}$. By assumption $\mathtt{cxt}$ cannot contain function calls, so by examining the semantics its simple to verify that for any environment σ there exists a polynomial $\Pi_{\mathtt{cxt}}$ such that

$$\|\mathcal{E}[\![\mathtt{cxt}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)})]\!]\sigma\| \quad \leq \quad \Pi_{\mathtt{cxt}}(\|\sigma(\mathtt{f}^{(1)}_{bnd})\|, \ldots, \|\sigma(\mathtt{f}^{(k)}_{bnd})\|).$$

By Lemma 8.1.1, all bounding parameters are bounded by the original input: $\forall i : \ \|\sigma(\mathtt{f}^{(i)}_{bnd})\| \leq max\{v_0, \ldots, v_m\} = v_{max}$. By substitution we obtain

$$\begin{array}{rcl} \|\mathcal{E}[\![e]\!]\sigma\| & = & \|\mathcal{E}[\![\mathtt{cxt}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)})]\!]\sigma\| \\ & \leq & \Pi_{\mathtt{cxt}}(\underbrace{v_{max}, \ldots, v_{max}}_{|bnd(\mathtt{f})|}) \\ & = & \Pi_{\mathtt{e}}(v_{max}), \end{array}$$

for some polynomial $\Pi_{\mathtt{e}}$, which proves the base case.

Induction step. Now suppose the expression $\mathtt{e}$ contains the calls

$$\mathtt{h}_1(\mathtt{e}'_{1,1}, \ldots, \mathtt{e}'_{1,p}), \ldots, \mathtt{h}_n(\mathtt{e}'_{n,1}, \ldots, \mathtt{e}'_{n,p}) :$$

$$\mathtt{e} = \mathtt{cxt}(\mathtt{h}_1(\mathtt{e}'_{1,1}, \ldots, \mathtt{e}'_{1,p}), \ldots, \mathtt{h}_n(\mathtt{e}'_{n,1}, \ldots, \mathtt{e}'_{n,p}), \mathtt{f}^{(1)}_{bnd}, \ldots, \mathtt{f}^{(k)}_{bnd})$$

where $\mathtt{cxt}$ does not contain any function calls – note that the assumption $\forall i : \mathtt{e} \rightsquigarrow \mathtt{f}^{(i)} \Rightarrow i \in \mathit{bnd}(\mathtt{f})$ implies that the context only depends on input bounded parameters.

By the induction assumption $\|\mathcal{E}[\![\mathtt{e}'_{i,j}]\!]\sigma\| \leq \Pi_{\mathtt{e}_{i,j}}(v_{max})$, for all i, j so

$$\|\mathcal{E}[\![\mathtt{h}_i(\mathtt{e}'_{i,1}, \ldots, \mathtt{e}'_{i,p})]\!]\sigma\| \leq \Pi_{\mathtt{h}_i}(v_{max})$$

follows from the assumption that all function in $\mathtt{e}$ terminate with a polynomially large value and that polynomials are closed under composition.

Again it follows from Lemma 8.1.1 that $\|\sigma(\mathtt{f}^{(k)}_{bnd})\| \leq v_{max}$ and similarly to the base case we conclude that the size of $\mathtt{cxt}$ is polynomial in the "input" for the context:

$$\|\mathcal{E}[\![\mathtt{cxt}(x_1, \ldots, x_{n+k})]\!]\sigma\| \leq \Pi_{\mathtt{cxt}}(\|\mathcal{E}[\![x_1]\!]\sigma\|, \ldots, \|\mathcal{E}[\![x_{n+k}]\!]\sigma\|).$$

The claim then follows from the closure property of polynomials.

Thus by induction, the size of the expression $\mathtt{e}$ is bounded by a polynomial, i.e. there exists a polynomial $\Pi_{\mathtt{e}}$ such that $\|\mathcal{E}[\![\mathtt{e}]\!]\sigma\| \leq \Pi_{\mathtt{e}}(v_{max})$. □

8.4 Sub-Function Calls

Using Lemma 8.3.1 it is easily verified that safety of inside and outside calls imply that iteration of safe inside and outside calls can only produce values bounded by a polynomial in the input.

Lemma 8.4.1. *Suppose C is a component and $\mathtt{g}(\mathtt{e}_1, \ldots \mathtt{e}_n)$ is* inside safe *for $\mathtt{f}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)}) \in C$, then for any input $v_0, \ldots, v_m \in$ Value for the function $\mathtt{f}$ and for any bounding argument position $\mathtt{e}_i$, there exists a polynomial Π such that*

$$\|\mathcal{E}[\![\mathtt{e}_i]\!]\sigma\| \leq \Pi(v_0, \ldots, v_m).$$

Proof. Let $v_0, \ldots v_m$ be the input for function $\mathtt{f}$, $v_{max} = max\{v_0, \ldots, v_m\}$ and let $\mathtt{e}_i$ a critical argument position for $\mathtt{g}$.

By assumption all inside calls terminate in polynomial time with a return value which is bounded by the input to that particular function. It then follows from Lemma 8.3.1, that the size of all $\mathtt{g}$-critical arguments are bounded by a polynomial $\Pi_{\mathtt{e}_i}(v_{max})$. □

Lemma 8.4.2. *Suppose C is a component and $\mathtt{g}(\mathtt{e}_1, \ldots \mathtt{e}_n)$ is* outside safe *for $\mathtt{f}(\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(m)}) \in C$, then for any input $v_0, \ldots, v_m \in$ Value for the function $\mathtt{f}$ and for any bounding argument position $\mathtt{e}_i$, there exists a polynomial $\Pi_{\mathtt{e}_i}$ such that*

$$\|\mathcal{E}[\![\mathtt{e}_i]\!]\sigma\| \leq \Pi_{\mathtt{e}_i}(v_0, \ldots, v_m).$$

Proof. Similarly to the proof for Lemma 8.4.1 the result follows almost directly from Lemma 8.3.1.

Let $v_0, \ldots v_m$ be the input for function $\mathtt{f}$, $v_{max} = max\{v_0, \ldots, v_m\}$ and let $\mathtt{e}_i$ a critical argument position for $\mathtt{g}$. By assumption all sub-functions relative to C calls terminate in polynomial time with a return value which is bounded by the input to that particular function. In order to apply Lemma 8.3.1, it remains to be shown that any recursive calls in $\mathtt{e}_i$ to function $\mathtt{h} \in C$ terminate with a polynomial value. By assumption, termination has already been proven by the algorithm, and the latter can be proven by induction:

Consider the context around all the recursive calls in the component: $\mathtt{cxt}(\mathtt{h}(\ldots))$. By assumption, all functions $\mathtt{g}$ in $\mathtt{cxt}$ are outside calls, so they are linear in the input – there exists $\gamma_{\mathtt{g}}, I_{\mathtt{g}}$ such that:

$$\gamma_{\mathtt{g}} + \sum_{i \in I_{\mathtt{g}}} \|\mathtt{g}^{(i)}\| \leq \Pi_{\mathtt{g}}(\|\sigma(\mathtt{g}^{(1)})\|, \ldots, \|\sigma(\mathtt{g}^{(|I_{\mathtt{g}}|)})\|).$$

Thus it follows from Lemma 8.3.1 that the size of the value that the context evaluates to is polynomial in the size of the "input": $\mathtt{h}(\ldots)$. Now we can prove

that the size of $\mathtt{e}_i$ is bounded by a polynomial in v_{max} by induction over the call tree for the evaluation of $\mathtt{e}_i$.

By assumption, all base calls return a value of polynomial size in the size of the input v_{max}.

For the nodes in the call tree, by assumption the recursive calls terminate with a polynomial value. Furthermore, the context around the recursive calls is polynomial in the size of the return value of the recursive calls, so return value for the node in the call tree is polynomial in v_{max}.

Thus by induction, the size of return value of the recursive calls in $\mathtt{e}_i$ is bounded by a polynomial in the size of v_{max}. Thus by Lemma 8.3.1, the size of $\mathtt{e}_i$ is bounded by a polynomial in v_{max}, i.e. all g-critical arguments are bounded by polynomial $\Pi_{\mathtt{e}_i}(v_{max})$.

□

8.5 Implications for the Component

Definition 8.5.1. (Sub-Call Safety)*: Suppose C is a component is the call graph. If for all functions $\mathtt{f} \in C$ it holds that all base calls are in the body $\mathtt{e}^{\mathtt{f}}$ of $\mathtt{f}$ are base safe, all inside calls in $\mathtt{e}^{\mathtt{f}}$ are inside safe and all outside calls in $\mathtt{e}^{\mathtt{f}}$ are outside safe, then the C is* sub-call safe.

Theorem 8.5.1. *Suppose π is a program and C is a component in the call graph for π, such that C is terminating, polynomially branching and has polynomial call depth. Furthermore suppose that for all sub-components for C, the running time for any function $\mathtt{g} \in C$ is bounded by a polynomial in the input.*

If C is sub-call safe, then the running time of any $\mathtt{f} \in C$ is bounded by a polynomial in the size of the input.

Proof. Lemma 8.4.1 and Lemma 8.4.2 state that all bounding parameters in any sub-function call are bounded by the original input. By assumption, the running time of each sub-functions is bounded by the size of the input to the function, so the running time is bounded by a polynomial in the size of the original input, as polynomials are closed under composition. □

Chapter 9

The Algorithm

9.1 Refinement

As noted in Section 6, the analysis must choose a suitable set of input bounded parameters as the set of parameters controlling recursion in the component being analyzed. The disjointness criteria (Chapter 7) and sub-call safety (Chapter 8) both enforce restrictions on the arguments that can be passed in the critical input bounded (i.e. bounding) argument positions, so smaller sets of critical input bounded parameters will require fewer restrictions to be made on the function calls in the program. For the purpose of simply deducing asymptotic running time bounds, it is sufficient to look for the smallest possible sets. One method would be to enumerate all subsets of the parameter sets and for each set test whether it can be used to prove that the program runs in polynomial time. Unfortunately such a search procedure takes exponential time in the size of the program being analyzed. Thus we propose a slightly better search procedure which is expected to perform well in the average case.

Observe that if the disjointness criteria fails or sub-call safety is violated, then the set of parameters causing the violations can be identified. This information can be used to speed up the search.

- A parameter that causes the disjointness criteria to fail or sub-call safety to be violated, is called a *conflicting parameter*.
- Sub-call safety is violated if
  - a bounding argument position for a sub-function call depends on a "non-linear" parameter. If this is the case, the bounding parameter is a conflicting parameter.
  - there exists an outside call with a recursive call in a bounding argument position such that the recursive call has not been found to be bounded by a bound of the form $\gamma + \sum_{i=1}^{n} x_i : t_i$. If this is the case, the parameter corresponding to the argument position with the recursive call is a conflicting parameter.

- If disjointness is violated, then two bounding argument positions in two different recursive calls are non-disjoint, so both parameters are conflicting parameters. But rather than excluding *both* parameters, the conflict can be resolved if either parameter can be excluded.

Thus if disjointness fails or sub-call safety is violated, then a set of conflicting parameters can be compiled. If the program can be proven to run in polynomial be the present method, then *all* conflicts must be resolved. Thus the conflicting parameters are pruned from the set of critical input bounded parameters. Since the critical input bounded set must satisfy input boundedness, the largest input bounded subset of the pruned set is found. The resulting set is then used as the next critical input bounded set for the component.

Example. Consider the following function definition, which was taken from a type inference algorithm:

```
tsubst(a, t, t1) =
  if equal(Tvar, 1st(t1))
    then if equal(a, 2nd(t1))
           then t
           else t1
    else Arr(1:tsubst(a, t, 1st(2nd(t1))),
             2:tsubst(a, t, 1st(2nd(t1))))
```

All parameters are input bounded, so the first guess at the bounding parameters is $\{\mathtt{a}, \mathtt{t}, \mathtt{t1}\}$. The disjointness criteria causes two conflicting pairs: $\{(1:\mathtt{a}, 2:\mathtt{a}), (1:\mathtt{t}, 2:\mathtt{t})\}$. Thus we must select $\{\mathtt{a}, \mathtt{t}\}$ as the set of conflicting parameters, which leaves $\{\mathtt{t1}\}$ as the set of critical input bounded parameters. With that set all criteria can be passed without non-termination or conflicts, so the function runs in polynomial time in the input.

Note that many sets can be selected based on the conflicting parameters for the disjointness criteria. Although expensive, our solution is to test *all* possible conflicting sets. This is important because for any conflicting pair it is not clear which parameter is the better parameter to remove and selecting the right parameter can be crucial for the success of the algorithm.

9.2 Algorithm

- Extract the call graph relation $\rightarrow$.
- Compute the strongly connected components in the call graph.
- For Each Component: $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$.
  - Compute the size dependency graph and the function dependency sets *dep*.

- Find bound on return values using the size analysis and a linear programming solver, Section 5.5.
- Extract the size-change graphs $\mathcal{G}$ for the component, Section 5.4.
- Test whether $\mathcal{G}$ is size-change terminating. If not, the analysis halts.
- Compute the largest possible set of input bounded parameters, *ibd*, Section 6.
- Test whether $\mathcal{G}|_{ibd}$ is size-change terminating. If not, the analysis halts. Otherwise the call depth is bounded by a polynomial in the input by Corollary 6.2.1.
- Verify that all recursive calls are disjoint, Chapter 7. any two recursive calls in the same recurrence equation must not share the same data structure in a critical argument position. If disjointness is violated, the set of critical parameters is refined and the analysis of the *component* is restarted.
- Verify sub-call safety:
  * Compute the set of potentially doubling parameters: *dbl*, Section 8.1.
  * For all sub-function calls: verify that no potentially doubling parameter can be passed in a bounding argument position. Otherwise the set of bounding parameters is refined and the analysis of the *program* is restarted.
  * Verify that all recursive calls occurring in a bounding argument position for some outside call, are bounded by a linear function. If not, the set of critical parameters is refined and the analysis of the *program* is restarted.
- If all tests are passed, the program runs in polynomial time in the size of the input.

Chapter 10

The Bellantoni-Cook Function Class

The Bellantoni-Cook function class [2] divides the parameters into *normal* and *safe* parameters. The arguments passed to normal parameters in a function call are required to satisfy some severe restrictions, which are sufficient to ensure that the program terminates in polynomial time in the input without special execution. The key point of the syntactic criterion is that the normal parameters must be input bounded, i.e. must be copied or decreased from input bounded parameters. Secondly, only *linear*, *non-mutual* recursion is allowed on a *single* parameter.

In this section we show that the programs handled by our approach includes the programs which satisfy Bellantoni-Cook's criterion.

Definition 10.0.1. (Bellantoni-Cook's Function class*): The parameters are classified as being either* normal *parameters of* safe *parameters. Function definitions declare the parameters as normal or safe:*

$$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m)$$

parameters $\mathtt{x}_1, \ldots, \mathtt{x}_n$ *are normal and* $\mathtt{y}_1, \ldots, \mathtt{y}_m$ *are safe.*

Any program accepted by the Bellantoni-Cook syntactic criteria is defined by a set of safe *function definitions: base functions, functions defined by safe composition and function defined by safe primitive recursion.*

- Base function *are the functions:*

$$\begin{aligned}
&\mathtt{zero}(;) = Z \\
&\mathtt{proj}_j^{n,m}(\mathtt{x}_1, \ldots, \mathtt{x}_n; \mathtt{x}_{n+1}, \ldots, \mathtt{x}_{n+m}) = \mathtt{x}_j \\
&\mathtt{succ}(;\mathtt{y}) = \mathtt{S}(\mathtt{y}) \\
&\mathtt{pred}(;\mathtt{y}) = \mathtt{1st}(\mathtt{y}) \\
&\mathtt{cond}(;\mathtt{a},\mathtt{b},\mathtt{c}) = \textit{if eq(a, Z) then b else c}
\end{aligned}$$

- Safe composition *of functions* $\mathtt{h}, \mathtt{r}_i, \mathtt{t}_j$, $i < k, j < l$ *defines a function of the form:*

$$\begin{aligned}&\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m) = \\ &\quad \mathtt{h}(\mathtt{r}_1(\mathtt{x}_1, \ldots, \mathtt{x}_n;), \ldots, \mathtt{r}_k(\mathtt{x}_1, \ldots, \mathtt{x}_n;); \\ &\qquad \mathtt{t}_1(\mathtt{x}_1, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m), \ldots, \mathtt{t}_l(\mathtt{x}_1, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m))\end{aligned}$$

  where the functions $\mathtt{r}_i$ *only accept* normal *parameters for all* i*, but* $\mathtt{t}_j$ *accept both normal and safe parameters for all* j*.*

- Safe primitive recursion *from the sub-functions* $\mathtt{g}$ *and* $\mathtt{h}$ *is defined as a function of the form:*

$$\begin{aligned}&\mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m) = \\ &\quad \mathtt{if}\ \mathtt{eq}(\mathtt{x}_1, \mathtt{Z}) \\ &\qquad \mathtt{then}\ \mathtt{g}(\mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m) \\ &\qquad \mathtt{else}\ \mathtt{h}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m, \\ &\qquad\qquad \mathtt{f}(\mathtt{1st}(\mathtt{x}_1), \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m))\end{aligned}$$

Theorem 10.0.1. *Suppose the program* π *satisfies the Bellantoni-Cook syntactic criteria, then* π *terminates in polynomial time by the methods given in the previous section.*

Proof. Suppose $\mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m)$ is a function defined in the Bellantoni-Cook framework and assume that all sub-functions run in polynomial time in the input.

The function $\mathtt{f}$ is either defined by a base function, safe composition or safe primitive recursion. The base functions clearly run in polynomial time in the input, so consider safe primitive recursion and safe composition:

Safe Primitive Recursion. If the function $\mathtt{f}$ is defined by primitive recursion then $\mathtt{f}$ must have the form

$$\begin{aligned}&\mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m) = \\ &\quad \mathtt{if}\ \mathtt{eq}(\mathtt{x}_1, \mathtt{Z}) \\ &\qquad \mathtt{then}\ \mathtt{g}(\mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m) \\ &\qquad \mathtt{else}\ \mathtt{h}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m, \\ &\qquad\qquad \mathtt{f}(\mathtt{1st}(\mathtt{x}_1), \mathtt{x}_2, \ldots, \mathtt{x}_n; \mathtt{y}_1, \ldots, \mathtt{y}_m)),\end{aligned}$$

where $\mathtt{g}, \mathtt{h}$ are sub-functions for $\mathtt{f}$.

The component only contains a single recursive function call, so disjointness is trivially satisfied. Thus the number of recursive calls is polynomial in the size of the input to $\mathtt{f}$.

Observe that all parameters are input bounded. The size change graph for the single call possible in the component, $\mathtt{f} \to \mathtt{f}$, has a decrease in $\mathtt{x}_1$, so size change termination is satisfied. Thus the set of normal parameters are both critical and input bounded. However this is not the set initially selected by refinement procedure, in fact the refinement procedure starts out with all

parameters as critical input bounded parameters. But if selecting the larger set leads to any conflicts, then the conflicting parameters will be pruned from the critical input bounded parameter set. Observe that $\mathtt{x}_i$ never can cause a conflict for any i: they can by definition only be passed normal arguments, i.e. arguments that are critical and input bounded, so $\mathtt{x}_i \notin \mathit{dbl}\mathtt{f}$ for all i. Thus the outside call is outside safe, so eventually the refinement procedure will zero in on the normal parameters (or any larger bounding parameter set).

The recursive definition only contains a single outside call, where the recursive call is in a safe argument position, i.e. in a non-bounding argument position. Thus the call must satisfy $\mathtt{x}_i \notin \mathit{dbl}(\mathtt{f})$, which is trivially satisfied by the bounding parameter set eventually selected by the refinement procedure.

In the base call, $\mathtt{f} \to \mathtt{g}$, $\mathtt{g}$ is only passed input bounded parameters, so no arguments of non-constant growth can be passed to $\mathtt{g}$, i.e. the call is base safe.

Thus the function is sub-call safe, so the result follows from Theorem 8.5.1.

Safe Composition. If the function $\mathtt{f}$ is defined by safe composition must have the form

$$\begin{aligned}&\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n;\mathtt{y}_1,\ldots,\mathtt{y}_m) = \\ &\quad \mathtt{h}(\mathtt{r}_1(\mathtt{x}_1,\ldots,\mathtt{x}_n;),\ldots,\mathtt{r}_k(\mathtt{x}_1,\ldots,\mathtt{x}_n;); \\ &\qquad \mathtt{t}_1(\mathtt{x}_1,\ldots,\mathtt{x}_n;\mathtt{y}_1,\ldots,\mathtt{y}_m),\ldots,\mathtt{t}_l(\mathtt{x}_1,\ldots,\mathtt{x}_n;\mathtt{y}_1,\ldots,\mathtt{y}_m))\end{aligned}$$

Clearly, $\mathtt{f}$ is a non-recursive component in the call graph for π. Thus π runs in polynomial time provided that the sub-functions run in polynomial time in the size of of the bounding parameters, which is given by the assumption. Note that the bounding parameters for $\mathtt{f}$ is the set of parameters passed in bounding argument positions in the sub-function calls. □

Chapter 11

Size-Measures

This section explores the advantages of using other size-measures than the node size-measure as defined in Section 5.1.

11.1 Homogeneous size-measures

A quite versatile size-measure for the terms is the *term size-measure*.

Definition 11.1.1. (Term size-measure)*: The term size-measure is given by:*

$$\begin{array}{lcl} \|\mathtt{con}\| & = & 1 \\ \|\mathtt{con}(\mathtt{t}_1, \ldots, \mathtt{t}_n)\| & = & 1 + \sum_{i=1}^{n} \|\mathtt{t}_i\|. \end{array}$$

The term size-measure measures the amount of store consumed by the term under the assumption that the sub-terms are disjoint – i.e. the "perceived size" of the term. This is important since the language has constructors with different arities – e.g. the terms $\mathtt{Cons}(\mathtt{Nil}, \mathtt{Nil})$ and $\mathtt{S}(\mathtt{S}(\mathtt{Z}))$ have the same size, but the number of node constructors [1] differs. Differentiating the sizes such terms can be important for proving termination for programs that change the structure of the decreasing term.

Since the language is untyped and deconstruction is performed using deconstructors, such as `1st` and `2nd`, it is not always possible to deduce the arity of the topmost constructor of the value being deconstructed. This can lead to overly conservative size estimates for deconstruction operations.

Consider the deconstruction $\mathtt{1st(x)}$, which when evaluated for some environment $\sigma = [\mathtt{x} \mapsto \mathtt{S}(y)]$ has the size $\|\mathtt{1st(x)}\| = \|\mathtt{x}\| - 1$. But if $\sigma = [\mathtt{x} \mapsto \mathtt{Cons}(y, z)]$ then the size is

$$\begin{array}{lcl} \|\mathtt{1st(x)}\| & = & \|y\| \\ & = & \|\mathtt{x}\| - (1 + \|\mathtt{2nd(x)}\|) \\ & = & \|\mathtt{x}\| - (1 + \|z\|) \\ & \leq & \|\mathtt{x}\| - 2, \end{array}$$

[1] Constructors with arity greater than zero.

since $\|z\| \geq 1$ according to the definition of the term size-measure. Thus in general, it would only be safe to assert that $\|\texttt{1st(x)}\| \leq \|\texttt{x}\| - 1$.

For typed languages or languages with explicit deconstruction (case matching), the arity is always known at the time of deconstruction so no precision is lost. If the language semantics is to remain as defined, one possible workaround would be to analyze the program and find the maximal arity used for any constructor. All constructors could then be considered as having that arity for analysis purposes.

Fortunately in practice, the termination argument rarely depends on proving that substituting constructors of different arities changes the size of a term.

A variant of the term size-measure is the *node size-measure*, where the size of zero-ary constructors is zero rather than one, which avoids the arity problem above. The size analysis in Chapter 5 was defined in terms of the node size measure.

Definition 11.1.2. (Node size-measure*): The node size-measure is given by:*

$$\begin{array}{lcl} \|\texttt{con}\| & = & 0 \\ \|\texttt{con}(\texttt{v}_1, \ldots, \texttt{v}_n)\| & = & 1 + \sum_{i=1}^{n} \|\texttt{v}_i\|, n \geq 1. \end{array}$$

Using this size-measure, two terms have equal size if they have the same number of node constructors. The advantage over the term size-measure is that it is not necessary to know the arity of the constructor being deconstructed to obtain a precise bound on the size.

In some cases, the termination argument relies on showing that the length of a list decreases, *regardless* of the behavior of the elements in the list. For this purpose, the *list size-measure* is the better size-measure to use.

Definition 11.1.3. (List size-measure (generalized)*): The list size-measure is given by:*

$$\begin{array}{lcl} \|\texttt{con}\| & = & 0 \\ \|\texttt{con}(v_1, \ldots, v_n)\| & = & 1 + \|v_n\|, n \geq 1. \end{array}$$

That is, only the right-most sub-term is considered when computing the size, which is the usual convention for Lisp programs. Again, complications arise due to the fact that the arity must be known at the time of deconstruction.

A further generalization of the list size-measure, would be to select a subset of sub-terms and ignore the remaining sub-terms. This is only relevant for rather specialized termination arguments, e.g. descent over a "left skewed" list. Thus since searching for the best subset is exponential in the maximal arity, it might prove not to be worth the effort.

11.2 Structural Size-Measures

If type information is available as given either by a type system or explicit deconstruction (or perhaps even by a type analysis or type inference), one can

utilize the types to associate different constructors with different size-measures, e.g. use the list size-measure for the `Cons` constructor, but use the node size-measure for `Node` constructors for use in trees [?]. This makes it possible to trace "size-changes" even more precisely, and to uncover subtle orderings on the terms. If a type system is available, it is possible to tailor the size-measures to the data types apriori, so expensive searches can be avoided.

For typical programs, the termination argument is centered around the types used, so type declarations can provide good clues as to which size measure to use in termination proofs. This can be use to improve search strategies for size-measures. Consider e.g. programs using terms with the type: trees of lists of numbers encoded as unary numbers, i.e. `Z`, `S(Z)`, etc. A possible search strategy could be based on the size-measures associated with the types:

First try to prove termination using the term size-measure, which would account only for node constructors in the terms. If termination cannot be proven, the termination might depend on the size of the elements, i.e. leaves, in the tree as well. Therefore, a good next guess would be to try to prove termination using a size measure, which measures the elements in the tree with the list size-measure. If termination still cannot be proven, the elements in the list can be measured as well. This essentially corresponds to using the node size-measure.

Measures Inducing Orderings on Constructors. Orderings between constructors present yet another problem. Consider the 10 zero-ary constructors `Zero`, ..., `Nine` and a decrement function operating on them `dec` satisfying:

$$\|\texttt{Zero}\| = \|\mathcal{E}[\![\texttt{pred(One)}]\!]\sigma\| < \|\texttt{One}\| \ldots$$

for any environment σ. The `pred` function decrements the "size" of the term, but this cannot be captured by the homogeneous size-measures. Fortunately, the number of constructors in a program is bounded, so this type of term descent can be unfolded away.

Lexicographical Measures. Lexicographical orderings on terms are hard to uncover, the prime example being binary numbers. Suppose the natural numbers are encoded by bit strings:

$$bin ::= Z \mid S0(bin) \mid S1(bin).$$

The predecessor function *does* decrease the size of the term using the list size-measure if a sufficiently long sequence is considered, but the problem is that the sequence is exponential in the list size. Thus it is not possible to use unfolding to make the analysis see the descent using the term-size measure, so this type of descent needs special attention if one wishes to handle it. This is unfortunate since this restricts the analysis to inefficient encodings (base-1 encodings) of the natural numbers. A cheap workaround would be to introduce natural numbers in the language and directly write the size analysis to deal with them, rather than to handle general lexicographical descent in terms.

Mixing Size-Measures. Once basic observation is that different size-measures *must not* be mixed when extracting size-change graphs.

```
f(xs) = let a = 1st(xs)
            b = 1st(2nd(xs))
            r = 2nd(2nd(xs))
        in 1:f(Cons(Cons(a, b), r)
        ...
        let a = 1st(1st(xs))
            b = 2st(1nd(xs))
            r = 2nd(2nd(xs))
        in 2:f(Cons(a, Cons(b, r))
```

The call at call site `1` shuffles elements from the tail of `xs` into the first element, and the call at call site `2` does the opposite. Now consider the call sequence $(12)^\omega$; `xs` is decreasing at call site `1` using the list size-measure, and size non-increasing at call site `2` using the node size-measure. But since `xs` contains exactly the same term after the two calls $1, 2$, the size analysis cannot be permitted to conclude that the two size-change graphs can be combined to obtain a decrease over the two calls.

It is, however, possible to apply different size-measures to different infinite call sequences for the same program. The size-analysis will, however, only be defined in terms of the node size-measure, but it is an easy task to modify the analysis and the accompanying proofs to use any other size-measure.

Chapter 12

Experiments

The analyses presented in the previous sections, have been implemented on top of the size-change termination analyzer [7], all though most of the analyzer has been completely rewritten. The main focus of the implementation has not directly been to obtain a fast implementation, but the performance has been improved in comparison to [7]. This is mainly due to the fact that the analyzer works with one component in the call graph at the time, rather than the entire program.

The analyzer was implemented using roughly 4000 lines of Haskell code, and interfaces to the "lp_solve 3.1" linear programming package [1] to compute linear bounds on the return values of functions.

12.1 Test Suites

The analyzer accepts any subject program written in the language described in section 3. The test suite is mainly the one used in [7], which in turn was collected from [14, 24, 8]. The subject programs used in the experiments can be found in section A.

The Jones test suite. The test suite from [14], (Appendix A.1) consists of the 6 small examples used in the paper to illustrate various interesting types of descent that the size-change termination principle is able to handle, even with a very simple size analysis. The original programs translate directly to the subject language used in the analysis.

The Wahlstedt test suite. Wahlstedt [24] (Appendix A.2) published his results on experiments with the size-change termination principle applied to a small first order function language. The subject language contains unary numbers represented by the constructors `Z` for zero and `S` for the successor and uses explicit deconstruction. The programs translate easily into the subject language for the analysis.

The Glenstrup test suite. Glenstrup [8] (Appendix **??**) collected an impressive test suite for his thesis on bounding time analysis for partial evaluation. The programs were translated from scheme to the subject language using a small compiler [7], but some minor changes were made in order to represent the programs. In particular, symbols, characters and strings are encoded using zero-ary constructors. We claim that the modifications have no impact on the running time of the programs, since termination arguments rely on lists exclusively for the programs in the test suite. A minor point is that some of the programs encode base-1 by list length, where elements in the list is `Cons(Nil, Nil)`, i.e. the number 2 would be encoded as `Cons(Cons(Nil, Nil), Cons(Cons(Nil, Nil), Nil))` as opposed to `S(S(Z))`.

Other subject programs. Finally a small collection of programs (Section **??**) is provided to illustrate various issues in the analyzer. Some of the programs are simply minor modifications of the programs from the other test suites.

12.2 Analysis Output

The analysis output can be found in Appendix **??**. The output consists of the name of the program being analyzed followed by a list of the refinements attempted during analysis and the actual results found. The results are broken down for each strongly connected component in the call graph: the function return bounds found, the critical parameters, the time taken to analyze the component and a list of violations of the analysis criteria if any. If component fails to be classified as computing a function in PTIMEF, then the analyzer outputs an error message according to the failure mode:

- The component might not terminate: a violating call sequence is printed plus the size-change graphs for the component. The violating call sequence is a cycle that does not have a decreasing thread according to the size-change graphs.
- The component does not terminate when restricted to the input bounded parameters. Again the size-change graphs are printed along with a violating trace.
- The component might not be polynomially branching: then bounding arguments positions in different function calls in the same branch are passed arguments that are not disjoint. Both the violated parameters and the violating arguments are printed.
- The component passes a potentially doubling value to a bounding argument position: Again both the violated argument position and the violating parameter are printed.
- The component contains a non-linear outside call in a bounding argument position: the function name and the bounding argument positions are printed.

	Analysis	Opt	Failures
Number of programs:	99	-	-
Potentially non-terminating programs:	30	20	10
Potentially exponential programs:	21	17	4
- Potentially non-polynomially branching:	6	-	3
- Potentially non-disjoint:	8	-	1
- Potentially doubling conflicts:	2	-	0
- Potentially non-linear outer calls:	5	-	0
PTIMEF programs:	48	62	-

Table 12.1: Summary of analysis results.

Program	Program name
Opt	Optimal result
Term	"NT" for non-termination, "T" for termination
Call depth	"Π" for polynomially branching, "exp" otherwise
Disj.	Disjointness test
Dbl.	Doubling parameters passed in bounding argument positions?
Outer	Non-linear outer calls?
Steps	Number of refinement steps
Time	Total analysis time for the program

Table 12.2: Legend

Thus the results can interpreted as follows: the program is possibly non-terminating for all input if some component might not size-change terminate, and the program terminates in polynomial time in the size of the input if no errors are listed for any component in in the call graph for the program.

A brief summary of the results is given in Table 12.1.

Summary analysis results are given in Tables 12.3, 12.4, 12.5, 12.6, 12.7, and the legend is given in Table 12.2. For the actual analysis output, the reader if referred to Appendix **??**.

Program	Opt	Term	Call depth	Disj	Dbl	Outer	Steps	Time
ex1	Π	T	Π	no	no	no	0	0.01s
ex2	Π	T	Π	no	no	no	0	0.00s
ex3	exp	T	exp	-	-	-	0	0.01s
ex4	Π	T	Π	no	no	no	0	0.02s
ex5	Π	T	Π	no	no	no	0	0.00s
ex6	Π	T	Π	no	no	no	0	0.00s

Table 12.3: Experiments: Jones test suite

Program	Opt	Term	Call depth	Disj	Dbl	Outer	Steps	Time
ack	exp	T	exp	-	-	-	0	0.01s
add	Π	T	Π	no	no	no	0	0.01s
boolprog	NT	NT	-	-	-	-	0	0.78s
div2	Π	T	Π	no	no	no	0	0.00s
eq	Π	T	Π	no	no	no	0	0.00s
ex6	Π	T	Π	no	no	no	0	0.01s
fgh	NT	NT	-	-	-	-	0	0.03s
oddeven	Π	T	Π	no	no	no	0	0.00s
permut	Π	T	Π	no	no	no	0	0.01s

Table 12.4: Experiments:Wahlstedt test suite

12.3 Positive Examples

The test suites contain many functions constructed from safe composition and safe primitive recursion compute by the Bellantoni-Cook criterion a function in PTIMEF. As was shown in Section 10, the present analysis will identify all programs that pass the Bellantoni-Cook criterion.

12.3.1 Greatest Common Divisor

`gcd-1`, `gcd-2`, `gexgcd1`, `gexgcd2` The functions `gcd-1` and `gcd-2` implement a greatest common divisor function, that repeatedly subtracts the smaller input from the larger. Proving size-change termination amounts to proving that the subtraction function `monus` is size decreasing, which is easily established by the function return bound algorithm. The functions are linear and use only input bounded parameters, so the analyzer is able to conclude that the programs run in polynomial time in the size of the input.

The program `gexgcd1` is a translation of an imperative implementation of a greatest common divisor algorithm. Each instruction in the original program has been translated to a function on the program state. One point to note is that the mechanical conversion generates a function program with rather long call sequences. Due to the implementation of the closure algorithm (the core of the size-change termination analyzer), the program takes relatively longer time to analyze then should be expected. This is illustrated by the program `gexgcd2` where unfolding has been performed to obtain a program with tighter call sequences. This causes the analyzer to terminate (in the order of 100 times) faster for the latter program. In recent work by Thiemann [23], this insight was used to improve the closure computation in the size-change termination analyzer. The above example illustrates that such optimizations can have a significant impact on automatically generated programs.

Program	Opt	Term	Call depth	Disj	Dbl	Outer	Steps	Time
add	Π	T	Π	no	no	no	0	0.00s
addlists	Π	T	Π	no	no	no	0	0.00s
anchored	Π	T	Π	no	no	no	0	0.00s
append	Π	T	Π	no	no	no	0	0.00s
assrewrite	Π	NT	-	-	-	-	0	0.02s
badd	NT	NT	-	-	-	-	0	0.00s
contrived1	Π	T	exp	-	-	-	0	0.04s
contrived2	NT	NT	-	-	-	-	0	0.04s
decrease	Π	T	Π	no	no	no	0	0.00s
deeprev	Π	T	Π	no	no	no	0	0.02s
disjconj	Π	T	Π	no	no	no	0	0.02s
duplicate	Π	T	Π	no	no	no	0	0.01s
equal	NT	NT	-	-	-	-	0	0.01s
evenodd	Π	T	Π	no	no	no	0	0.00s
fold	Π	T	Π	no	no	no	0	0.01s
game	Π	T	Π	no	no	no	0	0.01s
increase	NT	NT	-	-	-	-	0	0.00s
intlookup	NT	NT	-	-	-	-	0	0.00s
letexp	NT	NT	-	-	-	-	0	0.01s
list	Π	T	Π	no	no	no	0	0.01s
lte	Π	T	Π	no	no	no	0	0.00s
map0	Π	T	Π	no	no	no	0	0.00s
member	Π	T	Π	no	no	no	0	0.00s
mergelists	Π	T	Π	no	no	no	0	0.00s
mul	Π	T	Π	no	no	no	0	0.01s
naiverev	Π	T	Π	no	no	no	0	0.01s
nestdec	Π	T	Π	no	no	no	0	0.01s
nesteql	NT	NT	-	-	-	-	0	0.00s
nestimeql	NT	NT	-	-	-	-	0	0.00s
nestinc	NT	NT	-	-	-	-	0	0.01s
nolexicord	Π	T	Π	no	no	no	0	0.02s
ordered	Π	T	Π	no	no	no	0	0.00s
overlap	Π	T	Π	no	no	no	0	0.01s
permute	exp	NT	-	-	-	-	0	0.04s
revapp	Π	T	Π	no	no	no	0	0.00s
select	Π	T	Π	no	no	no	0	0.01s
shuffle	Π	T	Π	no	no	no	0	0.01s
sp1	NT	NT	-	-	-	-	0	0.01s
subsets	exp	T	Π	no	no	yes	0	0.01s
thetrick	Π	T	exp	-	-	-	0	0.01s
vangelder	NT	NT	-	-	-	-	0	0.08s

Table 12.5: Experiments: Glenstrup – basic algorithms

Program	Opt	Term	Call depth	Disj	Dbl	Outer	Steps	Time
graphcolour1	exp	NT	-	-	-	-	0	0.14s
graphcolour2	exp	NT	-	-	-	-	0	0.30s
graphcolour3	exp	T	Π	yes	no	no	0	0.12s
match	Π	T	Π	no	no	no	0	0.00s
reach	NT	NT	-	-	-	-	0	0.04s
rematch	Π	NT	-	-	-	-	0	1.07s
strmatch	Π	T	Π	no	no	no	0	0.01s
typeinf	exp	T	Π	yes	yes	yes	0	0.19s
int	NT	NT	-	-	-	-	0	0.30s
intdynscope	NT	NT	-	-	-	-	0	0.46s
intloop	exp	T	Π	yes	yes	no	1	6.36
intwhile	NT	NT	-	-	-	-	0	0.23s
lambdaint	NT	NT	-	-	-	-	0	0.06s
parsexp	NT	NT	-	-	-	-	0	0.04s
turing	NT	NT	-	-	-	-	0	0.18s
ack	exp	T	exp	-	-	-	0	0.01s
binom	exp	T	Π	yes	no	no	1	0.01s
gcd1	Π	T	Π	no	no	no	0	0.01s
gcd2	Π	T	Π	no	no	no	0	0.02s
power	exp	T	Π	no	no	yes	0	0.01s
mergesort	Π	NT	-	-	-	-	0	0.05s
minsort	Π	NT	-	-	-	-	0	0.03s
quicksort	Π	NT	-	-	-	-	0	0.11s

Table 12.6: Experiments: Glenstrup – algorithms, interpreters, simple, sorting

Program	Opt	Term	Call depth	Disj	Dbl	Outer	Steps	Time
assrewriteSize	Π	NT	-	-	-	-	0	0.06s
bubblesort	Π	T	Π	no	no	no	0	0.01s
inssort	Π	T	Π	no	no	no	0	0.00s
minsortSize	Π	T	Π	no	no	no	0	0.03s
deadcodeSize	Π	T	Π	no	no	no	0	0.00s
fghSize	Π	NT	-	-	-	-	0	0.04s
gexgcd	Π	T	Π	no	no	no	0	3.24s
gexgcd2	Π	T	Π	no	no	no	0	0.02s
graphcolour2Size	exp	T	Π	yes	no	no	0	0.29s
quicksortSize	Π	T	Π	yes	no	no	0	0.03s
thetrickSize	Π	T	exp	-	-	-	0	0.01s
power	exp	T	Π	no	no	yes	0	0.01s
disj1	exp	T	Π	yes	no	no	0	0.01s
disj2	exp	T	Π	yes	no	no	1	0.01s
dup1	Π	T	Π	no	no	no	2	0.01s
dup2	Π	T	Π	no	no	no	2	0.01s
ocall-safe	Π	T	Π	no	no	no	0	0.01s
ocall-unsafe	exp	T	Π	no	no	yes	0	0.01s
quicksortPtime	Π	T	Π	no	no	no	0	0.03s

Table 12.7: Experiments: other examples

12.4 False Negatives

Since the analysis is an *approximation* of the underlying problem: decide whether a program terminates in polynomial time, the analysis will fail to correctly classify some programs, i.e. the analysis will produce false negatives. In this section, the false negatives are investigated to highlight the causes of the incorrect classification.

The analysis subsumes termination analysis, so the false negatives fall into two types of false negatives: either the subject program terminates in super-polynomial time, but the analyzer fails to prove termination, or the subject program terminates in polynomial time, but the analyzer is unable to prove this.

12.4.1 Graph Coloring

The programs `graphcolour1`, `graphcolour2`, `graphcolour3`, `graphcolourSize` implement a graph coloring algorithm, so apriori the programs should not run in polynomial time. All programs are in fact small variations of the same terminating algorithm, but differ in the way they call themselves recursively. The programs `graphcolour3`, `graphcolourSize` both size-change terminate, but the analysis is unable to prove termination for the two first variants. This clearly represents two false negatives, due to the termination analysis.

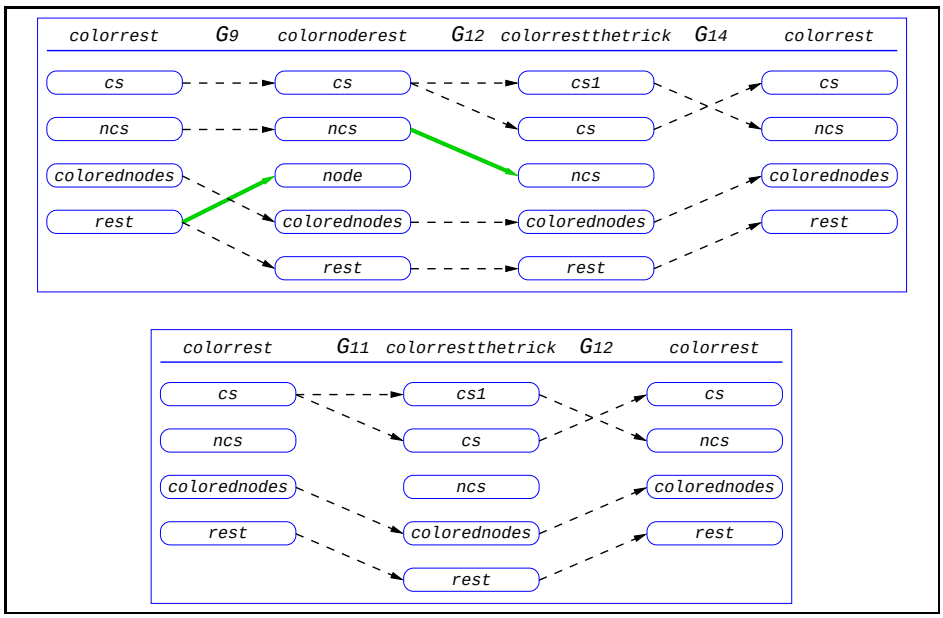

Figure 12.4.1: Critical call sequence in `graphcolour2` *(top),* `graphcolour1` *(bottom)*

In the program `graphcolour2`, the function `colourrestthetrick` is called instead of the recursive call to `colorrest` (as in `graphcolour3`). The call duplicates the parameter `cs` as `cs` and `cs1` in `colourrestthetrick`. The source of the termination problem lies with the last call: `colourrestthetrick` calls `colourrest` but the thread in `ncs` is discontinued, so termination cannot be deduced. The call is *only* made if `equal(ncs, cs1)` evaluates to `true`, implying that `ncs` and `cs1` are equal. Since the size analysis is able to deduce a size non-increasing arc from `cs1` to `ncs`, the equivalence can be used to obtain a size non-increasing arc from `ncs` to `ncs` in the last call. Given such a size-change graph for the last call, the analysis would be able to infer size-change termination for the program. This is illustrated in `graphcolour2Size` where `cs1` has been replaced by `ncs` in the last call.

The analysis also fails to prove size-change termination for the `graphcolour1` program. The problem lies with the call sequence:

$$\texttt{colourrest} \xrightarrow{11} \texttt{colourrestthetrick} \xrightarrow{12} \texttt{colourrest}$$

as before the call sequence cannot be repeated infinitely often since `ncs` then would decrease infinitely often. Again this is not captured by the size-change graphs. The situation is slightly worse the for the `graphcolour2` program. As before, it can be argued that the last call is made only when `ncs` = `cs1`, which would give a size non-increasing arc from `ncs` to `ncs` in the last call. But to show size-change termination, it must be proven that `ncs` decreases in the first call.

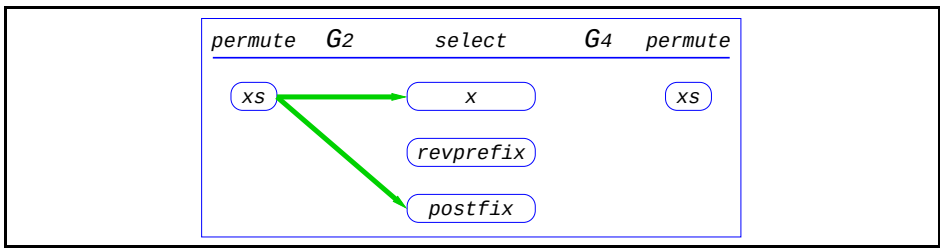

Figure 12.4.2: Critical call sequence in `permute`

Note that the sub-function `colornode` is bounded by $\|\texttt{ncs}\| + \|\texttt{1st(rest)}\| + 1$. This give the bound $\|\texttt{ncs}\| + \|\texttt{rest}\| - 2$ on the argument passed to `ncs` in the first call. Unfortunately, this is not sufficient, since the bound must at most depend on a single variable. The crucial thing to note is that `colornode` either returns `Nil` (in which case the call sequence cannot be repeated infinitely) or a pair where the head is materially dependent on `ncs`. The value passed to `ncs` in the first call, is a substructure of the head of the value returned by `colornode`, so `ncs` does decrease in the first call. However in general it is a complex problem to analyze how substructures of values returned by a function depend upon the input, so a lot more work is required to capture the information necessary to deduce size-change termination.

12.4.2 Permutation

The function `permute` computes all possible permutations of a list, so ideally the analyzer should be able to prove that the program terminates in super-polynomial time. Unfortunately, the current size-analysis is not strong enough to show this – the problem is caused by the call sequence

$$\texttt{permute} \overset{2}{\rightarrow} \texttt{select} \overset{4}{\rightarrow} \texttt{permute}.$$

The analysis is able to show that `revapp` is non-increasing in the size of the input, which establishes a size non-increase in from the parameters `revprefix` and `postfix` to the value passed to `xs` in the recursion. However, the size analysis does not track parameter *tuples*, so the information is not used. Clearly, this situation could be avoided by tracking parameter tuples.

12.4.3 The "fgh" function

The Wahlstedt test suite includes a small program, `fgh`. In it original formulation it is non terminating, due to the absence of decreases in the recursive call $\texttt{f} \overset{3}{\rightarrow} \texttt{f}$. This has been rectified in the program `fghSize`. The program does terminate, but the analyzer is unable to show size-change termination. This is due to the call sequence: $\texttt{f} \overset{2}{\rightarrow} \texttt{g} \overset{6}{\rightarrow} \texttt{g} \overset{5}{\rightarrow} \texttt{f} \overset{3}{\rightarrow} \texttt{f}$.

Examining the multi-path for the program, it is clear that the problem lies with the absence of a non-increasing transition from `x` to `x` of the call $\texttt{f} \overset{3}{\rightarrow} \texttt{f}$.

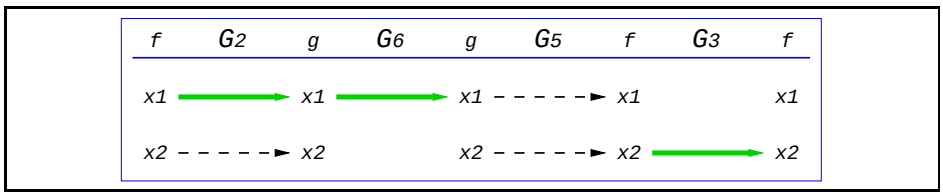

Figure 12.4.3: Critical call sequence in `fghSize`

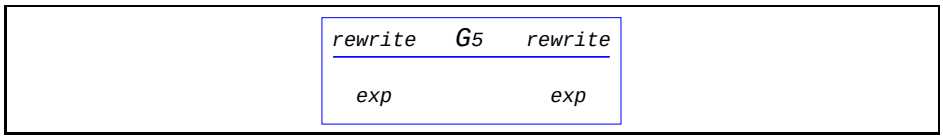

Figure 12.4.4: Critical call sequence in `assrewrite`

Consider the thread in `x` over the call sequence: `x` is decreased in the first two calls, copies and then increased in the last call. If the size information was tracked by bit more precisely, we would be able to deduce a decrease over the entire call sequence.

12.4.4 Associative String Rewriting

The program `assrewrite` takes an expression in the form of a Lisp-style tree. The expression is rewritten such that all occurrences of the associative operator `Op` are "pushed to the right":

$$\texttt{Op}(\texttt{Op}(\texttt{a},\texttt{b}),\texttt{c}) \to \texttt{Op}(\texttt{a},\texttt{Op}(\texttt{b},\texttt{c}))$$

The expression is contained in a single variable which is rewritten using a single function `rewrite`, so clearly the amount of storage used to represent the expression does not change in the recursion. Thus the analysis fails to prove termination for many different size measures (including the node size measure, which is used in the implementation). Thus is an instance where unusual size measures are needed in order to prove termination. Consider the size measure

$$\begin{array}{lcl} \|\texttt{con}\|_{elem} & = & 0 \\ \|\texttt{con}(v_1,\ldots,v_n)\|_{elem} & = & 1+\sum_{i=1}^{n-1}\|v_i\|_{node} \end{array}$$

where $\|\cdot\|_{node}$ is the node size measure. The $\|\cdot\|_{elem}$ size measure is intuitively the size of the term using the node size-measure minus the size of the term using the list size measure:

$$\|\texttt{e}\|_{elem} = \|\texttt{e}\|_{node} - \|\texttt{e}\|_{list}.$$

Using this size-measure, the program can be shown to size-change terminate.

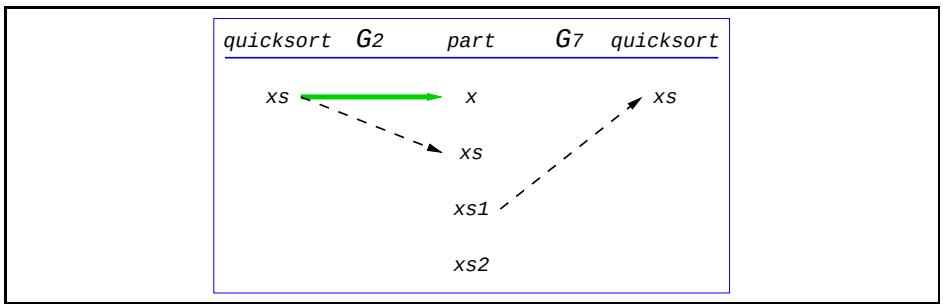

Figure 12.4.5: Critical call sequence in `quicksort`

12.4.5 Quicksort

The quicksort programs, `quicksort`, `quicksortSize`, `quicksortPtime`, are prime examples of the "divide and conquer" programming paradigm, so naturally we hope to be able to successfully show that quicksort terminates in polynomial time in the size of the input. Unfortunately, the current methods fall a bit short of the goal. Quicksort has been implemented in three flavors: `quicksort` is the most "natural" functional implementation of quick sort. The function `quicksort` recursively calls a split function `part` to divide to list into a low and a high part. The important point here is that `part` recursively calls quicksort again on the low and the high part. To prove termination, it is important to realize that the low and high parts depend *immaterially* on the list to be sorted – i.e. the low and high parts are *constructed* from the input list and are passed to a mutually recursive function. This confuses the size analysis. The size analysis is primarily designed to capture immaterial dependencies for *sub-function* calls and it is unable to provide useful information when immaterial dependencies arise between mutually recursive functions.

One way to view the situation is that `quicksort` makes a continuation passing style call to the "sub-function" `part` (where the "continuation" is implicit in the context).

It is possible to apply program transformation to avoid such problems, due to the way `part` is called. This is illustrated by the program `quicksortPtime`: the splitting function `part` has been rewritten as a sub-function relative to `quicksort`. Instead of recursively calling quicksort on the low and high parts, the two parts are returned in a tuple, which are then extracted by the `quicksort` function. This allows the size-analysis to deduce that `part` is size non-increasing, which is sufficient to show size-change termination.

The final implementation of quicksort, `quicksortSize`, employs two functions to split to input list: `partLt` to extract the low part and `partGt` to extract the hight part. This is an interesting examples since the terminating argument is quite obvious, so the program size-change terminates by the algorithm. But proving that the program terminates in polynomial time is the size of the input is a different matter: the disjointness criterion fails, since the analysis cannot deduce that the output of `partLt` and `partGt` are in fact disjoint. It is in gen-

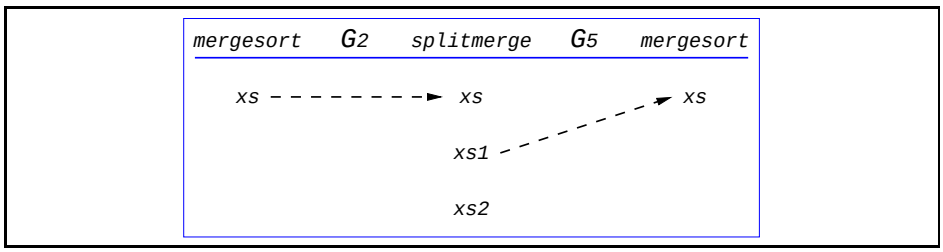

Figure 12.4.6: Critical call sequence in `mergesort`

eral a complex problem to prove that two functions compute disjoint values. One method for handling this type of complications could be to use a "loop fusion" procedure to construct a single function that computes both passes over the input list; this essentially would result in the `part` function as used in `quicksortPtime`.

12.4.6 Mergesort

The mergesort program, `mergesort`, suffers from the same problems as the `quicksort` program: the termination problems are caused by the recursive calls between the `mergesort` function and the splitting function `splitmerge`. Again due to immaterial dependencies arising between mutually recursive functions, the size analysis is unable to extract sufficient information to prove size-change termination.

12.4.7 Minsort

The minsort programs, `minsort`, `minsortSize`, can be quite tricky to handle. The problem arises in the recursive call from `appmin` to `minsort` – the minimum element in the list has been found and minsort is called recursively on the list with the minimum element has been removed by the `remove` function. The analyzer must be able to prove that the call to `remove` *decreases* the size of the input. The problem is that the "natural" implementation of `remove` would not assume that the element to be removed actually exists in the list – thus the function is *not* size decreasing for all input. The minsort program however always maintains the invariant that the minimum element is in the list when `remove` is called, thus the function is size decreasing for all input satisfying the invariant. Interestingly, this type of invariants is very hard to uncover.

One workaround is to realize that since that element to be removed is always in the list, the `remove` function can be unfolded one step, yielding a size decreasing function for all input. This is illustrated by the terminating version of minsort: `minsortSize`. The analysis is even able to correctly deduce that the program runs in polynomial time in the size of the input, due to the facts that minsort contains only linear calls, is size linear and all parameters are input bounded.

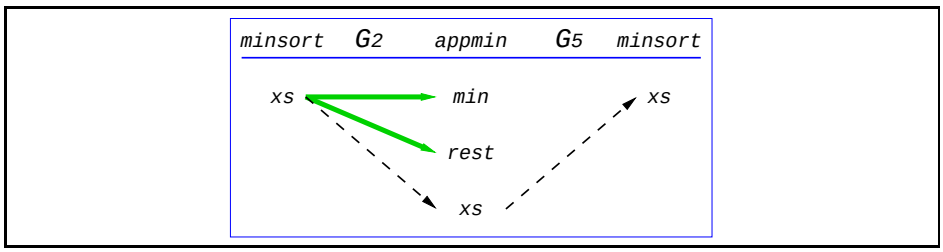

Figure 12.4.7: Critical call sequence in `minsort`

12.4.8 Contrived 1

The `contrived1` program addresses another useful point; The program contains a function `h` that is easily seen to size-change terminate: `h(r, s)` contains two function calls: `h(r, 2nd(s))` and `h(2nd(r), number42(Nil))`, where the function `number42` simply returns the number 42 encoded as a list. Due to the second call, the analyzer is unable to prove that the call depth is bounded by polynomial in the size of th input. The second argument is *not* bounded by the input, so the analyzer cannot conclude that the function is polynomially branching. This suggests that the input bounded criterion should be extended to "bounded by the size of the input *for sufficiently large input*".

12.4.9 The Trick

The program `thetrick` computes a function in polynomial time in the size of the input. The program `thetrickSize` is a variation where the conditionals have been pulled out of context and the tests have been reduced to a single test. Thus only the feasible recursive calls are accounted for by the analyzer, so size-change termination can be deduced.

The program runs in polynomial time but fail the analyzer fails to prove it – the problem arises from an immaterial dependency between the first and second argument: the program decreases the base-1 number in the first parameter, adding it to the second parameter. When the first argument reaches zero, the program starts to recur in the second argument. Thus the program recurs on a non-input bounded parameter, so the analysis cannot deduce that the program has a polynomial call depth in the size of the input.

12.4.10 Regular Expression Pattern Matcher

The program `rematch` takes a regular expression and a string and checks whether the string can be generated from the regular expression. This is done by first parsing the string containing the regular expression and then using the parse tree to perform the check. The analyzer fails to prove termination. Consider the call sequence: `parsep` $\overset{14}{\rightarrow}$ `parsepchar` $\overset{23}{\rightarrow}$ `parsep`, for which no decreasing thread exists. The problem is again that the analysis does not take conditionals

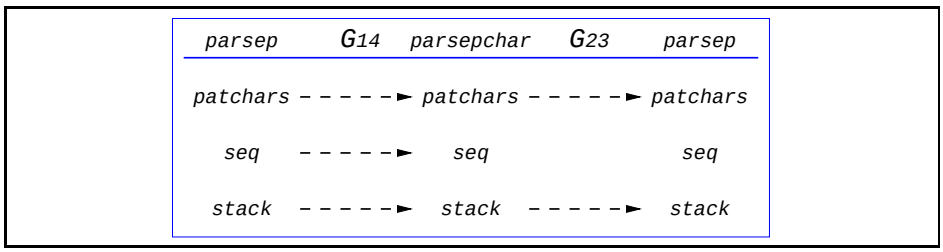

Figure 12.4.8: Critical call sequence in `rematch`

into account – the above call sequence is in fact infeasible due to the conditionals guarding the calls.

Chapter 13

Conclusion

In the present work, we presented an approach to conservatively decide whether a program a function in PTIMEF [12, 13] for a first order generally recursive function language with a well-founded data domain. The approach extends is size-change termination principle [14], and takes a program analysis perspective on complexity. For size-change terminating programs, the running time can be bound using three principles for obtaining bounds on the call depth, for handing non-linear calls and for handling sub-function calls respectively. The analysis is fully automatic and does not require special execution for the programs to conform to the deduced running time bounds. Furthermore, the analysis can automaticly correctly classify programs definable from primitive recursion on notation as computing functions in PTIMEF – i.e. the normal and safe parameters are automaticly identified.

The method is capable of handling a large set of naturally occurring algorithms, including programs with non-linear mutual recursion, but as is evident from the experiments, the analysis is sensitive to the implementation.

The function return size analysis proves to be quite useful. By employing linear programming combined with size analysis, it is possible to derive simple linear bounds. This is precisely what is needed by the algorithm, since the bounding parameters are not allowed to double during recursion.

Clearly there are truly many paths that can be taken to extend the capabilities of the analysis.

From a programmers perspective, it would be desirable to reduce the number of false positives and extend the class of programs that can be safely certified as computing functions in PTIMEF. By examining the test results, it is evident that the main bottle neck is the termination analysis, followed by the criterion for bounding the call depth. A problem that causes problems for the natural formulations of divide and conquer programs is the fact that the implementation of size-change termination is unable to handle mutually recursive functions with immaterial dependencies. This is usually the case when the program must recur on return values from a multi-valued function – implemented in "continuation passing style" – e.g. the part function for quicksort. The size analysis could

be extended to treat such functions as proper sub-function calls and there by utilizing the power of the function return size analysis.

Classification is also made difficult by the fact that the analysis does not account for the outcome of conditionals. A simple way to gain some extra power, would be to use program transformation prior to the analysis, e.g. super compilation [22] can be beneficial for some programs. In general, however, a more integrated approach would be desirable.

In interesting direction for future work would be to further investigate how arguments context transformations can be used to obtain closed form expressions for bounds on running time.

Appendix A

Examples

A.1 Jones Examples

ex1

```
1  --
2  -- Example 1: Reverse function, with accumulating parameter
3  --
4  rev(ls)    = r1(ls, Nil)
5  r1(ls, a) = if eq(ls, Nil) then a
6                 else r1(2nd(ls), Cons(1st(ls), a))
```

ex2

```
1  --
2  -- Example 2: Function with indirect recursion
3  --
4  f(i, x)     = if eq(i, Nil) then x else g(2nd(i), x, i)
5  g(a, b, c) = f(a, Cons(b, c))
```

ex3

```
1  --
2  -- Example 3: Function with lexically ordered parameters
3  --
4  a(m, n) = if eq(m, Z) then S(n) else
5            if eq(n, Z) then a(1st(m), S(Z))
6                        else a(1st(m), a(m, 1st(n)))
```

ex4

```
1  --
2  -- Example 4: Program with permuted parameters
3  --
4  p(m, n, r) = if gt(r, Z) then p(m, 1st(r), n) else
5               if gt(n, Z) then p(r, 1st(n), m)
6                           else m
```

ex5

```
--
-- Example 5: Program with permuted and possibly discarded parameters
--
f(x, y) = if eq(y, Nil) then x else
          if eq(x, Nil) then f(y, 2nd(y))
                        else f(y, 2nd(x))
```

ex6

```
--
-- Example 6:
-- Program with late starting sequence of descending parameter values
--
f(a, b) = if eq(b, Nil) then g(a, Nil)
                        else f(Cons(1st(b), a), 2nd(b))
g(c, d) = if eq(c, Nil) then d
                        else g(2nd(c), Cons(1st(c), d))
```

A.2 Wahlstedt Examples

ack

```
ack(x1,x2) = if eq(x1, Z)
               then S(x2)
               else if eq(x2, Z)
                      then ack(1st(x1), S(Z))
                      else ack(1st(x1), ack(x1, 1st(x2)))
```

add

```
add0(x1,x2) = if eq(x1, Z)
                then x2
                else S(add0(x2, 1st(x1)))
```

boolprog

```
-- Neil Jones et al. example of hard time and space complexity:
--
-- This example was used in [Jones00] to show PSPACE-hardness of the algorithm.
-- The hard part is to generate all the finite compositions of
-- size-change graphs. the more swappings the harder.
--
-- Should be hard to compute and does not size-change terminate.
--
 -- 1: X := not X
 -- 2: if Y goto 5 else 3
 -- 3: Y := not Y
 -- 4: if X goto 2 else 3
 -- 5: X := not X
 -- 6: Y := not Y
--
-- Each line is a function that calls another function(line). The
-- parameters are all the variables in the program, and their boolean
-- inverses, plus one extra argument z. We always have a call from the
-- last line to the first.  From each line there is one call to the next
-- line in the program, except for if-statements, where there are two
-- calls, one for each branch to corresponding line number.
--
-- So now we get nine graphs.
--
-- Below we construct a program that give the same set of size-change
-- graphs and the same call graph as the example above would give:
```

```
--
f0(x1,x2,x3,x4,x5)=if eq(x1, Z)
                       then Z
                       else if eq(x4,Z)
                              then Z
                              else f1(1st(x2),x2,1st(x4),x4,S(S(x5)))

f1(x1,x2,x3,x4,x5)=if eq(x5, Z)
                       then Z
                       else f2(x2,x1,x3,x4,1st(x5))

f2(x1,x2,x3,x4,x5)=
  if eq(x3, Z)
    then Z
    else if eq(x4, Z)
           then if eq(x5, Z)
                  then Z
                  else f3(x1,x2,1st(x3),x4,1st(x5))
           else if eq(x5, Z)
                  then Z
                  else f5(x1,x2,x3,1st(x4),1st(x5))

f3(x1,x2,x3,x4,x5)=if eq(x5, Z)
                       then Z
                       else f4(x1,x2,x4,x3,1st(x5))

f4(x1,x2,x3,x4,x5)=
  if eq(x1, Z)
    then Z
    else if eq(x2, Z)
           then if eq(x5, Z)
                  then Z
                  else f3(1st(x1),x2,x3,x4,1st(x5))
           else if eq(x5, Z)
                  then Z
                  else f2(x1,1st(x2),x3,x4,1st(x5))

f5(x1,x2,x3,x4,x5)=if eq(x5, Z)
                       then Z
                       else f6(x2,x1,x3,x4,1st(x5))

f6(x1,x2,x3,x4,x5)=if eq(x5, Z)
                       then Z
                       else f0(x1,x2,x4,x3,1st(x5))
```

div2

```
-- example of nested case on one variable:
div2(x1) = if eq(x1, Z)
             then Z
             else if eq(1st(x1), Z)
                    then Z
                    else S(div2(1st(1st(x1))))
```

eq

```
-- equality test
eq0(x1,x2) = if eq(x1, Z)
               then if eq(x2, Z)
                      then S(Z)
                      else Z
               else if eq(x2, Z)
                      then Z
                      else eq0(1st(x1), 1st(x2))
```

ex6

```
--
-- Ex 6 from [Jones 00]:
-- Program with late-starting sequence of decreasing parameter values:
--
-- f(a,b) = if b = [] then g(a,[]) else f(hd b:a, tl b)
-- g(c,d) = if c = [] then d else g(tl c, hd c:d)
--
--
--  in this language:
f(x1,x2) = if eq(x2, Z)
               then g(x1, Z)
               else f(S(x1), 1st(x2))
g(x1,x2) = if eq(x1, Z)
               then x2
               else g(1st(x1), S(x2))
```

fgh

```
-- example from Andreas Abel 3.12:
f(x1,x2) = if eq(x1, Z)
               then x1
               else if eq(x2, Z)
                      then x2
                      else h(g(1st(x1), x2), f(S(S(x1)), 1st(x2)))

g(x1,x2) = if eq(x1, Z)
               then x1
               else if eq(x2, Z)
                       then x2
                       else h(f(x1,x2), g(1st(x1), S(x2)))

h(x1,x2) = if eq(x1, Z)
               then if eq(x2, Z)
                      then x2
                      else h(x1, 1st(x2))
               else h(1st(x1), x2)
```

oddeven

```
-- odd / even function
even(x1) = if eq(x1, Z)
               then S(Z)
               else odd(1st(x1))
odd(x1)  = if eq(x1, Z)
               then Z
               else even(1st(x1))
```

permut

```
-- simple example of swapped parameters
f(x1,x2)=if eq(x1, Z)
            then Z
            else f(x2, 1st(x1))
```

A.3 Size Examples

assrewriteSize

```
--
-- Source file: basic\assrewrite.scm
-- Modified to call with multiple arguments
--
```

```
--- Rewrite expression with associative operator 'op'
---      a -> a1     b -> b1     c -> c1
--- ------------------------------------
--- '(op (op a b) c) -> '(op a1 (op b1 c1))
--- a != 'op   a -> a1   b -> b1              a != '(op ...)
--- -------------------------                -------------
---  '(op a b) -> '(op a1 b1)                   a -> a
assrewrite(exp) = rewrite(exp)
rewrite(exp) =
  if and(eq(exp, Cons), eq(Op, 1st(exp)))
    then rw(1st(2nd(exp)), 1st(2nd(2nd(exp))))
    else exp
rw(opab, c) =
        if and(eq(opab, Cons), eq(Op, 1st(opab)))
          then
            let
              a1 = rewrite(1st(2nd(opab)))
              b1 = rewrite(1st(2nd(2nd(opab))))
              c1 = rewrite(c)
            in
              rw(a1, Cons(Op, Cons(b1, Cons(c1, Nil))))
          else Cons(Op, Cons(rewrite(opab), Cons(rewrite(c), Nil)))
```

bubblesort

```
--
-- Bubblesort
--
bubblesort(xs) = bsort(len(xs), xs)
bsort(l, xs) =
  if eq(l, Z)
    then xs
    else bsort(1st(l), bubble(xs))
bubble(x, xs) =
  if eq(xs, Nil)
    then Cons(x, Nil)
    else if lt(x, 1st(x))
      then Cons(x, bubble(1st(xs), 2nd(xs)))
      else Cons(1st(xs), bubble(x, 2nd(xs)))
len(xs) =
  if eq(xs, Nil)
    then Z
    else S(len(2nd(xs)))
```

inssort

```
--
-- Insertion sort
--
inssort(xs) = isort(xs, Nil)
isort(xs, r) =
  if eq(xs, Nil)
    then Nil
    else isort(2nd(xs), insert(1st(xs), r))
insert(x, r) =
  if lt(x, 1st(x))
    then Cons(x, r)
    else Cons(1st(r), insert(x, 2nd(r)))
```

minsortSize

```
--
-- Minsort
--
minsort(xs) =
  if eq(xs, Cons)
```

```
6       then appmin(1st(xs), 2nd(xs), xs)
7       else Nil
8   appmin(min, rest, xs) =
9     if eq(rest, Cons)
10      then
11        if lt(1st(rest), min)
12          then appmin(1st(rest), 2nd(rest), xs)
13          else appmin(min, 2nd(rest), xs)
14      else Cons(min, minsort(remove(min, xs)))
15  remove(x, xs) =
16      if equal(x, 1st(xs))
17        then 2nd(xs)
18        else Cons(1st(xs), remove(x, 2nd(xs)))
```

deadcodeSize

```
1   --
2   -- Dead code example
3   --
4   f(x) =
5     if eq(x, Z)
6       then x
7       else 1st(x)
8   g(x) = g(x)
```

fghSize

```
1   -- example from Andreas Abel 3.12:
2   -- Modified to size-change terminate.
3   f(x1,x2) = if eq(x1, Z)
4                 then x1
5                 else if eq(x2, Z)
6                        then x2
7                        else h(g(1st(x1), x2), f(S(x1), 1st(x2)))
8
9   g(x1,x2) = if eq(x1, Z)
10                then x1
11                else if eq(x2, Z)
12                       then x2
13                       else h(f(x1,x2), g(1st(x1), S(x2)))
14
15  h(x1,x2) = if eq(x1, Z)
16                then if eq(x2, Z)
17                       then x2
18                       else h(x1, 1st(x2))
19                else h(1st(x1), x2)
```

gexgcd

```
1   ------------------------------------------
2   -- gcd paramenters: x, y, res, tmp, mtmp, t --
3   ------------------------------------------
4   gcd(x, y) = l1(x, y, Z, Z, Z, Z)
5
6   -- if x<0
7   l1(x, y, res, tmp, mtmp, t) = l2(x, y, res, tmp, mtmp, lt(x, Z))
8   l2(x, y, res, tmp, mtmp, t) = if t then l16(x, y, res, tmp, mtmp, t)
9                                      else l3(x, y, res, tmp, mtmp, t)
10  -- 4:res = 0;
11  l3(x, y, res, tmp, mtmp, t) = l4(x, y, Z, tmp, mtmp, t)
12
13  -- if y<0
14  l4(x, y, res, tmp, mtmp, t) = l5(x, y, res, tmp, mtmp, lt(y, Z))
15  l5(x, y, res, tmp, mtmp, t) = if t then l6(x, y, res, tmp, mtmp, t)
16                                     else l7(x, y, res, tmp, mtmp, t)
```

```
17
18  -- 5: res=0;
19  l6(x, y, res, tmp, mtmp, t) = l16(x, y, Z, tmp, mtmp, t)
20
21  -- 6: tmp=equal0(x,y)
22  l7(x, y, res, tmp, mtmp, t) = l8(x, y, res, equal0(x, y), mtmp, t)
23
24  -- if tmp
25  l8(x, y, res, tmp, mtmp, t) = if tmp then l9(x, y, res, tmp, mtmp, t)
26                                       else l10(x, y, res, tmp, mtmp, t)
27
28  -- 7: res=x;
29  l9(x, y, res, tmp, mtmp, t) = l16(x, y, x, tmp, mtmp, t)
30
31  -- if x<y
32  l10(x, y, res, tmp, mtmp, t) = l11(x, y, res, tmp, mtmp, lt(x, y))
33  l11(x, y, res, tmp, mtmp, t) = if t then l12(x, y, res, tmp, mtmp, t)
34                                     else l14(x, y, res, tmp, mtmp, t)
35  -- mtmp = monus(x,y);
36  l12(x, y, res, tmp, mtmp, t) = l13(x, y, res, tmp, monus(x, y), t)
37
38  -- res = gcd(mtmp, y);
39  l13(x, y, res, tmp, mtmp, t) = l16(x, y, gcd(mtmp, y), tmp, mtmp, t)
40
41  -- 8: mtmp = monus(y, x);
42  l14(x, y, res, tmp, mtmp, t) = l15(x, y, res, tmp, monus(y, x), t)
43
44  -- 9: res = gcd(y, mtmp);
45  l15(x, y, res, tmp, mtmp, t) = l16(x, y, gcd(y, mtmp), tmp, mtmp, t)
46
47  -- return res;
48  l16(x, y, res, tmp, mtmp, t) = res
49
50  ------------------------------------
51  -- equal parameters: a, b, res, t --
52  ------------------------------------
53  equal0(a,b) = e1(a, b, Z, Z)
54
55  -- if a<b
56  e1(a, b, res, t) = e2(a, b, res, lt(a, b))
57  e2(a, b, res, t) = if t then e3(a, b, res, t)
58                          else e7(a, b, res, t)
59
60  -- if b<a
61  e3(a, b, res, t) = e4(a, b, res, lt(b, a))
62  e4(a, b, res, t) = if t then e5(a, b, res, t)
63                          else e6(a, b, res, t)
64
65  -- 1:res=1;
66  e5(a, b, res, t) = e8(a, b, S(Z), t)
67
68  -- 2:res=0;
69  e6(a, b, res, t) = e8(a, b, Z, t)
70
71  -- 3:res=0;
72  e7(a, b, res, t) = e8(a, b, Z, t)
73
74  -- return res;
75  e8(a, b, res, t) = res
76
77
78  ------------------------------------
79  -- monus parameters: a, b, res, t --
80  ------------------------------------
81  monus(a, b) = m1(a, b, Z, Z)
82
83  -- if b<0
84  m1(a, b, res, t) = m2(a, b, res, lt(b, Z))
```

```
m2(a, b, res, t) = if t then m3(a, b, res, t)
                           else m4(a, b, res, t)

-- 10:res=pred(a);
m3(a, b, res, t) = m5(a, b, 1st(a), t)

-- 11:res=monus(pred(a), pred(b))
m4(a, b, res, t) = m5(a, b, monus(1st(a), 1st(b)), t)

-- return res;
m5(a, b, res, t) = res
```

gexgcd2

```
-------------------------------------------
-- gcd paramenters: x, y, res, tmp, mtmp, t --
-------------------------------------------
gcd(x, y) =
  if lt(x,Z) then
    Z
  else
    if lt(y,Z) then
      Z
    else
      if equal0(x,y) then
        x
      else
        if lt(x,y) then
          gcd(monus(x,y),y)
        else
          gcd(y, monus(y,x))

-----------------------------------
-- equal parameters: a, b, res, t --
-----------------------------------
equal0(a,b) =
  if lt(a, b) then
    if lt(b,a) then
      S(Z)
    else
      Z
  else
    Z

-----------------------------------
-- monus parameters: a, b, res, t --
-----------------------------------
monus(a, b) =
  if lt(b, Z) then
    1st(a)
  else
    monus(1st(a), 1st(b))
```

graphcolour2Size

```
--
-- Source file: algorithms\graphcolour2.scm
--
--- Colour graph G with colours cs so that neighbors have different
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
```

```
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
--
-- Modified to call with ncs instead of cs1 in colorrestthetrick.
--
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
    else colorednodes
colornoderest(cs, ncs, node, colorednodes, rest) =
  if eq(ncs, Cons)
    then
      if possible(1st(ncs), 2nd(node), colorednodes)
        then
          let
            colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes),
                                2nd(rest))
          in
            if eq(colored, Cons)
              then colored
              else
                if eq(ncs, Cons)
                  then colorrestthetrick(cs, cs, 2nd(ncs), colorednodes, rest)
                  else Nil
        else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
    else Nil
colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
  if equal(cs1, ncs)
    then colorrest(cs, ncs, colorednodes, rest)
    else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

quicksortSize

```
1  --
2  -- Source file: sorting\quicksort.scm
3  --
4  -- Quicksort
5  -- Modified from Glenstrup, note that duplets are thrown away
6  -- as in the original implementation.
7  goal(xs) = quicksort(xs)
8  quicksort(xs) =
9    if eq(xs, Cons)
10     then
11       if eq(2nd(xs), Cons)
12         then part(1st(xs), 2nd(xs))
13         else xs
14     else xs
15 part(x, xs) =
16     app(quicksort(partLt(x, xs)), Cons(x, quicksort(partGt(x, xs))))
17 partLt(x, xs) =
18   if eq(xs, Cons)
19     then
20       if lt(1st(xs), x)
21         then Cons(1st(xs), partLt(x, 2nd(xs)))
22         else partLt(x, 2nd(xs))
23     else Nil
24 partGt(x, xs) =
25   if eq(xs, Cons)
26     then
27       if gt(1st(xs), x)
28         then Cons(1st(xs), partGt(x, 2nd(xs)))
29         else partGt(x, 2nd(xs))
30     else Nil
31 app(xs, ys) =
32   if eq(xs, Cons)
33     then Cons(1st(xs), app(2nd(xs), ys))
34     else ys
```

thetrickSize

```
1  --
2  -- Source file: basic\thetrick.scm
3  --
4  -- The trick: pulling out the conditional into the context
5  -- Modified: conditional pulled out of the context, 42 -> 4
6  goal(x, y) = Cons(f(x, y), Cons(g(x, y), Nil))
7  f(x, y) =
8    if eq(y, Nil)
9      then number4(Nil)
10     else if lt0(x, Cons(Nil, Nil))
11            then f(x, 2nd(y))
12            else f(2nd(x), Cons(Cons(Nil, Nil), y))
13 g(x, y) =
14   if eq(y, Nil)
15     then number4(Nil)
16     else
17       if lt0(x, Cons(Nil, Nil))
18         then g(x, 2nd(y))
19         else g(2nd(x), Cons(Cons(Nil, Nil), y))
20 lt0(x, y) =
21   if eq(y, Nil)
22     then False
23     else
24       if eq(x, Nil)
25         then True
26         else lt0(2nd(x), 2nd(y))
27 number4(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
```

power

```
--
-- Source file: simple\power.scm
--
-- Power function: x to the nth power
-- (numbers represented in unary)
power(x, n) =
  if equal(n, Z)
    then S(Z)
    else mult(x, power(x, 2nd(n)))
mult(x, y) =
  if equal(y, Z)
    then Z
    else add0(x, mult(x, 2nd(y)))
add0(x, y) =
  if equal(y, Z)
    then x
    else S(add0(x, 2nd(y)))
```

A.4 Ptime Examples

disj1

```
--
-- Source: disj1.l
--
f(x) =
  if eq(x, Z)
    then x
    else Cons(f(1st(x)), f(2nd(immatcp(x))))
immatcp(x) =
  if eq(x, Z)
    then x
    else Cons(1st(x), immatcp(2nd(x)))
```

disj2

```
--
-- Source: Disj2.l
--
f(x) = if eq(x, Z) then Z else Cons(f(1st(x)), f(2nd(g(x))))
g(x) = x
```

dup1

```
--
-- Source: dup1.l
--
-- Example of duplication between a critical var and a non-critical var
--
a(x,y) = if eq(x,Z) then x else Cons(a(1st(x),y), a(2nd(x),x))
```

dup2

```
--
-- Source: dup2.l
--
-- Example of duplication between two non-critical vars
--
a(x,y,z) = if eq(x,Z) then x else Cons(a(1st(x),y,z), a(2nd(x),y,y))
```

ocall-safe

```
1  --
2  -- Source: ocall-safe.l
3  --
4  safe(x) = if eq(x,Z) then x else dbl(Z,safe(1st(x)))
5
6  dbl(x,y) = if eq(x,Z) then y else dbl(1st(x),S(S(S(y))))
```

ocall-unsafe

```
1  --
2  -- Source: ocall-unsafe.l
3  --
4  unsafe(x) = if eq(x,Z) then x else dbl(unsafe(1st(x)),Z)
5
6  dbl(x,y) = if eq(x,Z) then y else dbl(1st(x),S(S(S(y))))
```

quicksortPtime

```
1  --
2  -- Source: quicksortPtime.l
3  --
4  -- Quicksort, split is performed by a sub-function
5  --
6  quicksort(xs) =
7    if eq(xs, Cons)
8      then
9        if eq(2nd(xs), Cons)
10         then qs(1st(xs), part(1st(xs), 2nd(xs), Nil, Nil))
11         else xs
12     else xs
13 qs(x, ps) =
14   app(quicksort(1st(ps)), Cons(x, quicksort(2nd(ps))))
15 part(x, xs, xs1, xs2) =
16   if eq(xs, Cons)
17     then
18       if gt(x, 1st(xs))
19         then part(x, 2nd(xs), Cons(1st(xs), xs1), xs2)
20         else
21           if lt(x, 1st(xs))
22             then part(x, 2nd(xs), xs1, Cons(1st(xs), xs2))
23             else part(x, 2nd(xs), xs1, xs2)
24     else Cons(xs1, xs2)
25 app(xs, ys) =
26   if eq(xs, Cons)
27     then Cons(1st(xs), app(2nd(xs), ys))
28     else ys
```

A.5 Glenstrup Examples - Algorithms

graphcolour1

```
1  --
2  -- Source file: algorithms\graphcolour1.scm
3  --
4  --- Colour graph G with colours cs so that neighbors have different colours
5  --- The graph is represented as a list of nodes with adjacency lists
6  --- Example:
7  ---  '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
8  ---    (e . (b c f)) (f . (c d e)))
9  --- Colour a node by appending a colour list to the node. The head of
10 --- the list is the chosen colour, the tail are the yet untried
11 --- colours. If impossible, return nil.
12 --- Example of coloured node: '((red blue yellow) . (a . (b c d)))
```

```
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then
      let
        colorednode = colornode(ncs, 1st(rest), colorednodes)
      in

        if eq(colorednode, Cons)
          then
            let
              colored = colorrest(cs, cs, Cons(colorednode, colorednodes), 2nd(rest))
            in

              if eq(colored, Cons)
                then colored
                else
                  if eq(1st(colorednode), Cons)
                    then colorrestthetrick(cs, cs, 2nd(1st(colorednode)), colorednodes, rest)
                    else Nil
          else Nil
    else colorednodes
colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
  if equal(cs1, ncs)
    then colorrest(cs, cs1, colorednodes, rest)
    else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

graphcolour2

```
--
-- Source file: algorithms\graphcolour2.scm
--
--- Colour graph G with colours cs so that neighbors have different
```

```
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with these adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
    else colorednodes
colornoderest(cs, ncs, node, colorednodes, rest) =
  if eq(ncs, Cons)
    then
      if possible(1st(ncs), 2nd(node), colorednodes)
        then
          let
            colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes), 2nd(rest))
          in

            if eq(colored, Cons)
              then colored
              else
                if eq(ncs, Cons)
                  then colorrestthetrick(cs, cs, 2nd(ncs), colorednodes, rest)
                  else Nil
        else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
    else Nil
colorrestthetrick(cs1, cs, ncs, colorednodes, rest) =
  if equal(cs1, ncs)
    then colorrest(cs, cs1, colorednodes, rest)
    else colorrestthetrick(2nd(cs1), cs, ncs, colorednodes, rest)
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
```

```
      then revapp(2nd(xs), Cons(1st(xs), rest))
      else rest
```

graphcolour3

```
--
-- Source file: algorithms\graphcolour3.scm
--
--- Colour graph G with colours cs so that neighbors have different
--- colours (slightly tail recursive version)
--- The graph is represented as a list of nodes with adjacency lists
--- Example:
--- '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
---   (e . (b c f)) (f . (c d e)))
--- Colour a node by appending a colour list to the node. The head of
--- the list is the chosen colour, the tail are the yet untried
--- colours. If impossible, return nil.
--- Example of coloured node: '((red blue yellow) . (a . (b c d)))
--- Can we use color with adjacent nodes and current coloured nodes
--- Return colour of node. If no colour yet, return nil.
--- Colour the first node of rest with colours from ncs, and
--- colour remaining nodes. If impossible, return nil.
--- Like colornode, only continue with colouring the rest
graphcolour(g, cs) =
  let
    ns = g
  in
    reverse(colorrest(cs, cs, Cons(colornode(cs, 1st(ns), Nil), Nil), 2nd(ns)))
colornode(cs, node, colorednodes) =
  if eq(cs, Cons)
    then
      if possible(1st(cs), 2nd(node), colorednodes)
        then Cons(cs, node)
        else colornode(2nd(cs), node, colorednodes)
    else Nil
possible(color, adjs, colorednodes) =
  if eq(adjs, Cons)
    then
      if equal(color, colorof(1st(adjs), colorednodes))
        then False
        else possible(color, 2nd(adjs), colorednodes)
    else True
colorof(node, colorednodes) =
  if eq(colorednodes, Cons)
    then
      if equal(1st(2nd(1st(colorednodes))), node)
        then 1st(1st(1st(colorednodes)))
        else colorof(node, 2nd(colorednodes))
    else Nil
colorrest(cs, ncs, colorednodes, rest) =
  if eq(rest, Cons)
    then colornoderest(cs, ncs, 1st(rest), colorednodes, rest)
    else colorednodes
colornoderest(cs, ncs, node, colorednodes, rest) =
  if eq(ncs, Cons)
    then
      if possible(1st(ncs), 2nd(node), colorednodes)
        then
          let
            colored = colorrest(cs, cs, Cons(Cons(ncs, node), colorednodes), 2nd(rest))
          in

            if eq(colored, Cons)
              then colored
              else
                if eq(ncs, Cons)
                  then colorrest(cs, 2nd(ncs), colorednodes, rest)
```

```
                    else Nil
          else colornoderest(cs, 2nd(ncs), node, colorednodes, rest)
      else Nil
reverse(xs) = revapp(xs, Nil)
revapp(xs, rest) =
  if eq(xs, Cons)
    then revapp(2nd(xs), Cons(1st(xs), rest))
    else rest
```

match

```
--
-- Source file: algorithms\match.scm
--
-- Simple pattern matcher
match(p, s) = loop(p, s, p, s)
loop(p, s, pp, ss) =
  if equal(p, Nil)
    then True
    else
      if equal(s, Nil)
        then False
        else
          if equal(1st(p), 1st(s))
            then loop(2nd(p), 2nd(s), pp, ss)
            else loop(pp, 2nd(ss), pp, 2nd(ss))
```

reach

```
--
-- Source file: algorithms\reach.scm
--
--- How can node v be reached from node u in a directed graph.
--- Graph example: '((a . b) (a . d) (b . d) (c . a))
goal(u, v, edges) = reach(u, v, edges)
reach(u, v, edges) =
  if member(Cons(u, v), edges)
    then Cons(Cons(u, v), Nil)
    else via(u, v, edges, edges)
via(u, v, rest, edges) =
  if equal(rest, Nil)
    then Nil
    else
      if equal(u, 1st(1st(rest)))
        then
          let
            path = reach(2nd(1st(rest)), v, edges)
          in

            if equal(path, Nil)
              then via(u, v, 2nd(rest), edges)
              else Cons(1st(rest), path)
        else via(u, v, 2nd(rest), edges)
member(x, xs) =
  if equal(xs, Nil)
    then False
    else
      if equal(x, 1st(xs))
        then True
        else member(x, 2nd(xs))
```

rematch

```
--
-- Source file: algorithms\rematch.scm
--
```

```
--- Regular expression pattern matcher
---
--- pat ::= . | character | \ character
---       |  pat* | (pat) | pat ... pat
--- When parsed, this is represented by:
--- pat ::= ('dot) | ('char c) | ('star pat) | ('seq pat ... pat)
--
-- Modification: char -> symbols, string ->symbols
rematch(patstr, str) =
  let
    matchrest = domatch(parsepat(patstr), stringlist(str))
  in

    if eq(matchrest, Cons)
      then Cons(liststring(reverse(1st(matchrest))), liststring(2nd(matchrest)))
      else matchrest
parsepat(patstr) = parsep(stringlist(patstr), Nil, Nil)
parsep(patchars, seq, stack) =
  if eq(patchars, Cons)
    then
      if equal(Cdot, 1st(patchars))
        then parsepdot(patchars, seq, stack)
        else
          if equal(Cstar, 1st(patchars))
            then parsepstar(patchars, seq, stack)
            else
              if equal(Clpar, 1st(patchars))
                then parsepopenb(patchars, seq, stack)
                else
                  if equal(Crpar, 1st(patchars))
                    then parsepcloseb(patchars, seq, stack)
                    else
                      if equal(Cslash, 1st(patchars))
                        then parsepchar(2nd(patchars), seq, stack)
                        else parsepchar(patchars, seq, stack)
    else
      if eq(stack, Cons)
        then error(Unmatchedlparinpattern)
        else Cons(Seq, reverse(seq))
parsepdot(patchars, seq, stack) = parsep(2nd(patchars), Cons(Cons(Cdot, Nil), seq), stack)
parsepstar(patchars, seq, stack) =
  if eq(seq, Cons)
    then parsep(2nd(patchars), Cons(Cons(Cstar, Cons(1st(seq), Nil)), 2nd(seq)), stack)
    else parsep(2nd(patchars), Cons(Cons(Char, Cons(Cstar, Nil)), Nil), stack)
parsepopenb(patchars, seq, stack) = parsep(2nd(patchars), Nil, Cons(seq, stack))
parsepcloseb(patchars, seq, stack) =
  if eq(stack, Cons)
    then parsep(2nd(patchars), Cons(Cons(Seq, reverse(seq)), 1st(stack)), 2nd(stack))
    else error(Unmatchedrparinpattern)
parsepchar(patchars, seq, stack) =
  if eq(patchars, Cons)
    then parsep(2nd(patchars), Cons(Cons(Char, Cons(1st(patchars), Nil)), seq), stack)
    else parsep(patchars, Cons(Cons(Char, Cons(Cslash, Nil)), seq), stack)
domatch(pat, cs) =
  if eq(pat, Cons)
    then
      if equal(1st(pat), Dot)
        then domatchdot(cs)
        else
          if equal(1st(pat), Char)
            then domatchchar(cs, 1st(2nd(pat)))
            else
              if equal(1st(pat), Star)
                then domatchstar(cs, 1st(2nd(pat)), Nil)
                else
                  if equal(1st(pat), Seq)
                    then domatchseq(cs, Nil, 2nd(pat))
                    else error(Unknownpatterndata)
```

```
72      else Cons(Nil, cs)
73  domatchdot(cs) =
74    if eq(cs, Cons)
75      then Cons(Cons(1st(cs), Nil), 2nd(cs))
76      else Nomatch
77  domatchchar(cs, c) =
78    if eq(cs, Cons)
79      then
80        if equal(1st(cs), c)
81          then Cons(Cons(1st(cs), Nil), 2nd(cs))
82          else Nomatch
83      else Nomatch
84  domatchstar(cs, pat, init) =
85    if eq(cs, Cons)
86      then
87        let
88          first = domatch(pat, cs)
89        in
90
91          if eq(first, Cons)
92            then domatchstar(2nd(first), pat, append(1st(first), init))
93            else Cons(init, cs)
94      else Cons(init, cs)
95  domatchseq(cs, rest, pats) =
96    if eq(pats, Cons)
97      then
98        let
99          first = domatch(1st(pats), cs)
100       in
101
102         if eq(first, Cons)
103           then
104             let
105               next = domatchseq(append(2nd(first), rest), Nil, 2nd(pats))
106             in
107
108               if eq(next, Cons)
109                 then Cons(append(1st(next), 1st(first)), 2nd(next))
110                 else
111                   if eq(1st(first), Cons)
112                     then domatchseq(reverse(2nd(1st(first))), Cons(1st(1st(first)), append(2nd(fi
113                     else Nomatch
114           else Nomatch
115     else Cons(Nil, append(cs, rest))
116 liststring(x) = dummy(x)
117 stringlist(x) = dummy(x)
118 reverse(x) = x
119 append(x, y) = Cons(x, y)
120 dummy(x) =
121   if True
122     then x
123     else Cons(Nil, stringlist(x))
```

strmatch

```
1  --
2  -- Source file: algorithms\strmatch.scm
3  --
4  --- Naive pattern string matcher
5  --
6  -- Modified to not call scm conversion functions
7  strmatch(patstr, str) = domatch(patstr, str, Nil)
8  domatch(patcs, cs, n) =
9    if eq(cs, Cons)
10     then
11       if prefix(patcs, cs)
12         then Cons(n, domatch(patcs, 2nd(cs), add(n, Cons(Nil, Nil))))
13         else domatch(patcs, 2nd(cs), add(n, Cons(Nil, Nil)))
```

```
      else
        if equal(patcs, cs)
          then Cons(n, Nil)
          else Nil
prefix(precs, cs) =
  if eq(precs, Cons)
    then
      if eq(cs, Cons)
        then and(equal(1st(precs), 1st(cs)), prefix(2nd(precs), 2nd(cs)))
        else False
    else True
```

typeinf

```
--
-- Source file: algorithms\typeinf.scm
--
--- Type inference for the Typed Lambda Calculus
--- e ::= ('var . x)              variable x
---     | ('apply . (e1 . e2))    apply abstraction e1 to expression e2
---     | ('lambda . (x . e1))    make lambda abstraction
--- t ::= ('tyvar . a)
---     | ('arrow . (t1 . t2))
-----------------------------------------
--- (define inittenv 1) - tenv simply holds the next fresh type variables
-- fixed a call to error arity 2 -> 1
typeinf(inittenv, e) =
  let
    atenv = freshtvar(inittenv)
  in
    1st(etype(Nil, 2nd(atenv), e, 1st(atenv)))
freshtvar(tenv) = Cons(Cons(Tvar, tenv), add(tenv, Cons(Nil, Nil)))
vtype(venv, x) =
  if equal(x, 1st(1st(venv)))
    then 2nd(1st(venv))
    else vtype(2nd(venv), x)
tsubst(a, t, t1) =
  if equal(Tvar, 1st(t1))
    then
      if equal(a, 2nd(t1))
        then t
        else t1
    else
      if equal(Arr, 1st(t1))
        then Cons(Arr, Cons(tsubst(a, t, 1st(2nd(t1))), tsubst(a, t, 2nd(2nd(t1)))))
        else error(Tsubstt1)
subst(venv, a, t) =
  if eq(venv, Cons)
    then Cons(Cons(1st(1st(venv)), tsubst(a, t, 2nd(1st(venv)))), subst(2nd(venv), a, t))
    else Nil
unify(venv, t1, t2) =
  if equal(Tvar, 1st(t1))
    then Cons(subst(venv, 2nd(t1), t2), t2)
    else
      if equal(Arr, 1st(t1))
        then
          if equal(Tvar, 1st(t2))
            then Cons(subst(venv, 2nd(t2), t1), t1)
            else
              if equal(Arr, 1st(t2))
                then
                  let
                    venv1tx1 = unify(venv, 1st(2nd(t1)), 2nd(2nd(t1)))
                    venv2tx2 = unify(1st(venv1tx1), 1st(2nd(t2)), 2nd(2nd(t2)))
                  in
                    Cons(1st(venv2tx2), Cons(Arr, Cons(2nd(venv1tx1), 2nd(venv2tx2))))
                else error(Unifyt2)
```

```
54          else error(Unifyt1)
55  etype(venv, tenv, e, t) =
56    if equal(Var, 1st(e))
57      then
58        let
59          venv1t1 = unify(venv, vtype(venv, 2nd(e)), t)
60        in
61          Cons(2nd(venv1t1), Cons(1st(venv1t1), tenv))
62      else
63        if equal(App, 1st(e))
64          then
65            let
66              atenv1 = freshtvar(tenv)
67              t2venv2tenv2 = etype(venv, 2nd(atenv1), 1st(2nd(e)), 1st(atenv1))
68              t1venv3tenv3 = etype(1st(2nd(t2venv2tenv2)), 2nd(2nd(t2venv2tenv2)), 2nd(2nd(e)), Con
69              t1 = 1st(t1venv3tenv3)
70            in
71              Cons(2nd(2nd(t1)), 2nd(t1venv3tenv3))
72          else
73            if equal(Lam, 1st(e))
74              then
75                let
76                  atenv1 = freshtvar(tenv)
77                  t1venv2tenv2 = etype(Cons(Cons(1st(2nd(e)), 1st(atenv1)), venv), 2nd(atenv1), 2nd
78                  venv3t3 = unify(1st(2nd(t1venv2tenv2)), Cons(Arr, Cons(vtype(2nd(2nd(t1venv2tenv2
79                in
80                  Cons(2nd(venv3t3), Cons(2nd(1st(venv3t3)), 2nd(2nd(t1venv2tenv2))))
81              else error(Errorinlambdaexpression)
```

A.6 Glenstrup Examples - Basic

add

```
1  --
2  -- Source file: basic\add.scm
3  --
4  -- Add two numbers unarily represented as '(s s s ... s)
5  goal(x, y) = add0(x, y)
6  add0(x, y) =
7    if equal(y, Nil)
8      then x
9      else add0(Cons(Cons(Nil, Nil), x), 2nd(y))
```

addlists

```
1  --
2  -- Source file: basic\addlists.scm
3  --
4  --- Add two lists elementwise
5  goal(xs, ys) = addlist(xs, ys)
6  addlist(xs, ys) =
7    if eq(xs, Cons)
8      then Cons(add(1st(xs), 1st(ys)), addlist(2nd(xs), 2nd(ys)))
9      else Nil
```

anchored

```
1  --
2  -- Source file: basic\anchored.scm
3  --
4  -- Parameter y anchored in parameter x
5  goal(x, y) = anchored(x, y)
6  anchored(x, y) =
7    if equal(x, Nil)
8      then y
9      else anchored(2nd(x), Cons(Cons(Nil, Nil), y))
```

append

```
--
-- Source file: basic\append.scm
--
goal(x, y) = append(x, y)
append(xs, ys) =
  if equal(xs, Nil)
    then ys
    else Cons(1st(xs), append(2nd(xs), ys))
```

assrewrite

```
--
-- Source file: basic\assrewrite.scm
--
--- Rewrite expression with associative operator `op'
---      a -> a1     b -> b1     c -> c1
--- ---------------------------------
--- '(op (op a b) c) -> '(op a1 (op b1 c1))
---a != 'op  a -> a1  b -> b1              a != '(op ...)
--- ------------------------               ------------
---  '(op a b) -> '(op a1 b1)                 a -> a
assrewrite(exp) = rewrite(exp)
rewrite(exp) =
  if and(eq(exp, Cons), equal(Op, 1st(exp)))
    then
      let
        opab = 1st(2nd(exp))
      in

        if and(eq(opab, Cons), equal(Op, 1st(opab)))
          then
            let
              a1 = rewrite(1st(2nd(opab)))
              b1 = rewrite(1st(2nd(2nd(opab))))
              c1 = rewrite(1st(2nd(2nd(exp))))
            in
              rewrite(Cons(1st(exp), Cons(a1, Cons(Cons(1st(opab), Cons(b1, Cons(c1, 2nd(2nd(2nd(
          else Cons(1st(exp), Cons(rewrite(1st(2nd(exp))), Cons(rewrite(1st(2nd(2nd(exp))))), 2nd(
    else exp
```

badd

```
--
-- Source file: basic\badd.scm
--
goal(x, y) = badd(x, y)
badd(x, y) =
  if equal(y, Nil)
    then x
    else badd(Cons(Nil, Nil), badd(x, 2nd(y)))
```

contrived1

```
--
-- Source file: basic\contrived1.scm
--
-- A contrived example from Arne Glenstrup's Master's Thesis
-- Numbers represented by list length
contrived1(a, b) = f(a, Cons(Cons(Nil, Nil), Cons(Cons(Nil, Nil), a)), a, b)
f(x, y, z, d) =
  if and(gt(z, Zero), gt(d, Zero))
    then f(Cons(Cons(Nil, Nil), x), z, 2nd(z), 2nd(d))
    else
      if gt(y, Zero)
```

```
12          then g(x, 2nd(y), d)
13          else x
14  g(u, v, w) =
15    if gt(w, Zero)
16      then f(Cons(Cons(Nil, Nil), u),
17           if equal(h(v, Zero), Zero)
18             then v
19             else dec(v), 2nd(v), 2nd(w))
20      else u
21  h(r, s) =
22    if gt(r, Zero)
23      then h(2nd(r), number42(Nil))
24      else
25        if gt(s, Zero)
26          then h(r, 2nd(s))
27          else r
28  dec(n) = 2nd(n)
29  number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

contrived2

```
1   --
2   -- Source file: basic\contrived2.scm
3   --
4   -- A contrived example from Arne Glenstrup's Master's Thesis
5   -- Numbers represented by list length
6   contrived2(a, b) = f(a, Cons(Cons(Nil, Nil), Cons(Cons(Nil, Nil), a)), a, b)
7   f(x, y, z, d) =
8     if and(gt(z, Zero), gt(d, Zero))
9       then f(Cons(Cons(Nil, Nil), x), z, 2nd(z), 2nd(d))
10      else
11        if gt(y, Zero)
12          then g(x, 2nd(y), d)
13          else x
14  g(u, v, w) =
15    if gt(w, Zero)
16      then f(Cons(Cons(Nil, Nil), u),
17           if equal(h(v, Zero), Zero)
18             then v
19             else dec(v), 2nd(v), 2nd(w))
20      else u
21  h(r, s) =
22    if gt(r, Zero)
23      then h(2nd(r), number42(Nil))
24      else
25        if gt(s, Zero)
26          then h(number17(Nil), 2nd(s))
27          else r
28  dec(n) = 2nd(n)
29  number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
30  number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

decrease

```
1   --
2   -- Source file: basic\decrease.scm
3   --
4   goal(x) = decrease(x)
5   decrease(x) =
6     if equal(x, Nil)
7       then number42(Nil)
8       else decrease(2nd(x))
9   number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

deeprev

```
--
-- Source file: basic\deeprev.scm
--
--- Recursively reverse all list elements in a data structure
--- Example: (deeprev '((1 2 3) 4 5 6 (8 (9 10 11)) . 12))
---       ===>((3 2 1) 4 5 6 ((11 10 9) 8) . 12)
goal(x) = deeprev(x)
deeprev(x) =
  if eq(x, Cons)
    then deeprevapp(x, Nil)
    else x
deeprevapp(xs, rest) =
  if eq(xs, Cons)
    then deeprevapp(2nd(xs), Cons(deeprev(1st(xs)), rest))
    else
      if equal(xs, Nil)
        then rest
        else revconsapp(rest, xs)
revconsapp(xs, r) =
  if eq(xs, Cons)
    then revconsapp(2nd(xs), Cons(1st(xs), r))
    else r
```

disjconj

```
--
-- Source file: basic\disjconj.scm
--
--- Predicates for disjunctive and conjunctive terms p
disjconj(p) = disj(p)
disj(p) =
  if eq(p, Cons)
    then
      if equal(Or, 1st(p))
        then and(conj(1st(2nd(p))), disj(2nd(2nd(p))))
        else conj(p)
    else conj(p)
conj(p) =
  if eq(p, Cons)
    then
      if equal(And, 1st(p))
        then and(disj(1st(2nd(p))), conj(2nd(2nd(p))))
        else bool(p)
    else bool(p)
bool(p) = or(equal(F, p), equal(T, p))
```

duplicate

```
--
-- Source file: basic\duplicate.scm
--
--- Compute a list where each element is duplicated
goal(x) = duplicate(x)
duplicate(xs) =
  if equal(xs, Nil)
    then Nil
    else Cons(1st(xs), Cons(1st(xs), duplicate(2nd(xs))))
```

equal

```
--
-- Source file: basic\equal.scm
--
```

```
4  goal(x) = equal0(x)
5  equal0(x) =
6    if equal(x, Nil)
7      then number42(Nil)
8      else equal0(x)
9  number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

evenodd

```
1  --
2  -- Source file: basic\evenodd.scm
3  --
4  --- Predicate: is x, unarily represented as '(s s s ... s), even/odd
5  evenodd(x) = even(x)
6  even(x) =
7    if eq(x, Nil)
8      then True
9      else odd(2nd(x))
10 odd(x) =
11   if eq(x, Cons)
12     then even(2nd(x))
13     else False
```

fold

```
1  --
2  -- Source file: basic\fold.scm
3  --
4  -- The fold operators, using a fixed operator, op
5  fold(a, xs) = Cons(foldl(a, xs), Cons(foldr(a, xs), Nil))
6  foldl(a, xs) =
7    if eq(xs, Cons)
8      then foldl(op(a, 1st(xs)), 2nd(xs))
9      else a
10 foldr(a, xs) =
11   if eq(xs, Cons)
12     then op(1st(xs), foldr(a, 2nd(xs)))
13     else a
14 op(x1, x2) = add(x1, x2)
```

game

```
1  --
2  -- Source file: basic\game.scm
3  --
4  -- The game function from Manuvir Das' PhD Thesis (p. 137)
5  goal(p1, p2, moves) = game(p1, p2, moves)
6  game(p1, p2, moves) =
7    if equal(moves, Nil)
8      then Cons(p1, p2)
9      else
10       if equal(1st(moves), Swap)
11         then game(p2, p1, 2nd(moves))
12         else
13           if equal(1st(moves), Capture)
14             then game(Cons(1st(p2), p1), 2nd(p2), 2nd(moves))
15             else Error
```

increase

```
1  --
2  -- Source file: basic\increase.scm
3  --
4  goal(x) = increase(x)
5  increase(x) =
```

```
    if equal(x, Nil)
      then number42(Nil)
      else increase(Cons(Cons(Nil, Nil), x))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

intlookup

```
--
-- Source file: basic\intlookup.scm
--
-- The function call case of an interpreter
-- Function number represented as list length
run(e, p) = intlookup(e, p)
intlookup(e, p) = intlookup(lookup(e, p), p)
lookup(fnum, p) =
  if equal(fnum, Nil)
    then 1st(p)
    else lookup(2nd(fnum), 2nd(p))
```

letexp

```
--
-- Source file: basic\letexp.scm
--
-- Testing the let construction
goal(x, y) = letexp(x, y)
letexp(x, y) =
  let
    z = Cons(Cons(Nil, Nil), x)
  in
    letexp(z, y)
```

list

```
--
-- Source file: basic\list.scm
--
-- The predicate for checking whether the argument is a list
goal(x) = list(x)
list(xs) =
  if eq(xs, Cons)
    then list(2nd(xs))
    else eq(xs, Nil)
```

lte

```
--
-- Source file: basic\lte.scm
--
-- Less than or equal
goal(x, y) = and(lte(x, y), even(x))
lte(x, y) =
  if eq(x, Nil)
    then True
    else
      if and(eq(x, Cons), eq(y, Cons))
        then lte(2nd(x), 2nd(y))
        else False
even(x) =
  if eq(x, Nil)
    then True
    else
      if eq(2nd(x), Nil)
        then False
        else even(2nd(2nd(x)))
```

map0

```
1  --
2  -- Source file: basic\map0.scm
3  --
4  -- The map function with fixed function f
5  goal(xs) = map(xs)
6  map(xs) =
7    if equal(xs, Nil)
8      then Nil
9      else Cons(f(1st(xs)), map(2nd(xs)))
10 f(x) = mul(x, x)
```

member

```
1  --
2  -- Source file: basic\member.scm
3  --
4  -- The member function
5  goal(x, xs) = member(x, xs)
6  member(x, xs) =
7    if equal(xs, Nil)
8      then
9        if equal(x, 1st(xs))
10         then True
11         else member(x, 2nd(xs))
12     else False
```

mergelists

```
1  --
2  -- Source file: basic\mergelists.scm
3  --
4  --- Merge two lists
5  goal(xs, ys) = merge(xs, ys)
6  merge(xs, ys) =
7    if equal(xs, Nil)
8      then ys
9      else
10       if equal(ys, Nil)
11         then xs
12         else
13           if lte(1st(xs), 1st(ys))
14             then Cons(1st(xs), merge(2nd(xs), ys))
15             else Cons(1st(ys), merge(xs, 2nd(ys)))
```

mul

```
1  --
2  -- Source file: basic\mul.scm
3  --
4  --- Unary multiplication and addition, e.g. (mul '(s s z) '(s s s z))
5  goal(x, y) = mul0(x, y)
6  mul0(x, y) =
7    if equal(x, Nil)
8      then Nil
9      else add0(mul0(2nd(x), y), y)
10 add0(x, y) =
11   if equal(x, Nil)
12     then y
13     else add0(2nd(x), Cons(S, y))
```

naiverev

```
--
-- Source file: basic\naiverev.scm
--
-- Naive reverse function
goal(xs) = naiverev(xs)
naiverev(xs) =
  if equal(xs, Nil)
    then xs
    else app(naiverev(2nd(xs)), Cons(1st(xs), Nil))
app(xs, ys) =
  if equal(xs, Nil)
    then ys
    else Cons(1st(xs), app(2nd(xs), ys))
```

nestdec

```
--
-- Source file: basic\nestdec.scm
--
-- Parameter decrease by nested call in recursion
goal(x) = nestdec(x)
nestdec(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nestdec(dec(x))
dec(x) =
  if equal(x, Cons(Nil, Nil))
    then 2nd(x)
    else dec(2nd(x))
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

nesteql

```
--
-- Source file: basic\nesteql.scm
--
-- Parameter equality by nested call in recursion
goal(x) = nesteql(x)
nesteql(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nesteql(eql(x))
eql(x) =
  if equal(x, Nil)
    then x
    else eql(x)
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

nestimeql

```
--
-- Source file: basic\nestimeql.scm
--
-- Using an immaterial copy as recursive argument
goal(x) = nestimeql(x)
nestimeql(x) =
  if equal(x, Nil)
    then number42(Nil)
    else nestimeql(immatcopy(x))
immatcopy(x) =
  if equal(x, Nil)
    then Nil
    else Cons(Nil, immatcopy(2nd(x)))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

nestinc

```
--
-- Source file: basic\nestinc.scm
--
-- Parameter increase by nested call in recursion
goal(x) = nestinc(x)
nestinc(x) =
  if equal(x, Nil)
    then number17(Nil)
    else nestinc(inc(x))
inc(x) =
  if equal(x, Nil)
    then Cons(Nil, Nil)
    else Cons(Cons(Nil, Nil), inc(2nd(x)))
number17(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

nolexicord

```
--
-- Source file: basic\nolexicord.scm
--
-- Example not termination-provable by simple lexicographical ordering
goal(a1, b1, a2, b2, a3, b3) = nolexicord(a1, b1, a2, b2, a3, b3)
nolexicord(a1, b1, a2, b2, a3, b3) =
  if equal(a1, Nil)
    then number42(Nil)
    else
      if equal(a1, b1)
        then nolexicord(2nd(b1), 2nd(a1), 2nd(a2), 2nd(b2), 2nd(b3), 2nd(a3))
        else nolexicord(2nd(b1), 2nd(a1), 2nd(b2), 2nd(a2), 2nd(a3), 2nd(b3))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

ordered

```
--
-- Source file: basic\ordered.scm
--
-- Predicate that checks whether a list is ordered
goal(xs) = ordered(xs)
ordered(xs) =
  if eq(xs, Cons)
    then
      if eq(2nd(xs), Cons)
        then
          if lte(1st(xs), 1st(2nd(xs)))
            then ordered(2nd(2nd(xs)))
            else False
        else True
    else True
```

overlap

```
--
-- Source file: basic\overlap.scm
--
-- Predicate for checking whether there is an overlap of two sets
goal(xs, ys) = overlap(xs, ys)
overlap(xs, ys) =
  if eq(xs, Cons)
    then
      if member(1st(xs), ys)
        then True
        else overlap(2nd(xs), ys)
    else False
member(x, xs) =
```

```
14      if eq(xs, Cons)
15        then
16          if equal(1st(xs), x)
17            then True
18            else member(x, 2nd(xs))
19        else False
```

permute

```
1   --
2   -- Source file: basic\permute.scm
3   --
4   -- Compute all the permutations of a list
5   -- Select x as the first element and cons it onto
6   -- permutations of the remaining list represented by
7   -- the list of elements before x (reversed) and the
8   -- list of elements after x. Finally, recurse by moving
9   -- on to the next element in postfix
10  -- Map '(cons x' onto the list of lists xss and append the rest
11  -- Reverse xs and append the rest
12  goal(xs) = permute(xs)
13  permute(xs) =
14    if equal(xs, Nil)
15      then Cons(Nil, Nil)
16      else select(1st(xs), Nil, 2nd(xs))
17  select(x, revprefix, postfix) = mapconsapp(x, permute(revapp(revprefix, postfix)),
18      if equal(postfix, Nil)
19        then Nil
20        else select(1st(postfix), Cons(x, revprefix), 2nd(postfix)))
21  mapconsapp(x, xss, rest) =
22    if equal(xss, Nil)
23      then rest
24      else Cons(Cons(x, 1st(xss)), mapconsapp(x, 2nd(xss), rest))
25  revapp(xs, rest) =
26    if equal(xs, Nil)
27      then rest
28      else revapp(2nd(xs), Cons(1st(xs), rest))
```

revapp

```
1   --
2   -- Source file: basic\revapp.scm
3   --
4   --- Reverse list and append to rest
5   goal(x, y) = revapp(x, y)
6   revapp(xs, rest) =
7     if equal(xs, Nil)
8       then rest
9       else revapp(2nd(xs), Cons(1st(xs), rest))
```

select

```
1   --
2   -- Source file: basic\select.scm
3   --
4   -- Compute a list of lists. Each list is computed by picking out an
5   -- element of the original list and consing it onto the rest of the list
6   -- Reverse xs and append to rest
7   select(xs) =
8     if equal(xs, Nil)
9       then Nil
10      else selects(1st(xs), Nil, 2nd(xs))
11  selects(x, revprefix, postfix) = Cons(Cons(x, revapp(revprefix, postfix)),
12      if equal(postfix, Nil)
13        then Nil
```

```
14          else selects(1st(postfix), Cons(x, revprefix), 2nd(postfix)))
15  revapp(xs, rest) =
16    if equal(xs, Nil)
17      then rest
18      else revapp(2nd(xs), Cons(1st(xs), rest))
```

shuffle

```
1  --
2  -- Source file: basic\shuffle.scm
3  --
4  -- Shuffle List
5  goal(xs) = shuffle(xs)
6  shuffle(xs) =
7    if equal(xs, Nil)
8      then Nil
9      else Cons(1st(xs), shuffle(reverse(2nd(xs))))
10 reverse(xs) =
11   if equal(xs, Nil)
12     then xs
13     else append(reverse(2nd(xs)), Cons(1st(xs), Nil))
14 append(xs, ys) =
15   if equal(xs, Nil)
16     then ys
17     else Cons(1st(xs), append(2nd(xs), ys))
```

sp1

```
1  --
2  -- Source file: basic\sp1.scm
3  --
4  -- Mutual recursion requiring specialisation points
5  sp1(x, y) = f(x, y)
6  f(x, y) =
7    if equal(x, Nil)
8      then g(x, y)
9      else h(x, y)
10 g(x, y) =
11   if equal(x, Nil)
12     then h(x, y)
13     else r(x, y)
14 h(x, y) =
15   if equal(x, Nil)
16     then h(x, y)
17     else f(x, y)
18 r(x, y) = x
```

subsets

```
1  --
2  -- Source file: basic\subsets.scm
3  --
4  -- Compute all subsets
5  -- map `(cons x' ont the list of lists xss, and append rest
6  goal(xs) = subsets(xs)
7  subsets(xs) =
8    if eq(xs, Cons)
9      then
10       let
11         subs = subsets(2nd(xs))
12       in
13         mapconsapp(1st(xs), subs, subs)
14     else Cons(Nil, Nil)
15 mapconsapp(x, xss, rest) =
16   if eq(xss, Cons)
17     then Cons(Cons(x, 1st(xss)), mapconsapp(x, 2nd(xss), rest))
18     else rest
```

thetrick

```
--
-- Source file: basic\thetrick.scm
--
-- The trick: pulling out the conditional into the context
goal(x, y) = Cons(f(x, y), Cons(g(x, y), Nil))
f(x, y) =
  if equal(y, Nil)
    then number42(Nil)
    else f(
        if lt0(x, Cons(Nil, Nil))
          then x
          else 2nd(x),
        if lt0(x, Cons(Nil, Nil))
          then 2nd(y)
          else Cons(Cons(Nil, Nil), y))
g(x, y) =
  if equal(y, Nil)
    then number42(Nil)
    else
      if lt0(x, Cons(Nil, Nil))
        then g(x, 2nd(y))
        else g(2nd(x), Cons(Cons(Nil, Nil), y))
lt0(x, y) =
  if equal(y, Nil)
    then False
    else
      if equal(x, Nil)
        then True
        else lt0(2nd(x), 2nd(y))
number42(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Con
```

vangelder

```
--
-- Source file: basic\vangelder.scm
--
--- Following is an example due to Allen Van Gelder.
--- Note that in the following example there is a
--- cycle involving q, p, r, t, and q again, such that
--- nothing gets smaller along that cycle.
--- e(a,b).
--- q(X,Y)        :- e(X,Y).
--- q(X,f(f(X))) :- p(X,f(f(X))), q(X,f(X)).
--- q(X,f(f(Y))) :- p(X,f(Y)).
---
--- p(X,Y)        :- e(X,Y).
--- p(X,f(Y))     :- r(X,f(Y)), p(X,Y).
---
--- r(X,Y)        :- e(X,Y).
--- r(X,f(Y))     :- q(X,Y), r(X,Y).
--- r(f(X),f(X)) :- t(f(X),f(X)).
---
--- t(X,Y)        :- e(X,Y).
--- t(f(X),f(Y)) :- q(f(X),f(Y)), t(X,Y).
goal(x, y) = q(x, y)
e(a, b) = and(equal(a, A), equal(b, B))
q(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(y, Cons), and(equal(1st(y), F), and(eq(2nd(y), Cons), equal(1st(2nd(y)), F))))
        then
          if and(p(x, y), q(x, 2nd(y)))
            then True
            else p(x, 2nd(y))
        else False
```

```
p(x, y) =
  if e(x, y)
    then True
    else
      if equal(F, 1st(y))
        then and(r(x, y), p(x, 2nd(y)))
        else False
r(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(y, Cons), equal(1st(y), F))
        then
          if and(q(x, 2nd(y)), r(x, 2nd(y)))
            then True
            else
              if and(eq(x, Cons), equal(1st(x), F))
                then t(x, y)
                else False
        else False
t(x, y) =
  if e(x, y)
    then True
    else
      if and(eq(x, Cons), and(equal(1st(x), F), and(eq(y, Cons), equal(1st(y), F))))
        then and(q(x, y), t(2nd(x), 2nd(y)))
        else False
```

A.7 Glenstrup Examples - Interpreters

int

```
--
-- Source file: interpreters\int.scm
--
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), Cst)
    then 2nd(e)
    else
      if equal(1st(e), Var)
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Bsf)
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
              in
                apply(1st(2nd(e)), v1, v2)
            else
              if equal(1st(e), If)
                then
                  if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                    then eeval(1st(2nd(2nd(e))), ns, vs, p)
                    else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                else
                  if equal(1st(e), Eq)
                    then
                      if equal(eeval(1st(2nd(e)), ns, vs, p), eeval(1st(2nd(2nd(e))), ns, vs, p))
```

```
                          then T
                          else F
                      else
                        let
                          ef = lookbody(1st(2nd(e)), p)
                          nf = lookname(1st(2nd(e)), p)
                          v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                        in
                          eeval(ef, Cons(nf, Nil), Cons(v, Nil), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Eqop)
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

intdynscope

```
--
-- Source file: interpreters\intdynscope.scm
--
-- Small 1st order interpreter with dynamic scoping
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), Cons(Nil, Nil))
    then 2nd(e)
    else
      if equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
              in
                apply(1st(2nd(e)), v1, v2)
            else
              if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
                then
                  if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                    then eeval(1st(2nd(2nd(e))), ns, vs, p)
                    else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                else
                  if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
                    then
                      if equal(eeval(1st(2nd(e)), ns, vs, p), eeval(1st(2nd(2nd(e))), ns, vs, p))
                        then T
```

```
                          else F
                      else
                        let
                          ef = lookbody(1st(2nd(e)), p)
                          nf = lookname(1st(2nd(e)), p)
                          v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                        in
                          eeval(ef, Cons(nf, ns), Cons(v, vs), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

intloop

```
--
-- Source file: interpreters\intloop.scm
--
-- Small 1st order interpreter for LOOP programs
run(p, l, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), l, p)
eeval(e, ns, vs, l, p) =
  if equal(1st(e), Cons(Nil, Nil))
    then 2nd(e)
    else
      if equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, l, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, l, p)
              in
                apply(1st(2nd(e)), v1, v2)
            else
              if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
                then
                  if equal(eeval(1st(2nd(e)), ns, vs, l, p), T)
                    then eeval(1st(2nd(2nd(e))), ns, vs, l, p)
                    else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, l, p)
                else
                  if equal(1st(e), Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
                    then
                      if equal(eeval(1st(2nd(e)), ns, vs, l, p), eeval(1st(2nd(2nd(e))), ns, vs,
                        then T
                        else F
```

```
                      else
                        let
                          ef = lookbody(1st(2nd(e)), p)
                          nf = lookname(1st(2nd(e)), p)
                          v = eeval(1st(2nd(2nd(e))), ns, vs, l, p)
                        in

                          if equal(l, Nil)
                            then Nil
                            else eeval(ef, Cons(nf, Nil), Cons(v, Nil), 2nd(l), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
```

intwhile

```
--
-- Source file: interpreters\intwhile.scm
--
-- Small 1st order interpreter with static scoping
run(p, input) =
  let
    f0 = 1st(1st(p))
    ef = lookbody(f0, p)
    nf = lookname(f0, p)
  in
    eeval(ef, Cons(nf, Nil), Cons(input, Nil), p)
eeval(e, ns, vs, p) =
  if equal(1st(e), number1(Nil))
    then 2nd(e)
    else
      if equal(1st(e), number2(Nil))
        then lookvar(2nd(e), ns, vs)
        else
          if equal(1st(e), number3(Nil))
            then
              let
                v1 = eeval(1st(2nd(2nd(e))), ns, vs, p)
                v2 = eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
              in
                apply(1st(2nd(e)), v1, v2)
            else
              if equal(1st(e), number4(Nil))
                then
                  if equal(eeval(1st(2nd(e)), ns, vs, p), T)
                    then eeval(1st(2nd(2nd(e))), ns, vs, p)
                    else eeval(1st(2nd(2nd(2nd(e)))), ns, vs, p)
                else
                  if equal(1st(e), number5(Nil))
                    then
                      if equal(eeval(1st(2nd(e)), ns, vs, p), eeval(1st(2nd(2nd(e))), ns, vs, p))
                        then T
```

```
                            else F
                        else
                          let
                            ef = lookbody(1st(2nd(e)), p)
                            nf = lookname(1st(2nd(e)), p)
                            v = eeval(1st(2nd(2nd(e))), ns, vs, p)
                          in
                            eeval(ef, Cons(nf, Nil), Cons(v, Nil), p)
lookvar(x, ns, vs) =
  if equal(x, 1st(ns))
    then 1st(vs)
    else lookvar(x, 2nd(ns), 2nd(vs))
lookbody(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(2nd(1st(p))))
    else lookbody(f, 2nd(p))
lookname(f, p) =
  if equal(1st(1st(p)), f)
    then 1st(2nd(1st(p)))
    else lookname(f, 2nd(p))
apply(op, v1, v2) =
  if equal(op, number5(Nil))
    then
      if equal(v1, v2)
        then T
        else F
    else Cons(v1, v2)
number1(n) = Cons(Nil, Nil)
number2(n) = Cons(Nil, Cons(Nil, Nil))
number3(n) = Cons(Nil, Cons(Nil, Cons(Nil, Nil)))
number4(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil))))
number5(n) = Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Cons(Nil, Nil)))))
```

lambdaint

```
--
-- Source file: interpreters\lambdaint.scm
--
-- Reducer for the lambda calculus
--   Representation:
--     R [[n]]     = (1 0 ... 0)  n zeros
--     R [[\n.e]] = (2 R [[n]] R [[e]])
--     R [[e e']] = (3 R [[e]] R [[e']])
lambdaint(e) = red(e)
red(e) =
  if isvar(e)
    then e
    else
      if islam(e)
        then e
        else
          let
            f = red(appe1(e))
            a = red(appe2(e))
          in

            if islam(f)
              then red(subst(lamvar(f), a, lambody(f)))
              else mkapp(f, a)
subst(x, a, e) =
  if isvar(e)
    then
      if equal(x, e)
        then a
        else e
    else
      if islam(e)
        then
```

```
            if equal(x, lamvar(e))
              then e
              else mklam(lamvar(e), subst(x, a, lambody(e)))
          else mkapp(subst(x, a, appe1(e)), subst(x, a, appe2(e)))
isvar(e) = equal(1st(e), Cons(Nil, Nil))
islam(e) = equal(1st(e), Cons(Nil, Cons(Nil, Nil)))
mklam(n, e) = Cons(Cons(Nil, Cons(Nil, Nil)), Cons(n, Cons(e, Nil)))
lamvar(e) = 1st(2nd(e))
lambody(e) = 1st(2nd(2nd(e)))
mkapp(e1, e2) = Cons(Cons(Nil, Cons(Nil, Cons(Nil, Nil))), Cons(e1, Cons(e2, Nil)))
appe1(e) = 1st(2nd(e))
appe2(e) = 1st(2nd(2nd(e)))
```

parsexp

```
--
-- Source file: interpreters\parsexp.scm
--
--- Parse a list of atoms as an expression. Return remaining list.
--- e.g. '(5 * ( 3 + 2 * 4 ))
--
-- Modified to use symbols instead of strings
parsexp(xs) = expr(xs)
expr(xs) =
  let
    rs1 = term(xs)
  in

    if equal(Nil, rs1)
      then Nil
      else
        if member(1st(rs1), Cons(Plus, Cons(Minus, Nil)))
          then
            let
              rs2 = expr(2nd(rs1))
            in

              if equal(Nil, rs2)
                then rs1
                else rs2
          else rs1
term(xs) =
  let
    rs1 = factor(xs)
  in

    if equal(rs1, Nil)
      then Nil
      else
        if member(1st(rs1), Cons(Mul, Cons(Div, Nil)))
          then
            let
              rs2 = term(2nd(rs1))
            in

              if equal(Nil, rs2)
                then rs1
                else rs2
          else rs1
factor(xs) =
  if equal(Lpar, 1st(xs))
    then
      let
        rs1 = expr(xs)
      in

        if and(not(equal(rs1, Nil)), equal(Rpar, 1st(rs1)))
```

```
            then 2nd(rs1)
            else atom(xs)
      else atom(xs)
member(x, xs) =
   if eq(xs, Cons)
     then
       if equal(x, 1st(xs))
         then True
         else member(x, 2nd(xs))
     else False
atom(xs) =
   if eq(xs, Cons)
     then 2nd(xs)
     else Nil
```

turing

```
--
-- Source file: interpreters\turing.scm
--
--- Turing machine interpreter
--- instrs ::= '(instr . instrs)
---          |  '()
--- instr ::= '(Halt)          - Stop interpretation
---         |  '(Write . x)     - Write x onto the tape at current pos
---         |  '(Left)          - Move pos left, extend tape if needed
---         |  '(Right)         - Move pos right, extend tape if needed
---         |  '(Goto . i)      - Continue at instruction i
---         |  '(IfGoto x . i) - If current pos contains x, goto i
run(prog, tapeinput) = turing(prog, Nil, tapeinput, prog)
turing(instrs, revltape, rtape, prog) =
   if eq(instrs, Cons)
     then
       if equal(Halt, 1st(1st(instrs)))
         then rtape
         else
           if equal(Write, 1st(1st(instrs)))
             then turing(2nd(instrs), revltape, Cons(2nd(1st(instrs)), 2nd(rtape)), prog)
             else
               if equal(Left, 1st(1st(instrs)))
                 then
                   if eq(revltape, Cons)
                     then turing(2nd(instrs), 2nd(revltape), Cons(1st(revltape), rtape), prog)
                     else turing(2nd(instrs), Nil, Cons(Blank, rtape), prog)
                 else
                   if equal(Right, 1st(1st(instrs)))
                     then
                       if eq(rtape, Cons)
                         then turing(2nd(instrs), Cons(1st(rtape), revltape), 2nd(rtape), prog)
                         else turing(2nd(instrs), Cons(Blank, revltape), Nil, prog)
                     else
                       if equal(Goto, 1st(1st(instrs)))
                         then turing(lookup(2nd(1st(instrs)), prog), revltape, rtape, prog)
                         else
                           if equal(Ifgoto, 1st(1st(instrs)))
                             then
                               if equal(1st(rtape), 1st(2nd(1st(instrs))))
                                 then turing(lookup(2nd(2nd(1st(instrs))), prog), revltape, rtape,
                                 else turing(2nd(instrs), revltape, rtape, prog)
                             else rtape
     else rtape
lookup(i, instrs) =
   if eq(i, Cons(Nil, Nil))
     then instrs
     else lookup(sub(i, Cons(Nil, Nil)), 2nd(instrs))
```

A.8 Glenstrup Examples - Simple

ack

```
1  --
2  -- Source file: simple\ack.scm
3  --
4  --- Ackermann's function, numbers represented by list length
5  goal(m, n) = ack(m, n)
6  ack(m, n) =
7    if equal(Nil, m)
8      then Cons(Cons(Nil, Nil), n)
9      else
10       if equal(Nil, n)
11         then ack(2nd(m), Cons(Nil, Nil))
12         else ack(2nd(m), ack(m, 2nd(n)))
```

binom

```
1  --
2  -- Source file: simple\binom.scm
3  --
4  --- Binomial function, numbers represented by list length
5  goal(n, k) = binom(n, k)
6  binom(n, k) =
7    if equal(Nil, n)
8      then Cons(Nil, Nil)
9      else
10       if equal(Nil, k)
11         then Cons(Nil, Nil)
12         else add(binom(2nd(n), 2nd(k)), binom(2nd(n), k))
```

gcd1

```
1  --
2  -- Source file: simple\gcd1.scm
3  --
4  -- Greatest common divisor, numbers represented by list length
5  goal(x, y) = gcd(x, y)
6  gcd(x, y) =
7    if or(equal(x, Nil), equal(y, Nil))
8      then Error
9      else
10       if equal(x, y)
11         then x
12         else
13           if gt0(x, y)
14             then gcd(monus(x, y), y)
15             else gcd(x, monus(y, x))
16 gt0(x, y) =
17   if equal(x, Nil)
18     then False
19     else
20       if equal(y, Nil)
21         then True
22         else gt0(2nd(x), 2nd(y))
23 monus(x, y) =
24   if equal(lgth(y), Cons(Nil, Nil))
25     then 2nd(x)
26     else monus(2nd(x), 2nd(y))
27 lgth(x) =
28   if equal(x, Nil)
29     then Nil
30     else add(Cons(Nil, Nil), lgth(2nd(x)))
```

gcd2

```
--
-- Source file: simple\gcd2.scm
--
-- Greatest common divisor, numbers represented by list length
goal(x, y) = gcd(x, y)
gcd(x, y) =
  if or(equal(x, Nil), equal(y, Nil))
    then Error
    else
      if equal(x, y)
        then x
        else
          if gt0(x, y)
            then gcd(y, monus(x, y))
            else gcd(monus(y, x), x)
gt0(x, y) =
  if equal(x, Nil)
    then False
    else
      if equal(y, Nil)
        then True
        else gt0(2nd(x), 2nd(y))
monus(x, y) =
  if equal(lgth(y), Cons(Nil, Nil))
    then 2nd(x)
    else monus(2nd(x), 2nd(y))
lgth(x) =
  if equal(x, Nil)
    then Nil
    else add(Cons(Nil, Nil), lgth(2nd(x)))
```

power

```
--
-- Source file: simple\power.scm
--
-- Power function: x to the nth power
-- (numbers represented by list length)
goal(x, n) = power(x, n)
power(x, n) =
  if equal(n, Nil)
    then Cons(Nil, Nil)
    else mult(x, power(x, 2nd(n)))
mult(x, y) =
  if equal(y, Nil)
    then Nil
    else add0(x, mult(x, 2nd(y)))
add0(x, y) =
  if equal(y, Nil)
    then x
    else Cons(Cons(Nil, Nil), add0(x, 2nd(y)))
```

A.9 Glenstrup Examples - Sorting

mergesort

```
--
-- Source file: sorting\mergesort.scm
--
-- Mergesort
goal(xs) = mergesort(xs)
mergesort(xs) =
  if eq(xs, Cons)
```

```
      then
        if eq(2nd(xs), Cons)
          then splitmerge(xs, Nil, Nil)
          else xs
      else xs
  splitmerge(xs, xs1, xs2) =
    if eq(xs, Cons)
      then splitmerge(2nd(xs), Cons(1st(xs), xs2), xs1)
      else merge(mergesort(xs1), mergesort(xs2))
  merge(xs1, xs2) =
    if eq(xs1, Cons)
      then
        if eq(xs2, Cons)
          then
            if lte(1st(xs1), 1st(xs2))
              then Cons(1st(xs1), merge(2nd(xs1), xs2))
              else Cons(1st(xs2), merge(xs1, 2nd(xs2)))
          else xs1
      else xs2
```

minsort

```
  --
  -- Source file: sorting\minsort.scm
  --
  --- Minimum sort: remove minimum and cons it onto the rest, sorted.
  goal(xs) = minsort(xs)
  minsort(xs) =
    if eq(xs, Cons)
      then appmin(1st(xs), 2nd(xs), xs)
      else Nil
  appmin(min, rest, xs) =
    if eq(rest, Cons)
      then
        if lt(1st(rest), min)
          then appmin(1st(rest), 2nd(rest), xs)
          else appmin(min, 2nd(rest), xs)
      else Cons(min, minsort(remove(min, xs)))
  remove(x, xs) =
    if eq(xs, Cons)
      then
        if equal(x, 1st(xs))
          then 2nd(xs)
          else Cons(1st(xs), remove(x, 2nd(xs)))
      else Nil
```

quicksort

```
  --
  -- Source file: sorting\quicksort.scm
  --
  -- Quicksort
  goal(xs) = quicksort(xs)
  quicksort(xs) =
    if eq(xs, Cons)
      then
        if eq(2nd(xs), Cons)
          then part(1st(xs), xs, Cons(1st(xs), Nil), Nil)
          else xs
      else xs
  part(x, xs, xs1, xs2) =
    if eq(xs, Cons)
      then
        if gt(x, 1st(xs))
          then part(x, 2nd(xs), Cons(1st(xs), xs1), xs2)
          else
```

```
19              if lt(x, 1st(xs))
20                then part(x, 2nd(xs), xs1, Cons(1st(xs), xs2))
21                else part(x, 2nd(xs), xs1, xs2)
22        else app(quicksort(xs1), quicksort(xs2))
23  app(xs, ys) =
24    if eq(xs, Cons)
25      then Cons(1st(xs), app(2nd(xs), ys))
26      else ys
```

Appendix B

Results

In the following sections only the output from the terminating examples for which the analysis failed to show size-change termination are listed. This done for the sake of brevity, the output is meant as a reference for the more interesting examples. The results of the other examples are listed in the tables in section ??.

B.1 Jones Results

ex1

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex1.l
3  ----------------------------------------
4
5       --- Analysis of component: ["r1"] ---
6       size: ("r1",Lin ["ls","a"] 0)
7       crit: [r1.ls]
8       time: 0.00s
9       <No failures>
10
11 ----------Analysis Complete----------
12 TOTAL Time : 0.01s
13 ----------------------------------------
```

ex2

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex2.l
3  ----------------------------------------
4
5       --- Analysis of component: ["f","g"] ---
6       size: ("f",Exp ["i","x"])
7             ("g",Exp ["a","b","c"])
8       crit: [f.i,g.a,g.c]
9       time: 0.00s
10      <No failures>
11
12 ----------Analysis Complete----------
13 TOTAL Time : 0.00s
14 ----------------------------------------
```

ex3

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex3.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["a"] ---
7       size: ("a",Exp ["m","n"])
8       crit: [a.m]
9       time: 0.01s
10 !>  Component terminates, but call depth may be super-polynomial:
11       a-a:[(1,>=,1)]*[3]
12      Using the graphs:
13       a-a:[(1,>,1)]*[1]
14       a-a:[(1,>,1)]*[2]
15       a-a:[(1,>=,1)]*[3]
16
17 ----------Analysis Complete----------
18 TOTAL Time : 0.01s
19 ----------------------------------------
```

ex4

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex4.l
3  ----------------------------------------
4
5       --- Analysis of component: ["p"] ---
6       size: ("p",Lin ["m","r","n"] 0)
7       crit: [p.m,p.n,p.r]
8       time: 0.02s
9       <No failures>
10
11 ----------Analysis Complete----------
12 TOTAL Time : 0.02s
13 ----------------------------------------
```

ex5

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex5.l
3  ----------------------------------------
4
5       --- Analysis of component: ["f"] ---
6       size: ("f",Exp ["y","x"])
7       crit: [f.x,f.y]
8       time: 0.00s
9       <No failures>
10
11 ----------Analysis Complete----------
12 TOTAL Time : 0.00s
13 ----------------------------------------
```

ex6

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/sct/ex6.l
3  ----------------------------------------
4
5       --- Analysis of component: ["f"] ---
6       size: ("f",Lin ["b","a"] 0)
7              ("g",Lin ["c","d"] 0)
8       crit: [f.b,g.c]
9       time: 0.00s
10      <No failures>
```

```
11
12      --- Analysis of component: ["g"] ---
13      size: ("g",Lin ["c","d"] 0)
14      crit: [g.c]
15      time: 0.00s
16      <No failures>
17
18  ----------Analysis Complete----------
19  TOTAL Time : 0.00s
20  ----------------------------------------
```

B.2 Wahlstedt Results

ack

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/ack.l
3   ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["ack"] ---
7       size: ("ack",Exp ["x1","x2"])
8       crit: [ack.x1]
9       time: 0.01s
10  !>  Component terminates, but call depth may be super-polynomial:
11       ack-ack:[(1,>=,1)]*[3]
12      Using the graphs:
13       ack-ack:[(1,>,1)]*[1]
14       ack-ack:[(1,>,1)]*[2]
15       ack-ack:[(1,>=,1)]*[3]
16
17  ----------Analysis Complete----------
18  TOTAL Time : 0.01s
19  ----------------------------------------
```

add

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/add.l
3   ----------------------------------------
4
5       --- Analysis of component: ["add0"] ---
6       size: ("add0",Lin ["x2","x1"] 0)
7       crit: [add0.x1,add0.x2]
8       time: 0.01s
9       <No failures>
10
11  ----------Analysis Complete----------
12  TOTAL Time : 0.01s
13  ----------------------------------------
```

boolprog

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/boolprog.l
3   ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["f0","f1","f2","f3","f5","f4","f6"] ---
7       size: ("f0",Lin [] 0)
8             ("f1",Lin [] 0)
9             ("f2",Lin [] 0)
10            ("f3",Lin [] 0)
11            ("f5",Lin [] 0)
12            ("f4",Lin [] 0)
```

```
13              ("f6",Lin [] 0)
14        crit: [f0.x1,f0.x2,f0.x3,f0.x4,f1.x1,f1.x2,f1.x3,f1.x4,f2.x1,f2.x2,f2.x3,f2.x4,f3.x1,f3.x2,f3
15        time: 0.78s
16  !>    Component may not terminate
17         f0-f0:[(2,>,1),(2,>=,2),(3,>,4),(3,>=,3)]*[1,2,3,5,7,4,8,9]
18        Using the graphs:
19         f0-f1:[(2,>,1),(2,>=,2),(4,>,3),(4,>=,4)]*[1]
20         f1-f2:[(2,>=,1),(1,>=,2),(3,>=,3),(4,>=,4),(5,>,5)]*[2]
21         f2-f3:[(1,>=,1),(2,>=,2),(3,>,3),(4,>=,4),(5,>,5)]*[3]
22         f2-f5:[(1,>=,1),(2,>=,2),(3,>=,3),(4,>,4),(5,>,5)]*[4]
23         f3-f4:[(1,>=,1),(2,>=,2),(4,>=,3),(3,>=,4),(5,>,5)]*[5]
24         f4-f3:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4),(5,>,5)]*[6]
25         f4-f2:[(1,>=,1),(2,>,2),(3,>=,3),(4,>=,4),(5,>,5)]*[7]
26         f5-f6:[(2,>=,1),(1,>=,2),(3,>=,3),(4,>=,4),(5,>,5)]*[8]
27         f6-f0:[(1,>=,1),(2,>=,2),(4,>=,3),(3,>=,4),(5,>,5)]*[9]
28
29  ----------Analysis Complete----------
30  TOTAL Time : 0.78s
31  ----------------------------------------
```

div2

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/div2.l
3   ----------------------------------------
4
5         --- Analysis of component: ["div2"] ---
6         size: ("div2",Lin ["x1"] 0)
7         crit: [div2.x1]
8         time: 0.00s
9         <No failures>
10
11  ----------Analysis Complete----------
12  TOTAL Time : 0.00s
13  ----------------------------------------
```

eq

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/eq.l
3   ----------------------------------------
4
5         --- Analysis of component: ["eq0"] ---
6         size: ("eq0",Lin [] 1)
7         crit: [eq0.x1,eq0.x2]
8         time: 0.00s
9         <No failures>
10
11  ----------Analysis Complete----------
12  TOTAL Time : 0.00s
13  ----------------------------------------
```

ex6

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/ex6.l
3   ----------------------------------------
4
5         --- Analysis of component: ["f"] ---
6         size: ("f",Lin ["x2","x1"] 0)
7               ("g",Lin ["x1","x2"] 0)
8         crit: [f.x2,g.x1]
9         time: 0.00s
10        <No failures>
11
12        --- Analysis of component: ["g"] ---
```

```
      size: ("g",Lin ["x1","x2"] 0)
      crit: [g.x1]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

fgh

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/fgh.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["f","g"] ---
     size: ("f",Exp ["x2","x1"])
           ("g",Exp ["x1","x2"])
           ("h",Lin ["x2"] 0)
     crit: [h.x1,h.x2]
     time: 0.03s
!>   Component may not terminate
      f-f:[]*[2,6,5,3]
     Using the graphs:
      f-g:[(1,>,1),(2,>=,2)]*[2]
      f-f:[(2,>,2)]*[3]
      g-f:[(1,>=,1),(2,>=,2)]*[5]
      g-g:[(1,>,1)]*[6]

     --- Analysis of component: ["h"] ---
     size: ("h",Lin ["x2"] 0)
     crit: [h.x1,h.x2]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.03s
-------------------------------------
```

oddeven

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/oddeven.l
-------------------------------------

     --- Analysis of component: ["even","odd"] ---
     size: ("even",Lin [] 1)
           ("odd",Lin [] 1)
     crit: [even.x1,odd.x1]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

permut

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/exjobb/permut.l
-------------------------------------

     --- Analysis of component: ["f"] ---
     size: ("f",Lin [] 0)
```

```
      crit: [f.x1,f.x2]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

B.3 Size Results

assrewriteSize

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/assrewriteSize.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["rewrite","rw"] ---
     size: ("rewrite",Exp ["exp"])
           ("rw",Exp ["c","opab"])
     crit: []
     time: 0.06s
!>   Component may not terminate
      rw-rw:[]*[6]
      rewrite-rewrite:[]*[2,6,3]
     Using the graphs:
      rewrite-rw:[(1,>,1),(1,>,2)]*[2]
      rw-rewrite:[(1,>,1)]*[3]
      rw-rewrite:[(1,>,1)]*[4]
      rw-rewrite:[(2,>=,1)]*[5]
      rw-rw:[]*[6]
      rw-rewrite:[(1,>=,1)]*[7]
      rw-rewrite:[(2,>=,1)]*[8]

----------Analysis Complete----------
TOTAL Time : 0.06s
-------------------------------------
```

bubblesort

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/bubblesort.l
-------------------------------------

     --- Analysis of component: ["bsort"] ---
     size: ("bsort",Lin ["l","xs"] 0)
           ("len",Lin ["xs"] 0)
           ("bubble",Lin ["x","xs"] 1)
     crit: [bsort.l,len.xs,bubble.x,bubble.xs]
     time: 0.00s
     <No failures>

     --- Analysis of component: ["len"] ---
     size: ("len",Lin ["xs"] 0)
           ("bubble",Lin ["x","xs"] 1)
     crit: [len.xs,bubble.x,bubble.xs]
     time: 0.01s
     <No failures>

     --- Analysis of component: ["bubble"] ---
     size: ("bubble",Lin ["x","xs"] 1)
     crit: [bubble.x,bubble.xs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
```

```
TOTAL Time : 0.01s
-----------------------------------------
```

inssort

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/inssort.l
-----------------------------------------

    --- Analysis of component: ["isort"] ---
    size: ("isort",Lin [] 0)
          ("insert",Lin ["x","r"] 1)
    crit: [isort.xs,insert.x,insert.r]
    time: 0.00s
    <No failures>

    --- Analysis of component: ["insert"] ---
    size: ("insert",Lin ["x","r"] 1)
    crit: [insert.x,insert.r]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-----------------------------------------
```

minsortSize

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/minsortSize.l
-----------------------------------------

    --- Analysis of component: ["minsort","appmin"] ---
    size: ("minsort",Exp ["xs"])
          ("appmin",Exp ["rest","xs","min"])
          ("remove",Lin ["xs"] (-1))
    crit: [minsort.xs,appmin.min,appmin.rest,appmin.xs,remove.x,remove.xs]
    time: 0.03s
    <No failures>

    --- Analysis of component: ["remove"] ---
    size: ("remove",Lin ["xs"] (-1))
    crit: [remove.x,remove.xs]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.03s
-----------------------------------------
```

deadcodeSize

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/deadcodeSize.l
-----------------------------------------
    No conflicting parameters.

----------Analysis Complete----------
TOTAL Time : 0.00s
-----------------------------------------
```

fghSize

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/fghSize.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["f","g"] ---
7       size: ("f",Lin ["x2","x1"] 0)
8             ("g",Lin ["x1","x2"] 0)
9             ("h",Lin ["x2"] 0)
10      crit: [h.x1,h.x2]
11      time: 0.03s
12 !>  Component may not terminate
13       f-f:[]*[2,6,5,3]
14      Using the graphs:
15       f-g:[(1,>,1),(2,>=,2)]*[2]
16       f-f:[(2,>,2)]*[3]
17       g-f:[(1,>=,1),(2,>=,2)]*[5]
18       g-g:[(1,>,1)]*[6]
19
20      --- Analysis of component: ["h"] ---
21      size: ("h",Lin ["x2"] 0)
22      crit: [h.x1,h.x2]
23      time: 0.01s
24      <No failures>
25
26 ----------Analysis Complete----------
27 TOTAL Time : 0.04s
28 ----------------------------------------
```

gexgcd

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/gexgcd.l
3  ----------------------------------------
4
5       --- Analysis of component: ["gcd","l13","l15","l12","l14","l11","l10","l8","l7","l5","l4","l3
6       size: ("gcd",Exp ["y","x"])
7             ("l13",Exp ["y","mtmp"])
8             ("l15",Exp ["mtmp","y"])
9             ("l12",Exp ["y","x"])
10            ("l14",Exp ["y"])
11            ("l11",Exp ["y","x"])
12            ("l10",Exp ["y","x"])
13            ("l8",Exp ["y","x"])
14            ("l7",Exp ["y","x"])
15            ("l5",Exp ["y","x"])
16            ("l4",Exp ["y","x"])
17            ("l3",Exp ["y","x"])
18            ("l2",Exp ["y","x","res"])
19            ("l1",Exp ["y","x","res"])
20            ("equal0",Lin [] 1)
21            ("e1",Lin [] 1)
22            ("e2",Lin [] 1)
23            ("e3",Lin [] 1)
24            ("e4",Lin [] 1)
25            ("monus",Lin ["a"] (-1))
26            ("m1",Lin ["a"] (-1))
27            ("m2",Lin ["a"] (-1))
28            ("m4",Lin ["a"] (-2))
29            ("e7",Lin [] 0)
30            ("e6",Lin [] 0)
31            ("e5",Lin [] 1)
32            ("m3",Lin ["a"] (-1))
33            ("l9",Lin ["x"] 0)
34            ("l6",Lin [] 0)
35            ("e8",Lin ["res"] 0)
```

```
36              ("m5",Lin ["res"] 0)
37              ("l16",Lin ["res"] 0)
38      crit: [gcd.x,gcd.y,l13.x,l13.y,l13.mtmp,l15.x,l15.y,l15.mtmp,l12.x,l12.y,l14.x,l14.y,l11.x,l1
39      time: 3.18s
40      <No failures>
41
42      --- Analysis of component: ["monus","m1","m2","m4"] ---
43      size: ("monus",Lin ["a"] (-1))
44              ("m1",Lin ["a"] (-1))
45              ("m2",Lin ["a"] (-1))
46              ("m4",Lin ["a"] (-2))
47              ("e7",Lin [] 0)
48              ("e6",Lin [] 0)
49              ("e5",Lin [] 1)
50              ("m3",Lin ["a"] (-1))
51              ("l9",Lin ["x"] 0)
52              ("l6",Lin [] 0)
53              ("e8",Lin ["res"] 0)
54              ("m5",Lin ["res"] 0)
55              ("l16",Lin ["res"] 0)
56      crit: [monus.a,monus.b,m1.a,m1.b,m2.a,m2.b,m4.a,m4.b]
57      time: 0.04s
58      <No failures>
59
60  ----------Analysis Complete----------
61  TOTAL Time : 3.24s
62  ----------------------------------------
```

gexgcd2

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/gexgcd2.l
3   ----------------------------------------
4
5       --- Analysis of component: ["gcd"] ---
6       size: ("gcd",Exp ["y","x"])
7               ("monus",Lin ["a"] (-1))
8               ("equal0",Lin [] 1)
9       crit: [gcd.x,gcd.y,monus.a,monus.b]
10      time: 0.01s
11      <No failures>
12
13      --- Analysis of component: ["monus"] ---
14      size: ("monus",Lin ["a"] (-1))
15              ("equal0",Lin [] 1)
16      crit: [monus.a,monus.b]
17      time: 0.01s
18      <No failures>
19
20  ----------Analysis Complete----------
21  TOTAL Time : 0.02s
22  ----------------------------------------
```

graphcolour2Size

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/graphcolour2Size.l
3   ----------------------------------------
4
5       --- Analysis of component: ["colornode"] ---
6       size: ("colornode",Lin ["cs","node"] 1)
7               ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
8               ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
9               ("colorrestthetrick",Exp ["cs","cs1","ncs","colorednodes","rest"])
10              ("possible",Lin [] 0)
11              ("reverse",Lin ["xs"] 0)
```

```
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colornode.cs,colornode.node,colornode.colorednodes,colorrest.cs,colorrest.ncs,colorres
      time: 0.01s
      <No failures>

      --- Analysis of component: ["colorrest","colornoderest","colorrestthetrick"] ---
      size: ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
            ("colorrestthetrick",Exp ["cs","cs1","ncs","colorednodes","rest"])
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorrest.cs,colorrest.ncs,colorrest.rest,colornoderest.cs,colornoderest.ncs,colornode
      time: 0.27s
!>    Possible duplication of variables: [colornoderest.cs,colornoderest.rest]
      in the bounding argument positions: [11:colorrest.cs,11:colorrest.ncs,11:colorrest.rest,12:co

      --- Analysis of component: ["possible"] ---
      size: ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [possible.color,possible.adjs,possible.colorednodes,colorof.node,colorof.colorednodes,r
      time: 0.00s
      <No failures>

      --- Analysis of component: ["colorof"] ---
      size: ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorof.node,colorof.colorednodes,revapp.xs]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["revapp"] ---
      size: ("revapp",Lin ["xs","rest"] 0)
      crit: [revapp.xs]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.29s
-------------------------------------
```

quicksortSize

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/quicksortSize.l
-----------------------------------------

      --- Analysis of component: ["quicksort","part"] ---
      size: ("quicksort",Exp ["xs"])
            ("part",Exp ["xs","x"])
            ("partGt",Lin ["xs"] 0)
            ("partLt",Lin ["xs"] 0)
            ("app",Lin ["ys","xs"] 0)
      crit: [quicksort.xs,part.x,part.xs,partGt.x,partGt.xs,partLt.x,partLt.xs,app.xs,app.ys]
      time: 0.02s
!>    Possible duplication of variables: [part.xs]
      in the bounding argument positions: [4:quicksort.xs,6:quicksort.xs]

      --- Analysis of component: ["partGt"] ---
      size: ("partGt",Lin ["xs"] 0)
            ("partLt",Lin ["xs"] 0)
            ("app",Lin ["ys","xs"] 0)
      crit: [partGt.x,partGt.xs,partLt.x,partLt.xs,app.xs,app.ys]
```

```
      time: 0.00s
      <No failures>

      --- Analysis of component: ["partLt"] ---
      size: ("partLt",Lin ["xs"] 0)
            ("app",Lin ["ys","xs"] 0)
      crit: [partLt.x,partLt.xs,app.xs,app.ys]
      time: 0.01s
      <No failures>

      --- Analysis of component: ["app"] ---
      size: ("app",Lin ["ys","xs"] 0)
      crit: [app.xs,app.ys]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.03s
-------------------------------------
```

thetrickSize

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/thetrickSize.l
-----------------------------------------
     Non-termination detected.

     --- Analysis of component: ["g"] ---
     size: ("g",Lin [] 4)
           ("f",Lin [] 4)
           ("lt0",Lin [] 0)
           ("number4",Lin [] 4)
     crit: [g.x,f.x,lt0.x,lt0.y]
     time: 0.01s
!>   Component terminates, but call depth may be super-polynomial:
      g-g:[(1,>=,1)]*[9]
     Using the graphs:
      g-g:[(1,>=,1)]*[9]
      g-g:[(1,>,1)]*[10]

     --- Analysis of component: ["f"] ---
     size: ("f",Lin [] 4)
           ("lt0",Lin [] 0)
           ("number4",Lin [] 4)
     crit: [f.x,lt0.x,lt0.y]
     time: 0.00s
!>   Component terminates, but call depth may be super-polynomial:
      f-f:[(1,>=,1)]*[5]
     Using the graphs:
      f-f:[(1,>=,1)]*[5]
      f-f:[(1,>,1)]*[6]

     --- Analysis of component: ["lt0"] ---
     size: ("lt0",Lin [] 0)
           ("number4",Lin [] 4)
     crit: [lt0.x,lt0.y]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

power

```
-------------------------------------
```

```
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/size/power.l
----------------------------------------

    --- Analysis of component: ["power"] ---
    size: ("power",Exp ["x"])
          ("mult",Exp ["x"])
          ("add0",Lin ["y","x"] 0)
    crit: [power.x,power.n,mult.x,mult.y,add0.x,add0.y]
    time: 0.00s
!>  Non size-linear outer calls: [1:mult]
    due to the bounding argument positions: [1:mult.y]

    --- Analysis of component: ["mult"] ---
    size: ("mult",Exp ["x"])
          ("add0",Lin ["y","x"] 0)
    crit: [mult.x,mult.y,add0.x,add0.y]
    time: 0.01s
    <No failures>

    --- Analysis of component: ["add0"] ---
    size: ("add0",Lin ["y","x"] 0)
    crit: [add0.x,add0.y]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
----------------------------------------
```

B.4 Ptime Results

disj1

```
----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/disj1.l
----------------------------------------

    --- Analysis of component: ["f"] ---
    size: ("f",Exp ["x"])
          ("immatcp",Lin ["x"] 0)
    crit: [f.x,immatcp.x]
    time: 0.00s
!>  Possible duplication of variables: [f.x]
    in the bounding argument positions: [1:f.x,2:f.x]

    --- Analysis of component: ["immatcp"] ---
    size: ("immatcp",Lin ["x"] 0)
    crit: [immatcp.x]
    time: 0.01s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
----------------------------------------
```

disj2

```
----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/disj2.l
----------------------------------------
    Refining parameters: [f.x]

    --- Analysis of component: ["f"] ---
    size: ("f",Exp ["x"])
          ("g",Lin ["x"] 0)
```

```
      crit: [f.x]
      time: 0.00s
!>    Possible duplication of variables: [f.x]
      in the bounding argument positions: [1:f.x,2:f.x]

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

dup1

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/dup1.l
-------------------------------------
      Refining parameters: [a.x]
      Refining parameters: [a.y]

      --- Analysis of component: ["a"] ---
      size: ("a",Lin ["x"] 0)
      crit: [a.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

dup2

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/dup2.l
-------------------------------------
      Refining parameters: [a.y]
      Refining parameters: [a.y,a.z]

      --- Analysis of component: ["a"] ---
      size: ("a",Lin ["x"] 0)
      crit: [a.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

ocall-safe

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/ocall-safe.l
-------------------------------------

      --- Analysis of component: ["safe"] ---
      size: ("safe",Exp ["x"])
            ("dbl",Exp ["x","y"])
      crit: [safe.x,dbl.x]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["dbl"] ---
      size: ("dbl",Exp ["x","y"])
      crit: [dbl.x]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

ocall-unsafe

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/ocall-unsafe.l
3  ----------------------------------------
4
5       --- Analysis of component: ["unsafe"] ---
6       size: ("unsafe",Exp ["x"])
7             ("dbl",Exp ["x","y"])
8       crit: [unsafe.x,dbl.x]
9       time: 0.00s
10 !>   Non size-linear outer calls: [1:dbl]
11      due to the bounding argument positions: [1:dbl.x]
12
13      --- Analysis of component: ["dbl"] ---
14      size: ("dbl",Exp ["x","y"])
15      crit: [dbl.x]
16      time: 0.01s
17      <No failures>
18
19 ---------- Analysis Complete ----------
20 TOTAL Time : 0.01s
21 ----------------------------------------
```

quicksortPtime

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/ptime/quicksortPtime.l
3  ----------------------------------------
4
5       --- Analysis of component: ["quicksort","qs"] ---
6       size: ("quicksort",Lin ["xs"] 0)
7             ("qs",Lin ["ps","x"] 0)
8             ("part",Lin ["xs","xs1","xs2"] 1)
9             ("app",Lin ["ys","xs"] 0)
10      crit: [quicksort.xs,qs.x,qs.ps,part.x,part.xs,app.xs,app.ys]
11      time: 0.01s
12      <No failures>
13
14      --- Analysis of component: ["part"] ---
15      size: ("part",Lin ["xs","xs1","xs2"] 1)
16            ("app",Lin ["ys","xs"] 0)
17      crit: [part.x,part.xs,app.xs,app.ys]
18      time: 0.02s
19      <No failures>
20
21      --- Analysis of component: ["app"] ---
22      size: ("app",Lin ["ys","xs"] 0)
23      crit: [app.xs,app.ys]
24      time: 0.00s
25      <No failures>
26
27 ---------- Analysis Complete ----------
28 TOTAL Time : 0.03s
29 ----------------------------------------
```

B.5 Glenstrup Results - Algorithms

graphcolour1

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/graphcolour1.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["colorrest","colorrestthetrick"] ---
```

```
      size: ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colorrestthetrick",Exp ["cs","cs1","rest","colorednodes"])
            ("colornode",Lin ["cs","node"] 1)
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorrest.cs,colorrest.ncs,colorrest.rest,colorrestthetrick.cs1,colorrestthetrick.cs,c
      time: 0.12s
!>    Component may not terminate
       colorrest-colorrest:[(2,>=,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[11,12]
       colorrest-colorrest:[(2,>,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[11,13,12]
      Using the graphs:
       colorrest-colorrest:[(1,>=,1),(1,>=,2),(4,>,4)]*[10]
       colorrest-colorrestthetrick:[(1,>=,1),(1,>=,2),(3,>=,4),(4,>=,5)]*[11]
       colorrestthetrick-colorrest:[(2,>=,1),(1,>=,2),(4,>=,3),(5,>=,4)]*[12]
       colorrestthetrick-colorrestthetrick:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4),(5,>=,5)]*[13]

      --- Analysis of component: ["colornode"] ---
      size: ("colornode",Lin ["cs","node"] 1)
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colornode.cs,colornode.node,colornode.colorednodes,possible.color,possible.adjs,possib
      time: 0.01s
      <No failures>

      --- Analysis of component: ["possible"] ---
      size: ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [possible.color,possible.adjs,possible.colorednodes,colorof.node,colorof.colorednodes,r
      time: 0.00s
      <No failures>

      --- Analysis of component: ["colorof"] ---
      size: ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorof.node,colorof.colorednodes,revapp.xs]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["revapp"] ---
      size: ("revapp",Lin ["xs","rest"] 0)
      crit: [revapp.xs]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.14s
-----------------------------------------
```

graphcolour2

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/graphcolour2.l
-----------------------------------------
      Non-termination detected.

      --- Analysis of component: ["colornode"] ---
      size: ("colornode",Lin ["cs","node"] 1)
            ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
            ("colorrestthetrick",Exp ["cs","cs1","colorednodes","rest"])
            ("possible",Lin [] 0)
```

```
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colornode.cs,colornode.node,colornode.colorednodes,colorrest.cs,colorrest.ncs,colorres
      time: 0.01s
      <No failures>

      --- Analysis of component: ["colorrest","colornoderest","colorrestthetrick"] ---
      size: ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
            ("colorrestthetrick",Exp ["cs","cs1","colorednodes","rest"])
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorrest.cs,colorrest.ncs,colorrest.rest,colornoderest.cs,colornoderest.ncs,colornode
      time: 0.28s
!>    Component may not terminate
       colorrest-colorrest:[(1,>=,1),(1,>=,2),(4,>=,4),(5,>,3),(5,>=,5)]*[9,12,14]
       colorrest-colorrest:[(1,>,2),(1,>=,1),(4,>=,4),(5,>,3),(5,>=,5)]*[9,13,12,14]
       colorrest-colorrest:[(1,>,2),(1,>=,1),(3,>=,3),(4,>=,4)]*[9,12,15,14]
       colorrest-colorrest:[(2,>,3),(2,>=,1),(2,>=,2),(4,>=,4),(5,>=,5)]*[9,12,14,9,12,14]
       colorrest-colorrest:[(2,>,1),(2,>,3),(2,>=,2),(4,>=,4),(5,>=,5)]*[9,12,14,9,12,15,14]
      Using the graphs:
       colorrest-colornoderest:[(1,>=,1),(2,>=,2),(4,>,3),(3,>=,4),(4,>=,5)]*[9]
       colornoderest-colorrest:[(1,>=,1),(1,>=,2),(5,>,4)]*[11]
       colornoderest-colorrestthetrick:[(1,>=,1),(1,>=,2),(2,>,3),(4,>=,4),(5,>=,5)]*[12]
       colornoderest-colornoderest:[(1,>=,1),(2,>,2),(3,>=,3),(4,>=,4),(5,>=,5)]*[13]
       colorrestthetrick-colorrest:[(2,>=,1),(1,>=,2),(4,>=,3),(5,>=,4)]*[14]
       colorrestthetrick-colorrestthetrick:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4),(5,>=,5)]*[15]

      --- Analysis of component: ["possible"] ---
      size: ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [possible.color,possible.adjs,possible.colorednodes,colorof.node,colorof.colorednodes,r
      time: 0.00s
      <No failures>

      --- Analysis of component: ["colorof"] ---
      size: ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorof.node,colorof.colorednodes,revapp.xs]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["revapp"] ---
      size: ("revapp",Lin ["xs","rest"] 0)
      crit: [revapp.xs]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.30s
-------------------------------------
```

graphcolour3

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/graphcolour3.l
-----------------------------------------

      --- Analysis of component: ["colornode"] ---
      size: ("colornode",Lin ["cs","node"] 1)
            ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
```

```
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colornode.cs,colornode.node,colornode.colorednodes,colorrest.cs,colorrest.ncs,colorres
      time: 0.01s
      <No failures>

      --- Analysis of component: ["colorrest","colornoderest"] ---
      size: ("colorrest",Exp ["cs","ncs","rest","colorednodes"])
            ("colornoderest",Exp ["cs","rest","ncs","node","colorednodes"])
            ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorrest.cs,colorrest.ncs,colorrest.rest,colornoderest.cs,colornoderest.ncs,colornode
      time: 0.10s
!>    Possible duplication of variables: [colornoderest.cs,colornoderest.rest]
      in the bounding argument positions: [11:colorrest.cs,11:colorrest.ncs,11:colorrest.rest,12:co

      --- Analysis of component: ["possible"] ---
      size: ("possible",Lin [] 0)
            ("reverse",Lin ["xs"] 0)
            ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [possible.color,possible.adjs,possible.colorednodes,colorof.node,colorof.colorednodes,r
      time: 0.00s
      <No failures>

      --- Analysis of component: ["colorof"] ---
      size: ("colorof",Lin ["colorednodes"] 0)
            ("revapp",Lin ["xs","rest"] 0)
      crit: [colorof.node,colorof.colorednodes,revapp.xs]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["revapp"] ---
      size: ("revapp",Lin ["xs","rest"] 0)
      crit: [revapp.xs]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.12s
-------------------------------------
```

match

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/match.l
-------------------------------------

      --- Analysis of component: ["loop"] ---
      size: ("loop",Lin [] 0)
      crit: [loop.p,loop.s,loop.pp,loop.ss]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

reach

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/reach.l
-------------------------------------
```

```
      Non-termination detected.

      --- Analysis of component: ["reach","via"] ---
      size: ("reach",Exp ["edges","u","v"])
            ("via",Exp ["v","edges","rest"])
            ("member",Lin [] 0)
      crit: [reach.u,reach.v,reach.edges,via.u,via.v,via.rest,via.edges,member.x,member.xs]
      time: 0.03s
!>    Component may not terminate
       reach-reach:[(2,>=,2),(3,>,1),(3,>=,3)]*[3,4]
       reach-reach:[(2,>=,2),(4,>,1),(4,>=,3),(4,>=,4)]*[3,4,3,4]
       reach-reach:[(2,>=,2),(4,>,1),(4,>,3),(4,>=,4)]*[3,4,3,5,4]
      Using the graphs:
       reach-via:[(1,>=,1),(2,>=,2),(3,>=,3),(3,>=,4)]*[3]
       via-reach:[(3,>,1),(2,>=,2),(4,>=,3)]*[4]
       via-via:[(1,>=,1),(2,>=,2),(3,>,3),(4,>=,4)]*[5]
       via-via:[(1,>=,1),(2,>=,2),(3,>,3),(4,>=,4)]*[6]

      --- Analysis of component: ["member"] ---
      size: ("member",Lin [] 0)
      crit: [member.x,member.xs]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.04s
-------------------------------------
```

rematch

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/rematch.l
-------------------------------------
      Non-termination detected.

      --- Analysis of component: ["parsep","parsepdot","parsepstar","parsepopenb","parsepcloseb","p
      size: ("parsep",Exp ["seq","stack","patchars"])
            ("parsepdot",Exp ["patchars","seq","stack"])
            ("parsepstar",Exp ["patchars","stack","seq"])
            ("parsepopenb",Exp ["patchars","seq","stack"])
            ("parsepcloseb",Exp ["patchars","seq","stack"])
            ("parsepchar",Exp ["seq","stack","patchars"])
            ("domatch",Exp ["pat","cs"])
            ("domatchstar",Exp ["cs","init","pat"])
            ("domatchseq",Exp ["pats","rest","cs"])
            ("liststring",Exp ["x"])
            ("append",Lin ["x","y"] 1)
            ("domatchchar",Lin ["cs"] 1)
            ("domatchdot",Lin ["cs"] 1)
            ("reverse",Lin ["x"] 0)
            ("dummy",Exp ["x"])
            ("stringlist",Exp ["x"])
      crit: [parsep.patchars,parsepdot.patchars,parsepstar.patchars,parsepopenb.patchars,parsepclos
      time: 0.86s
!>    Component may not terminate
       parsep-parsep:[(1,>=,1),(3,>=,3)]*[14,23]
      Using the graphs:
       parsep-parsepdot:[(1,>=,1),(2,>=,2),(3,>=,3)]*[9]
       parsep-parsepstar:[(1,>=,1),(2,>=,2),(3,>=,3)]*[10]
       parsep-parsepopenb:[(1,>=,1),(2,>=,2),(3,>=,3)]*[11]
       parsep-parsepcloseb:[(1,>=,1),(2,>=,2),(3,>=,3)]*[12]
       parsep-parsepchar:[(1,>,1),(2,>=,2),(3,>=,3)]*[13]
       parsep-parsepchar:[(1,>=,1),(2,>=,2),(3,>=,3)]*[14]
       parsepdot-parsep:[(1,>,1),(3,>=,3)]*[16]
       parsepstar-parsep:[(1,>,1),(3,>=,3)]*[17]
       parsepstar-parsep:[(1,>,1),(3,>=,3)]*[18]
       parsepopenb-parsep:[(1,>,1)]*[19]
       parsepcloseb-parsep:[(1,>,1),(3,>,3)]*[20]
```

```
39        parsepchar-parsep:[(1,>,1),(3,>=,3)]*[22]
40        parsepchar-parsep:[(1,>=,1),(3,>=,3)]*[23]
41
42      --- Analysis of component: ["domatch","domatchstar","domatchseq"] ---
43      size: ("domatch",Exp ["pat","cs"])
44            ("domatchstar",Exp ["cs","init","pat"])
45            ("domatchseq",Exp ["pats","rest","cs"])
46            ("liststring",Exp ["x"])
47            ("append",Lin ["x","y"] 1)
48            ("domatchchar",Lin ["cs"] 1)
49            ("domatchdot",Lin ["cs"] 1)
50            ("reverse",Lin ["x"] 0)
51            ("dummy",Exp ["x"])
52            ("stringlist",Exp ["x"])
53      crit: [domatch.pat,domatchstar.pat,domatchseq.pats,dummy.x,stringlist.x]
54      time: 0.20s
55  !>  Component may not terminate
56        domatchseq-domatchseq:[(3,>=,3)]*[35]
57        domatchstar-domatchstar:[(2,>=,2)]*[29]
58      Using the graphs:
59        domatch-domatchstar:[(2,>=,1),(1,>,2)]*[26]
60        domatch-domatchseq:[(2,>=,1),(1,>,3)]*[27]
61        domatchstar-domatch:[(2,>=,1),(1,>=,2)]*[28]
62        domatchstar-domatchstar:[(2,>=,2)]*[29]
63        domatchseq-domatch:[(3,>,1),(1,>=,2)]*[31]
64        domatchseq-domatchseq:[(3,>,3)]*[32]
65        domatchseq-domatchseq:[(3,>=,3)]*[35]
66
67      --- Analysis of component: ["dummy","stringlist"] ---
68      size: ("dummy",Exp ["x"])
69            ("stringlist",Exp ["x"])
70      crit: [dummy.x,stringlist.x]
71      time: 0.00s
72  !>  Component may not terminate
73        stringlist-stringlist:[(1,>=,1)]*[40,41]
74      Using the graphs:
75        stringlist-dummy:[(1,>=,1)]*[40]
76        dummy-stringlist:[(1,>=,1)]*[41]
77
78  ---------- Analysis Complete----------
79  TOTAL Time : 1.07s
80  ----------------------------------------
```

strmatch

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/strmatch.l
3   ----------------------------------------
4
5       --- Analysis of component: ["domatch"] ---
6       size: ("domatch",Lin ["cs","n"] 1)
7             ("prefix",Lin ["precs","cs"] 0)
8       crit: [domatch.patcs,domatch.cs,prefix.precs,prefix.cs]
9       time: 0.01s
10      <No failures>
11
12      --- Analysis of component: ["prefix"] ---
13      size: ("prefix",Lin ["precs","cs"] 0)
14      crit: [prefix.precs,prefix.cs]
15      time: 0.00s
16      <No failures>
17
18  ---------- Analysis Complete----------
19  TOTAL Time : 0.01s
20  ----------------------------------------
```

typeinf

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/algorithms/typeinf.l
3  ----------------------------------------
4
5      --- Analysis of component: ["etype"] ---
6      size: ("etype",Exp ["tenv","venv","t","e"])
7            ("unify",Exp ["venv","t2","t1"])
8            ("subst",Exp ["venv","t"])
9            ("freshtvar",Lin ["tenv"] 2)
10           ("vtype",Lin ["venv"] (-2))
11           ("tsubst",Exp ["t","t1"])
12     crit: [etype.e,unify.t1,unify.t2,subst.venv,subst.a,subst.t,vtype.venv,vtype.x,tsubst.a,tsubs
13     time: 0.15s
14 !>  Variables of dangerous growth: [etype.t,etype.tenv,etype.venv,etype.t,etype.tenv,etype.venv,e
15     passed in bounding argument positions: [19:unify.t2,19:unify.t2,19:unify.t2,19:unify.t1,19:un
16 !>  Non size-linear outer calls: [19:unify]
17     due to the bounding argument positions: [19:unify.t1]
18
19     --- Analysis of component: ["unify"] ---
20     size: ("unify",Exp ["venv","t2","t1"])
21           ("subst",Exp ["venv","t"])
22           ("freshtvar",Lin ["tenv"] 2)
23           ("vtype",Lin ["venv"] (-2))
24           ("tsubst",Exp ["t","t1"])
25     crit: [unify.t1,unify.t2,subst.venv,subst.a,subst.t,vtype.venv,vtype.x,tsubst.a,tsubst.t,tsub
26     time: 0.03s
27 !>  Variables of dangerous growth: [unify.venv,unify.venv]
28     passed in bounding argument positions: [9:subst.venv,8:subst.venv]
29
30     --- Analysis of component: ["subst"] ---
31     size: ("subst",Exp ["venv","t"])
32           ("freshtvar",Lin ["tenv"] 2)
33           ("vtype",Lin ["venv"] (-2))
34           ("tsubst",Exp ["t","t1"])
35     crit: [subst.venv,subst.a,subst.t,vtype.venv,vtype.x,tsubst.a,tsubst.t,tsubst.t1]
36     time: 0.00s
37     <No failures>
38
39     --- Analysis of component: ["vtype"] ---
40     size: ("vtype",Lin ["venv"] (-2))
41           ("tsubst",Exp ["t","t1"])
42     crit: [vtype.venv,vtype.x,tsubst.a,tsubst.t,tsubst.t1]
43     time: 0.01s
44     <No failures>
45
46     --- Analysis of component: ["tsubst"] ---
47     size: ("tsubst",Exp ["t","t1"])
48     crit: [tsubst.a,tsubst.t,tsubst.t1]
49     time: 0.00s
50 !>  Possible duplication of variables: [tsubst.a,tsubst.t]
51     in the bounding argument positions: [4:tsubst.a,4:tsubst.t,5:tsubst.t,5:tsubst.a]
52
53 ----------Analysis Complete----------
54 TOTAL Time : 0.19s
55 ----------------------------------------
```

B.6 Glenstrup Results - Basic

add

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/add.l
3  ----------------------------------------
4
5      --- Analysis of component: ["add0"] ---
```

```
      size: ("add0",Exp ["y","x"])
      crit: [add0.y]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

addlists

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/addlists.l
-----------------------------------------

      --- Analysis of component: ["addlist"] ---
      size: ("addlist",Lin ["xs","ys"] 0)
      crit: [addlist.xs,addlist.ys]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

anchored

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/anchored.l
-----------------------------------------

      --- Analysis of component: ["anchored"] ---
      size: ("anchored",Exp ["x","y"])
      crit: [anchored.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

append

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/append.l
-----------------------------------------

      --- Analysis of component: ["append"] ---
      size: ("append",Lin ["ys","xs"] 0)
      crit: [append.xs,append.ys]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

assrewrite

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/assrewrite.l
-----------------------------------------
      Non-termination detected.

      --- Analysis of component: ["rewrite"] ---
```

```
      size: ("rewrite",Exp ["exp"])
      crit: []
      time: 0.01s
!>  Component may not terminate
       rewrite-rewrite:[]*[5]
      Using the graphs:
       rewrite-rewrite:[(1,>,1)]*[2]
       rewrite-rewrite:[(1,>,1)]*[3]
       rewrite-rewrite:[(1,>,1)]*[4]
       rewrite-rewrite:[]*[5]
       rewrite-rewrite:[(1,>,1)]*[6]
       rewrite-rewrite:[(1,>,1)]*[7]

----------Analysis Complete----------
TOTAL Time : 0.02s
-------------------------------------
```

badd

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/badd.l
-------------------------------------
      Non-termination detected.

      --- Analysis of component: ["badd"] ---
      size: ("badd",Exp ["x"])
      crit: []
      time: 0.00s
!>  Component may not terminate
       badd-badd:[]*[2]
      Using the graphs:
       badd-badd:[]*[2]
       badd-badd:[(1,>=,1),(2,>,2)]*[3]

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

contrived1

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/contrived1.l
-------------------------------------
      Non-termination detected.

      --- Analysis of component: ["f","g"] ---
      size: ("f",Exp ["y","z","d","x"])
            ("g",Exp ["w","v","u"])
            ("h",Lin ["r"] 0)
            ("dec",Lin ["n"] (-1))
            ("number42",Lin [] 42)
      crit: [f.y,f.z,f.d,g.v,g.w,h.r]
      time: 0.03s
      <No failures>

      --- Analysis of component: ["h"] ---
      size: ("h",Lin ["r"] 0)
            ("dec",Lin ["n"] (-1))
            ("number42",Lin [] 42)
      crit: [h.r]
      time: 0.01s
!>  Component terminates, but call depth may be super-polynomial:
       h-h:[(1,>=,1)]*[9]
      Using the graphs:
       h-h:[(1,>,1)]*[7]
       h-h:[(1,>=,1)]*[9]

```

```
----------Analysis Complete----------
TOTAL Time : 0.04s
-------------------------------------
```

contrived2

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/contrived2.l
-------------------------------------
    Non-termination detected.

    --- Analysis of component: ["f","g"] ---
    size: ("f",Exp ["y","z","d","x"])
          ("g",Exp ["w","v","u"])
          ("h",Exp ["r"])
          ("dec",Lin ["n"] (-1))
          ("number17",Lin [] 17)
          ("number42",Lin [] 42)
    crit: [f.y,f.z,f.d,g.v,g.w]
    time: 0.03s
    <No failures>

    --- Analysis of component: ["h"] ---
    size: ("h",Exp ["r"])
          ("dec",Lin ["n"] (-1))
          ("number17",Lin [] 17)
          ("number42",Lin [] 42)
    crit: []
    time: 0.00s
!>  Component may not terminate
     h-h:[]*[7,9]
    Using the graphs:
     h-h:[(1,>,1)]*[7]
     h-h:[(2,>,2)]*[9]

----------Analysis Complete----------
TOTAL Time : 0.04s
-------------------------------------
```

decrease

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/decrease.l
-------------------------------------

    --- Analysis of component: ["decrease"] ---
    size: ("decrease",Lin [] 42)
          ("number42",Lin [] 42)
    crit: [decrease.x]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

deeprev

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/deeprev.l
-------------------------------------

    --- Analysis of component: ["deeprev","deeprevapp"] ---
    size: ("deeprev",Lin ["x"] 0)
          ("deeprevapp",Lin ["xs","rest"] 0)
          ("revconsapp",Lin ["xs","r"] 0)
```

```
      crit: [deeprev.x,deeprevapp.xs,revconsapp.xs]
      time: 0.01s
      <No failures>

      --- Analysis of component: ["revconsapp"] ---
      size: ("revconsapp",Lin ["xs","r"] 0)
      crit: [revconsapp.xs]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.02s
-------------------------------------
```

disjconj

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/disjconj.l
-------------------------------------

      --- Analysis of component: ["disj","conj"] ---
      size: ("disj",Lin ["p"] 0)
            ("conj",Lin ["p"] 0)
            ("bool",Lin ["p"] 0)
      crit: [disj.p,conj.p]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.02s
-------------------------------------
```

duplicate

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/duplicate.l
-------------------------------------

      --- Analysis of component: ["duplicate"] ---
      size: ("duplicate",Exp ["xs"])
      crit: [duplicate.xs]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

equal

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/equal.l
-------------------------------------
      Non-termination detected.

      --- Analysis of component: ["equal0"] ---
      size: ("equal0",Lin [] 42)
            ("number42",Lin [] 42)
      crit: [equal0.x]
      time: 0.01s
!>    Component may not terminate
       equal0-equal0:[(1,>=,1)]*[3]
      Using the graphs:
       equal0-equal0:[(1,>=,1)]*[3]

```

```
16  ----------Analysis Complete----------
17  TOTAL Time : 0.01s
18  ----------------------------------------
```

evenodd

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/evenodd.l
3   ----------------------------------------
4
5       --- Analysis of component: ["even","odd"] ---
6       size: ("even",Lin [] 0)
7             ("odd",Lin [] 0)
8       crit: [even.x,odd.x]
9       time: 0.00s
10      <No failures>
11
12  ----------Analysis Complete----------
13  TOTAL Time : 0.00s
14  ----------------------------------------
```

fold

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/fold.l
3   ----------------------------------------
4
5       --- Analysis of component: ["foldr"] ---
6       size: ("foldr",Lin ["xs","a"] 0)
7             ("foldl",Lin ["xs","a"] 0)
8             ("op",Lin ["x1","x2"] 0)
9       crit: [foldr.a,foldr.xs,foldl.xs]
10      time: 0.00s
11      <No failures>
12
13      --- Analysis of component: ["foldl"] ---
14      size: ("foldl",Lin ["xs","a"] 0)
15            ("op",Lin ["x1","x2"] 0)
16      crit: [foldl.xs]
17      time: 0.01s
18      <No failures>
19
20  ----------Analysis Complete----------
21  TOTAL Time : 0.01s
22  ----------------------------------------
```

game

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/game.l
3   ----------------------------------------
4
5       --- Analysis of component: ["game"] ---
6       size: ("game",Lin ["p1","moves","p2"] 1)
7       crit: [game.moves]
8       time: 0.01s
9       <No failures>
10
11  ----------Analysis Complete----------
12  TOTAL Time : 0.01s
13  ----------------------------------------
```

increase

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/increase.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["increase"] ---
7       size: ("increase",Lin [] 42)
8             ("number42",Lin [] 42)
9       crit: []
10      time: 0.00s
11 !>   Component may not terminate
12       increase-increase:[]*[3]
13      Using the graphs:
14       increase-increase:[]*[3]
15
16 ----------Analysis Complete----------
17 TOTAL Time : 0.00s
18 ----------------------------------------
```

intlookup

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/intlookup.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["intlookup"] ---
7       size: ("intlookup",Exp [])
8             ("lookup",Lin ["p"] (-1))
9       crit: [intlookup.e,intlookup.p,lookup.fnum,lookup.p]
10      time: 0.00s
11 !>   Component may not terminate
12       intlookup-intlookup:[(2,>,1),(2,>=,2)]*[2]
13      Using the graphs:
14       intlookup-intlookup:[(2,>,1),(2,>=,2)]*[2]
15
16      --- Analysis of component: ["lookup"] ---
17      size: ("lookup",Lin ["p"] (-1))
18      crit: [lookup.fnum,lookup.p]
19      time: 0.00s
20      <No failures>
21
22 ----------Analysis Complete----------
23 TOTAL Time : 0.00s
24 ----------------------------------------
```

letexp

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/letexp.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["letexp"] ---
7       size: ("letexp",Exp [])
8       crit: [letexp.y]
9       time: 0.00s
10 !>   Component may not terminate
11       letexp-letexp:[(2,>=,2)]*[2]
12      Using the graphs:
13       letexp-letexp:[(2,>=,2)]*[2]
14
15 ----------Analysis Complete----------
16 TOTAL Time : 0.01s
17 ----------------------------------------
```

list

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/list.l
-------------------------------------

    --- Analysis of component: ["list"] ---
    size: ("list",Lin ["xs"] 0)
    crit: [list.xs]
    time: 0.01s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

lte

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/lte.l
-------------------------------------

    --- Analysis of component: ["even"] ---
    size: ("even",Lin [] 0)
    crit: [even.x]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

map0

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/map0.l
-------------------------------------

    --- Analysis of component: ["map"] ---
    size: ("map",Lin ["xs"] 0)
          ("f",Lin ["x"] 0)
    crit: [map.xs]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

member

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/member.l
-------------------------------------

    --- Analysis of component: ["member"] ---
    size: ("member",Lin [] 0)
    crit: [member.x,member.xs]
    time: 0.00s
    <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

mergelists

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/mergelists.l
-------------------------------------

      --- Analysis of component: ["merge"] ---
      size: ("merge",Lin ["ys","xs"] 0)
      crit: [merge.xs,merge.ys]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

mul

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/mul.l
-------------------------------------

      --- Analysis of component: ["mul0"] ---
      size: ("mul0",Exp ["y"])
            ("add0",Lin ["x","y"] 0)
      crit: [mul0.x,mul0.y,add0.x]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["add0"] ---
      size: ("add0",Lin ["x","y"] 0)
      crit: [add0.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

naiverev

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/naiverev.l
-------------------------------------

      --- Analysis of component: ["naiverev"] ---
      size: ("naiverev",Lin ["xs"] 0)
            ("app",Lin ["ys","xs"] 0)
      crit: [naiverev.xs,app.xs,app.ys]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["app"] ---
      size: ("app",Lin ["ys","xs"] 0)
      crit: [app.xs,app.ys]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

nestdec

```
------------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/nestdec.l
------------------------------------------

     --- Analysis of component: ["nestdec"] ---
     size: ("nestdec",Lin [] 17)
           ("dec",Lin ["x"] (-1))
           ("number17",Lin [] 17)
     crit: [nestdec.x,dec.x]
     time: 0.00s
     <No failures>

     --- Analysis of component: ["dec"] ---
     size: ("dec",Lin ["x"] (-1))
           ("number17",Lin [] 17)
     crit: [dec.x]
     time: 0.01s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
------------------------------------------
```

nesteql

```
------------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/nesteql.l
------------------------------------------
     Non-termination detected.

     --- Analysis of component: ["nesteql"] ---
     size: ("nesteql",Lin [] 17)
           ("eql",Lin ["x"] 0)
           ("number17",Lin [] 17)
     crit: [nesteql.x,eql.x]
     time: 0.00s
!>   Component may not terminate
      nesteql-nesteql:[(1,>=,1)]*[3]
     Using the graphs:
      nesteql-nesteql:[(1,>=,1)]*[3]

     --- Analysis of component: ["eql"] ---
     size: ("eql",Lin ["x"] 0)
           ("number17",Lin [] 17)
     crit: [eql.x]
     time: 0.00s
!>   Component may not terminate
      eql-eql:[(1,>=,1)]*[5]
     Using the graphs:
      eql-eql:[(1,>=,1)]*[5]

----------Analysis Complete----------
TOTAL Time : 0.00s
------------------------------------------
```

nestimeql

```
------------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/nestimeql.l
------------------------------------------
     Non-termination detected.

     --- Analysis of component: ["nestimeql"] ---
     size: ("nestimeql",Lin [] 42)
           ("immatcopy",Lin ["x"] 0)
           ("number42",Lin [] 42)
```

```
      crit: [nestimeql.x,immatcopy.x]
      time: 0.00s
!>    Component may not terminate
       nestimeql-nestimeql:[(1,>=,1)]*[3]
      Using the graphs:
       nestimeql-nestimeql:[(1,>=,1)]*[3]

      --- Analysis of component: ["immatcopy"] ---
      size: ("immatcopy",Lin ["x"] 0)
            ("number42",Lin [] 42)
      crit: [immatcopy.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
-------------------------------------
```

nestinc

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/nestinc.l
-------------------------------------
      Non-termination detected.

      --- Analysis of component: ["nestinc"] ---
      size: ("nestinc",Lin [] 17)
            ("inc",Exp ["x"])
            ("number17",Lin [] 17)
      crit: [inc.x]
      time: 0.00s
!>    Component may not terminate
       nestinc-nestinc:[]*[3]
      Using the graphs:
       nestinc-nestinc:[]*[3]

      --- Analysis of component: ["inc"] ---
      size: ("inc",Exp ["x"])
            ("number17",Lin [] 17)
      crit: [inc.x]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

nolexicord

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/nolexicord.l
-------------------------------------

      --- Analysis of component: ["nolexicord"] ---
      size: ("nolexicord",Lin [] 42)
            ("number42",Lin [] 42)
      crit: [nolexicord.a1,nolexicord.b1,nolexicord.a2,nolexicord.b2,nolexicord.a3,nolexicord.b3]
      time: 0.02s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.02s
-------------------------------------
```

ordered

```
------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/ordered.l
------------------------------------

     --- Analysis of component: ["ordered"] ---
     size: ("ordered",Lin [] 0)
     crit: [ordered.xs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.00s
------------------------------------
```

overlap

```
------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/overlap.l
------------------------------------

     --- Analysis of component: ["overlap"] ---
     size: ("overlap",Lin [] 0)
           ("member",Lin [] 0)
     crit: [overlap.xs,overlap.ys,member.x,member.xs]
     time: 0.00s
     <No failures>

     --- Analysis of component: ["member"] ---
     size: ("member",Lin [] 0)
     crit: [member.x,member.xs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
------------------------------------
```

permute

```
------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/permute.l
------------------------------------
     Non-termination detected.

     --- Analysis of component: ["permute","select"] ---
     size: ("permute",Exp ["xs"])
           ("select",Exp ["postfix","revprefix","x"])
           ("revapp",Lin ["xs","rest"] 0)
           ("mapconsapp",Exp ["rest","x","xss"])
     crit: [revapp.xs,mapconsapp.x,mapconsapp.xss,mapconsapp.rest]
     time: 0.03s
!>   Component may not terminate
      permute-permute:[]*[2,4]
     Using the graphs:
      permute-select:[(1,>,1),(1,>,3)]*[2]
      select-permute:[]*[4]
      select-select:[(3,>,1),(3,>,3)]*[6]

     --- Analysis of component: ["revapp"] ---
     size: ("revapp",Lin ["xs","rest"] 0)
           ("mapconsapp",Exp ["rest","x","xss"])
     crit: [revapp.xs,mapconsapp.x,mapconsapp.xss,mapconsapp.rest]
     time: 0.01s
     <No failures>

```

```
27      --- Analysis of component: ["mapconsapp"] ---
28      size: ("mapconsapp",Exp ["rest","x","xss"])
29      crit: [mapconsapp.x,mapconsapp.xss,mapconsapp.rest]
30      time: 0.00s
31      <No failures>
32
33  ----------Analysis Complete----------
34  TOTAL Time : 0.04s
35  ----------------------------------------
```

revapp

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/revapp.l
3   ----------------------------------------
4
5       --- Analysis of component: ["revapp"] ---
6       size: ("revapp",Lin ["xs","rest"] 0)
7       crit: [revapp.xs]
8       time: 0.00s
9       <No failures>
10
11  ----------Analysis Complete----------
12  TOTAL Time : 0.00s
13  ----------------------------------------
```

select

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/select.l
3   ----------------------------------------
4
5       --- Analysis of component: ["selects"] ---
6       size: ("selects",Exp ["revprefix","postfix","x"])
7             ("revapp",Lin ["xs","rest"] 0)
8       crit: [selects.x,selects.postfix,revapp.xs]
9       time: 0.00s
10      <No failures>
11
12      --- Analysis of component: ["revapp"] ---
13      size: ("revapp",Lin ["xs","rest"] 0)
14      crit: [revapp.xs]
15      time: 0.01s
16      <No failures>
17
18  ----------Analysis Complete----------
19  TOTAL Time : 0.01s
20  ----------------------------------------
```

shuffle

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/shuffle.l
3   ----------------------------------------
4
5       --- Analysis of component: ["shuffle"] ---
6       size: ("shuffle",Lin ["xs"] 0)
7             ("reverse",Lin ["xs"] 0)
8             ("append",Lin ["ys","xs"] 0)
9       crit: [shuffle.xs,reverse.xs,append.xs,append.ys]
10      time: 0.00s
11      <No failures>
12
13      --- Analysis of component: ["reverse"] ---
14      size: ("reverse",Lin ["xs"] 0)
15            ("append",Lin ["ys","xs"] 0)
```

```
16      crit: [reverse.xs,append.xs,append.ys]
17      time: 0.00s
18      <No failures>
19
20      --- Analysis of component: ["append"] ---
21      size: ("append",Lin ["ys","xs"] 0)
22      crit: [append.xs,append.ys]
23      time: 0.01s
24      <No failures>
25
26  ----------Analysis Complete----------
27  TOTAL Time : 0.01s
28  ----------------------------------------
```

sp1

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/sp1.l
3   ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["f","g","h"] ---
7       size: ("f",Lin ["x"] 0)
8             ("g",Lin ["x"] 0)
9             ("h",Lin ["x"] 0)
10            ("r",Lin ["x"] 0)
11      crit: [f.x,f.y,g.x,g.y,h.x,h.y]
12      time: 0.01s
13  !>  Component may not terminate
14       h-h:[(1,>=,1),(2,>=,2)]*[6]
15       f-f:[(1,>=,1),(2,>=,2)]*[3,7]
16       f-f:[(1,>=,1),(2,>=,2)]*[2,4,7]
17      Using the graphs:
18       f-g:[(1,>=,1),(2,>=,2)]*[2]
19       f-h:[(1,>=,1),(2,>=,2)]*[3]
20       g-h:[(1,>=,1),(2,>=,2)]*[4]
21       h-h:[(1,>=,1),(2,>=,2)]*[6]
22       h-f:[(1,>=,1),(2,>=,2)]*[7]
23
24  ----------Analysis Complete----------
25  TOTAL Time : 0.01s
26  ----------------------------------------
```

subsets

```
1   ----------------------------------------
2   Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/subsets.l
3   ----------------------------------------
4
5       --- Analysis of component: ["subsets"] ---
6       size: ("subsets",Exp ["xs"])
7             ("mapconsapp",Exp ["rest","x","xss"])
8       crit: [subsets.xs,mapconsapp.x,mapconsapp.xss,mapconsapp.rest]
9       time: 0.00s
10  !>  Non size-linear outer calls: [3:mapconsapp]
11      due to the bounding argument positions: [3:mapconsapp.rest,3:mapconsapp.xss]
12
13      --- Analysis of component: ["mapconsapp"] ---
14      size: ("mapconsapp",Exp ["rest","x","xss"])
15      crit: [mapconsapp.x,mapconsapp.xss,mapconsapp.rest]
16      time: 0.01s
17      <No failures>
18
19  ----------Analysis Complete----------
20  TOTAL Time : 0.01s
21  ----------------------------------------
```

thetrick

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/thetrick.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["g"] ---
7       size: ("g",Lin [] 42)
8             ("f",Lin [] 42)
9             ("lt0",Lin [] 0)
10            ("number42",Lin [] 42)
11      crit: [g.x,f.x,f.y,lt0.x,lt0.y]
12      time: 0.00s
13 !>  Component terminates, but call depth may be super-polynomial:
14       g-g:[(1,>=,1)]*[9]
15      Using the graphs:
16       g-g:[(1,>=,1)]*[9]
17       g-g:[(1,>,1)]*[10]
18
19      --- Analysis of component: ["f"] ---
20      size: ("f",Lin [] 42)
21            ("lt0",Lin [] 0)
22            ("number42",Lin [] 42)
23      crit: [f.x,f.y,lt0.x,lt0.y]
24      time: 0.01s
25      <No failures>
26
27      --- Analysis of component: ["lt0"] ---
28      size: ("lt0",Lin [] 0)
29            ("number42",Lin [] 42)
30      crit: [lt0.x,lt0.y]
31      time: 0.00s
32      <No failures>
33
34 ----------Analysis Complete----------
35 TOTAL Time : 0.01s
36 ----------------------------------------
```

vangelder

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/basic/vangelder.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["q","r","t","p"] ---
7       size: ("q",Lin [] 0)
8             ("r",Lin [] 0)
9             ("t",Lin [] 0)
10            ("p",Lin [] 0)
11            ("e",Lin ["a","b"] 0)
12      crit: [q.x,q.y,r.x,r.y,t.x,t.y,p.x,p.y]
13      time: 0.08s
14 !>  Component may not terminate
15       q-q:[(1,>=,1),(2,>=,2)]*[3,7,12,14]
16      Using the graphs:
17       q-p:[(1,>=,1),(2,>=,2)]*[3]
18       q-q:[(1,>=,1),(2,>,2)]*[4]
19       q-p:[(1,>=,1),(2,>,2)]*[5]
20       p-r:[(1,>=,1),(2,>=,2)]*[7]
21       p-p:[(1,>=,1),(2,>,2)]*[8]
22       r-q:[(1,>=,1),(2,>,2)]*[10]
23       r-r:[(1,>=,1),(2,>,2)]*[11]
24       r-t:[(1,>=,1),(2,>=,2)]*[12]
25       t-q:[(1,>=,1),(2,>=,2)]*[14]
26       t-t:[(1,>,1),(2,>,2)]*[15]
27
```

```
28 ----------Analysis Complete----------
29 TOTAL Time : 0.08s
30 ----------------------------------------
```

B.7 Glenstrup Results - Interpreters

int

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/int.l
3  ----------------------------------------
4       Non-termination detected.
5
6       --- Analysis of component: ["eeval"] ---
7       size: ("eeval",Exp ["p","e","vs"])
8             ("lookname",Lin ["p"] (-3))
9             ("lookbody",Lin ["p"] (-4))
10            ("apply",Lin ["v1","v2"] 1)
11            ("lookvar",Lin ["vs"] (-1))
12      crit: [eeval.e,eeval.ns,eeval.p,lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar
13      time: 0.29s
14 !>   Component may not terminate
15       eeval-eeval:[(4,>,1),(4,>,2),(4,>=,4)]*[16]
16      Using the graphs:
17       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[5]
18       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[6]
19       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[8]
20       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[9]
21       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[10]
22       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[11]
23       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[12]
24       eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[15]
25       eeval-eeval:[(4,>,1),(4,>,2),(4,>=,4)]*[16]
26
27      --- Analysis of component: ["lookname"] ---
28      size: ("lookname",Lin ["p"] (-3))
29            ("lookbody",Lin ["p"] (-4))
30            ("apply",Lin ["v1","v2"] 1)
31            ("lookvar",Lin ["vs"] (-1))
32      crit: [lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
33      time: 0.01s
34      <No failures>
35
36      --- Analysis of component: ["lookbody"] ---
37      size: ("lookbody",Lin ["p"] (-4))
38            ("apply",Lin ["v1","v2"] 1)
39            ("lookvar",Lin ["vs"] (-1))
40      crit: [lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
41      time: 0.00s
42      <No failures>
43
44      --- Analysis of component: ["lookvar"] ---
45      size: ("lookvar",Lin ["vs"] (-1))
46      crit: [lookvar.x,lookvar.ns,lookvar.vs]
47      time: 0.00s
48      <No failures>
49
50 ----------Analysis Complete----------
51 TOTAL Time : 0.30s
52 ----------------------------------------
```

intdynscope

```
1  ----------------------------------------
2  Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/intdynscope.l
```

```
----------------------------------------
     Non-termination detected.

     --- Analysis of component: ["eeval"] ---
     size: ("eeval",Exp ["p","e","vs"])
           ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [eeval.e,eeval.p,lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookv
     time: 0.45s
!>   Component may not terminate
      eeval-eeval:[(4,>,1),(4,>=,4)]*[16]
     Using the graphs:
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[5]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[6]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[8]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[9]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[10]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[11]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[12]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[15]
      eeval-eeval:[(4,>,1),(4,>=,4)]*[16]

     --- Analysis of component: ["lookname"] ---
     size: ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.00s
     <No failures>

     --- Analysis of component: ["lookbody"] ---
     size: ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.01s
     <No failures>

     --- Analysis of component: ["lookvar"] ---
     size: ("lookvar",Lin ["vs"] (-1))
     crit: [lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.46s
----------------------------------------
```

intloop

```
----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/intloop.l
----------------------------------------
     Refining parameters: [eeval.e,eeval.ns,eeval.l,eeval.p,loopvar.vs]

     --- Analysis of component: ["eeval"] ---
     size: ("eeval",Exp ["p","e","vs"])
           ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [eeval.e,eeval.ns,eeval.l,eeval.p,lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x
     time: 0.40s
!>   Possible duplication of variables: [eeval.e,eeval.ns,eeval.l,eeval.p]
```

```
     in the bounding argument positions: [5:eeval.e,6:eeval.e,5:eeval.ns,6:eeval.ns,5:eeval.l,6:ee
!>   Variables of dangerous growth: [eeval.vs]
     Passed in bounding arument positions: [4:lookvar.vs]

     --- Analysis of component: ["lookname"] ---
     size: ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.01s
     <No failures>

     --- Analysis of component: ["lookbody"] ---
     size: ("lookbody",Lin ["p"] (-4))
           ("apply",Lin ["v1","v2"] 1)
           ("lookvar",Lin ["vs"] (-1))
     crit: [lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.00s
     <No failures>

     --- Analysis of component: ["lookvar"] ---
     size: ("lookvar",Lin ["vs"] (-1))
     crit: [lookvar.x,lookvar.ns,lookvar.vs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 6.36s
-------------------------------------
```

intwhile

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/intwhile.l
-----------------------------------------
     Non-termination detected.

     --- Analysis of component: ["eeval"] ---
     size: ("eeval",Exp ["p","e","vs"])
           ("apply",Lin ["v1","v2"] 1)
           ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("number4",Lin [] 4)
           ("number5",Lin [] 5)
           ("number3",Lin [] 3)
           ("lookvar",Lin ["vs"] (-1))
           ("number2",Lin [] 2)
           ("number1",Lin [] 1)
     crit: [eeval.e,eeval.ns,eeval.p,lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar
     time: 0.31s
!>   Component may not terminate
      eeval-eeval:[(4,>,1),(4,>,2),(4,>=,4)]*[21]
     Using the graphs:
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[8]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[9]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[12]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[13]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[14]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[16]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[17]
      eeval-eeval:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[20]
      eeval-eeval:[(4,>,1),(4,>,2),(4,>=,4)]*[21]

     --- Analysis of component: ["lookname"] ---
     size: ("lookname",Lin ["p"] (-3))
           ("lookbody",Lin ["p"] (-4))
           ("number4",Lin [] 4)
```

```
            ("number5",Lin [] 5)
            ("number3",Lin [] 3)
            ("lookvar",Lin ["vs"] (-1))
            ("number2",Lin [] 2)
            ("number1",Lin [] 1)
      crit: [lookname.f,lookname.p,lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
      time: 0.01s
      <No failures>

      --- Analysis of component: ["lookbody"] ---
      size: ("lookbody",Lin ["p"] (-4))
            ("number4",Lin [] 4)
            ("number5",Lin [] 5)
            ("number3",Lin [] 3)
            ("lookvar",Lin ["vs"] (-1))
            ("number2",Lin [] 2)
            ("number1",Lin [] 1)
      crit: [lookbody.f,lookbody.p,lookvar.x,lookvar.ns,lookvar.vs]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["lookvar"] ---
      size: ("lookvar",Lin ["vs"] (-1))
            ("number2",Lin [] 2)
            ("number1",Lin [] 1)
      crit: [lookvar.x,lookvar.ns,lookvar.vs]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.33s
-------------------------------------
```

lambdaint

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/lambdaint.l
-----------------------------------------
      Non-termination detected.

      --- Analysis of component: ["red"] ---
      size: ("red",Exp ["e"])
            ("subst",Exp ["a","e"])
            ("appe2",Lin ["e"] (-3))
            ("appe1",Lin ["e"] (-2))
            ("mkapp",Lin ["e1","e2"] 6)
            ("lambody",Lin ["e"] (-3))
            ("lamvar",Lin ["e"] (-2))
            ("mklam",Lin ["n","e"] 5)
            ("islam",Lin ["e"] 0)
            ("isvar",Lin ["e"] 0)
      crit: [subst.x,subst.a,subst.e]
      time: 0.03s
!>  Component may not terminate
       red-red:[]*[9]
      Using the graphs:
       red-red:[(1,>,1)]*[4]
       red-red:[(1,>,1)]*[6]
       red-red:[]*[9]

      --- Analysis of component: ["subst"] ---
      size: ("subst",Exp ["a","e"])
            ("appe2",Lin ["e"] (-3))
            ("appe1",Lin ["e"] (-2))
            ("mkapp",Lin ["e1","e2"] 6)
            ("lambody",Lin ["e"] (-3))
            ("lamvar",Lin ["e"] (-2))
            ("mklam",Lin ["n","e"] 5)
```

```
            ("islam",Lin ["e"] 0)
            ("isvar",Lin ["e"] 0)
     crit: [subst.x,subst.a,subst.e]
     time: 0.02s
!>   Possible duplication of variables: [subst.x,subst.a,subst.e]
     in the bounding argument positions: [22:subst.x,22:subst.a,22:subst.e,24:subst.e,24:subst.a,2

----------Analysis Complete----------
TOTAL Time : 0.06s
-------------------------------------
```

parsexp

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/parsexp.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["expr","factor","term"] ---
     size: ("expr",Lin ["xs"] 0)
           ("factor",Lin ["xs"] 0)
           ("term",Lin ["xs"] 0)
           ("atom",Lin ["xs"] 0)
           ("member",Lin [] 0)
     crit: [expr.xs,factor.xs,term.xs,member.x,member.xs]
     time: 0.04s
!>   Component may not terminate
      expr-expr:[(1,>=,1)]*[2,5,8]
     Using the graphs:
      expr-term:[(1,>=,1)]*[2]
      expr-expr:[(1,>,1)]*[4]
      term-factor:[(1,>=,1)]*[5]
      term-term:[(1,>,1)]*[7]
      factor-expr:[(1,>=,1)]*[8]

     --- Analysis of component: ["member"] ---
     size: ("member",Lin [] 0)
     crit: [member.x,member.xs]
     time: 0.00s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.04s
-------------------------------------
```

turing

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/interpreters/turing.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["turing"] ---
     size: ("turing",Exp ["prog","instrs","rtape","revltape"])
           ("lookup",Lin ["instrs"] 0)
     crit: [turing.instrs,turing.prog,lookup.instrs]
     time: 0.18s
!>   Component may not terminate
      turing-turing:[(2,>=,2),(3,>=,3),(4,>=,1),(4,>=,4)]*[7]
      turing-turing:[(2,>=,2),(3,>=,3),(4,>,1),(4,>=,4)]*[7,11]
      turing-turing:[(4,>,1),(4,>=,4)]*[4,7]
      turing-turing:[(2,>=,2),(4,>,1),(4,>=,4)]*[2,7]
     Using the graphs:
      turing-turing:[(1,>,1),(2,>=,2),(4,>=,4)]*[2]
      turing-turing:[(1,>,1),(2,>,2),(4,>=,4)]*[3]
      turing-turing:[(1,>,1),(4,>=,4)]*[4]
```

```
        turing-turing:[(1,>,1),(3,>,3),(4,>=,4)]*[5]
        turing-turing:[(1,>,1),(4,>=,4)]*[6]
        turing-turing:[(4,>=,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[7]
        turing-turing:[(4,>=,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[9]
        turing-turing:[(1,>,1),(2,>=,2),(3,>=,3),(4,>=,4)]*[11]

      --- Analysis of component: ["lookup"] ---
      size: ("lookup",Lin ["instrs"] 0)
      crit: [lookup.instrs]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.18s
-------------------------------------
```

B.8 Glenstrup Results - Simple

ack

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/simple/ack.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["ack"] ---
     size: ("ack",Exp ["m","n"])
     crit: [ack.m]
     time: 0.00s
!>   Component terminates, but call depth may be super-polynomial:
       ack-ack:[(1,>=,1)]*[4]
     Using the graphs:
       ack-ack:[(1,>,1)]*[2]
       ack-ack:[(1,>,1)]*[3]
       ack-ack:[(1,>=,1)]*[4]

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

binom

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/simple/binom.l
-------------------------------------
     Refining parameters: [binom.n,binom.k]

     --- Analysis of component: ["binom"] ---
     size: ("binom",Lin [] 1)
     crit: [binom.n,binom.k]
     time: 0.01s
!>   Possible duplication of variables: [binom.n,binom.k]
     in the bounding argument positions: [2:binom.n,2:binom.k,3:binom.k,3:binom.n]

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

gcd1

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/simple/gcd1.l
-------------------------------------

     --- Analysis of component: ["gcd"] ---
```

```
      size: ("gcd",Lin ["y","x"] 0)
            ("monus",Lin ["x"] (-1))
            ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [gcd.x,gcd.y,monus.x,monus.y,lgth.x,gt0.x,gt0.y]
      time: 0.01s
      <No failures>

      --- Analysis of component: ["monus"] ---
      size: ("monus",Lin ["x"] (-1))
            ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [monus.x,monus.y,lgth.x,gt0.x,gt0.y]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["lgth"] ---
      size: ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [lgth.x,gt0.x,gt0.y]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["gt0"] ---
      size: ("gt0",Lin [] 0)
      crit: [gt0.x,gt0.y]
      time: 0.00s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

gcd2

```
-----------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/simple/gcd2.l
-----------------------------------------

      --- Analysis of component: ["gcd"] ---
      size: ("gcd",Lin ["x","y"] 0)
            ("monus",Lin ["x"] (-1))
            ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [gcd.x,gcd.y,monus.x,monus.y,lgth.x,gt0.x,gt0.y]
      time: 0.01s
      <No failures>

      --- Analysis of component: ["monus"] ---
      size: ("monus",Lin ["x"] (-1))
            ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [monus.x,monus.y,lgth.x,gt0.x,gt0.y]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["lgth"] ---
      size: ("lgth",Lin [] 0)
            ("gt0",Lin [] 0)
      crit: [lgth.x,gt0.x,gt0.y]
      time: 0.00s
      <No failures>

      --- Analysis of component: ["gt0"] ---
      size: ("gt0",Lin [] 0)
      crit: [gt0.x,gt0.y]
      time: 0.01s
```

```
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.02s
-------------------------------------
```

power

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/simple/power.l
-------------------------------------

     --- Analysis of component: ["power"] ---
     size: ("power",Exp ["n","x"])
           ("mult",Exp ["y","x"])
           ("add0",Exp ["y","x"])
     crit: [power.x,power.n,mult.x,mult.y,add0.x,add0.y]
     time: 0.00s
!>   Non size-linear outer calls: [2:mult]
     due to the bounding argument positions: [2:mult.y]

     --- Analysis of component: ["mult"] ---
     size: ("mult",Exp ["y","x"])
           ("add0",Exp ["y","x"])
     crit: [mult.x,mult.y,add0.x,add0.y]
     time: 0.00s
!>   Non size-linear outer calls: [4:add0]
     due to the bounding argument positions: [4:add0.y]

     --- Analysis of component: ["add0"] ---
     size: ("add0",Exp ["y","x"])
     crit: [add0.x,add0.y]
     time: 0.01s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.01s
-------------------------------------
```

B.9 Glenstrup Results - Sorting

mergesort

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/sorting/mergesort.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["mergesort","splitmerge"] ---
     size: ("mergesort",Lin ["xs"] 0)
           ("splitmerge",Lin ["xs1","xs2","xs"] 0)
           ("merge",Lin ["xs1","xs2"] 0)
     crit: [merge.xs1,merge.xs2]
     time: 0.04s
!>   Component may not terminate
      mergesort-mergesort:[]*[2,5]
      mergesort-mergesort:[]*[2,3,5]
     Using the graphs:
      mergesort-splitmerge:[(1,>=,1)]*[2]
      splitmerge-splitmerge:[(1,>,1),(2,>=,3)]*[3]
      splitmerge-mergesort:[(2,>=,1)]*[5]
      splitmerge-mergesort:[(3,>=,1)]*[6]

     --- Analysis of component: ["merge"] ---
     size: ("merge",Lin ["xs1","xs2"] 0)
```

```
      crit: [merge.xs1,merge.xs2]
      time: 0.01s
      <No failures>

----------Analysis Complete----------
TOTAL Time : 0.05s
-------------------------------------
```

minsort

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/sorting/minsort.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["minsort","appmin"] ---
     size: ("minsort",Exp ["xs"])
           ("appmin",Exp ["xs","rest","min"])
           ("remove",Lin ["xs"] 0)
     crit: [minsort.xs,appmin.min,appmin.rest,appmin.xs,remove.x,remove.xs]
     time: 0.02s
!>   Component may not terminate
      minsort-minsort:[(3,>,1),(3,>,2),(3,>=,3)]*[2,5]
     Using the graphs:
      minsort-appmin:[(1,>,1),(1,>,2),(1,>=,3)]*[2]
      appmin-appmin:[(2,>,1),(2,>,2),(3,>=,3)]*[3]
      appmin-appmin:[(1,>=,1),(2,>,2),(3,>=,3)]*[4]
      appmin-minsort:[(3,>=,1)]*[5]

     --- Analysis of component: ["remove"] ---
     size: ("remove",Lin ["xs"] 0)
     crit: [remove.x,remove.xs]
     time: 0.01s
     <No failures>

----------Analysis Complete----------
TOTAL Time : 0.03s
-------------------------------------
```

quicksort

```
-------------------------------------
Source File: /home/disk17/xeno/ptime/thesis/doc/auto/pgm/t2/sorting/quicksort.l
-------------------------------------
     Non-termination detected.

     --- Analysis of component: ["quicksort","part"] ---
     size: ("quicksort",Exp ["xs"])
           ("part",Exp ["xs1","xs2","xs"])
           ("app",Lin ["ys","xs"] 0)
     crit: [app.xs,app.ys]
     time: 0.10s
!>   Component may not terminate
      quicksort-quicksort:[(3,>,1),(3,>=,2),(3,>=,3)]*[2,7]
      quicksort-quicksort:[]*[2,8]
      quicksort-quicksort:[(3,>,1),(3,>,2),(3,>=,3)]*[2,4,7]
      quicksort-quicksort:[]*[2,3,7]
     Using the graphs:
      quicksort-part:[(1,>,1),(1,>=,2),(1,>=,3)]*[2]
      part-part:[(1,>=,1),(2,>,2),(4,>=,4)]*[3]
      part-part:[(1,>=,1),(2,>,2),(3,>=,3)]*[4]
      part-part:[(1,>=,1),(2,>,2),(3,>=,3),(4,>=,4)]*[5]
      part-quicksort:[(3,>=,1)]*[7]
      part-quicksort:[(4,>=,1)]*[8]

     --- Analysis of component: ["app"] ---
```

```
26        size: ("app",Lin ["ys","xs"] 0)
27        crit: [app.xs,app.ys]
28        time: 0.01s
29        <No failures>
30
31  ----------Analysis Complete----------
32  TOTAL Time : 0.11s
33  ----------------------------------------
```

Bibliography

[1] Michael Berkelaar "LP solve 3.1" Available from `ftp://ftp.es.ele.tue.nl/pub/lp solve`.

[2] Stephen Bellantoni and Stephen Cook "A New Recursion-Theoretic Characterization of the Polytime Functions" Proceedings of the 24th Annual ACM STOC, pp 283-293, 1992

[3] Vuokko H. Caseiro "Equations for Defining Poly-time Functions" Ph.D. thesis, Dept. of informatics, Faculty of Mathematics and Natural Sciences University of Oslo, February 1997

[4] Peter Clote. "Computation Models and Function Algebras" Preliminary condensed version, to appear in the *Handbook of Recursive Theory*, ed. E. Griffor.

[5] Michael Codish and Cohavit Taboch "A Semantic Basis for Termination Analysis of Logic Programming and its Realization using Symbolic Norm Constraints" Algebraic and Logic Programming, ALP '97-HOA '97, LNCS vol. 983 pp 154-171, September 1997

[6] Michael A. Cólon, Henny B. Sipma "Practical Methods for Proving Program Termination" LNCS 2404, *Computer Aided Verification* Prodeedings of the 14th International Conference, CAV 2002, Copenhagen, Denmark, July 2002

[7] Carl C. Frederiksen. "A Simple Implementation of the Size-Change Termination Principle." Unpublished TOPPS report D-442, University of Copenhagen, February 2001.

[8] Arne J. Glenstrup. "Terminator II: Stopping Partial Evaluation of Fully Recursive Programs" Master's Thesis. DIKU, University of Copenhagen. Universitetsparken 1, DK-2100 Copenhagen Ø. June, 1999

[9] Bernd Grobauer "Cost Recurrences for DML Programs" ACM ICFP'01, September 2001

[10] Martin Hofmann "Programming Languages Capturing Complexity Classes" SIGACT News, Logic Column 9, February 2000

[11] Martin Hofmann "Linear Types and Non-size Increasing Polynomial Time Computation" Proceedings of the Symposium on Logic in Computer Science (LICS) 1999.

[12] Neil D. Jones "Program Analysis for Implicit Computational Complexity" Invited talk at PADO 2001, Abstract in Proceedings of LNCS 2053: *Programs as Data Objects*, Second Symposium, PADO 2001, Aarhus, Denmark, May 2001

[13] Neil D. Jones, Carl C. Frederiksen "Program Analysis for Implicit Computational Complexity" Invited talk at ICC'02, Abstract in Proceddings of the 3rd International Workshop on *Implicit Computational Complexity*, Copenhagen, Denmark, July 2002.

[14] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram "The Size-Change Principle for Program Termination" POPL 2001: Proceedings 28^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2001.

[15] Chin Soon Lee "Program Termination Analysis and the Termination of Offline Partial Evaluation" Ph.D. thesis, University of Western Australia, March 2001

[16] Naomi Lindenstrauss and Yehoshua Sagiv "Automatic Termination Analysis of Prolog Programs" Proceedings of the 14th International Conference on Logic Programming pp 64-77, July 1997

[17] Yanhong A. Liu, Scott D. Stoller. "Dynamic Programming via Static Incrementalization" Computer Science Department, Indiana University, Bloomington, ESOP 2000.

[18] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. "Principles of Program Analysis" Preliminary copy, Dagstuhl 1998.

[19] Mads Rosendahl "Automatic Program Analysis" M.Sc. thesis, Dept. of Computer Science, University of Copenhagen, November 1986

[20] Mads Rosendahl "Automatic Complexity Analysis" ACM FPCA (ICFP) 1989

[21] Helmut Schwichtenberg and Klaus Aehlig "A Syntactical Analysis of Non-size Increasing Polynomial Time Computation" ACM Transactions on Computational Logic, July 2002

[22] Jens Peter Secher "Perfect Supercompilation" DIKU-TR-99/1. Department of Computer Science (DIKU), University of Copenhagen. February, 1999.

[23] Rene Thiemann Work in progress on improving the closure algorithm for SCT (Student at RWTH-Achen)

[24] David Wahlstedt. "Detecting termination using size-change in parameter values" M.Sc. thesis, Chalmers University of Technology, 2000. `http://www.cs.chalmers.se/ davidw/`

[25] Ben Wegbreit "Mechanical Program Analysis" Communications of the ACM, Vol 18, Number 9, pp 528-539, September 1975

[26] Ben Wegbreit "Goal-Directed Program Transformation" IEEE Transactions of Software Engeneering, vol. SE-2, No. 2, pp 69-80, June 1976.

[27] Hongwei Xi "Dependent Types in Practical Programming" Ph.D. Thesis, Carnegie Mellon University, 1998

[28] Hongwei Xi "Dependent Typed Data Structures" In C. Okasaki, ed., Proceedings of WAAAPL'99 pp 17-32, September 1999