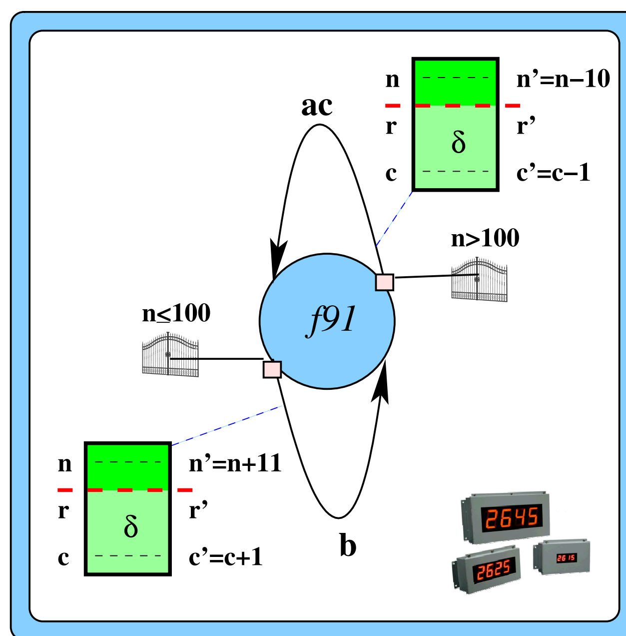
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

EXPLORING LINEAR SIZE-CHANGE TERMINATING PROGRAMS



HUGH ANDERSON

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2007

Statement

I, Hugh Anderson, declare that this thesis is my own work and that, to the best of my knowledge, it contains no material substantially overlapping with material submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Hugh Anderson

Acknowledgments

I would principally like to thank Dr Khoo Siau-Cheng for his suggestions and continual encouragement throughout the development of this thesis, and of course those times that he paid for lunch! I found that the challenges he set for me from time to time helped me immeasurably. Thanks Siau-Cheng, for all the things you have taught me, for being positive, and for always finding answers when we were stuck!

Thanks also to Lindsay Groves, Thiagu, KwangKeun and Wei-Ngan for their comments. Martin, Roland, Stefan and Jin-Song provided spirited lunch-time solace. During their visit(s) to NUS, Gill Dobbie, Gabriel Ciobanu, Luke Ong, John McCallum, Krzysztof Apt, Bruce Brown and Dines Bjørner challenged me to get finished, and also provided numerous off-the-cuff suggestions that were invaluable. I'd like to thank my co-authors, Stefan, Beatrice, Siau-Cheng and Gabriel; my colleagues and friends at NUS: Bim and Atreyi, Joxan, Tiow Seng, Min Yen, Thiagu, Sandeep, Eng Wee, Wei-Ngan, Theresa, Colin, Martin S, Frank, Sun Teck, Razvan, Siau-Cheng's lunch group (Beatrice, David, Ping, Meng...), the PL&S lab and the team at Spinellis. Neil Jones initially proposed the form of the time-cost problem [Jon03], and I should not forget my dear students, and Rhys, Tim and Clive.

Finally, my thanks to the National University of Singapore for sponsoring my study, and for employing me for the period of my candidature.

Dedication

To my dear family, Rebecca, Boyd, Ford and Judge. Rebecca and I have enjoyed each other's company for 36 years or so, and now we have three sons to amuse us. The last few years have had their ups and downs, but throughout this time Rebecca and the kids have supported my activities, even when it meant Dad wasn't home for dinner.

Contents

Statement i

Acknowledgements ii

Dedication iii

Abstract viii

I Prologue 1

1 Introduction 2

1.1 Motivation 2

1.2 Termination, and other difficult problems 4

1.3 The road followed in this thesis 6

2 Termination analysis and related work 7

2.1 Program analysis 7

2.2 Preliminary notions 8

2.2.1 The notion of termination 8

2.2.2 Functional versus other programming styles 9

2.2.3 Decidable theories . . . 9
2.3 General termination analysis . . . 13
2.3.1 Traditional imperative termination analysis . . . 15
2.3.2 Termination and term-rewrite systems . . . 16
2.3.3 Termination and logic programming . . . 17
2.3.4 Termination and other programming styles . . . 18
2.4 Size-change termination analysis . . . 20
2.5 Type inference for generation of SCGs . . . 26
2.5.1 Type systems and analysis . . . 27
2.6 Type directed implementation for size-change graphs . . . 28
2.6.1 Notation and syntax for typing . . . 29
2.6.2 Type inference rules for size-change graphs . . . 30
2.6.3 SCT termination analysis using size-type inference . . . 32
2.7 Preliminary definitions . . . 35
2.7.1 The language . . . 35
2.7.2 Affine relations and Presburger formulæ . . . 36
2.8 Commentary . . . 38

II Body 39

3 Affine-based analysis 40
3.1 Affine-based size-change termination . . . 41
3.2 Affine-based termination analysis . . . 46
3.2.1 Associating affine with abstract graphs . . . 47
3.2.2 Affine-based closure algorithm . . . 49
3.2.3 A sample widening operator . . . 51

3.3 Properties of affine-based termination analysis 53
3.3.1 Termination of the analysis algorithm 53
3.3.2 Accuracy of the analysis algorithm 54
3.3.3 Correctness of the analysis algorithm 54
3.4 Examples using affine termination analysis 56
3.4.1 First example - constant size cancellation 56
3.4.2 Second example - linear combinations of source parameters . . 58
3.4.3 Third example - context analysis 59
3.5 δSCT (Delta size-change termination) 60
3.6 Commentary 61

4 Extending affine size-change termination 62

4.1 Introduction 62
4.2 Extended size-change graphs 63
4.2.1 Capturing function return values: Value SCGs 64
4.3 Refining the graph composition algorithm 65
4.3.1 Deriving an approximation to the CFG for traces 67
4.3.2 Adding counters to restrict graph composition 69
4.4 From well-founded types to boundedness 71
4.4.1 Bounded termination 73
4.5 Summary of changes to affine-SCT 76
4.6 Termination analysis examples 77
4.6.1 Termination for $f91$ 77
4.6.2 Termination of Ackermann's function 79
4.6.3 Termination for function $\mathcal{X}$ 80
4.6.4 Functions which cannot be analyzed 81
4.7 Related work 81
4.7.1 Avery's SCT 81
4.7.2 Cook's Terminator 83
4.8 Commentary 84

5 Time and stack costs for SCT programs **85**

5.1 Introduction 85

5.1.1 Preliminaries 87

5.2 Runtime analysis 88

5.2.1 Characterization as a quantifier-elimination problem 90

5.2.2 Quantifier elimination 91

5.2.3 Tool support 94

5.3 Calculating other program costs 95

5.3.1 Stack depth calculation 95

5.3.2 Relative runtime costs 96

5.4 Exponential program costs 97

5.4.1 Explanation of the form of the relation 98

5.4.2 Using the relation for exponential runtimes 101

5.4.3 Exponential examples 102

5.5 Commentary 104

III Conclusion **105**

6 Concluding remarks **106**

6.1 Research directions 106

6.1.1 Improving the algorithms 107

6.1.2 Other application areas 108

6.1.3 Extending and changing the formalisms 108

6.2 Summary of contribution 109

Bibliography **111**

Abstract

This thesis explores an area of research into the analysis of a particular set of programs. The set of programs is known as the "linear size-change terminating" programs, a significant subset of all programs, whose termination can be decided with an automated technique: size-change termination analysis. This analysis technique begins by generating size-change graphs from program sources, which are then analyzed using graphical analysis techniques.

The work outlines progressive research developments, firstly replacing the traditional size-change graphs used in size-change termination analysis, with affine-based graphs, in which affine relations among parameters are expressed by Presburger formulæ. In this approach, the data-types of the parameters that control the termination of a program must be well-founded, and (eventually) reducing. The termination analysis is extended in various ways to include not just reducing well-founded parameters, but those which are bounded in some domain. This development is termed "Bounded termination", and extends the set of size-change terminating programs. Finally, the polynomial and exponential runtime and stack depth costs of a subset of the programs can be precisely calculated.

The thesis concludes with an outline of some possibilities of future research directions.

List of Tables

2.1 The simplest language syntax 29

2.2 Inference rules for simple variables 30

2.3 Inference rules for simple functions 30

2.4 Inference rules for function application 31

2.5 Inference rules for the rest of the language 31

2.6 The language syntax, extended with operators 36

2.7 Syntax of Presburger formulæ 36

3.1 Abstract graphs for the example program 48

3.2 Definition of $E(r_i)$ 52

3.3 Definition of $K(r'_j)$ 52

4.1 CFG generation rules for single function definition 67

4.2 Extra counters derived from the CFG $P \rightarrow a \mid bXcYdP \mid eZfQ$ 70

5.1 Extended language syntax 87

5.2 Inference rules for Tarski formulæ 93

List of Figures

2.1 LJB size-change graph for function f calling g 21

2.2 LJB size-change graphs 22

2.3 Control-flow graph for functions 22

2.4 LJB size-change graphs for functions $f \circ_G g$ and $g \circ_G f$ 23

2.5 Ordering of the operators for LJB graph composition 23

2.6 LJB size-change graphs for function c 25

2.7 Overview of LJB size-change termination analysis 26

2.8 Simple SCG for Ackermann function call 29

2.9 Final SCGs for example programs 33

3.1 Overview of affine-SCT analysis 42

3.2 Ordering of the operators for Abstract graph composition 47

3.3 Relationships between f, g, F_f and F_g 55

4.1 The *value* and *call* SCGs for the $f91$ function 64

4.2 Composition of nested call 64

4.3 A hierarchy of program traces 66

4.4 Counters for extended SCGs of the $f91$ function 70

4.5 Overview of changes to affine SCT for bounded termination 72

4.6 Flow graph for sample program, with size-change graphs 82

5.1 Trajectories of guard variables with time 99

Part I

Prologue

Chapter 1

Introduction

This thesis outlines a particular area of research into the use of arithmetic techniques for the analysis of programs, for the purpose of discovering termination, runtime and stack depth properties. There are many avenues in the general area of program analysis, and in this thesis, related work in this area is outlined, and then various specific new avenues are developed, including:

1. A translation of the size-change graphs used in *size-change* program termination analysis, to affine-based graphs, in which affine relations among parameters are expressed by Presburger formulæ.
2. Extensions of the affine-based termination analysis in various ways including one using *bounds*. These extensions allow for the parameters to be integers (rather than just naturals), and to increase or decrease towards some explicit or implicit bound.
3. The use of affine-based size-change graphs to derive an explicit program runtime measure.

In all, these developments provide useful and practical extensions to existing work on termination and runtime analysis, and have been presented at various conferences over the last few years, listed in the summary at the end of this section.

1.1 Motivation

At one stage of my life, I spent a lot of time consulting and developing software projects for embedded and distributed systems. From this experience I developed

a personal, rather cynical, view that the software industry was in a state of crisis, with software continuing to be produced that is buggy, and prone to failure. There is nothing new in this, and industry pundits have been harping on this theme for many years.

Computer *hardware,* by contrast, is unlikely to fail in the same way, provided it is used within its design constraints. In general when computer hardware fails, it fails completely and one is left in no doubt that the hardware needs replacement. Erratic faults are unlikely to be caused by hardware failures.

However, again my personal view, the way forward to improve software is for us to adopt *formal* methods to ensure that our software works correctly. My thesis coincides strongly with this personal view, and in some sense is my attempt to try to develop mathematical tools for software which leave us in no doubt about the correctness or otherwise of software and systems. Let me expand on this.

The complexity of computer software, and the undesired interactions between programs and operating systems continue to give rise to catastrophic software failures. There is no known silver bullet [Bro95] for software developers, but formal methods (i.e. *mathematical* methods) have been effective in the development, analysis and testing of software.

It is by now well established that techniques such as testing do not yield the required silver bullets. A famous (36 year old) quote from Edsger Dijkstra that is often mentioned in this context is: "*Program testing can be used to show the presence of bugs, but never to show their absence*" [Dij70]. In keeping with other engineering disciplines, the adoption of mathematical approaches to establishing or ensuring the properties of programs is a sensible approach, yielding greater assurance. If a software product's adherence to a specification is at the level of a formal mathematical *proof*, then this may be the best level of assurance that may be achieved. In a contentious paper [MLP79], DeMillo, Lipton and Perlis argue that most proofs are just *outlines* of proofs crafted to persuade others of the correctness of an assertion, and that the process of proof is a social process as much as a formal process. However, my view is that *something* is better than *nothing*.

Consider the following *formal-methods* approaches:

Refinement - The refinement calculus elaborated by Morgan in [Mor94] and Back and von Wright in [BvW98] provides a formal way of deriving code directly from specifications using mathematically justified transformation rules. The resultant code is guaranteed to match the original specification: correctness by construction. In the refinement calculus approach, specifications and programs are considered equivalent, a program just being a more concrete version of a specification.

Analysis - An alternative approach is to attempt to confirm properties of existing code by analysis of the program using representations of the semantics of the code components. In this approach, the meaning of the program is represented in an

abstract manner, and mathematical techniques are used to derive properties of the code. These properties may be considered to be partial specifications of the code.

Exhaustive Testing - If the testing of a program is exhaustive, then we can raise our assurance to the level of a proof. However, in general, it is not possible to exhaustively check any but the most trivial programs. However a suitably abstract model may be exhaustively tested for a particular property of interest, and model-checking techniques [McM93, CGP99] have had success in modeling protocols, electronic circuits, and parallel programs.

Given this classification, the research strategy used here lies in the middle approach, using mathematical techniques on representations of the program. The focus is in a specific area of this kind of program *analysis*: termination, runtime and stack depth cost calculations.

1.2 Termination, and other difficult problems

The focus of this thesis is on certain *difficult* problems found in computer science. One such problem is that of the *termination* of a program or function. In the obvious initial scenario, a system implementor or designer may need assurance that a function or group of functions will always finish running. If it does not, and perhaps endlessly loops, this may cause a system to fail. In reactive systems, we may have a main function which is intended to run forever, but would still have a requirement that the functions it calls terminate.

However, termination appears in other contexts as well. For example, in RT (Real-Time) systems, the stability of the whole system may be dependant on the *timely* completion of a response to an impulse. In [AAMK06], ideas of termination and runtime analysis are applied to a-priori verification of the stability of a PID (Proportional, Integral, Derivative) controller, implemented in an RT system as a succession of timed interrupt response routines. Stability for PID systems is traditionally calculated using Matlab, but with the assumption that all the components of the system are analog devices. In a computer driven PID controller, responses and measurements are discrete approximations, taken at specific times. The reason that this particular analysis is interesting is that it copes with the discrete nature of the computer driven PID system, rather than approximating it as in Matlab.

Another context for termination analysis is found in the termination of program *generation*. For example, in [GJ05], the authors outline the use of termination analysis in program specialization. Given a program P, and a partition of the known values of the input data s, a new program P_s is generated. This program is a specialized version of the original program, which efficiently generates the expected results for the known values. If the input data does not belong to the set s, the program will revert to the non-specialized code. Program generators of this form have been successful in generating far more efficient code than simple generators that just follow

the original source program structure. A vital point about these generators is that it is imperative to ensure that the program specialization process itself terminates.

The general problem of testing a program for termination is a difficult one, and there can be no program that *decides* if any arbitrary program terminates. Such a program is called a decider, and no decider can be developed for testing if an arbitrary program terminates. This is of course one of the founding results from computer science, Alan Turing's undecidability result [Tur37].

However, a semi-decider can be written for problems such as the termination problem. A semi-decider will always terminate (sic), and can return either YES - the program terminates, or NO - the program *may* (or may *not*) terminate. Such a semi-decider is useful if it returns a decision quickly.

There are many approaches to discovering if a program terminates, discussed elsewhere in this thesis. However, this thesis starts with size-change termination (SCT), the size-change termination technique [LJBA01] which requires each (possibly infinite) cyclic chain of function calls to have a chain of parameter changes in which each parameter either reduces in size or stays the same, and in which the total chain must reduce. In addition, the parameter values must be drawn from well-founded sets. The size-change termination technique has been shown to be PSPACE-complete.

The thesis begins by introducing the idea of expressing size-change termination in terms of affine relations between arbitrary combinations of inter-procedural parameters, rather than the traditional expression of size-change termination which uses explicit size comparison (less-than or less-than-or-equal) between specific inter-procedural parameters. This idea also provides a way of including in the analysis other aspects of the program, for example the conditional expressions in an if-then-else.

In bounded-termination, the affine SCT is extended to include arbitrary bounds expressed as linear constraints, a sufficient condition is established which ensures that a cyclic chain of function calls monotonically moves the parameters towards some bound which terminates the chain of calls. In short parameters may be integers (not well founded), and size changes might increase or reduce.

Turing's result for termination is extended by Rice's theorem [Ric53], which asserts that *any* (non-trivial) assertion about arbitrary functions is undecidable. For example the runtime of a function is a non-trivial assertion and hence undecidable, and so obviously the calculation of program runtimes is *another* difficult problem.

A subset of the set of terminating programs (the linear size-change terminating ones) may have runtimes that can be calculated automatically with a decision procedure. In particular, if the runtime is polynomial in the program parameters, or if it has a certain exponential form, then the runtime and stack depth may be calculated automatically.

1.3 The road followed in this thesis

In summary, this thesis begins by re-casting size-change termination in terms of *affine* graphs. These graphs allow for more refined analysis, including the effects of the conditionals in programs. The thesis continues with improvements to the approach including size-change termination over integers, termed *bounded* size-change termination. Having developed the affine-SCT framework sufficiently, the thesis shows an approach to calculating the runtime or stack depth of bounded size-change terminating programs.

There are three parts to the thesis. Part I (Prologue) introduces the work, and in Chapter 2 continues with related and foundational work, along with some of the notation used in the later analysis. Section 2.2 introduces preliminary notions of termination and decidable theories used throughout the thesis. Section 2.3 surveys various methods of automatic termination analysis. Section 2.4 describes in detail the size-change termination analysis technique. Section 2.5 shows how type inference can be used to derive size-change information, and finally Section 2.7 defines the language used for programs in the thesis, and Presburger formulæ, and gives preliminary definitions of operations over affine relations useful for calculation using affine-based graphs.

Part II (Body) has three chapters, describing in detail the contribution of this thesis. In Chapter 3 a translation of size-change graphs to affine-based graphs useful for program termination analysis is introduced along with the algorithm for affine termination analysis. This section is largely based on work done in 2003, and reported in [AK03a] and [AK03b], but extended here for clarification. Chapter 4 extends this approach, allowing the technique to be used on non well-founded datatypes, and to allow an argument that variables can approach some arbitrary bound, rather than the zero bound used in well-founded size-change termination. This section is reported on in [KA05], but again clarified and extended in this thesis. Chapter 5 discusses time and stack cost calculation. This work is published in [AKAL05, AKL07].

Finally Part III (Conclusion) summarizes the contribution of the research and the proposed continuation direction.

Chapter 2

Termination analysis and related work

Here are brief outlines of foundational work directly related to this thesis. In particular the use of program analysis, decidable theories, and type inference to derive arithmetic and size-change information useful in the analysis of program properties, will each be introduced. Various techniques for program termination analysis are outlined, in particular the size-change termination technique of [LJBA01]. The notation and concepts common to size-change termination, and the affine based analysis are progressively introduced while explaining this related work.

2.1 Program analysis

Programs are analyzed for different purposes. Sometimes the analysis is performed to confirm some properties that our program *must* possess. Programs may also be analyzed to allow optimizations of the program (for example, by moving computations out of loops). However, even though the purpose of the analysis may vary, the method of analysis in general follows a two-step process. Firstly, the programs are used to derive an abstract view of the program amenable to (mathematical) analysis. Secondly, the abstract view is analyzed using some algorithm that is guaranteed to return an answer.

There are many forms of program analysis, with examples including Hoare reasoning [Hoa69], in which axioms and inference rules are applied to program sources augmented with pre and postconditions of the program statements, and abstract interpretation introduced by Cousot and Cousot in [CC77] (see also [CH78]), in which an iterative fixed point computation can be used to derive variable interdependencies. The book [NNH99] gives a good overview of program analysis. In this book, Nielson et. al. classify the field into four main analysis approaches: data-flow, constraint-based, abstract interpretation, and type and effect systems.

In constraint-based program analysis, the source of the program is used to derive sets of constraints for a property of interest. The constraints may then be solved by an efficient tool. Heintze and Aiken explored set-based analysis in [Hei92] and [Aik99], where program variables are viewed as sets of values. A calculus of set constraints is used to carry out an analysis. The general style of program analysis developed here follows in the same vein, but with deviations in some areas. For example, constraints are often simple inequalities or equalities. This is so that the solutions of the constraints are decidable. In [RR00], Rugina and Rinard use more general symbolic polynomial constraint systems in performing memory and array bounds analysis. In this thesis, more general symbolic polynomial constraint systems are considered, although the application area is runtime and stack-depth analysis.

The close associations of new languages and their type systems have clear benefits, including early detection of program errors through static type checking, the support of large-scale software construction through interface type checking, and assurance or runtime safety through type checking. Pierce's book [Pie02] surveys the relation between types and programming languages, providing a basis for the type and effect systems used for program analysis. Type systems are formalized through axioms and inference rules, and may be used to generate conservative type information about a program.

Dependent type systems [DP97] are a class of type systems in which the types depend on the values of the terms. A subclass of dependent type systems are the size-type systems, in which the types depend on some expression of the size of a term. In [HPS96], Hughes, Pareto and Sabry introduced the use of sized types for the correctness of reactive systems. In [CK00], Chin and Khoo use size-type inference systems and efficient calculation of constraints over programs.

2.2 Preliminary notions

In this section, some preliminary ideas are introduced. The first explains the notion of termination used throughout the thesis. The second explains the use of the functional style throughout the rest of the thesis. Finally, the particular utility of decidable theories in program analysis is explained, introducing the two main theories used in the thesis.

2.2.1 The notion of termination

The term *termination* arises often in algorithm and program analysis. It is associated with assurance that a given program or part of a program will not just keep on running forever. If it were to do so, other important activities may not occur, or they may be delayed, by the processor wasting execution cycles on this faulty program.

There is a small difference between the notion of termination employed in this thesis, and a more standard notion of termination. In the context introduced here, there is no differentiation between the termination of a program, and error conditions. For example, consider a program over *natural* numbers which reduces a value for ever like this:

```
f(x) = if true then
           f(x − 1);
```

In the context of traditional SCT, this is considered a *terminating* program. The program cannot run forever, as eventually the (*natural*) number x will be reduced to zero, and the operation to subtract 1 from zero is not defined over the naturals. The runtime support for the program is presumed to generate an exception, and the program will terminate. As can be seen, in this sense there is no differentiation between a program which terminates normally, and another program which terminates through an exception/error condition.

2.2.2 Functional versus other programming styles

The functional style adopted in the thesis can easily accommodate imperative and other styles of programming, and it is just a matter of personal choice that the functional style is used here. Other authors have recast the number-of-function-calls used for termination of simple functional languages in other programming styles.

For example, in [Ave06] SCT is discussed using a C-like language with for loops. In [TG05] SCT is applied to the termination of term-rewrite systems.

2.2.3 Decidable theories

Decidable arithmetic theories form a useful tool for reasoning about programs. In this thesis, "Presburger first-order theory" is used, and "quantifier elimination in the theory of real closed fields". Both of these are decidable, and projected future work is to use the conjectured decidability of exponential real closed fields [Wei00].

Presburger integer arithmetic

Presburger integer arithmetic (PIA), first described in [Pre27], but reprinted in English in [Pre91], is a particular axiomatization of a first order theory of arithmetic over integers in which addition is the only operation. In this arithmetic, the elements are the constant 0, integer variables, binary addition $+$, the relations $<, \leq, =, \geq, >$

and the quantifiers and connectives of first-order predicate calculus. A formula such as $x \geq 3y$ is allowed, because this is $x \geq y+y+y$, but it is not possible to write $x \geq yz$. A sentence in the Presburger theory might be something like $\forall x \exists y : x \geq 3y$. Within the theory, it is possible to formulate all sorts of useful assertions about a program. For example it is possible to assert that an array index variable i is always less than the size of the array. However, it is not possible to formulate notions (say) of primality or divisibility.

Presburger arithmetic is consistent (i.e. all provable statements in Presburger arithmetic are true), and complete (i.e. all true statements in Presburger arithmetic are provable from the axioms). However, in the present context the most important property of the theory is that it is decidable (i.e. there is an algorithm which decides if any given statement in Presburger arithmetic is true or false). The decidability of Presburger arithmetic is shown using successive quantifier elimination.

When Presburger arithmetic is restricted to the non-negative integers, it is termed Presburger natural arithmetic (PNA). PNA is in some sense more fundamental than PIA, as any integer variable x may be represented using the difference between two natural variables $x_a - x_b$. In addition there is a Presburger rational arithmetic (PRA) defined over the rationals, which is also decidable.

Any decision algorithm for Presburger arithmetic has a worst-case runtime of at least $2^{2^{kn}}$ [FR74], where n is the length of a sentence, and k is some constant greater than 0. Note that it requires more than polynomial or even exponential run time. The best known upper bound on such an algorithm is $2^{2^{2^n}}$ [Opp78].

However, in practice, these super-exponential runtimes rarely occur, and in [NO78] an expression simplifier is shown which uses the simplex algorithm on a variant of Presburger arithmetic, with exponential worst-case run time for specially constructed problems, but a much better average run time. Such simplex-based approaches are practical in most cases.

The Omega library [KMP$^+$96] is an example of an efficient implementation of C++ classes to manipulate sets of numbers and tuple relations, with constraints that can be described by Presburger (PIA) formulæ. For example, the following session shows how the Omega Calculator may be used to define two sets, S (the set of integers from 0 to 9) and T (the set of integers from 1 to 9) and then show how the two sets are related, specifically that $T \subseteq S$, but not vice-versa.

```
[hugh@pnp176-44 hugh]$ oc
# Omega Calculator v1.2
S := { [x] : x>=0 && x<10 };
T := { [y] : y>1 && y<10 };
S subset T;
False
T subset S;
True
```

The library can also be used to define and manipulate *tuple* relations. In the following example, two simple relations are defined, and then we can take the composition of the two relations.

```
[hugh@pnp176-44 hugh]$ oc
# Omega Calculator v1.2
F := { [m] -> [m']: m'=m+1 };
G := { [n] -> [n']: n'=n-2 };
F.G;
{[m] -> [m-1] }
```

In the example, F represents the set of pairs $\{(m,m') \mid m' = m+1\}$ and G represents the set of pairs $\{(n,n') \mid n' = n-2\}$. The composition of the two relations is the set $\{(m,m') \mid m' = m-1\}$. Note that in the Omega Calculator, the variables and expressions range over the integers Z.

Apart from standard Presburger simplification, the Omega library also includes efficient implementations of various other high-level operations, such as ones to discover upper and lower bounds on inexact relations, and provide various forms of convex hulls approximating a relation.

The authors of the Omega library state that they have no reason to believe that their method will provide better worst-case performance than $2^{2^{2^n}}$, but that the method may be more efficient in many simple cases that occur in their applications [KMP+96].

QE in the First-order Theory of Real Closed Fields

A real closed field is an ordered field F in which every non-negative element of F has a square root in F, and any polynomial of odd degree with coefficients in F has at least one root in F.

Tarski's theorem [Tar51] tells us that the theory of real closed fields, provides for elimination of quantifiers, and hence it is a complete and decidable theory. What this means is that given a formula with quantified variables, there is a decision procedure which will give an equivalent solution formula from the original one, with no quantified variables. For example, we can remove the quantifier x from $\exists x : ax^2+bx+c = 0$, giving the solution $a \neq 0 \wedge 4ac - b^2 \leq 0$.

Unfortunately, quantifier elimination has high complexity. Tarski's original strategy for eliminating quantifiers is impractical for large problems, as the complexity is high: no tower $2^{2^{2^{\cdots^n}}}$ can bound the execution time of the algorithm if n is the size of the problem. The problem in general is doubly exponential [DH88] in the number of quantifiers. In 1973, Collins discovered a more efficient algorithm for quantifier elimination [Col75].

The algorithm is called CAD (Cylindrical Algebraic Decomposition), and the maximum computing time is of the order of $(mn)^{k^r}d^k$ where k is some constant, r is the number of variables, m is the number of polynomials, n is the maximum degree of the polynomials, and d is the maximum length of any integer coefficient of any such polynomial. If r is fixed, then the time is dominated by a polynomial in m, n and d. This may still seem to have a high complexity, but CAD has been implemented and shown to be capable of solving non-trivial problems. Many practical problems can be cast as problems in quantifier elimination, and in this thesis, we characterize a problem in termination and runtime as a QE problem, and exploit CAD software systems to discover solutions. A computer software system for performing quantifier elimination automatically is the `redlog` package [DS97] which can be added to the computer algebra system `reduce`.

Here is an example of QE using the `redlog` system on $\exists x : ax^2 + bx + c = 0$:

```
[hugh@pnp176-44 hugh]$ /usr/local/reduce/reduce
load_package redlog;
rlset OFSF;
ex1    := rlqea ex({x}, ( a*x^2+b*x+c=0 )  );

ex1 := {{a = 0 and b = 0 and c = 0,{x = infinity1}},
                  2
        {4*a*c - b  <= 0 and a <> 0,
                                  2
               - sqrt( - 4*a*c + b ) - b
        {x = ----------------------------}},
                         2*a
                  2
        {4*a*c - b  <= 0 and a <> 0,
                               2
              sqrt( - 4*a*c + b ) - b
        {x = -------------------------}},
                        2*a
                               - c
        {a = 0 and b <> 0,{x = ------}}}
                                  b
```

After initializing the system, the `rlqea` function is used on the redlog representation of $\exists x : ax^2 + bx + c = 0$:

```
ex1    := rlqea ex({x}, ( a*x^2+b*x+c=0 )  );
```

The `rlqea` function performs quantifier elimination on the formula, providing a possible answer. The resultant value is stored in the `ex1` variable. The output shows

that either $a = b = c = 0$ in which case x can have any value, or that $a \neq 0$ and $4ac - b^2 \leq 0$ and $x = \frac{\pm\sqrt{b^2-4ac}-b}{2a}$, or finally that $a = 0 \neq b$ and $x = \frac{-c}{b}$.

In this thesis, quantifier elimination is used to derive program runtime measures. However other techniques such as the Gröbner basis method can be used to discover similar program invariants [SSM04]. In [Kap04], the author compares a theory of Presburger arithmetic, a theory involving parametric Gröbner bases, and Tarski's theory of real closed fields, and claims that the last theory is the most expressive, and generates stronger invariants than the Gröbner basis approach.

On reflection, effective cost calculations may result from a more detailed investigation of the use of the Gröbner basis method in our application domain, using its computational efficiencies to outweigh the lack of expressiveness.

Real exponential fields?

The decidability result for the real closed fields, and the success of the practical application of CAD, give rise to the following question, posed by Weispfenning: "Can this result be extended to a language including in addition the real exponential function e^x?". There is a conditionally positive answer to this [Wei00].

Of more interest though is that by restricting the form of the exponential expressions, there is a decidability result that is still unconditional [Wei99]. In this restricted form there are complex expressions containing e^x, and linear relations between the variables. Weispfenning has shown a suitable algorithm, with an implementation in `redlog` [Wei00].

The reason that this is of interest in this thesis, is that many programs/algorithms have exponential runtimes that may be expressed in this form.

2.3 General termination analysis

The analysis of programs for termination is of particular interest to computer scientists. We are all familiar with the meta-equation

totalCorrectness = partialCorrectness + Termination

which reminds us of the special place held by termination analysis, clearly partitioning a program correctness argument into two camps. Floyd and Hoare provided two of the seminal papers [Flo67, Hoa69] formalizing program language theory.

Floyd points out the role of *termination* when *assigning meaning to programs*:

> ... and in particular, if the program ever halts, the proposition on the path leading to the selected exit will be true. Thus, we have a basis for proofs of relations between input and output in a program. The attentive reader, however, will have observed that we have not proved that an exit will ever be reached; ... [Flo67]

Later, Floyd points out that

> Because no infinite decreasing sequence is possible in a well-ordered set, the program must sooner or later terminate. Thus we prove termination, a global property of a flowchart, by local arguments, just as we prove the correctness of an algorithm. [Flo67]

Floyd's approach is to associate the values of program variables with elements in a well founded set, with execution of the program resulting in inevitable reduction of the related elements in the set. This approach can also be applied to multisets, with multiset orderings [DM79].

In [Hoa69], Hoare also explored logical foundations of computer programs, and clearly separated these two concerns (although he seemed to prefer the term *conditional* correctness, rather than *partial* correctness). He clearly separates the termination of the program from the correctness:

> Another deficiency in the rules and axioms quoted above is that they give no basis for a proof that a program successfully terminates.
> ... Thus the notation "$P\{Q\}R$" should be interpreted "provided that the program successfully terminates, the properties of its results are described by R." [Hoa69]

In "The Science of Programming", Gries develops a formal approach to the development of programs, and mentions the termination problem in the context of constructing loops such that they terminate:

> **Strategy for developing a loop:** Develop guarded commands, creating each command so that it makes progress towards termination and creating the corresponding guard to ensure that the invariant is maintained. The process of developing guarded commands is finished when enough of them have been developed ... [Gri81]

In Gries work, the (slightly different) notation $\{Q\}S\{R\}$ is used, where the meaning is that the execution of S begun in any state satisfying the predicate Q terminates in a state satisfying R. An equivalent notion is given by the notation $Q \Rightarrow \text{wp}(S, R)$. The term $\text{wp}(S, R)$ is the **w**eakest **p**recondition of S with respect to R, and represents the set of all states such that execution begun in any one will terminate with R true.

There are many approaches to termination analysis, and it has been studied in a range of programming arenas. Each of these approaches have common features, but exploit unique qualities of the particular programming style. In the remainder of this section, brief introductions are given to these approaches to termination analysis. In each of the arenas, the terminology changes and notations are different, but in general, the underlying idea is to automatically find some relation between the program and a well-founded ordering such that execution of the program leads to reduction through the ordering.

2.3.1 Traditional imperative termination analysis

Traditionally, there are two concepts of termination, termed *strong* and *weak* termination [Dij82]. A strongly terminating system must terminate within some finite amount of time. A weakly terminating system must terminate, but there may be no upper bound on the time it will take.

In [Gri79], Gries succinctly formalizes two methods of performing strong and weak termination analysis using the predicate transformer (weakest-precondition) semantics as follows:

1. **Strong Termination.** Derive an integer function $t(\tilde{x})$ of the program variables $\tilde{x}$, show that $t \geq 0$ whenever the loop is still executing, and show that each execution of the loop body decreases t by at least 1. For a loop do $B \rightarrow S$ od with invariant P this means proving that

   $$(P \wedge B) \Rightarrow t \geq 0 \text{ AND } \{P \wedge B\}\, T := t;\;\; S\,\{t \leq T - 1\}$$

   where T is a new variable.

2. **Weak Termination.** Choose a "well-founded" set $(W, \succ)$, i.e., $\succ$ is a partial ordering with the property that for any w in W there is no infinite chain $w \succ w_1 \succ w_2 \succ \ldots$. Then choose a function $f(\tilde{x})$ of the program variables $\tilde{x}$, and prove that

   $$\{P \wedge B\}\; \tilde{w} := \tilde{x};\;\; S\,\{f(\tilde{w}) \succ f(\tilde{x})\}$$

   where $\tilde{w}$ is a new set of variables.

These methods are not particularly amenable to *automatic* analysis, as in each case a function has to be derived, appropriate for the program being analysed ("Derive an integer function...", "Choose a well founded set..."). The methods are however appropriate for (human) guided proofs of termination.

2.3.2 Termination and term-rewrite systems

Term rewrite systems [BN98] have a clear relationship to equational logic systems. In an equational system there are equality rules ($l = r$) and in an expression, it is possible to always replace a left-hand side of a rule (l) with a right hand side (r) and vice-versa. By contrast, in a term rewrite system, rules can only be applied one way ($l \rightarrow r$). An (l) can be replaced with a (r) in an expression, but not vice-versa. The term redex (reducible expression) is used to refer to an instance of the left-hand side of a rule.

This mechanism of rewriting in a directed fashion forms a basic computational model, amenable to analysis. Strong results (perhaps confluence/termination) from term-rewrite systems can be applied to programming languages. For example, it can be shown that a certain class of TRS's always terminates, then this may lead to a certain subset of functional or logic programs also always terminating.

Termination for term rewrite systems can be shown by considering two rewrite rules for numbers:

$$x + 0 \rightarrow x \tag{2.1}$$

$$x + s(y) \rightarrow s(x + y) \tag{2.2}$$

then it would be possible to have the following sequence of reductions, applying the rules to any redexes:

$$\begin{array}{rcll} s(0) + s(s(0)) & \rightarrow & s(s(0) + s(0)) & \text{using rule 2.2} \\ & \rightarrow & s(s(s(0) + 0)) & \text{using rule 2.2} \\ & \rightarrow & s(s(s(0))) & \text{using rule 2.1} \end{array}$$

At this stage the reductions terminate, as there are no more rewrite rules that may be applied. Thus, termination for a term rewrite system over a set $\mathcal{T}$ of all the terms t is defined if there are no infinite derivations $t_1 \rightarrow t_2 \rightarrow t_3 \ldots$. Proofs of termination for term rewrite systems must take into account that the ordering of reductions may be non-deterministic.

In [Der87, Der95], Dershowitz shows various techniques for discovering if a term rewrite system terminates. Examples are:

- The application of any rule to any redex of term t decreases the number of symbols $|t|$ in the term.
- (More generally) Any measure $[\![\cdot]\!] : \mathcal{T} \rightarrow \mathcal{W}$ such that $s \rightarrow t \Rightarrow [\![t]\!] \prec [\![s]\!]$, where $\prec$ is a well-founded ordering on $\mathcal{W}$, for example

  $$f(f(x)) \rightarrow f(g(f(x)))$$

  terminates because the measure *number-of-adjacent-fs* decreases.

- (Lexicographic) Orderings of well-founded orderings, for example

$$\begin{array}{rcl} f(f(x)) & \to & g(f(x)) \\ g(g(x)) & \to & f(x) \end{array}$$

terminates by considering the pair $\langle |t|, \#f \rangle$, where $\#f$ is the number of fs.

- (Interpretation) Assigning an n-ary function $[\![f]\!] : \mathcal{W}^n \to \mathcal{W}$ to each n-ary function f ($\mathcal{W}$ is a well-founded set), such that $x > y \Rightarrow [\![f]\!](\ldots x \ldots) > [\![f]\!](\ldots y \ldots)$, and for each symbol f, for all the variables $\bar{x}$ appearing in l and r, $[\![l]\!] > [\![r]\!]$. For example, with a set of *differentiation* rules containing

$$\begin{array}{rcl} D(x^y) & \to & (y \times x^{y-1} \times Dx) + (x^y \times (\ln x) \times Dy) \\ D(\ln x) & \to & Dx/x \end{array}$$

we could use positive integer polynomial interpretations such as

$$\begin{array}{rclcrcl} [\![Dx]\!] & = & [\![x]\!]^2 & & [\![x/y]\!] & = & [\![x]\!] + [\![y]\!] \\ [\![x^y]\!] & = & [\![x]\!] + [\![y]\!] & & [\![x]\!] & = & x \\ [\![\ln x]\!] & = & [\![x]\!] + 1 & & & & \end{array}$$

Interpreting the terms result in reducing sequences. For example, the left-hand side of the second differentiation rule $D(\ln x)$ is interpreted as $(x+1)^2$, whereas the right hand side of this rule is interpreted as $x^2 + x$. Since $(x+1)^2 > x^2 + x$ (for positive x) every application of this rule will result in a reduction of this interpretation. All the differentiation rules can be interpreted using a polynomial interpretation like this. Note that integer polynomial interpretations like this do not form a general termination technique (they cannot be applied to programs that do not terminate in polynomial time - an exponential system may need exponential interpretations).

Note that in [Der95] Dershowitz shows 33 examples of term rewrite systems, with 33 different techniques to show termination. More detail may be found in [Der87], and an automated termination prover in [GTSKF04].

These approaches (and others like them) seem a little ad-hoc in nature. Given a particular term-rewrite system, there is no clear indication which technique should be applied to show termination. As such, automatic termination checkers have to try a range of techniques, using heuristics to suggest which techniques are appropriate for a particular system.

2.3.3 Termination and logic programming

In the logic programming (LP) arena, there is considerable research, summarized in [DD94, Apt97] including the structure of termination proofs, and techniques to handle mutual recursion, and loop detection.

In LP, an atom without variables is termed a ground atom, and the set of all such ground instances is termed the Herbrand base B_P underlying the program P. A unique identifying characteristic of LP termination analysis is that it is performed in reference to the query, or a set of queries of interest.

LP inherits the same sort of non-deterministic rule application found in term rewrite systems, but the terms in a term rewrite system may not be directly mapped to the atoms of an LP computation. For example, the obvious rules for an *append(x,y,z)* program to append x and y giving z, can be shown to obey a well-founded order, and so if these rules were found in a term-rewrite system, *append* would be terminating. However, the semantics of LP is such that it is possible to query *append* without *ground* input, to derive all pairs of lists; obviously non-terminating.

LP termination has had many successes, and termination analysis is a normal part of LP software systems.

In the interpretation of logic programs, the computation method is known as SLD-resolution, a computation method based on replacement and unification. The sequence of steps is called an SLD-derivation. An initial notion of termination of logic programs is a strong one based on finite SLD-derivations from ground queries. However, this notion is not all that useful, as the normal mode of use of LP in (say) prolog involves using non-ground queries. In the LP world, classes of non-ground queries are checked for specific programs.

In the analysis of termination of LP, it is common to associate atoms with natural numbers, by using a mapping [Bez93]. These mappings are known as level mappings. A level mapping $|\cdot| : B_P \rightarrow \mathbb{N}$ for a program is a function which maps ground atoms to natural numbers. $|A|$ is called the *level* of A.

Level mapping is also defined in reference to non-ground atoms, by considering if it is bound with respect to some level mapping. Clauses in P are described as *recurrent* with respect to a particular level mapping $|\cdot|$ if for every variable-free instance $B \leftarrow A_1, \ldots, A_n$ of a clause from P, for all $i \leq n$, $|B| > |A_i|$. P is termed recurrent if it is recurrent with respect to some level mapping for P. Given these concepts, if every query is bounded, then every SLD-derivation from a recurrent program P will terminate.

The TermiLog system [LSS97] is an example of a system for SICStus Prolog, which automatically checks the termination of queries to logic programs. The authors claim that more than 82% of the programs were correctly checked.

This is necessarily a very brief introduction to termination in logic programming, but the field may be explored more deeply in [CT97, CT99, DLSS99, DLSS01, LS97].

2.3.4 Termination and other programming styles

In the functional programming arena, most functional languages allow the construction of divergent recursive expressions which will not terminate. By limiting the

language, it is possible to ensure termination. For example, a theory of inductive types [CP90] may be used for termination. If any function f defined over an inductive type θ is restricted in the form of the definition to one using the elimination rule of θ, then the function is known to terminate. An alternative form of recursion known as *guarded by destructors* [Gim95] limits recursive calls to be applied to ever-reducing terms.

In the Mercury system for termination of Mercury logic-functional programs [SSS97], a measure of the difference in total size between the input parameters and output arguments is constructed for each procedure. The measures for the procedure input parameters and output arguments are solved as a set of linear inequalities, and if they reduce for each procedure then the program terminates. In this approach, it may be that the constraints do not lead to a reducing solution, and in this case nothing can be said about the termination properties of the program. The "total size difference" argument can be viewed as a coarse-grained size-change termination analysis argument.

> **Note:** The term *size* requires some explanation. In this context it refers to any well-founded ordering over the arguments to a function. For example, if the type of the argument was a natural number, then the size of this could be its value, and the size of this natural number cannot be reduced indefinitely, as eventually it will reach zero and may no longer be reduced. If the type of the arguments were an inductively defined list of items, then the size cannot reduce indefinitely, as eventually it may no longer be reduced. Both of these types are well-founded, and it is on the use of these types that the termination argument relies.

Colón and Sipma [CS01, CS02] use linear ranking functions to prove termination of program loops in an imperative framework. An invariant generator automatically extracts linear ranking functions (over well-founded domains) for the program variables from each loop. The existence of any such ranking function implies that the program loop must terminate. The approach has been applied to reasonably large Java software systems (30,000 lines), and automatically confirms the termination of about a third of the program loops.

In [CPR06], a path and context-sensitive analysis is performed on C code (specifically Windows device drivers), using the Terminator tool. It has been applied to *large* program fragments. Terminator constructs over-approximated linear ranking functions on demand (rather than all at once), iterating between this construction, and checking that at least one of the ranking functions reduces. In analyses like these, least fixpoints correspond to reachability: eventually the state is reached. By contrast, greatest fixpoints correspond to properties that should always hold. In the case of Terminator, the checking process is a greatest fixpoint operation.

2.4 Size-change termination analysis

In 2001, Lee, Jones and Ben-Amram introduced a deceptively simple and yet beautiful termination analysis technique [LJBA01], giving a complexity analysis of the algorithm used in the technique (it is PSPACE-complete). The method derives from the observation that, in the case of a program with program values drawn from well-founded sets:

> *"a program terminates on all inputs if every infinite call sequence would cause an infinite descent in some program values"* [LJBA01].

Size-change termination is not a general termination method, but it is still useful, and it is particularly interesting because it appears to subsume some other termination criterion such as lexicographic descent [BA02]. Size-change termination analysis is an active area of research, with extensions for higher-order analysis [JB, SJ05, Blu04], for integer programs [Ave06, BA06], and applied to term-rewrite systems [TG05] as well as functional and imperative languages.

In the SCT framework, there are a finite number of functions, with the underlying data types of at least some of the parameters expected to be well-founded. In addition, the only technique for repetition is recursion. Given this framework, the intuition behind the size-change termination method is clear.

Consider a non-terminating program. This program can only be non-terminating through an infinite recursion through at least one function entry point, since the number of different functions is finite. Considering the chain of possible function calls within each one of those functions, and the size-change of each of its parameters between successive calls, then if any one of those parameters reduces on each call, there is a conflict. In particular, if it reduces infinitely often, then the data is not well-founded. As a result of this, it is possible to assert that: *"if every infinite call sequence in a program would cause an infinite descent in some program values then the program terminates on all inputs"*.

In this form of size-change termination analysis (hereafter termed LJB analysis after the authors of [LJBA01, Lee02]), size-change graphs approximate the relation between the sizes of each of the source parameters and destination arguments for each call to a function. In the size-change graph, only the following information about each destination argument is recorded: it is the same size ($=$) as some source parameter; (i.e. $m = x$), it is smaller ($\downarrow$ or $<$) than some source parameter; (i.e. $o < y$), it is either the same size or smaller ($\overset{=}{\downarrow}$ or $\leq$) than some source parameter; (i.e. $p \leq z$), or finally it is larger, or has no clearly defined relation to the source parameters (`unknown` or $\top$).

These simple relations are the only ones used to form the size-change graphs, and this style of size-change graph will be referred to as an LJB *size-change graph.*

Definition 1 *An LJB size-change graph is a size-change graph, with each destination argument having a simple relation (=, $\overline{\overline{\downarrow}}$,$\downarrow$ or* `unknown`*) to some source parameter.*

When it is necessary to refer to a particular relation in a size-change graph the notation src $\stackrel{\text{op}}{\rightarrow}$ dest is used (where op $\in \{=, \overline{\overline{\downarrow}}\ \downarrow, \text{unknown}\}$). For example in Figure 2.1, we have $x \stackrel{=}{\rightarrow} m$ and $y \stackrel{\downarrow}{\rightarrow} o$.

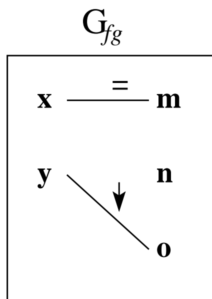


Figure 2.1: LJB size-change graph for function f calling g

This may be visualized with a graphical representation like that shown in Figure 2.1. In this graph, function $f(x, y)$ is calling function $g(m, n, o)$, the first argument is the same *size* as x (=), the second has no relation to the parameters of f (`unknown`), and the third is always smaller than y ($\downarrow$). Note that in the graphical representations of the size-change graphs, the symbols $\downarrow$ and $\overline{\overline{\downarrow}}$ are used instead of $<$ and $\leq$.

An LJB size-change graph can also be encoded by specifying a tuple relation between the source parameters and destination arguments, although the existing literature normally uses the graphical representation just given.

Definition 2 *The syntax for a tuple relation is*

$$\{[\bar{x}] \rightarrow [\bar{y}] : \mathcal{R}\}$$

where $\bar{x}$ and $\bar{y}$ are lists of tuple elements, with $\bar{x}$ corresponding to the source parameters, and $\bar{y}$ corresponding to the destination arguments. $\mathcal{R}$ is a conjunction of simple relations between an element of $\bar{x}$ and $\bar{y}$: $y_i \,\text{op}\, x_j$ where $\text{op} \in \{=, \leq, <\}$. Note that when using the tuple relation notation we use $<$ and $\leq$ instead of $\downarrow$ and $\overline{\overline{\downarrow}}$, and also that if a relation is unknown*, then it is not shown. It was only included to show the weakest relation between tuple elements.*

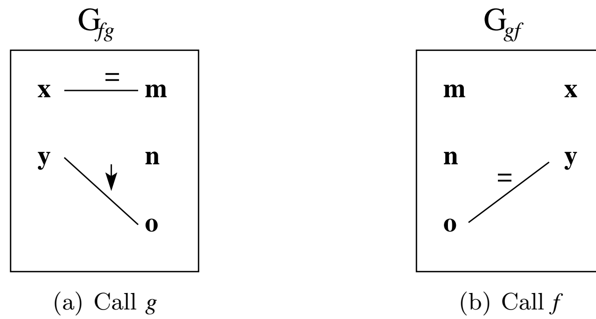
We could use the tuple relation notation just given to express the LJB size-change graph in Figure 2.1 as

$$\{[x, y] \rightarrow [m, n, o] : m = x \wedge o < y]$$

Consider the following functions:

```
f(x,y)   = if x ≥ 0 then
              y
           else
              g(x,0,y − 1);
g(m,n,o) = if o = 0 then
              m + n + 1
           else
              f(m + 1,o);
```

Figure 2.2(a) specifies that in function $f(x,y)$, calling function $g(m,n,o)$, the first argument is the same *size* as x (=), the second has no relation to the parameters of f (`unknown`), and the third is always smaller than y (↓). Similarly for the other function in Figure 2.2(b). The `unknown` relation is not drawn on the diagrams.



(a) Call g (b) Call f

Figure 2.2: LJB size-change graphs

The two graphs may then be used to analyze the mutually recursive functions for termination. A non-terminating sequence beginning with function f would involve an infinite succession of calls to (alternately) the functions f and g.

At this stage, the *flow-of-control* of the program is of interest in the discussion, although for the purposes of size-change termination, only the flow of control of function calls needs to be recorded. In the illustrative example, a possibly infinite call sequence may be expressed as the expressions $(fg)^n$ or $f(gf)^n$.

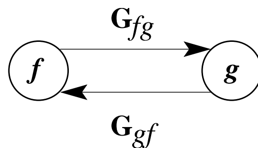


Figure 2.3: Control-flow graph for functions

An alternative way that this flow-of-control can be expressed is by drawing a control-flow graph, such as that seen in Figure 2.3. In this figure the arcs are annotated with the corresponding size-change graph.

When the control flow of a program is such that (say) a function calls another function which then calls a third function, the net effect of the composition of the size-change graphs for the functions is of interest. There are only a finite number of different compositions of graphs. A transitive closure of these size-change graphs must be calculated to ensure that the behaviour of all sequences of function calls is analyzed, and this closure may be computed in a conventional fashion by composing any possible pairs of graphs, and checking if the composite graph is a new one, or just a repetition of an existing one. Pairs of graphs are continually composed until no new graphs are created.

Consider the *idempotent* graphs, the graphs that when composed with themselves result in the same graph (note again that there will be formal definitions of graphs, idempotency and graph composition later). These can be viewed as graphs that represent a recursive chain of calls through a particular function entry point. For each of these idempotent graphs, examine the size-change information. If all of them have at least one parameter that reduces, then it is possible to conclude that the program terminates on all inputs.

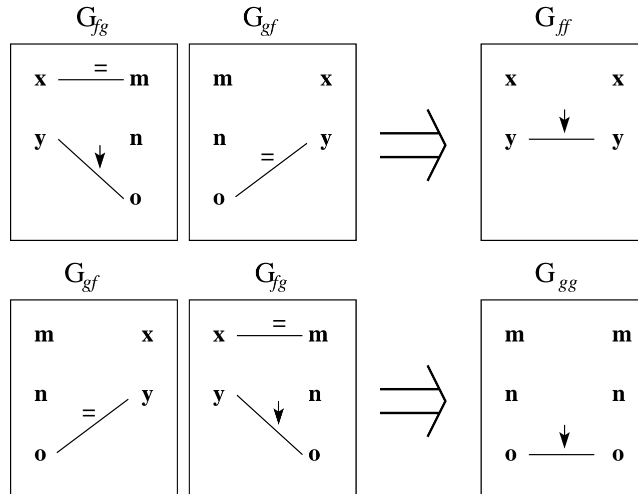


Figure 2.4: LJB size-change graphs for functions $f \circ_G g$ and $g \circ_G f$

Composite graphs are constructed as shown in Figure 2.4. The graph composition is easy to understand by inspection of the diagrams, and a formal definition of LJB graph composition is given using the tuple relation notation.

$$\boxed{=} \; < \; \boxed{\overline{\downarrow}} \; < \; \boxed{\downarrow} \; < \; \boxed{\text{unknown}}$$

Figure 2.5: Ordering of the operators for LJB graph composition

When composing graphs, the matching operators on a continuous path through the composite graph are compared, and used to generate a new operator. For this generation process, there is an obvious ordering of the operators, as seen in Figure 2.5. For example, in the lower graph composition in Figure 2.4, we have $o \stackrel{=}{\to} y$ and $y \stackrel{\downarrow}{\to} o'$, and the final operator is the maximum of the two operators: $\lceil =, \downarrow \rceil$. This generates the final relation $o \stackrel{\downarrow}{\to} o'$.

Definition 3 *LJB graph composition* $\circ_G$*: The composition of two LJB graphs such as* $r_1 = \{[\bar{x}] \to [\bar{y}] : D_1\}$ *and* $r_2 = \{[\bar{y}] \to [\bar{z}] : D_2\}$ *is a new LJB graph*

$$r = r_1 \circ_G r_2 = \{[\bar{x}] \to [\bar{z}] : D\}$$

with $D = \bigwedge_i d_i$*, and for each element* $d_i = z_j \,\mathrm{op}\, x_k$ *there exists a matching* y *such that* $x_k \stackrel{\mathrm{op}_1}{\to} y \in r_1$ *and* $y \stackrel{\mathrm{op}_2}{\to} z_j \in r_2$*. The new operator* op *is the maximum of* op_1 *and* op_2*. (*$\mathrm{op} = \lceil \mathrm{op}_1, \mathrm{op}_2 \rceil$ *given the ordering defined before). Note that the* `unknown` *relation in either* op_1 *or* op_2 *results in the* `unknown` *relation in* op *if there are no other matching relations, as it is the largest relation.*

When function f calls function g the corresponding graph is G_{fg}, and when function g calls function f the corresponding graph is G_{gf}. The sequence of f calling g calling f again has a corresponding graph $\mathrm{G}_{ff} = \mathrm{G}_{fg} \circ_G \mathrm{G}_{gf}$. Note that the use of the composition operator here is reversed from some other presentations, to correspond with the time sequence of the function calls, with $a \circ_G b$ corresponding to the call a followed by the call b.

The graphs in Figure 2.4 show the results of the two successive calls, and demonstrate that the y (in function f) or o (in function g) parameter must reduce. The sequences with graphs $\mathrm{G}_{ff} = \mathrm{G}_{fg} \circ_G \mathrm{G}_{gf}$ and $\mathrm{G}_{gg} = \mathrm{G}_{gf} \circ_G \mathrm{G}_{fg}$ must occur infinitely often in any possible infinite call sequences. As a result, since every infinite call sequence in a program would cause an infinite descent in a program value, then the program terminates on all inputs.

Note that during this argument, the only information that needs to be recorded is the (possibly infinite) call sequences; i.e. one aspect of the *flow-of-control* in the program, and the size change for parameters during each function call.

The size-change termination argument is independent of other factors such as the condition tests in if-then-else statements, or the precise changes in data values. It is only dependent on the above two elements. This is surprising when you are first introduced to size-change termination.

The technique is simple, and by itself can often automatically show termination without appealing to higher-level reasoning techniques. For example, it is common to use a lexicographic ordering to demonstrate termination of a function, and such a function may require some inspection to determine a suitable ordering of the parameters.

The following recursive countdown function c can be inspected, and termination confirmed by noting that the lexicographic ordering **h:t:o** (hundreds:tens:ones) always reduces on every call[1]. Note that each call is annotated to emphasize that information related to each call is captured independently.

```
c(h,t,o) = if h+t+o = 0 then
              0
           else
              if o = 0 then
                 if t = 0 then
                    c1(h-1,9,9)
                 else
                    c2(h,t-1,9)
              else
                 c3(h,t,o-1);
```

However the LJB size-change termination method is independent of this order, and is able to successfully confirm termination for the countdown example without guidance.

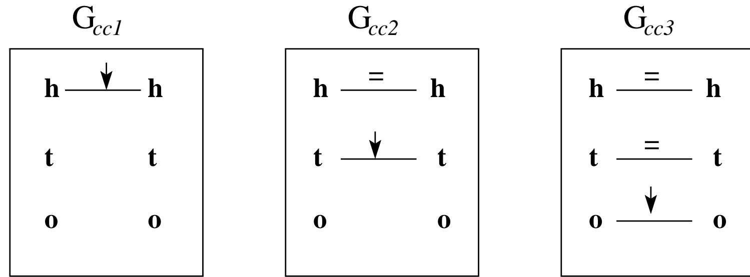


Figure 2.6: LJB size-change graphs for function c

In any given program there is a limit in the number of LJB size-change graphs and hence in their compositions. A transitive closure procedure can be used to generate all the different size-change graphs for a program, by using LJB graph composition over all reachable paths. When a graph is idempotent (i.e. $\mathrm{G} = \mathrm{G} \circ_G G$), then there is no need to generate more graphs like $\mathrm{G} \circ_G \mathrm{G} \circ_G \mathrm{G}$. This closure of the size-change graphs for the countdown function is shown in Figure 2.6, and each graph is idempotent, and has a reducing parameter.

The central theorem of the LJB size-change graph construction algorithm for establishing termination, explained and proved in [LJBA01], is:

Theorem 1 *Program P is not size-change terminating iff S contains $G : f \to f$ such that $G = G \circ_G G$ and G has no arc of the form $x \xrightarrow{\downarrow} x$, where S is the set of possible size-change graphs.*

[1]This function is chosen deliberately to highlight the suitable lexicographic ordering.

This theorem gives rise to a relatively efficient technique for automatically deriving termination properties from a closure computation over the size-change graphs, without appealing to higher-level reasoning techniques such as lexicographic ordering.

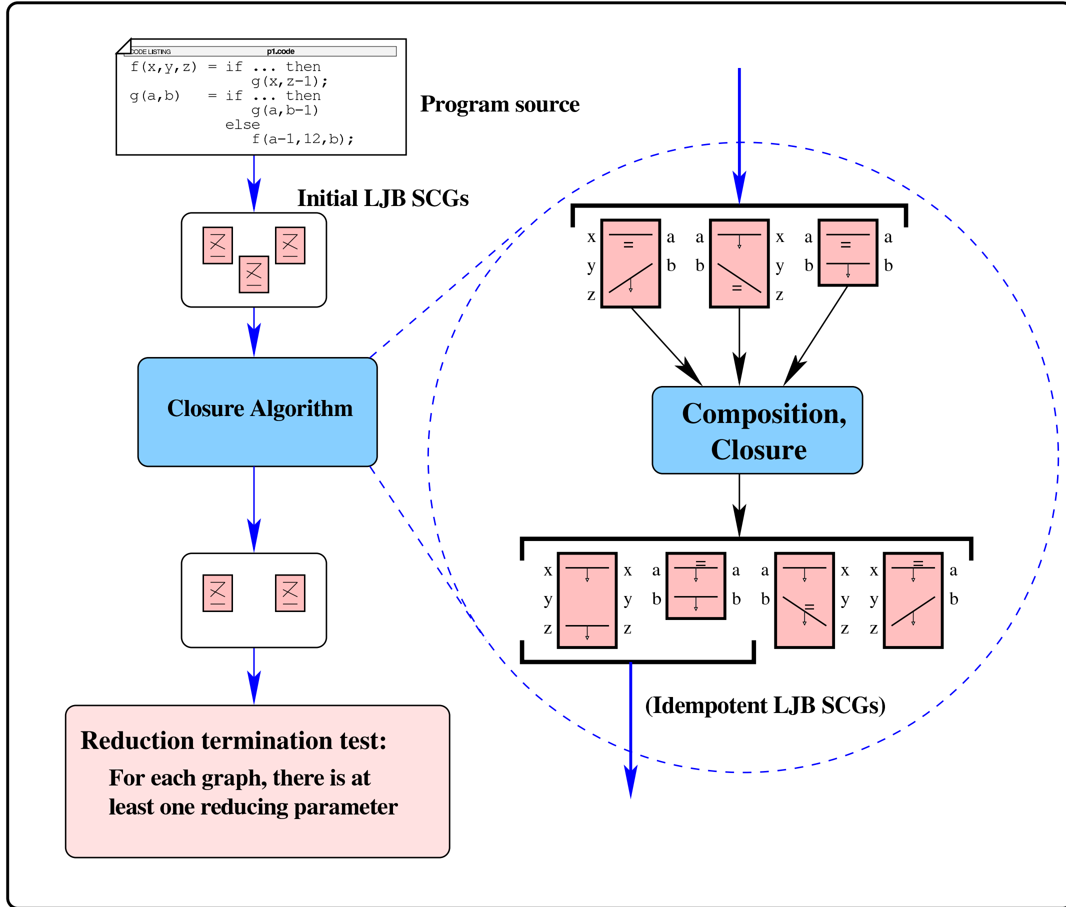


Figure 2.7: Overview of LJB size-change termination analysis

Figure 2.7 shows a summary of the flow of (LJB-style) SCT analysis, starting from the initial program source, which is analyzed to obtain the initial SCGs. A closure algorithm is applied to these SCGs, leading to a final set of SCGs. Only the idempotent SCGs are examined, to see if each one has a reducing arc, representing a reducing parameter.

2.5 Type inference for generation of SCGs

Tool support for the approaches to termination and run time analysis in this thesis raise some implementation issues. This section presents an approach to the inference of size-change graphs, within a *dependent/sized types* framework. In the approach, un-annotated recursive functions are typed, using a specialised set of type inference

rules. The resultant type annotations either directly indicate that the function terminates, or may be used to attempt to derive the termination properties of the function using the techniques described here.

2.5.1 Type systems and analysis

The process of effective abstraction underlies most facets of software production and analysis. The study of type systems has been useful in this context, providing mechanisms to statically determine program properties.

Type inference refers to finding the types of the expressions in a program at compile time, without explicit type annotations added to the source of the program. It derives from type inference for the simply typed lambda calculus of Church. Curry developed a type inference mechanism for the calculus, but now the Hindley/Damas-Milner algorithm [Hin69, Mil78, DM82] is normally used to perform type inference for modern typed languages.

One focus of the theory of types involves the construction of types dependent on terms. This is known as the theory of *dependent types*. A sub-theory of this involves types that also define the size of the type. These *sized types* may indicate the *length* of a list, or the *size* of an array or some other measure of interest. Dependent types and sized types are used in various areas of program analysis and have been shown to provide effective mechanisms for abstraction. The area of interest for this thesis is in the use of dependent types to establish termination properties of programs.

For example, program termination has been explored within type analysis frameworks, in [Xi02], where Xi annotates program fragments to assist in termination analysis, and a well-typed program must terminate. In Xi's approach, the annotations indicate a metric to be used for establishing a decreasing sequence, and hence termination. The approach has been embedded in Dependent ML (DML), and provides a facility to add metrics to type annotations for functions.

There are two main variants of the use of dependent/sized types. Firstly, it has been used in the context of *annotating* programs and then checking properties of the program. Secondly, type rules can be directly used to *infer* the types.

Program annotation/type checking variant: Most presentations of dependent types concentrate on the use of the theory to provide much more precise *type* information about program fragments. For example, dependent types have been embedded in languages as annotations to programs, with automated type-checkers verifying the consistency of the annotations [XP99, Xi98, Aug98]. The technique commonly results in verification of program properties, which can then be used to optimize the program in some way - for example verification of valid array-bounds access, leading to the elimination of runtime array-bounds checks.

This process (annotation of programs followed by type-consistency checks) requires a level of formal evaluation of the annotations. The annotations are considered to

be constraints over program fragments, and type-consistency is a process of finding a satisfaction set for the constraints. In general this is a *hard* problem, as the formal evaluation of the annotations and derivation of satisfaction may require proofs. However most implementations of the theory use a simple notion of constraints (such as sets of linear inequalities) which have solutions that are known to be tractable. In [XP99], Xi presents a general constraint domain framework, but only considers its use with linear relations over integers.

Program inference/type construction variant: Other presentations, by contrast, develop techniques for deriving sized type information by inference (that is - with no annotations). In [CK00], Chin and Khoo present algorithms which infer size information from an un-annotated program fragment. The algorithms deduce sized types for expressions, restricted by affine constraints over variables. These constraints are checked for satisfaction using the *Omega* library, which manipulates the integer tuple relations and sets. The result of this is the inference of size information over program fragments, with no prior annotations. The approach adopted in this research is to use the second variant, type *inference*, to extract size-change graphs automatically from program sources.

2.6 Type directed implementation for size-change graphs

To demonstrate the use of dependent/sized types in generating size-change graphs, a very simple function definition language is used with simple functions like `suc`, `head`, and `cons`. This language makes the arithmetic "size" operations explicit (in particular, the pred and tail functions directly map to a size reduction), and shows how they are used in typing rules. An example of the use of this language is the following implementation of Ackermann's function:

```
ack(m,n) = if iszero(m) then
               succ(n)
           else
               if iszero(n) then
                  ack(pred(m),1)
               else
                  ack(pred(m),ack(m,pred(n)));
```

Static dependent type inference analysis is then performed on a sequence of these function definitions, returning a mapping for each function. The mappings either indicate termination, or return complex structured types. For example - given the function above, static dependent type inference analysis returns the following structure:

```
1.  ("ack", [("ack", [(0, ↓); (-, -)]),
             ("ack", [(0, =); (1, ↓)]),
             ("ack", [(0, ↓); (-, -)])])
```

This indicates that the termination of function **ack** is dependent on three other (named) functions. The mapping for the **ack** function is a representation of the three size-change graphs for this function. For example, in the first line of the structure, (**"ack"**, [(**0**, ↓); (**-**, **-**)]) represents the size-change graph shown in Figure 2.8.

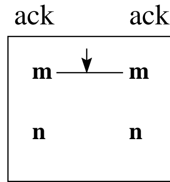


Figure 2.8: Simple SCG for Ackermann function call

The two dependent-type pairs (**0**, ↓) and (**-**, **-**) represent the changes in the two parameters to the call to the **ack** function. The first pair (**0**, ↓), represents that this parameter is dependent on the zeroth parameter of the parent function (m) and reduces, and the second pair indicates no discovered dependence for this parameter.

2.6.1 Notation and syntax for typing

The simple language is specified in Table 2.1.

v	$\in$	`Var`	⟨Well founded variables⟩
f,g,h	$\in$	`FName`	⟨Function names⟩
c	$\in$	`Const`	⟨Constants⟩
t	$\in$	`Term`	⟨Terms⟩
		t ::= **if** t_0 **then** t_1 **else** t_2	
		$\mid x \mid c \mid f(t_1,\ldots,t_n)$	
d	$\in$	`Decl`	⟨Definitions⟩
		d ::= $f(x_1,\ldots,x_n) = t$	

Table 2.1: The simplest language syntax

The language allows new functions to be defined, and has in addition, the predefined functions **iszero**, **notzero**, **succ**, **pred**, **head**, **tail** and **cons** which operate over the (well founded) variables. These functions have their normal meanings, and are commonly found in simple functional languages with natural numbers and lists.

In the following description of the type environment for this simple language, the following notation is used: t or t_i refer to the terms, T or T_i refer to the type of terms, v or v_i refer to the variables, C or C_i refer to constants, and v_ℓ refers to the value of a variable v when used in a term within a function.

2.6.2 Type inference rules for size-change graphs

The style of inference used for type analysis is borrowed here in order to generate size-change graphs. The conventional type of a term is not recorded, but instead its size "relative to" parameter variables. To show the general idea, consider the inference of a term containing a parameter m. The type of a parameter m is the pair $(m_\ell, =)$, in which m_ℓ represents the value of the parameter at the time it was instantiated, and $=$ indicates that the size is unchanged. If there was a more complex term, its type might be $(n_\ell, \downarrow)$, indicating that the size is less than the parameter n. The simplest dependent types are deduced from the use of the simple variable type inference rules as seen in Table 2.2.

$\Gamma \vdash v \,:\, (v_\ell, =)$	T-var
$\Gamma \vdash C \,:\, (-,-)$	T-constant
$\Gamma \vdash \text{iszero}(t) \,:\, (-,-)$	T-iszero
$\Gamma \vdash \text{notzero}(t) \,:\, (-,-)$	T-notzero

Table 2.2: Inference rules for simple variables

In addition, there are rules for the simple functions **suc**, **pred**, **head**, **tail** and **cons** as seen in Table 2.3.

$\dfrac{\Gamma \vdash t \,:\, (v_\ell, =)}{\Gamma \vdash \text{cons}(t, []) \,:\, (v_\ell, =)}$	T-cons1	$\dfrac{\Gamma \vdash t \,:\, (v_\ell, \downarrow)}{\Gamma \vdash \text{cons}(t, []) \,:\, (v_\ell, \downarrow)}$	T-cons2
$\Gamma \vdash \text{cons}(t_1, t_2) \,:\, (-,-)$	T-cons3	$\dfrac{\Gamma \vdash t \,:\, (v_\ell, =)}{\Gamma \vdash \text{pred}(t) \,:\, (v_\ell, \downarrow)}$	T-pred1
$\dfrac{\Gamma \vdash t \,:\, (v_\ell, \downarrow)}{\Gamma \vdash \text{pred}(t) \,:\, (v_\ell, \downarrow)}$	T-pred2	$\dfrac{\Gamma \vdash t \,:\, (-,-)}{\Gamma \vdash \text{pred}(t) \,:\, (-,-)}$	T-pred3
$\Gamma \vdash \text{succ}(t) \,:\, (-,-)$	T-succ	$\dfrac{\Gamma \vdash t \,:\, (v_\ell, =)}{\Gamma \vdash \text{head}(t) \,:\, (v_\ell, \downarrow)}$	T-head1
$\dfrac{\Gamma \vdash t \,:\, (v_\ell, \downarrow)}{\Gamma \vdash \text{head}(t) \,:\, (v_\ell, \downarrow)}$	T-head2	$\dfrac{\Gamma \vdash t \,:\, (-,-)}{\Gamma \vdash \text{head}(t) \,:\, (-,-)}$	T-head3
$\dfrac{\Gamma \vdash t \,:\, (v_\ell, =)}{\Gamma \vdash \text{tail}(t) \,:\, (v_\ell, \downarrow)}$	T-tail1	$\dfrac{\Gamma \vdash t \,:\, (v_\ell, \downarrow)}{\Gamma \vdash \text{tail}(t) \,:\, (v_\ell, \downarrow)}$	T-tail2
$\dfrac{\Gamma \vdash t \,:\, (-,-)}{\Gamma \vdash \text{tail}(t) \,:\, (-,-)}$	T-tail3		

Table 2.3: Inference rules for simple functions

Dependent types such as the ones just given are *simple* types. More complex typing rules are needed for function application as seen in Figure 2.4. The last two rules are not strictly needed, but provide a little more type information for simple single-parameter functions.

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \dots \quad \Gamma \vdash t_n : Tn}{\Gamma \vdash f(t_1, t_2, \dots, t_n) : [(\mathrm{f}, [T_1, T_2, \dots, T_n])]} \quad \text{T-app1}$$

$$\frac{\Gamma \vdash t : (v_\ell, =) \quad \Gamma \vdash f : (f, [(0, \downarrow)])}{\Gamma \vdash f(t) : (v_\ell, \downarrow)} \quad \text{T-app2}$$

$$\frac{\Gamma \vdash t : (v_\ell, \downarrow) \quad \Gamma \vdash f : (f, [(0, \downarrow)])}{\Gamma \vdash f(t) : (v_\ell, \downarrow)} \quad \text{T-app3}$$

Table 2.4: Inference rules for function application

Lastly there are rules for the remaining components of the language. The rules in Table 5.2 collect more complex type/size-change information.

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash f(a, \dots, n) = t : (\mathrm{f}, [T])} \quad \text{T-def}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \mathrm{if}\, t_1 \,\mathrm{then}\, t_2 \,\mathrm{else}\, t_3 : [T_1, T_2, T_3]} \quad \text{T-if}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ;\ t_2 : [T_1, T_2]} \quad \text{T-seq}$$

Table 2.5: Inference rules for the rest of the language

The preceding rules give a mechanism for automatically inferring the size-change information about functions. In the case that the resultant type of a function f is like $(f, [X, \dots, Z])$ where each component of $X \dots Z$ is a simple type, then it is possible to conclude that the function terminates. Why? Because it makes no function calls.

In the case that the resultant type of a function f is like $(f, [X, \dots, Z])$ where at least one component of $X \dots Z$ is a *complex* type, then it it is not possible to conclude that the function terminates. However, the type information collected in $[X, \dots, Z]$ may be enough to attempt to calculate the termination properties of the function.

2.6.3 SCT termination analysis using size-type inference

The SCT termination analysis process may be performed in four steps. The four steps are: (1) Inference - collect the size-change graph information; (2) Flattening - simplify the information, removing anything not relevant; (3) Closure - construct the transitive closure of the graphs for each function; (4) Check - examine the graphs, and try to deduce termination properties.

The approach is shown by considering the following three functions:

```
ack(m, n) = if iszero(m) then
                succ(n)
              else
                if iszero(n) then
                  ack(pred(m), 1)
                else
                  ack(pred(m), ack(m, pred(n)));
f(x, y)     = if iszero(x) then
                y
              else
                g(x, y, 0);
g(u, v, w) = if notzero(v) then
                g(u, pred(v), succ(succ(w)))
              else
                f(pred(u), w);
```

Using these functions, here is this four-step process:

Inference. Using the type inference rules given before, the following dependent type/size-change information is derived:

```
[ ("ack", [ (-,-),
            (-,-),
            [ (-,-),
              ("ack", [ (0, ↓); (-, -)]),
              ("ack", [ (0, ↓),
                        ("ack", [(0, =); (1, ↓)])
                      ])
            ]
          ]),
  [ ("f", [(-,-),(1,=),("g",[(0,=),(1,=),(-,-)])]),
    ("g", [(-,-),("g",[(0,=),(1,↓),(-,-)]),
          ("f",[(0,↓),(2,=)])])
  ]
]
```

This has extra information, which could possibly be used for more advanced termination inference techniques than that described here. In the approach described here however, the structure can be flattened, and then only retain the function call size-change information.

Flatten the types. The process of type inference collects extra information, which is not needed for the algorithm for termination inference used here. The flattening operation removes extra information, by removing simple types at the first level, and any types at third and greater levels (we do not need size changes within size changes).

After flattening, we have:

```
[ ("ack", [("ack", [(0, ↓); (-, -)]),
           ("ack", [(0, =); (1, ↓)]),
           ("ack", [(0, ↓); (-, -)])
          ]),
  ("f",   [("g",   [(0,=),(1,=),(-,-)])]),
  ("g",   [("g",   [(0,=),(1,↓),(-,-)]),
           ("f",   [(0,↓),(2,=)])
          ])
]
```

This type information may be seen to map directly to the expected size-change graphs, as seen in Figure 2.9.

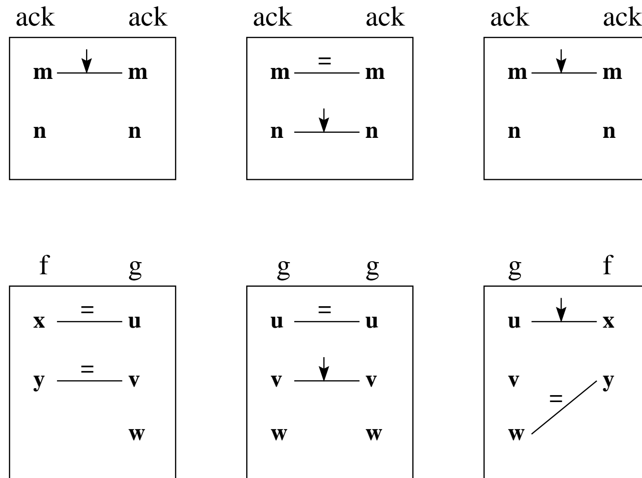


Figure 2.9: Final SCGs for example programs

Construct the closures. For each of the size-change dependent types, the transitive closure of the size-change graphs is constructed. Two graphs g_{ab} and g_{bc} may be composed to construct a new graph by considering the effect of function b calling

function c. The resultant size-change graph $g_{ac} = g_{ab} \circ_G g_{bc}$ represents the relation between the original parameters of function a and the values used to call function c.

The closure may be computed in a conventional fashion using Nuutila's technique [Nuu94]:

1. Compute the strongly connected components of the graph. If two vertices u and v are strongly connected, then u and v are reachable from each other.
2. Construct the condensation graph. A condensation graph is a new graph in which each new vertex corresponds to a strongly connected component and there exists an edge connecting any of the vertices in the pairs of components.
3. Compute the transitive closure on the condensation graph:

```
foreach Vertex u ∈ CondensationGraph
  foreach Vertex v ∈ Adjacent(u)
      if v ∉ Successor(u) then
          Successor(u) = Successor(u) ∪ {v} ∪ Successor(v);
```

The transitive closure of our example functions is:

```
[ ("ack", [("ack", [(0, ↓); (-, -)]),
           ("ack", [(0, =); (1, ↓)])
          ]),
  ("f",   [("g",   [(0,=),(1,=),(-,-)])]),
           ("g",   [(0,=),(1,↓),(-,-)]),
           ("f",   [(0,↓),(-,-)])
          ]),
  ("g",   [("g",   [(0,=),(1,↓),(-,-)]),
           ("f",   [(0,↓),(2,=)])
          ])
]
```

The final phase of the algorithm involves inspecting the idempotent graphs:

Termination. In this case, the size-change termination argument is used. Consider a non-terminating program. This program can only be non-terminating through an infinite recursion through (at least one) function, since the number of different functions is finite.

A second step is to consider the chain of possible function calls within each one of those functions, and determine the size change of each of its parameters between successive calls. If any one of those parameters reduces on each call, then there is a conflict. If it reduces infinitely often, then the data is not well-founded.

As a result of this negative argument, we restate a simple observation that may be made over a program which precludes it from being non-terminating:

> *"if every infinite call sequence in a program would cause an infinite descent in some program values then the program terminates on all inputs"*

The transitive closure G of all the size-change graphs is constructed, before looking at the idempotent graphs in this closure - i.e. the graphs $g_{ff} \in G : g_{ff} = g_{ff} \circ_G g_{ff}$. The graphs could be viewed as a representation of a recursive chain of calls through a particular function (f) entry point. For each of these idempotent graphs, the size change information is examined. If all of them have at least one parameter that reduces $(x, \downarrow)$, then there is a conflict, and the program terminates on all inputs. If there is an idempotent graph with no parameter that reduces on all inputs, then nothing can be said about the program. It may terminate, it may not. Returning to the example, the idempotent graphs are the following:

```
[ ("ack", [("ack", [(0, ↓); (-, -)]),
           ("ack", [(0, =); (1, ↓)])]),
  ("f",   [("f",   [(0,↓),(-,-)])]),
  ("g",   [("g",   [(0,=),(1,↓),(-,-)])])
]
```

For each of these, there is a reducing parameter, and hence all three functions terminate on all inputs.

The core of this section was to construct the SCGs found in the sort of program termination analysis outlined in [LJBA01], within a dependent/sized-types framework. This leads to an approach to verifying program termination on un-annotated programs.

2.7 Preliminary definitions

In this section the language used for programs is extended with primitive operators, along with affine relations and Presburger formulæ. Preliminary definitions and notation for affine functions are introduced.

2.7.1 The language

In order to show the mechanism behind the termination analysis, the simple first-order functional language is extended in Table 2.6. Additional language features could be included in the subject language, but as these will only complicate the generation of the initial set of size-change graphs, but not the analysis, they will not be included in the discussion.

x	$\in$	`Var`	⟨Variables⟩
op	$\in$	$\{<, \leq, =, \geq >\}$	⟨Primitive operators⟩
f, g, h	$\in$	`FName`	⟨Function names⟩
c	$\in$	`Const`	⟨Constants⟩
e	$\in$	`Exp`	⟨Expressions⟩
	e ::=	$\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2$	
		$\mid x \mid c \mid e_1\, op\, e_2 \mid f(e_1, \ldots, e_n)$	
d	$\in$	`Decl`	⟨Definitions⟩
	d ::=	$f(x_1, \ldots, x_n) = e$	

Table 2.6: The language syntax, extended with operators

As mentioned before, the functional style adopted in the thesis can easily accommodate imperative and other styles of programming, and it is just a matter of personal choice to use the functional style.

2.7.2 Affine relations and Presburger formulæ

The term "affine relation" is used in this thesis in the same sense as that of Karr's early paper which provided an algorithm for computing affine relationships between variables of a program [Kar76].

Definition 4 *An* affine relation *is a property of the form* $a_0 + \sum_{i=1}^{k} a_i x_i \leq 0$ *or* $a_0 + \sum_{i=1}^{k} a_i x_i = 0$ *where* $x_1, \ldots, x_k$ *are integer program variables, and* $a_0, \ldots, a_k$ *are integer constants.*

Affine relations are captured using Presburger formulæ, a class of logical formulæ built from affine constraints over the integers, the quantifier $\exists$, and $\wedge$, $\vee$ and $\neg$. In this context, if the parameters are of the *natural* (well-founded) type, then the affine relations have an extra constraint $\mathcal{D}$ which restricts the variables to be positive.

Formulæ:	ϕ	$\in$	`F`	⟨Formulæ⟩
		ϕ ::=	$\psi \mid \{[v_1, \ldots, v_m] \rightarrow [w_1, \ldots, w_n] : \psi\}$	
		ψ ::=	$\delta \mid \neg\psi \mid \exists\, v.\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$	
Size Formulæ:	δ	$\in$	`Fb`	⟨Boolean expressions⟩
		δ ::=	$\mathrm{T} \mid \mathrm{F} \mid a_1 = a_2 \mid a_1 \neq a_2$	
			$\mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2$	
	a	$\in$	`Aexp`	⟨Arithmetic expressions⟩
		a ::=	$n \mid v \mid n \star a \mid a_1 + a_2 \mid -a$	
	n	$\in$	$\mathcal{Z}$	⟨Integer constants⟩

Table 2.7: Syntax of Presburger formulæ

The tuple relations have explicitly identified source and destination parameters, and the syntax is defined in Table 2.7.

An affine tuple relation is interpreted as a set of pairs satisfying the affine relation. For example, the following affine tuple relation $\phi = \{[m,n] \to [p] : p = m+n+1 \wedge \mathcal{D}\}$ can be interpreted as a subset of $\mathbb{N}^2 \times \mathbb{N}$. Some of the pairs belonging to this set are: $([1,0],[2])$, $([3,4],[8])$.

This interpretation enables us to talk about subset inclusion between set solutions of affine tuple relations. It induces a partial ordering relationship among the affine relations, and corresponds nicely to the implication relation between two affine relations, when viewed as Presburger formulæ. As a result, ϕ implies the relation

$$\phi' = \{[m,n] \to [p] : p > m+n \wedge \mathcal{D}\}$$

because the set generated by ϕ is a subset of that generated by ϕ', denoted by $\phi \subseteq \phi'$. Subset notation is adopted as relation implication.

Operations over affine tuple relations. The first operation of interest is the composition operation for affine tuple relations. This operation is meaningful only when interpreting an affine tuple relation as a binary relation over two sets of parameters. The idea of composing two relations, as in $\phi_1 \circ_F \phi_2$, is to identify the second parameter set of ϕ_1 with the first parameter set of ϕ_2.

Definition 5 *Affine tuple relation composition is defined as follows:*

$$\phi_1 \circ_F \phi_2 \stackrel{def}{=} \{(\bar{x},\bar{z}) \mid \exists\, \bar{y} : (\bar{x},\bar{y}) \in \phi_1 \wedge (\bar{y},\bar{z}) \in \phi_2\}$$

Thus, for composition over two affine relations $\phi_1 \circ_F \phi_2$ to be definable, the number of parameters in the first set of ϕ_2 must be the same as the number of parameters in the second set of ϕ_1.

Fact 1. The composition operator over affine relations with the same set of parameters preserves any monotonic ordering of the parameters: if we had $(\bar{x},\bar{x}') \circ_F (\bar{x}',\bar{x}'')$ and an ordering[2] over the parameters $\preceq$ such that $\bar{x} \preceq \bar{x}'$ and $\bar{x}' \preceq \bar{x}''$, then $\bar{x} \preceq \bar{x}''$.

The second operation of interest is the union of affine tuple relations. This is definable when all the affine relations have the same set of parameters, modulo variable renaming.

Definition 6 *The union of affine tuple relations is given by:*

$$\phi_1 \cup \phi_2 \stackrel{def}{=} \{(\bar{x},\bar{y}) \mid (\bar{x},\bar{y}) \in \phi_1 \vee (\bar{x},\bar{y}) \in \phi_2\}$$

[2] For example if each individual parameter in the first set was less than the corresponding parameter in the second set.

Fact 2. The union operation over a set of affine relations is monotone. In fact, the union operation computes the least upper bound of the set of affine relations, if all affine relations with the same set of parameters are considered as a lattice partially ordered by set inclusion.

The third operation of interest is the transitive closure operation over an affine relation.

Definition 7 *The transitive closure over an affine tuple relation ϕ is as follows:*

$$\phi^{+} \stackrel{def}{=} \bigcup_{i \geq 0} \phi^{i}$$

where ϕ^{i} means composing ϕ with itself i times.

Note that for the closure operation to work properly, ϕ must be represented as a relation over two sets of parameters, with both sets of equal size. Facts 1 and 2, indicate that the transitive closure operation is monotone.

Fact3. The transitive closure operation is idempotent. That is, $\phi^{+} \circ_{F} \phi^{+} = \phi^{+}$.

2.8 Commentary

In this chapter, the groundwork has been laid for the contribution of this thesis. In particular, the use of program analysis, decidable theories, and type inference to derive arithmetic and size-change information useful in the analysis of program properties has been outlined. Various techniques for program termination analysis were described, in particular the size-change termination technique of [LJBA01]. In explaining this related work, the notation and concepts common to size-change termination, and the affine based analysis have been introduced.

Part II

Body

Chapter 3

Affine-based analysis

This section introduces affine-based SCT analysis, an improvement over the LJB-analysis introduced previously.

A more refined representation of size-change relations can widen the set of size-change terminating functions. Presburger formulæ, or affine relations in particular, can be a good candidate for encoding size-change graphs, for the following three reasons:

1. They allow the capturing of constant changes in parameter size. The effect of constant increment and decrement may be canceled out during the analysis.
2. They can express size change of a destination argument by a linear combination of source parameters. This enables more accurate representation of size change.
3. They can constrain the size change information with information about call context, thus naturally extending the analysis to be context-sensitive.

To illustrate that this termination analysis is strictly more powerful than the LJB-analysis, three example programs in the simple first-order function definition language are listed below. They can be successfully analyzed for termination by the more refined analysis, but are outside the scope of LJB-analysis.

The first example alternately increases and reduces a parameter on successive function calls corresponding to the first reason given.

```
f(m) = if m ≤ 0 then
           1
         else
           g_a(m + 1);
g(n) = if n ≤ 0 then
           1
         else
           f_a(n − 2);
```

An example of the second reason given is the following function in which the LJB-analysis is unable to establish that the first argument must decrease:

```
k(m,n) = if m ≤ 0 then
             1
           else
             k_a(m − n, n + 1);
```

An example of the third reason given is the following function in which the variables are all natural numbers, and only one of the two calls can be performed, constrained by the condition $m < n$. The LJB-analysis fails to establish termination:

```
j(m,n) = if m + n = 0 then
             0
           else
               if m < n then
                   j_a(m − 1, n + 1)
               else
                   j_b(m + 1, n − 1);
```

Affine relations can capture size-change information, and the soundness and termination of affine-relation based termination analysis is shown in the following sections.

3.1 Affine-based size-change termination

In *affine-based* size-change termination, size-change graphs are translated to affine-based graphs, in which affine relations among parameters are expressed by Presburger formulæ. The correctness of the translation is shown by defining the size-change graph composition in terms of affine relation manipulation, and identifying the idempotent size-change graphs with transitive closures in affine relations.

An affine-based termination analysis is developed, in which more refined termination size-change information is admissible by affine relations. Specifically, the affine-related analysis improves the effectiveness of the termination analysis by capturing constant changes in parameter sizes, affine relationships of the sizes of the source parameters, and contextual information pertaining to function calls.

This approach widens the set of size-change terminating functions: all LJB size-change terminating programs are still detectable using affine-based size-change termination, and there are affine-based size-change terminating programs not found by LJB-analysis.

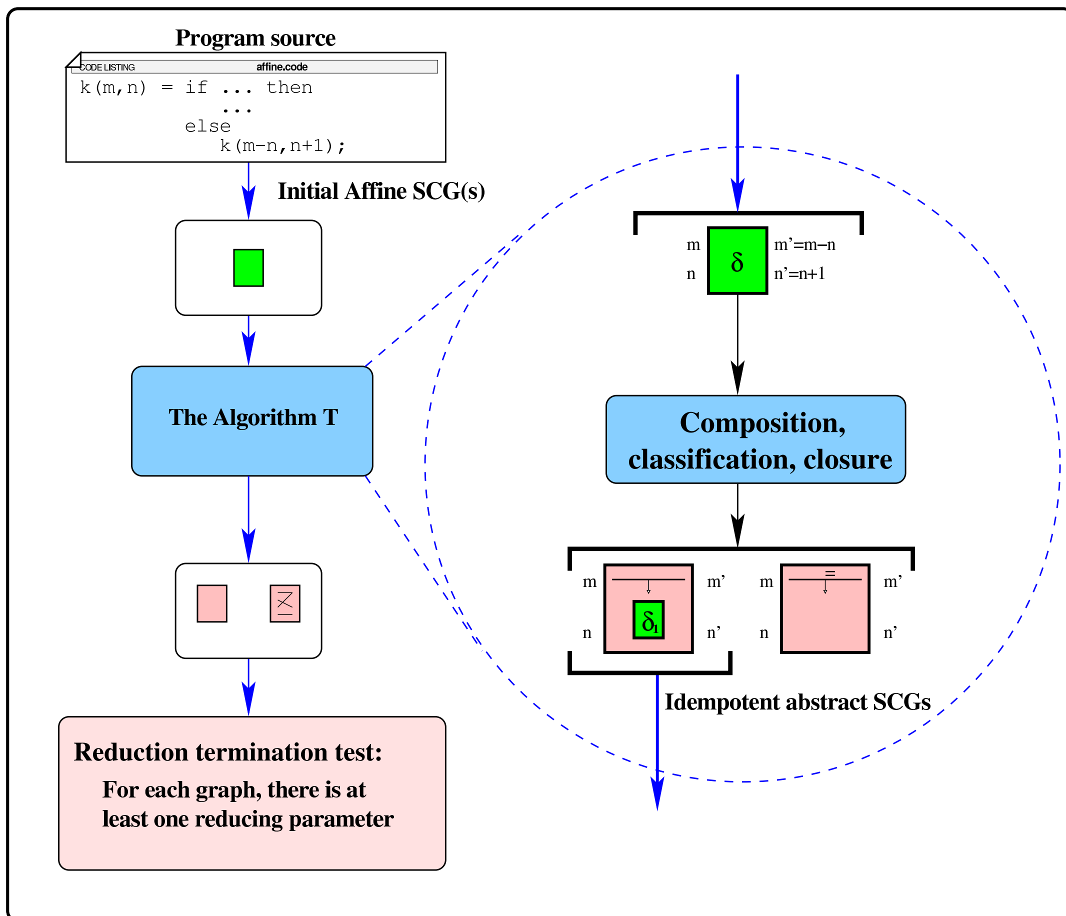


Figure 3.1: Overview of affine-SCT analysis

Figure 3.1 shows a summary of the flow of (affine-style) SCT analysis, starting from the initial program source, which is analyzed to obtain initial affine SCGs. A closure algorithm (algorithm $\mathcal{T}$) is applied to these affine SCGs, leading to a final set of SCGs (termed *abstract* SCGs). Only the idempotent abstract SCGs are examined, to see if each one has a reducing arc, representing a reducing parameter.

Size-change information is represented using the notation and syntax of the Omega calculator and library [KMP+96], which can manipulate Presburger formulæ. The

graphs thus represented are termed *affine size-change graphs*.

Definition 8 (Affine graph) *An affine size-change graph is a size-change graph such that each destination argument is constrained by the source parameters arranged in an affine relation.*

Such a graph is sometimes termed an affine tuple relation, mapping n-tuples to m-tuples constrained by an affine formula.

In an affine size-change graph, the possible relations between a source parameter m and a destination argument m' can be any expression represented by a Presburger formula. For example, $m' \leq 2m - 4$ is a valid relation in an affine size-change graph. The syntax introduced previously will be used:

$$\{[\bar{x}] \rightarrow [\bar{y}] : \mathcal{P}\}$$

where $\mathcal{P}$ is the Presburger formula.

Definition 9 *Affine graph composition $\circ_F$ is just the affine tuple relation composition defined in Section 2.7.2.*

The LJB size-change graphs described in [LJBA01] may be translated to the affine size-change graph form. A definition of the LJB-graph composition is provided in terms of operations on affine relations.

Given two lists of source parameters and destination arguments, consider a set $\mathcal{G}$ and a set $\mathcal{F}$ of (respectively) all possible LJB-graphs and affine graphs generated from these lists. A translation that maps LJB-graphs to affine graphs is defined as follows:

Definition 10 ($g2a$ Translation) *For each edge e_i in $g \in \mathcal{G}$, translation $g2a :: \mathcal{G} \rightarrow \mathcal{F}$ produces an affine relation r_i according to the following translation:*

e_i		r_i
$m \stackrel{=}{\rightarrow} n$	$\mapsto$	$n = m$
$m \stackrel{\downarrow}{\rightarrow} n$	$\mapsto$	$n < m$
$m \stackrel{\stackrel{=}{\downarrow}}{\rightarrow} n$	$\mapsto$	$n \leq m$

In addition, constraints are associated with each source parameter x_j and destination argument y_k ($x_j \geq 0$ and $y_k \geq 0$ respectively), giving:

$$\begin{aligned} g2a(g) &= \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : (\bigwedge_i r_i) \wedge \mathcal{D}\} \\ \mathcal{D} &= (\bigwedge_j x_j \geq 0) \wedge (\bigwedge_k y_k \geq 0) \end{aligned}$$

Note that edges with the `unknown` symbol are not mapped to any constraint, since the symbol implies no knowledge about how the parameter size is changed. As before, the relation $\mathcal{D}$ specifies the *boundary constraints*, and asserts that all parameters take non-negative values. Also note that $g2a$ is an injection, and that as a result, for an affine graph h containing only affine relations of the form r_i, $g2a^{-1}$ exists.

In order to show the correctness of the translation, an *abstraction function* is defined from affine graphs to LJB-graphs.

Definition 11 (Abstraction $a2g$) *The abstraction $a2g : \mathcal{F} \to \mathcal{G}$ is defined as follows:*

$$a2g(a) = g2a^{-1}(\pi(a))$$

where π is defined as follows:

$$\begin{array}{rcl} \pi(a) & = & \{[x_1, \ldots, x_m] \to [y_1, \ldots, y_n] : r \wedge \mathcal{D}\}; \\ r & = & \{\bigwedge(y_j \; op \; x_i) \;\mid\; a \subseteq P(x_i, y_j, op), \; 1 \le i \le m, \\ & & \qquad 1 \le j \le n, \; op \in \{=, <, \le\}\}; \\ P(u, v, op) & = & \{[x_1, \ldots, x_m] \to [y_1, \ldots, y_n] : v \; op \; u \wedge \mathcal{D}\}; \end{array}$$

Note that the result returned is the smallest in G satisfying a.

The function π is composed of projection functions that project a graph a onto a rectangular polygon enclosing a. This polygon is made up of affine relations of the following forms: $y \; op \; x$ where y is a destination argument, x is a source parameter, and op is either $=$, $<$, or $\le$. In addition, all source parameters and destination arguments are constrained to be non-negative. It is easy to show that the function π, and consequently $a2g$, are monotone. An example of the use of the function π is

$$\pi(\{[x, y] \to [x', y'] : x' = x + 2 \wedge x' = y - 1\}) = \{[x, y] \to [x', y'] : x' < y\}$$

where the π function retains exactly the information used in LJB-analysis.

Function π enjoys the following properties:

1. $id_a \subseteq \pi$.
2. It is *idempotent*: $\pi \circ \pi = \pi$.
3. For any a_1, $a_2 \in \mathcal{F}$,

   $$\pi(a_1 \circ_F a_2) \subseteq \pi(a_1) \circ_F \pi(a_2).$$

It is now possible to provide a definition of LJB-graph composition in terms of affine-graph operations, and give a characterization of the idempotent LJB-graphs using affine graphs.

Lemma 1 (LJB-graph composition $\circ_G$) *For all $g_1,\ g_2 \in \mathcal{G}$,*

$$g_1 \circ_G g_2 = a2g(g2a(g_1) \circ_F g2a(g_2))$$

Proof: The $g2a$ translation is just an encoding of LJB size-change graphs in affine tuple relation form, and from the definition of $\circ_F$, and assuming the graphs are composable, then $g2a(g_1) \circ_F g2a(g_2)$ contains the set of points

$$\{(\bar{x},\bar{z}) \mid \exists\, \bar{y} : (\bar{x},\bar{y}) \in g_1 \wedge (\bar{y},\bar{z}) \in g_2\}$$

Now consider each constituent element $x_i\, \mathrm{op}_1\, y_j$, $y_j\, \mathrm{op}_2\, z_k$ in $g2a(g_1) \circ_F g2a(g_2)$. These elements combine to form a new element $x_i\, \mathrm{op}\, z_k$ where op takes the weakest of the operators op_1 and op_2 (drawn from $\{=, \leq, <, \mathrm{unknown}\}$), and thus the resultant composed affine graph is an encoding of the expected LJB size-change graph $g_1 \circ_G g_2$. It remains to show that for any LJB size-change graph, the $a2g$ operation leaves it unchanged, by noting that in this case π is the identity function. □

Functions $a2g$ and $g2a$ are "tightly" related, in the following sense:

Let id_F be the identity function on $\mathcal{F}$, and id_G that of $\mathcal{G}$. The following holds:

$$\begin{array}{lcl} g2a \circ a2g & \subseteq & id_F \\ a2g \circ g2a & = & id_G \end{array}$$

The following theorem relates the idempotent LJB-graph to affine graphs:

Theorem 2 (Idempotent Graphs) *Let $g \in \mathcal{G}$ and $a = g2a(g)$.*

$$g \circ_G g \;=\; g \text{ if and only if } \pi(a) = a \text{ and } a = a^+.$$

Proof: ($\Rightarrow$) Suppose $g \circ_G g = g$, we show that $\pi(a) = a$ and $a = a^+$, where $a = g2a(g)$.

To show that $\pi(a) = a$, note that a comprises a constraint of the form $y_j\ op\ x_i$, which is identical to $P(x_i, y_j, op)$, the constituents of the result of applying π to any graph. Consequently, applying π on a will not modify any of a's constraints. Thus, $\pi(a) = a$.

To show that $a = a^+$, the approach is to show that $a = \bigcup_{i>0} a^i$ using the subset inclusion relation. Since $\bigcup_{i>0} a^i = a \cup \bigcup_{i>0} a^{i+1}$, thus $a \subseteq \bigcup_{i>0} a^i$. To show the other way round, the assumption that $g \circ_G g = g$ is used, giving that $\pi(a \circ_F a) = a$:

$$\begin{array}{lll} \multicolumn{2}{l}{a2g\ (g2a(g) \circ_F g2a(g)) = g} & \\ \Leftrightarrow & g2a^{-1}(\pi(a \circ_F a)) = g2a^{-1}(\pi(a)) & [\text{Definition of } g2a] \\ \Leftrightarrow & g2a^{-1}(\pi(a \circ_F a)) = g2a^{-1}(a) & [\pi(a) = a] \\ \Leftrightarrow & \pi(a \circ_F a) = a & [g2a^{-1} \text{ injective over range of } \pi] \end{array}$$

To show by induction that $a^i \subseteq a$ for all $i > 0$:

Case $i = 1$, clearly $a \subseteq a$.

Inductive case: Assume that $a^n \subseteq a$, then

$$\begin{array}{rcll} a^{n+1} & = & a^n \circ_F a & \\ & \subseteq & \pi(a^n \circ_F a) & [\mathit{property}\ 3.1.1] \\ & \subseteq & \pi(a^n) \circ_F \pi(a) & [\mathit{property}\ 3.1.3] \\ & \subseteq & a \circ_F \pi(a) & [\text{Induction hypothesis}] \\ & = & a \circ_F a & \\ & \subseteq & \pi(a \circ_F a) & [\text{property } 3.1.1] \\ & \subseteq & a & [\text{assumption}] \end{array}$$

Hence a is a upper bound of ${a^i}_{i>0}$, and therefore $\bigcup_{i>0} a^i \subseteq a$.

Thus, $a = a^+$.

($\Leftarrow$) Given that $a = g2a(g)$. Assume that $\pi(a) = a$, and $a = a^+$, the approach is to show that $g \circ_G g = g$. Equivalently, to show that $\pi(a \circ_F a) = a$.

Since $a = a^+$, by the property of transitive closure, $a \circ_F a = a$. So, it is necessary to show that $\pi(a) = a$, but this is the assumption for this case. □

3.2 Affine-based termination analysis

The affine-based termination analysis is computed in a similar fashion to the LJB-analysis. The analysis begins with the set of affine graphs representing parameter size-change for each function call in the program. Consequently, arguments' size changes are captured during the analysis for all possible call sequences. This change is computed by composing the existing affine size-change graphs in all the legitimate combinations.

The Omega library [KMP+96] can calculate the composition of affine relations efficiently, and assist in the calculation of the closure for a set of such relations.

For example, consider the following program:

```
f(m) = if m ≤ 0 then
          1
        else
          g_a(m + 1);
g(n) = if n ≤ 0 then
          1
        else
          f_a(n − 2);
```

The corresponding affine size-change graphs for the call to function g in f (let's label it g_a), and the call to function f in g (label it f_a), are encoded with two affine relations using the Omega library representation as follows:

$$g_a = \{[m] \rightarrow [m'] : m' = m + 1 \wedge \mathcal{D}\}$$
$$f_a = \{[n] \rightarrow [n'] : n' = n - 2 \wedge \mathcal{D}\}$$

In each of these representations, the source parameters and destination arguments have one member each, constrained by a relation expressed as a Presburger formula. The first expresses that the destination m' must be one more than the source m. The second expresses that the destination n' must be two less than the source n. From this it can be seen that the information about size reduction is captured, and also the size of parameter changes, and perhaps other more subtle relationships. There already exist well-documented ways for extracting affine relations from a program for other purposes. For example, [KS02] describes a contextual analysis to retrieve such information using sized types.

3.2.1 Associating affine with abstract graphs

A crucial administrative task in ensuring termination of the analysis is the association of each affine graph with an abstract graph. This could be viewed as a process of *classifying* the affine size-change graphs. The elements of the abstract size-change graphs provide the different classifications, and the affine graphs provide concrete instances of some of these classifications. An abstract graph is defined as follows:

Definition 12 (Abstract graph) *An affine size-change graph is called an abstract graph if all its parameters are non-negative, and each of its destination parameters y_i can only be related to the source parameters x_j in a simple relation y_i op x_j, where op $\in \{\leq, <, =, \geq, >\}$. Moreover, there is no other affine relation among the parameters.*

The set of affine graphs produced from LJB-graphs via the translation $g2a$ is a set of abstract graphs. To obtain a more accurate termination analysis, this set of abstract graphs must be extended to include those of which the destination parameter can have size greater than some source parameters.

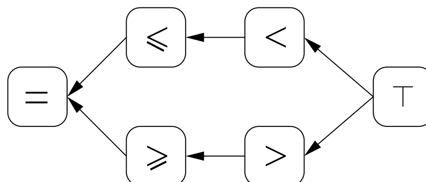


Figure 3.2: Ordering of the operators for Abstract graph composition

Graph composition for abstract graphs is similar to LJB graph composition, except that the new operator will be the least upper bound of the other operators, using the

ordering of operators in Figure 3.2. For example, if we have $o \xrightarrow{>} y$ and $y \xrightarrow{\geq} o'$, the final operator is $o \xrightarrow{>} o'$, but if we had $o \xrightarrow{>} y$ and $y \xrightarrow{\leq} o'$, the final operator would be $o \xrightarrow{\top} o'$ (i.e. the relation is unknown).

Given a program, it is possible to generate all its possible abstract graphs $\mathcal{A}$ representing the parameter size changes between the function entry and call points. Let $\mathcal{F}$ be the corresponding affine graphs that can be realized from the program. Each affine graph may then be associated with an abstract graph in $\mathcal{A}$, as follows:

$$\forall f \in \mathcal{F}, a \in \mathcal{A}, \text{ associate}(f,a) \stackrel{\text{def}}{=} a = \bigcap_i \{r_i \mid f \subseteq r_i,\ r_i \in \mathcal{A}\}$$

where $\phi_1 \cap \phi_2 \stackrel{def}{=} \{(\bar{x},\bar{y}) \mid (\bar{x},\bar{y}) \in \phi_1 \wedge (\bar{x},\bar{y}) \in \phi_2\}$. If two graphs $\langle f, a\rangle$ are associated in this way, then f is *contained* in a. The associated abstract graph a thus obtained is minimum, in that any other abstract graph that contains the affine graph f will also contain a. Consequently, it is called the *minimum association*.

It is important to point out that, in the implementation of the algorithm, not just the association of graphs is maintained, but also (approximated) information about call sequences leading to the creation of that particular graph. This call-sequence information is used to see if a particular graph composition is legal.

Consider the example given on page 46. The minimal set of possible abstract graphs A for the program is $A = \{A_1, A_2, \ldots, A_{12}\}$, given in Table 3.1.

For the label g_a:		
A_1	$=$	$\{[m] \to [m'] : m' < m\}$
A_2	$=$	$\{[m] \to [m'] : m' \leq m\}$
A_3	$=$	$\{[m] \to [m'] : m' = m\}$
A_4	$=$	$\{[m] \to [m'] : m' \geq m\}$
A_5	$=$	$\{[m] \to [m'] : m' > m\}$
A_6	$=$	$\{[m] \to [m']\}$

For the label f_a:		
A_7	$=$	$\{[n] \to [n'] : n' < n\}$
A_8	$=$	$\{[n] \to [n'] : n' \leq n\}$
A_9	$=$	$\{[n] \to [n'] : n' = n\}$
A_{10}	$=$	$\{[n] \to [n'] : n' \geq n\}$
A_{11}	$=$	$\{[n] \to [n'] : n' > n\}$
A_{12}	$=$	$\{[n] \to [n']\}$

Table 3.1: Abstract graphs for the example program

In the table, for clarity, the boundary constraints $\mathcal{D}$ are omitted. The initial set of affine graphs for the functions is $F = \{F_1, F_2\}$ where:

$$\begin{aligned} F_1 &= \{[m] \to [m'] : m' = m+1\} \\ F_2 &= \{[n] \to [n'] : n' = n-2\} \end{aligned}$$

The elements F_1 and F_2 are concrete instances of the elements A_5 and A_7 of A, as

$$\begin{aligned} \{[m] \to [m'] : m' = m+1\} &\subseteq \{[m] \to [m'] : m' > m\} \\ \{[n] \to [n'] : n' = n-2\} &\subseteq \{[n] \to [n'] : n' < n\} \end{aligned}$$

and each affine graph is associated with its classification using the set of pairs

$$\mathcal{C} = \{(F_1, A_5), (F_2, A_7)\}$$

The process of *classifying* the affine size-change graphs using the abstract size-change graphs, leads to a finite number of affine size-change graphs. This property is used in the proof of termination of the termination analysis algorithm.

3.2.2 Affine-based closure algorithm

The core of the termination analysis is the algorithm $\mathcal{T}$:

```
Classify initial affine graphs into C';
C := ∅;
while C' ≠ C do {
    C := C';
    F' := generate(C);
    foreach g ∈ F' {
       (r,A_g) := classify(g,C');
       if idempotent(g,A_g) then
           g := g⁺;
       g := ▽(hull(r ∪ g));
       if r =⊥ then
          C' := C' ∪ (g,A_g)
       else
          C' := (C'\(r,A_g)) ∪ (g,A_g);
    }
}
```

The algorithm $\mathcal{T}$ builds the closure of the new set of affine size-change graphs F, and uses a simple technique, constructing the compositions of the existing affine size-change graphs until no new affine graphs are created.

The main idea of the algorithm, ignoring the termination issues, can be described using the following metaphor: recall that each affine graph can be associated with a minimum abstract graph. Imagine each abstract graph (there are a finite number of them) as a container, which will contain those affine graphs under its minimal association. The containers will have a cap labeled with the corresponding abstract graph. Thus, some containers will be considered as idempotent containers when they are labeled/capped with an idempotent abstract graph.

Initially, only the initial set of affine graphs will be kept in some of the containers. At each iteration of the algorithm, a composition operation will be performed among all legitimate pairs of affine graphs, including self-composition. The resulting set of

affine graphs will again be placed in the respective containers they are (minimally) associated with. Assume that such an iteration process will eventually terminate, with no more new affine graphs created. Non-empty idempotent containers are then identified, checking their cap to see if the idempotent abstract graphs labeled therein contain a decreasing edge. If all these idempotent abstract graphs have decreasing edges, then it can be concluded that the associated program terminates. If one of these graphs does not have a decreasing edge, then it can be concluded that the associated program does not belong to the size-change terminating programs.

Suppose each affine size-change graph is associated with its classification, using the set of pairs $\mathcal{C} = \{(F_1, A_j), (F_2, A_k), \ldots\}$. This set grows in size as the algorithm runs, but has a maximum size determined by the size of $\mathcal{A}$. The function `classify` returns a pair (r, A_k), with r a null graph[1] if this classification has not been made before, or with the previous value for the affine size-change graph if this classification has been made before.

$$\text{classify}(g, C) \stackrel{\text{def}}{=} \begin{cases} (r, A_g) & \text{if } (r, A_g) \in C \wedge \triangledown(g) \subseteq \triangledown(r) \\ (\bot, A_g) & \text{otherwise} \end{cases}$$

The function $\triangledown$ performs a widening operation to produce an affine graph that is larger than the argument graph. This is a common technique used in ensuring finite generation of abstract values. The idea of using a widening operator to control termination is not new, and can be found in (for example) [CH78, NNH99]. Widening operators normally have two operands, for example $\triangledown(d_1, d_2)$, which widens d_2 with respect to d_1, keeping the constraints in d_2 that are already in d_1. However, for this context, the first operand is just the abstract graph, and since this can be derived from the associated affine graph (using $g2a$), it has been elided.

The function `generate` returns a new set consisting of all the original affine size-change graphs, and any new ones constructed by composing any possible pairs of existing size-change graphs. Given a set of classifications C, the function `generate` can extract the set of current affine graphs $F = \{g \mid (g, A) \in C\}$, and then return all legitimate compositions:

$$\text{generate}(C) \stackrel{\text{def}}{=} \{g_1 \circ_F g_2 \mid (g_1, g_2) \in (F \times F) \wedge \text{legitimate}(g_1, g_2)\} \cup F$$

Note that the legitimate compositions only include those which result in a legitimate call sequence, as is the case for LJB-analysis. In addition, the function may be deemed to be inefficient, as it can generate graphs that identify the same cycle; for example $(g \to f) \circ (f \to g)$ and $(f \to g) \circ (g \to f)$, starting at different points in the same cycle. However, both graphs must be retained, as later compositions may need to use either one of the cycles. At this stage the details of the legitimate test is deferred, as it is explored in more detail in Chapter 4. In the event that a graph g is idempotent, the transitive closure g^+ of the graph is kept, reflecting the idea that any idempotent graph may result in an infinite series of calls through itself.

[1] A null graph has all relations unknown, and the notation $\bot$ is used for such a graph. Note that $\bot \cup g = g$.

The function `idempotent` checks if a graph g and its self composition $g \circ_F g$ are both minimally associated with the same abstract graph A_g:

$$\text{idempotent}(g, A) \stackrel{\text{def}}{=} \text{associate}(g, A) \land \text{associate}(g \circ_F g, A)$$

The algorithm maintains at most *one* affine graph in each container. When a new graph is found to belong to an existing non-empty container, it is combined with the existing graph in the container, using the union, hull, and widening operations. Finally, the algorithm's main structure $\mathcal{C}$ is continually updated into $\mathcal{C}'$ until it reaches a fixed point.

3.2.3 A sample widening operator

In order to guarantee termination for the algorithm, a widening operation is used, to guarantee the stabilization of the affine graph in that container under graph composition. The particular widening operator outlined here is an exhaustive one, which retains as much information as possible from the container of affine graphs. Note that a simpler widening may do just as well for many termination problems.

To better understand how the widening operator $\bigtriangledown$ is defined, first note that the affine constraints in an affine graph can be divided into three sets: the boundary constraints ($\mathcal{D}$), the contextual constraints ($\mathcal{K}$) for the context in which the call represented by the affine graph is called, noting that there are no destination arguments in these constraints, and the output constraints ($\mathcal{E}$) which constrain the destination arguments by other variables, such as source parameters.

Furthermore, a constraint over a destination argument y_j can be expressed as

$$y_j \ op \ \Sigma_k a_k x_k - \Sigma_l b_l x'_l + c$$

where $op \in \{=, <, >, \leq, \geq\}$, x_k, x'_l are source parameters, $a_k, b_l \geq 0$ and c is any integer.

Given that the affine relations in an affine graph a can be divided into three sets of constraints, $\mathcal{D}$, $\mathcal{K}$, and $\mathcal{E}$, as described above.

Definition 13 *The widening operator $\bigtriangledown$ applying over a is defined as:*

$$\bigtriangledown(a) = \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : (\textstyle\bigwedge_i E(r_i)) \land \mathcal{D} \land (\bigwedge_j K(r'_j)) \mid r_i \in \mathcal{E}, r'_j \in \mathcal{K}\}$$

where $E(r_i)$ is a function defined in Table 3.2 (assuming $c > 0$).

and $K(r'_j)$ is a function defined in Table 3.3(assuming $c > 0$, and m and n are arbitrary variables).

Form of r_i	**Conditions**	$E(r_i)$
$y\ op\ x+c$	$op \in \{=,>,\geq\}$	$y\ op\ x+c$
$y=\Sigma_k a_k x_k + c$	$a_k>0, k>1$	$\bigwedge_k y \geq a_k x_k + c$
$y>\Sigma_k a_k x_k + c$	$a_k>0, k>1$	$\bigwedge_k y > a_k x_k + c$
$y\geq\Sigma_k a_k x_k + c$	$a_k>0, k>1$	$\bigwedge_k y \geq a_k x_k + c$
$y\ op\ x-c$	$op \in \{=,<,\leq\}$	$y\ op\ x-c$
$y=x-(\Sigma_l b_l x_l)-c$	$b_l>0, l>0$	$y \leq x-c$
$y<x-(\Sigma_l b_l x_l)-c$	$b_l>0, l>0$	$y < x-c$
$y\leq x-(\Sigma_l b_l x_l)-c$	$b_l>0, l>0$	$y \leq x-c$
all other forms		true

Table 3.2: Definition of $E(r_i)$

Form of r'_j	**Conditions**	$K(r'_j)$
$m\ op\ n$	$op \in \{=,<,>,\leq,\geq\}$	$m\ op\ n$
$m\ op\ c$	$op \in \{=,>,\geq\}$	$m\ op\ c$
all other forms		true

Table 3.3: Definition of $K(r'_j)$

Imagine the mn-tuple relations in $m+n$-space. For example if a source tuple had two parameters (x,y) and the destination arguments just one (z) then one could imagine a 3-space, with the axes x,y and z. The widening operation projects an arbitrary affine relation onto each of the two axes x and y. A relation such as $z=x+y+4$ would be widened to

$$z \geq x+4 \wedge z \geq y+4$$

The widening operator $\bigtriangledown$ defined on page 51 is monotone and idempotent. Furthermore, for any $a \in \mathcal{F}$, $\bigtriangledown(a) \subseteq \pi(a)$.

In addition to being monotone, $\bigtriangledown$ also ensures the generation of a finite number of affine graphs which are less precise than the initial one. For instance, if an application of $\bigtriangledown$ produces an affine graph having the following constraint:

$$y \geq 5x_1 + 2x_2 + 3$$

then, there are only a finite number of constraints which are of identical form

$$y \geq \Sigma_k a_k x_k + c$$

and are less precise, namely:

$$y \geq d_1 x_1 + d_2 x_2 + d_3$$

where $0 < d_1 < 5$, $0 < d_2 < 2$ and $0 < d_3 < 3$. This finiteness in the number of less precise constraints guarantees that iterative computation of affine graphs will stabilize in a finite number of steps.

An affine graph of the following form is said to be in stable form:

$$\{[x_1,\ldots,x_m] \to [y_1,\ldots,y_n] : \boldsymbol{Er} \wedge \boldsymbol{D} \wedge \boldsymbol{Kr}\}$$

where $\boldsymbol{Er}$ and $\boldsymbol{Kr}$ are conjunctions of constraints of the forms described by the range of functions $\boldsymbol{E}$ and $\boldsymbol{K}$ respectively.

Given an affine graph g defined in a stable form, there are a finite number of affine graphs of identical form which are less precise than g.

3.3 Properties of affine-based termination analysis

The termination, accuracy and correctness of the analysis algorithm are briefly discussed below.

3.3.1 Termination of the analysis algorithm

The key components of the analysis algorithm ensure that the algorithm terminates. The following proof uses the finite cardinality of the size-change graphs, and the monotonic nature of the graph operations.

Theorem 3 *The algorithm $\mathcal{T}$ terminates.*

Proof: The algorithm terminates when $\mathcal{C}'$ is no longer changing. $\mathcal{C}'$ can change in only two ways, either by *creating* an association for a new abstract graph A_g, or by *replacing* an existing set element (r, A_g) with a new (g', A_g). Proof of termination is done here by showing that neither of these actions can be done an infinite number of times.

Addressing the first point, the cardinality of the set $\mathcal{C}'$ is finite, bounded by the cardinality of the set of abstract graphs $\mathcal{A}$. As a result of this, $\mathcal{C}'$ cannot continue increasing in size forever, and hence we cannot keep adding in a new association (g', A_g).

Addressing the second point, consider the replacement of an existing set element (r, A_g) with a new set element (g', A_g). From the algorithm, we know that

$$g' = \bigtriangledown(\text{hull}(r \cup g))$$

Thus g' is in stable form. Since any update of g' (in further iterations of the algorithm) will result in an affine graph, g'', which cannot be more precise than g' (by Property 3), and the application of $\bigtriangledown$ ensures that g'' is also in stable form, by Property 4, there exists a finite number of iterations in which the update of the affine graph will stabilize. This completes the termination proof. $\square$

3.3.2 Accuracy of the analysis algorithm

Having established that the analysis algorithm terminates, it is easy to verify that the analysis computes more accurate information than the LJB-analysis.

To see that, drop all the contextual information in the initial affine graphs collected, by making all such contextual information true. Then replace the widening operator $\bigtriangledown$ in the analysis by π defined in Definition 11. This turns those affine graphs, which are associated with idempotent abstract graphs, into their respective abstract graphs during the analysis. With these changes, the analysis mimics the computation of LJB-analysis.

3.3.3 Correctness of the analysis algorithm

The correctness argument for SCT is presented in [LJBA01]. The overall strategy used for analyzing a program is to construct an abstract representation of the program, such that a property of the abstract representation captures the termination of the original program. The term *correctness* here requires that the abstract representation must conservatively[2] represent the behaviour of the original program. The abstract representation in SCT, is a set of idempotent SCGs which conservatively record the change of sizes of program parameters at all possible program points which could lead to infinite cyclic loops. The property of interest is that in each of these SCGs, at least one of the parameters must reduce, and if this property holds, then the corresponding program must terminate.

The abstract representation in affine SCT, is a set of idempotent affine SCGs which conservatively record the change of sizes of program parameters at all possible program points which could lead to infinite cyclic loops. The key components of the analysis algorithm ensure that the algorithm computes correctly, that is, that the final idempotent graphs represent the possible affine size-change information for a superset of all possible candidate infinite program traces. This can be confirmed by noting that the closure algorithm generates all possible idempotent (cyclic) graph compositions, and that at each stage a conservative graph is retained.

To show that the analysis computes conservative size-change information, at this stage the notion of *precise* tuple relations is introduced. Consider the *precise* tuple relations $f = \{[\bar{x}] \rightarrow [\bar{y}]\}$ and $g = \{[\bar{y}] \rightarrow [\bar{z}]\}$, representing *exactly* the realizable values of source and destination parameter-vectors for successive calls at location f and g, the affine relations $\boldsymbol{F_f}$ and $\boldsymbol{F_g}$ derived from each of them, and the composition $f \circ g$ meaning "the effect of successively calling f and then g".

Definition 14 *Precise tuple relation composition is affine tuple relation composition:*

$$\phi_1 \circ \phi_2 \stackrel{def}{=} \phi_1 \circ_F \phi_2$$

[2]In this context, *conservative* means that if a property holds for the conservative representation, then it must also hold for the original program.

By appealing to the interpretation of affine relations as sets of pairs of program parameter-vector values, we have that $f \subseteq F_f$ and $g \subseteq F_g$. For example, if we had a function with the precise relation $f = \{[x] \to [y] : y = x - 2\}$ an associated affine graph might be $F_f = \{[x] \to [y] : y \leq x - 2\}$, and $f \subseteq F_f$. In short, F_f is a *safe* (or conservative) approximation to the program, as whenever a pair of program parameter-vector values $(\bar{x}, \bar{y})$ is in f, it is also in F_f.

At the time the initial affine graphs are derived from the program source, the graphs are safe (by construction) approximations to the *precise* representation of the program. As long as the operations performed on the affine graphs retain this safe approximation property, the analysis is deemed *correct*. The significant operations are those of affine graph composition, and the closure and widening operations. The affine graph composition may be visualized as in Figure 3.3, where we see a diagram which shows the expected relationships between f, g, F_f and F_g. We have to prove that the affine composition $F_f \circ_{\mathrm{F}} F_g$ is still a safe approximation to $f \circ g$.

$$\begin{array}{ccccc} F_f & \circ_F & F_g & \longrightarrow & F_f \circ_F F_g \\ \cup\!\!| & & \cup\!\!| & & \cup\!\!| \\ f & \circ & g & \longrightarrow & f \circ g \end{array}$$

Figure 3.3: Relationships between f, g, F_f and F_g

Theorem 4 ***(Safe approximation for*** $\circ_F$***)*** *If f and g are precise tuple relations representing exactly the realizable values of source and destination parameters, and F_f and F_g are affine relations derived from each of them such that $f \subseteq F_f$ and $g \subseteq F_g$, then $f \circ g \subseteq F_f \circ_{\mathrm{F}} F_g$.*

Proof: From definition 14 we have that $f \circ g = f \circ_F g$. The proof that $f \circ_F g \subseteq F_f \circ_{\mathrm{F}} F_g$ comes directly from the monotonicity of affine composition:

$$\begin{array}{rl} & f \subseteq F_f \\ \Rightarrow & f \circ_F g \subseteq F_f \circ_F g \subseteq F_f \circ_F F_g \end{array}$$

and so $f \circ g \subseteq F_f \circ_{\mathrm{F}} F_g$. □

The transitive closure operation ϕ^+ over affine tuple relations is monotone (see Section 2.7.2), and so if $f \subseteq F_f$, then $f^+ \subseteq F_f^+$. As a result F_f^+ is a safe approximation to f^+. The widening operation $\bigtriangledown$ over affine tuple relations is monotone (see Section 3.2.3), and as a result $\bigtriangledown(F_f)$ is a safe approximation to f.

Consequently, all operations performed on the graphs ensure that analysis only generates graphs that are safe approximations to the initial program, and the correctness for affine-SCT is established.

3.4 Examples using affine termination analysis

The three examples given at the beginning of the chapter are revisited. Each one is an example of a program that cannot be tested using SCT analysis, but succumbs to affine SCT analysis.

3.4.1 First example - constant size cancellation

The first example from this chapter is repeated here, to show how constant change cancellation is captured, and how the analysis is done.

```
f(m) = if m ≤ 0 then
           1
         else
           g_a(m + 1);
g(n) = if n ≤ 0 then
           1
         else
           f_a(n − 2);
```

The following two affine size-change graphs are extracted from the source. Note that the constraints $\mathcal{D}$ (which specify that all the parameter values must be greater than, or equal to zero), have been omitted for clarity:

$$\begin{aligned} F_1 &= \{[m] \to [m'] : m' = m + 1\} \\ F_2 &= \{[n] \to [n'] : n' = n - 2\} \end{aligned}$$

The initial value for $\mathcal{C}$ is

$$c_1 = \{(F_1, A_5), (F_2, A_7)\}$$

The first call to `generate` returns the following new affine size-change graphs:

$$\begin{aligned} F_3 &= \{[n] \to [n'] : n' = n - 1\} && \text{(from the sequence } f_a g_a) \\ F_4 &= \{[m] \to [m'] : m' = m - 1\} && \text{(from the sequence } g_a f_a) \end{aligned}$$

Since these are idempotent, calculate the closure of F_3 and F_4:

$$\begin{aligned} F_3' &= \{[n] \to [n'] : n' < n\} \\ F_4' &= \{[m] \to [m'] : m' < m\} \end{aligned}$$

After this first iteration, $\mathcal{C}$ has the value

$$c_2 = \{(F_1, A_5), (F_3', A_7), (F_4', A_1)\}$$

where $A_1 = \{[m] \rightarrow [m'] : m' < m\}$. After a few more iterations there is one more graph, and $\mathcal{C}$ has a stable value. The new graph is:

$$F_5 \quad = \quad \{[m] \rightarrow [n'] : n' \leq m\} \qquad \text{(from the sequence } g_a(f_a g_a)^+\text{)}$$

In summary, the following table shows the initial affine and abstract graphs. Note that the constraints $\mathcal{D}$ have again been omitted for clarity, and the column headed "**Id**" indicates if the graph can be composed with itself, and if so, if the result is idempotent:

Sequence	**Id**	**Initial affine graphs**	**Initial abstract graphs**
g_a	**No**	$F_1 = \{[m] \rightarrow [m'] \quad : m' = m + 1\}$	$\{[m] \rightarrow [m'] \quad : m' > m\}$
f_a	**No**	$F_2 = \{[n] \rightarrow [n'] \quad : n' = n - 2\}$	$\{[n] \rightarrow [n'] \quad : n' < n\}$

These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs through affine-graph composition. These lead to a final classification of affine graphs into containers as follows:

Sequence	**Id**	**Final affine graphs**	**Final abstract graphs**
g_a	**No**	$F_1 = \{[m] \rightarrow [m'] \quad : m' = m + 1\}$	$\{[m] \rightarrow [m'] \quad : m' > m\}$
$(f_a g_a)^+$	**Yes**	$F_3' = \{[n] \rightarrow [n'] \quad : n' < n\}$	$\{[n] \rightarrow [n'] \quad : n' < n\}$
$(g_a f_a)^+$	**Yes**	$F_4' = \{[m] \rightarrow [m'] \quad : m' < m\}$	$\{[m] \rightarrow [m'] \quad : m' < m\}$
$g_a(f_a g_a)^+$	**No**	$F_5 = \{[m] \rightarrow [n'] \quad : n' \leq m\}$	$\{[m] \rightarrow [n'] \quad : n' \leq m\}$

The composition of any pairing of these graphs generates no new affine or abstract graphs. The idempotent functions are F_3' and F_4', which each have a reducing parameter, and so it can be concluded that the size-change termination property applies to this function.

In general, using this technique it is possible to capture termination for all the functions captured by the LJB technique, and also some others. In other words termination is captured for a larger set of programs.

3.4.2 Second example - linear combinations of source parameters

The second example from this chapter is repeated here, to show how linear combinations of source parameters are captured, and how the analysis is done.

```
k(m,n) = if m ≤ 0 then
           1
         else
           k_a(m − n, n + 1);
```

The following affine size-change graph is extracted from the source. Note that the constraints $\mathcal{D}$ have again been omitted for clarity:

$$F_1 = \{[m,n] \to [m',n'] : m' = m - n \wedge n' = n + 1\}$$

Note that $F_1 \subseteq A_1$ but $F_1 \not\subseteq A_2$. The first call to `generate` returns the following new affine size-change graph from the sequence $k_a k_a$:

$$F_1 = \{[m,n] \to [m',n'] : m' = m - 2n - 1 \wedge n' = n - 2 \wedge m > 2n\}$$

Note that $F_2 \subseteq A_1$ and $F_2 \subseteq A_2$, so F_2 is in a different classification (We choose the strongest classification, and $A_2 \subset A_1$). Since F_2 is idempotent, the closure is calculated:

$$F_2' = \{[m,n] \to [m',n'] : m' < m\}$$

After this first iteration, $\mathcal{C}$ has the value

$$c_2 = \{(F_1, A_1), (F_2, A_2)\}$$

There are no more graphs, and $\mathcal{C}$ has a stable value. The following table shows the initial affine and abstract graph:

Initial affine graph	**Initial abstract graph**
$F_1 = \{[m,n] \to [m',n'] \quad : m' = m - n \wedge n' = n + 1\}$	$\{[m,n] \to [m',n'] \quad : m' \leq m\}$

These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs through affine-graph composition. These lead to a final classification of affine graphs into containers as follows:

Final affine graphs	**Final abstract graphs**
$F_1 = \{[m,n] \to [m',n'] \quad : m' = m - n \wedge n' = n + 1\}$	$\{[m,n] \to [m',n'] \quad : m' \leq m\}$
$F_2 = \{[m,n] \to [m',n'] \quad : m' < m\}$	$\{[m,n] \to [m',n'] \quad : m' < m\}$

The composition of any pairing of these graphs generates no new affine or abstract graphs. The idempotent function is F_2, which has a reducing parameter, and so it can be concluded that the size-change termination property applies to this function.

3.4.3 Third example - context analysis

Chin and Khoo [CK00] have explored the use of the Omega calculator to infer other properties of arbitrary functions, and it seems appropriate to extend the termination analysis using this sort of inference. Consider the third example from this chapter:

```
j(m,n) = if m = 0 ∨ n = 0 then
             0
           else
                if m < n then
                   j_a(m − 1, n + 1)
                else
                   j_b(m + 1, n − 1);
```

It is easy to convince yourself that this function terminates, as only the first or the second recursive call to j will be made. However, the LJB size-change termination method cannot be applied, and similarly for the improvement to the LJB method just explored. The composition of the two size-change graphs ($F_3 = F_1 \circ F_2$) will be:

$$\begin{aligned} F_1 &= \{[m,n] \to [m',n'] : m' = m-1 \wedge n' = n+1\} \\ F_2 &= \{[m,n] \to [m',n'] : m' = m+1 \wedge n' = n-1\} \\ F_3 &= \{[m,n] \to [m,n]\} \end{aligned}$$

Termination cannot be inferred from this, as the composition of the two calls has no reducing parameters. However, extra information can be included that we can capture from the code segment. For example, the first call is only performed when $m < n$. The second call is only performed when $m \geq n$. By including this information in the constraining affine relation, the new calculation of the composition is:

$$\begin{aligned} F_1 &= \{[m,n] \to [m',n'] : m' = m-1 \wedge n' = n+1 \wedge m < n\} \\ F_2 &= \{[m,n] \to [m',n'] : m' = m+1 \wedge n' = n-1 \wedge m \geq n\} \\ F_3' &= \{[m,n] \to [m',n'] : \text{False}\} \end{aligned}$$

The result of the calculation of the composition of the functions indicates that this situation cannot occur (i.e. it is not possible for the two calls to occur). These lead to a final classification of affine graphs into containers as follows:

Final affine graphs	**Final abstract graphs**
$F_1' = \{[m,n] \to [m',n'] \quad : m' < m \wedge m < n\}$	$\{[m,n] \to [m',n'] \quad : m' < m\}$
$F_2' = \{[m,n] \to [m',n'] \quad : n' < n \wedge m \geq n\}$	$\{[m,n] \to [m',n'] \quad : n' < n\}$

Each of these is idempotent, and has a reducing parameter, and hence the function terminates. Again, by collecting and keeping more information, termination can be deduced for this function.

3.5 δSCT (Delta size-change termination)

In [BA06], Ben-Amram discusses in detail an approach similar to that adopted in this thesis, extending SCT to work with more refined changes in size. The approach uses constant size changes of parameters, introducing a new form of size-change graph, termed a δSCT graph (delta-S-C-T), which records the *size* of changes $(+1, -3, \ldots)$. SCT is viewed as a restricted form of δSCT and Ben Amram argues that δSCT is more expressive than SCT.

The article proves that the decision problem for δSCT is undecidable by a reduction from a known undecidable problem, but then continues by suggesting another restriction. This new (conservative) restricted set of graphs (the fan-in-free δSCT graphs) is more expressive than SCT, but is similarly decidable. The author proves that it is PSPACE-complete, and gives a closure-based algorithm for determining if a set of graphs is δSCT

The paper formally defines an annotated CFG (ACG) and the size-change graph (SCG), abstract states and state-transition sequences, safety of SCGs (conservative abstractions), multipaths and threads in a multipath, along with *infinite-descent* and the δSCT condition. Having defined these, a theorem is developed which asserts that if an ACG is safe and satisfies δSCT, then no reachable state-transition sequence can be infinite.

The δSCT approach is satisfying in that it is a graph-theoretic result, similar to the original LJB approach, and amenable to complexity (and decidability) analysis. By contrast the affine SCT approach relies on widening operators for termination of the algorithm, and on the decidability of Presburger arithmetic.

Both approaches are more expressive than SCT, but are not related otherwise - it is possible to construct examples that are δSCT, but not affine-SCT, and it is also possible to construct examples that are affine-SCT and not δSCT. In the two examples below, the function to the left is affine SCT, but not δSCT. The function to the right is δSCT, but not affine SCT.

```
k(m,n) = if ... then
            ...
         else
            k(m - n, n + 1);
```

```
k(m,n) = if  ... then
             ...
         else
           if  ... then
              k(m - 2, n + 1)
           else
              k(m + 1, n - 1);
```

Another significant difference between the two approaches is that the affine approach allows us to include in other information about the program (for example the conditional tests). This feature has proven useful in the analysis of many functions.

3.6 Commentary

This section presented an approach to improve the analysis of program termination properties based on the size-change termination method. Size-change graphs were encoded using Presburger formulæ representing affine relations, leading to more refined size-change graphs admissible by these affine relations. The algorithm for calculating the closure of the affine size-change graphs has been shown to terminate. Consequently, this affine-related analysis improves the effectiveness of the LJB termination analysis by capturing constant changes in parameter sizes, and affine relationships of the sizes of the source parameters. The approach widens the set of functions that are size-change terminating.

The way in which closures are found is different from the normal approach of finding closures and fixed points. Conventionally, program analysis will attempt to find a closure for each function definition, such as those found in [CK00] and in the work on termination done for the Mercury system [SSS97], and many works on program analysis, for example [Hei92, Aik99, CH78].

In this approach, the ultimate closure is expressed in terms of several closures called the idempotent graphs. This is similar to the idea of polyvariant program analysis [VB99, Con93], but differs in that there are also some graphs around during the analysis which cannot be made idempotent, yet are important for closure building. There are two ways to obtain polyvariant information for termination analysis: one way is to make use of the constraint enabling the call. Such constraints are commonly obtained from the conditional tests of the code leading to the call. The use of Presburger formulæ enables such information to be easily included in the analysis, resulting in a context-sensitive analysis [NNH99, Shi88].

Another way to capture polyvariant information is to capture the possible function call sequence, such as a function g can be called from another function f, but not from k. The LJB-analysis uses this information to achieve polyvariant analysis. While this information can be captured in the algorithm (by creating more distinct abstract graphs with call sequence information), it is not presented here. As it is, the termination analysis deals with a set of mutually recursive functions at a time. It would be interesting to investigate the modularity of the analysis, so that each function can be analyzed separately, and the results from various functions be composed to yield the final termination result. One such opportunity is to integrate the affine-based termination analysis with a type-based system.

The use of constraints in expressing argument change enables termination to be considered beyond the *well-founded* method. For example, through constraints, it is possible to express the fact that an increasing argument can be bounded above. This idea has been explored in the work on sized typing in [CK00]. The bounded-increment of an argument is also investigated in [CS02], which computes a linear ranking function for a loop-based program.

Chapter 4

Extending affine size-change termination

Building on the affine SCT approach introduced in Chapter 3, the size-change principle is complemented with a technique which allows the analysis of functions in which the return values are relevant to termination. A second new technique controls the composition of graphs in the algorithm $\mathcal{T}$, so that they more closely mirror the allowed call sequences. A final technique applies to functions in which repeated guarded calls are made, with call arguments that monotonically change and (in some sense) approach the *boundary* of the guard. This final approach can be viewed as an extension of affine size-change termination beyond the original well-founded approach. The analysis techniques exploit the decidability and expressive power of affine constraints. These techniques significantly extend the set of programs that are size-change terminating, allowing analysis of programs in a first order, strict functional language computing over the integers.

4.1 Introduction

In this section, significant extensions and variations to affine SCT analysis are described which exploit the capability of affine constraints and produce a more precise set of composite SCGs. The resulting analysis can handle functions in which return values are relevant to termination, and can also be applied to guarded functions which terminate through exceeding bounds. This is done with:

1. techniques for the construction and evaluation of more refined SCGs, and
2. a new approach to handle function call sequences that repeatedly change a value that approaches, and eventually exceeds, some bound.

These changes are a conservative extension of the affine-graph based size-change termination analysis: any program that previously could be shown to be size-change terminating can still be analysed. However, the new analysis can also show termination for many other functions.

At this stage McCarthy's 91 function [MM70] is introduced. Both LJB and affine-SCT style analyses work for many functions but not for the 91 function. It is different because the returned value of the function is relevant to its termination. For this reason it is used as a vehicle to demonstrate the techniques. The following annotated code shows the 91 function:

```
f91(n) = if n > 100 then
             n − 10_a
           else
             f91_c(f91_b(n + 11));
```

The function has the property that for all $n \leq 101$, the returned value of the function is 91 (and hence the name). Analysis of this function using LJB-analysis is not possible, as the SCG for function call c has no information relating the size of the parameter n and the call argument. The enhanced termination-detection ability is made possible by extending the affine graphs to include a result value to capture the return of function calls at specific affine-graphs, and a sufficient condition for detecting that the change of a call argument is bounded.

4.2 Extended size-change graphs

The proposed extension to SCGs can be visualized as in Figure 4.1. There are two parts: the upper part, called the *base*, is the original SCG, while the lower part is the *extension*. Affine relations may also be specified over the extension. In affine tuple relation form:

Definition 15 *An extended affine SCG is*

$$G' = \{[\overline{x}, \overline{e}] \rightarrow [\overline{y}, \overline{e'}] : (\bigwedge_i r_i) \wedge \mathcal{D} \wedge (\bigwedge_j s_j) \wedge \mathcal{D}'\}$$

where $r_i \stackrel{\text{def}}{=} y_i \operatorname{op} \delta_i(x_1, \ldots, x_m, e_1, \ldots, e_n)$*,* δ *is an affine relation over the input parameters and the extension elements* $\overline{e}$*, and* $\operatorname{op} \in \{\leq, <, =, \geq, >\}$*. The constraints* $\mathcal{D}$ *restrict the elements to be greater than zero:* $x_k \geq 0$ *and* $y_i \geq 0$*. The extension contains elements* e_i*,* e'_i *of* $\overline{e}$*,* $\overline{e'}$*, which are arguments/parameters used to record extra information about the graph, and* $s_j \stackrel{\text{def}}{=} e'_j \operatorname{op} \delta_j(x_1, \ldots, x_m, e_1, \ldots, e_n)$*. As for the affine SCGs, the constraints* $\mathcal{D}'$ *restrict the elements to be greater than zero:* $e_j \geq 0$ *and* $e'_j \geq 0$*.*

Note that this extended affine graph is still an ordinary affine graph, and does not affect the termination analysis method developed in Chapter 3.

4.2.1 Capturing function return values: Value SCGs

Since the language used in this thesis is a functional language, it seems appropriate to capture the returned values of functions. Sometimes these returned values are required for termination analysis (as in the $f91$ function). The first extension proposed here is to add in a global *result* value for function calls to the extension to the SCG.

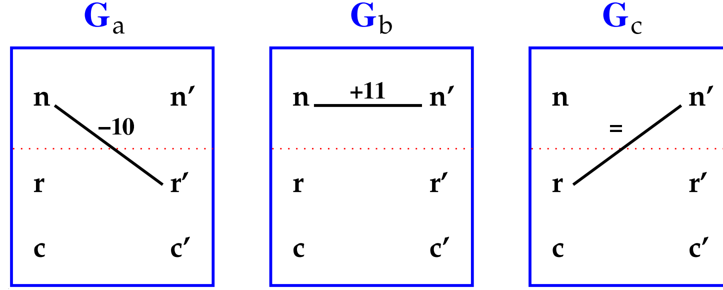


Figure 4.1: The *value* and *call* SCGs for the $f91$ function

To reflect the effect of a function result value on a following call, the return of function call is represented by an affine graph, called a *value* graph. The original size-change graphs are called *call* graphs. As an example, the value graph, G_a, for the $f91$ function is shown in Figure 4.1, signifying the return of value $n-10$ from a call.

Accompanying G_a are two *call* graphs available in the $f91$ function: G_b and G_c signifying two recursive calls in the function definition. Note that the graph G_c indicates a transfer of value from the result source (r) to the parameter destination (n). The corresponding affine relations are expressed as follows:

$$
\begin{aligned}
G_a &= \{[n,r,c] \rightarrow [n',r',c'] : r' = n-10\} \\
G_b &= \{[n,r,c] \rightarrow [n',r',c'] : n' = n+11\} \\
G_c &= \{[n,r,c] \rightarrow [n',r',c'] : n' = r\}
\end{aligned}
$$

The effect of a nested function call using the return value of previous call can be described by composing a value graph with a call graph, as in the case of the $f91$ function call shown in Figure 4.2.

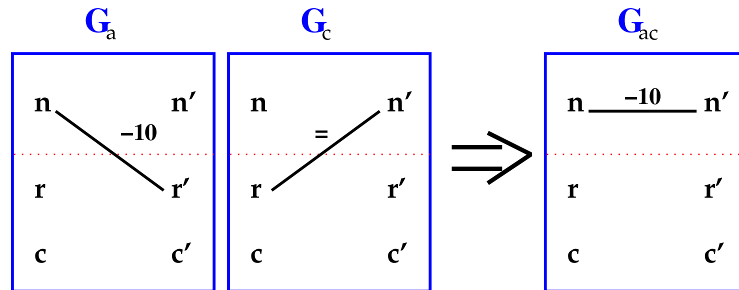


Figure 4.2: Composition of nested call

Note the composed graph shows a decrease in the $f91$'s only argument, n. This reflects the transfer of a value to a call c from some *last* recursive call. The corresponding tuple relation is expressed as follows:

$$G_{ac} = \{[n, r, c] \rightarrow [n', r', c'] : n' = n - 10\}$$

It is worth noticing that only *some*, and not all, of the returns to functions are captured in value graphs. Specifically, only the return of the last call in a series of nested calls is captured. Semantically, returns from a series of nested calls transfer the result from callees back to the callers, and this information is ignored in order to maintain clarity in this exposition. This results in a simpler classification of call sequences, as we will see in the following section.

Value graphs are composed with call graphs to construct new call graphs which are used as the initial call graphs for the closure algorithm $\mathcal{T}$. The value graphs are not used otherwise, and only appear at the beginning of the analysis.

It is possible to simply compose value SCGs with any call graphs, add these into the pool of existing call SCGs, and apply the LJB or affine-SCT analyses on this SCG pool to compute the termination of the $f91$ function. However, the result will not be satisfactory (*ie.*, termination of $f91$ cannot be detected). In addition, the composition of G_a with G_b is problematic, as this sequence (ab) cannot occur in any run of the program.

The analysis is therefore extended to take advantage of other information available from static analysis of the program functions: call sequences and relative numbers of calls.

4.3 Refining the graph composition algorithm

In LJB SCT analysis, the composition operation is applied to any SCGs in a current "pool" of graphs, so given the call graphs G_{ab}, G_{ac}, G_b and G_c, the approach is to consider all possible compositions, corresponding to all possible label sequences (traces) for the program.

Definition 16 *A trace is a sequence of labelled program points.*

The affine SCT algorithm constructs new affine SCGs by composing (labelled) graphs, and (if they are idempotent) taking the transitive closure. The traces can be compactly represented using the RE (regular expression) syntax, with brackets as needed. Given (say) a graph G_x, the trace is recorded as x, unless G_x is idempotent, in which case the closure is calculated, and the trace recorded as x^+. If the new graph is composed with (say) G_y, the new trace is x^+y. If this is idempotent, then the trace

becomes $(x^+y)^+$. It is easy to see that the traces constructed by the algorithm $\mathcal{T}$, and analyzed by affine SCT, form a *regular* language.

Many of these compositions may be inappropriate. For example, in the $f91$ case, it is obvious that a function return will *not* be followed by a call at label b, and so the composition G_{ab} should therefore be ruled out. By limiting the possible ways of composing a value graph with a call graph, it is possible to exclude the generation of graphs representing illegal call sequences.

A representation of the language of the allowed label sequences for LJB SCT would be $L_{\text{SCT}} = (ab \mid ac \mid b \mid c)^+$. When checking if two graphs can be composed, a candidate composition ($G_{ab}G_b$ for example) could be checked to see if the trace is an allowed subsequence: $abb \subseteq L_{\text{SCT}}$. In this case, the language L_{SCT} is very permissive, allowing all subsequences. In the $f91$ example just given, a better language would be $L_{\text{RE}} = (ac \mid b)^*a$, but the problem is how to automatically find a language which is as small as possible, but contains all possible traces of the program.

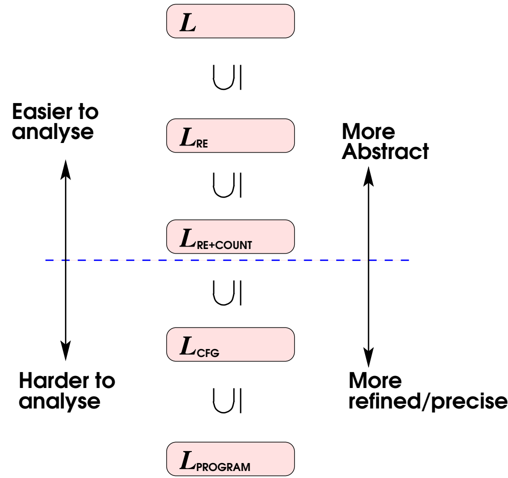


Figure 4.3: A hierarchy of program traces

Figure 4.3 shows a hierarchy of program traces. At the bottom is L_{PROGRAM} the sequence of calls allowed by the actual program - this is the most refined or precise language. If we use a CFG to represent the allowable traces, then we have the less precise, more abstract language L_{CFG} - it allows all the legal traces, but may allow some other traces. The other categories considered in the hierarchy are $L_{\text{RE+COUNT}}$ the regular expression (RE) sub-traces restricted by a counter, L_{RE} the RE sub-traces, and L_{SCT} the method used by traditional SCT which includes traces for all composable functions.

It is useful to construct more accurate traces. If $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ are progressively more abstract languages representing the allowable traces for a sequence of calls for a program P, then $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_3$, and if a (termination) property holds for $\mathcal{L}_3$, then the property will hold for $\mathcal{L}_1$. However, if the property does not hold for the language $\mathcal{L}_3$, then the property could be checked for $\mathcal{L}_2$ and so on. In summary, analysis of

a more refined trace may be slower or more difficult, but it can be used to derive termination properties for a larger set of programs.

4.3.1 Deriving an approximation to the CFG for traces

A CFG corresponding to the call/return traces for a set of functions may be mechanically extracted from the source of the functions, each function f_i generating a grammar rule such as $F_i \rightarrow X \mid Y \mid Z \mid ...$ where $X, Y, \ldots$ represent alternate paths through this function, and each $X, Y, \ldots$ is a term of the form $aAbBcC\ldots$ where $a, b, \ldots$ represent labels for each unique call or return found in a function, and $A, B, \ldots$ are identifiers for the rules for each of the other functions. The $A, B, \ldots$ identifiers represent non-terminals in the expression, and the $a, b, \ldots$ identifiers represent terminals.

The process can be formalized using the rules in Table 4.1.

$$\frac{\{f : F\} \cup \Gamma \vdash e \,:\, E}{\Gamma \vdash f = e \,:\, F \rightarrow E} \quad \text{CFG-def} \qquad \frac{\Gamma \vdash e_a \,:\, E_a \quad \Gamma \vdash e_b \,:\, E_b}{\Gamma \vdash e_a;\; e_b : E_a E_b} \quad \text{CFG-comp}$$

$$\frac{\Gamma \vdash e_a \,:\, E_a \quad \Gamma \vdash e_b \,:\, E_b}{\Gamma \vdash e_a, e_b : E_a E_b} \quad \text{CFG-args} \qquad \frac{\Gamma \vdash e \,:\, E \quad f : F \in \Gamma}{\Gamma \vdash f_a(e) \,:\, EaF} \quad \text{CFG-call}$$

$$\frac{}{\Gamma \vdash e_a \,:\, a} \quad \text{CFG-return} \quad (e_a \text{ is a return expression})$$

$$\frac{\Gamma \vdash b \,:\, B \quad \Gamma \vdash e_a \,:\, E_a \quad \Gamma \vdash e_b \,:\, E_b}{\Gamma \vdash \text{if}\, b \,\text{then}\, e_a \,\text{else}\, e_b \,:\, B(E_a \mid E_b)} \quad \text{CFG-ifthen}$$

Table 4.1: CFG generation rules for single function definition

The environment Γ keeps the association between a function f and its associated non-terminal F. The rules can easily be extended for multiple function definitions, with mutual recursion. Applying the rules to the $f91$ program yields:

$$F \rightarrow (a \mid bFcF)$$

The form of the CFG is a little difficult to relate back to the program. For the purposes of clarity in exposition, the $f91$ function is presented again as a *flattened* recursive procedure using global variables for the returned values for the functions. This flattened version has the same call trace structure, but the form has a clear relation back to the CFG for the function.

After *flattening*, the $f91$ function becomes:

```
global int result;
f91(n) = if n > 100 then
             result := n − 10_a
           else {
             f91_b(n + 11);
             f91_c(result)
           };
```

The flattened syntax reveals the clear relation between the program, and the CFG representing the *trace* of the sequence of labeled calls and returns to the function.

The purpose of constructing trace representations of the program is to control the generation of size-change graphs, allowing us to exclude any graphs which cannot represent any part of the behaviour of the program. This may be done by testing the labels for the two candidate SCGs (say G_{x^+} and G_y) to see if $x^+y \subseteq L_{\text{PROGRAM}}$, where L_{PROGRAM} is the language of realizable traces for the program. If the graphs pass this membership test, they are composable, and the new graph is annotated with the trace: G_{x^+y}. Note that the language of candidate traces is always a regular expression.

The CFG representation for allowable traces is reasonably precise, but not directly used for controlling the generation of size change graphs, as the above membership test is not decidable: *If $\mathcal{L}(g)$ is the language generated by a grammar g, G is a context-free grammar, and R is regular, then $\mathcal{L}(R) \subseteq \mathcal{L}(G)$ is not decidable* (found as a theorem in [HMU01]).

It may be that for specific choices of G and R, the problem is decidable, but since the *general* problem is undecidable, a regular grammar is constructed, approximating the CFG which represents the possibly infinite traces for the program. This must be a superset of the language for the CFG to be useful in our context, and if R_1 and R_2 are regular expressions, then $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$ is decidable and has quadratic complexity [HMU01]. This sort of approximation is described in [Ned97, PW97], and the GRM library [AMR04] is an efficient toolset to create regular approximations to CFLs.

The GRM library toolset takes a CFG such as the $f91$ one $F \rightarrow a \mid bFcF$, and automatically produces the regular grammar $F \rightarrow acF \mid bF \mid a$, corresponding to the traces found in the regular expression $(ac \mid b)^*a$. This is a superset of the allowable set of traces, and is termed L_{RE}. A candidate composition of graphs G_{C} may be efficiently checked to see if it is in this set of traces by checking if $\text{C} \subseteq L_{\text{RE}}$ (quadratic complexity). Note that if the membership test fails, then the candidate trace C is definitely not allowed by the program. Note also that the membership test may succeed sometimes when a candidate is not in L_{PROGRAM}. This means that an illegal graph composition has been accepted.

A set of initial graphs can be mechanically derived by starting with the component value and call graphs, and then applying the membership test to the composition of each value graph with each call graph. Any valid combinations are added to the list of initial graphs. Once this process is complete, any initial graphs with a trace that is not an initial trace of the function are removed, along with any value graphs.

For the $f91$ function the component graphs are G_a, G_b and G_c. The only valid composition of a value graph with a call graph is G_{ac} because $ac \subseteq (ac \mid b)^*a$. Note that $bc \nsubseteq (ac \mid b)^*a$. The initial graphs at this stage are thus G_a, G_b, G_c and G_{ac}. The graph G_c has a trace which is not an initial trace of the function and so is removed. In addition the graph G_a is a value graph and so is removed. Finally the initial affine graphs for $f91$ are G_b and G_{ac}.

The regular grammar $F \rightarrow acF \mid bF \mid a$ is automatically derived, and approximates the CFG for the $f91$ function, corresponding to the traces found in the regular expression $(ac \mid b)^*a$. This language of traces may be used to restrict the composition of new graphs, and closely approximates the call structure in the original program. By contrast, LJB graph composition would consider the language generated by $F \rightarrow bF \mid cF$.

4.3.2 Adding counters to restrict graph composition

One of the identifying characteristics of a CFG (as opposed to a regular grammar), is it's *counting* ability. For example, CFGs can be used to describe the strings a^nb^n (for example ab or $aaabbb$, where the number of a's is the same as the number of b's). The CFG for this is $P \rightarrow \epsilon \mid aPb$. No regular language can describe these strings.

In this thesis the illustrative functional programming language is described in Table 2.6 using the Backus-Naur formalism (BNF), which is a meta-syntax for context-free grammars. As such, CFGs can be used to describe all possible call sequences for the language. However, the method used in the algorithm $\mathcal{T}$ can only construct graphs representing call sequences that form a regular expression, with no *counting* ability.

Counting is important. Consider the $f91$ function (again). The function call f_b occurs before f_c, and so the number of f_b calls is greater than, or equal to the number of calls f_c. In terms of the annotated affine graphs, the number of compositions including G_b is greater than, or equal to the number of compositions including G_{ac}. In terms of the traces, the number of b's is greater than, or equal to the number of ac's ($\#b \geq \#ac$).

The counting behaviour of the CFG can be simulated by adding new parameters, called *counters* (denoted by $c_1 \dots c_n$) in the extended part of each initial graph.

The extended size-change graph in Figure 4.4 has a counter c for the graphs G_b and G_{ac}. Every composition involving G_b increments the destination counter c. On the other hand, every composition involving G_{ac} results in a decrement of the counter. With this arrangement, a composed graph is considered *legal* if it is composed from

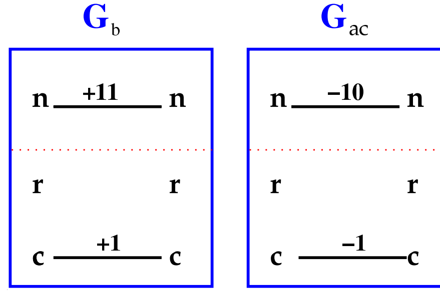


Figure 4.4: Counters for extended SCGs of the $f91$ function

other legal graphs and its counter destination remains positive. Such legal graphs mimic a (set of) legal call trace(s). With the counters and their constraints, the affine SCGs for G_b and G_{ac} are:

$$\begin{aligned} G_{ac} &= \{[n,r,c] \rightarrow [n',r',c'] : n' = n - 10 \wedge c' = c - 1 \wedge c \geq 0 \wedge c' \geq 0\} \\ G_b &= \{[n,r,c] \rightarrow [n',r',c'] : n' = n + 11 \wedge c' = c + 1 \wedge c \geq 0 \wedge c' \geq 0\} \end{aligned}$$

Counters can be mechanically included in a set of the initial graphs. For each recursive choice branch in the CFG for a function, allocate a counter for each embedded non-terminal. This counter is used to limit the relative number of calls made by the (terminal) call graphs on either side of the non-terminal. Each occurrence of left-hand call increases the counter by 1, and each occurrence of right-hand call decreases the counter by 1. The generation of any new affine graph g is limited by presetting the counter c_g value to 0 before constructing the composition, and by placing a contextual constraint $c_g > 0$ on the graph for the right-hand call.

For example, given a CFG representing the traces for a function p:

$$P \rightarrow a \mid bXcYdP \mid eZfQ$$

the component graphs generated would be G_a, G_b, G_c, G_d, G_e and G_f. The algorithm just described adds three counters C_X, C_Y and C_Z to all the component graphs, adding constraints to each graph to ensure the required restrictions.

	G_a	G_b	G_c	G_d	G_e	G_f	**Restriction**	**Constraint**
C_X	$=$	$+1$	-1	$=$	$=$	$=$	$\#b \geq \#c$	$C_X \geq 0$
C_Y	$=$	$=$	$+1$	-1	$=$	$=$	$\#c \geq \#d$	$C_Y \geq 0$
C_Z	$=$	$=$	$=$	$=$	$+1$	-1	$\#e \geq \#f$	$C_Z \geq 0$

Table 4.2: Extra counters derived from the CFG $P \rightarrow a \mid bXcYdP \mid eZfQ$

Table 4.2 shows the three extra counters that are added, showing the restriction that each is associated with, and the constraint added to the affine graph to ensure this restriction. The entry "=" in the table entry (C_X, G_a) indicates that SCG G_a has

an arc from $C_X \to C'_X$ with $C'_X = C_X$. The entry "+1" in the table entry (C_X, G_b) indicates that SCG G_b has an arc from $C_X \to C'_X$ with $C'_X = C_X + 1$. Note also that in the SCGs for the functions $x()$, $y()$, $z()$ and $q()$ corresponding to the non-terminals X, Y, Z and Q, the counters C_X, C_Y and C_Z are appended to the corresponding SCGs, with each arc from $C \to C'$ having $C' = C$.

As before, the extended affine size-change graphs add some time to the closure algorithm, but do not affect the algorithm otherwise. However, the *counter* extension enhances the analysis in two ways.

1. When composing two candidate affine graphs that belong to L_{RE}, if the result of the composition would result in the constraint going negative, then the composition fails, and returns FALSE, representing an empty relation. In this case, the candidate composition is (correctly) excluded and discarded.
2. Even if the composition succeeds, the counters continue to record the relative frequency of the function calls, perhaps resulting in some other graph being excluded at a later stage.

In summary, counters allow the (RE) graphs to record the relative frequency of function calls. The resultant language $L_{\mathrm{RE+COUNT}}$ is more refined than L_{RE}, and more closely approximates L_{CFG}.

4.4 From well-founded types to boundedness

With the introduction of affine constraints, it becomes feasible to lay the termination decision upon the concept of *boundedness* rather than (strictly) inductively defined variables. In the original paper on size-change termination [LJBA01], the principal concern was with program values drawn from well-founded sets. If it was possible to establish that every infinite sequence of function/procedure calls would result in an infinite descent in some program values, then termination was established.

However, this does not (for instance) handle the case where every infinite sequence of function/procedure calls would result in an infinite increase in some values, which are bounded above by some condition that inhibits the respective calls. We thus accept that either *monotonically increasing* and *bounded above* or *monotonically decreasing* and *bounded below* is sufficient to establish the termination property. This is realized by changing the presentation of size-change termination to include a *guard*, derived from the source program, in any idempotent function specification.

An example of this from the $f91$ function is in the argument which establishes that a series of calls to function $f91_b$ always terminates. The affine size-change graph G_b for call $f91_b$ is $G_b = \{[n] \to [n'] : n' = n + 11\}$, and when the guard derived from the

source program is included ($n \leq 100$), then there is a graph of the form $b^{\omega} : n' > n$, where b^{ω} represents possibly infinitely many recursive calls to $f91_b$. The graph is evaluated in two steps: first that n is monotonically increasing in $f91_b$, and second that $n \leq 100$ guards all calls to $f91_b$. As a result of these two statements it can be said that $f91_b$ is terminating (i.e., ω is finite).

To record this property, attach a guard to the graphs, derived from a static analysis of the program. Given a graph $G_b : n' > n$ that is guarded by the condition $n \leq 100$, then this will be denoted by $b_{[n \leq 100]} : n' > n$ for notational brevity, and when the meaning is clear.

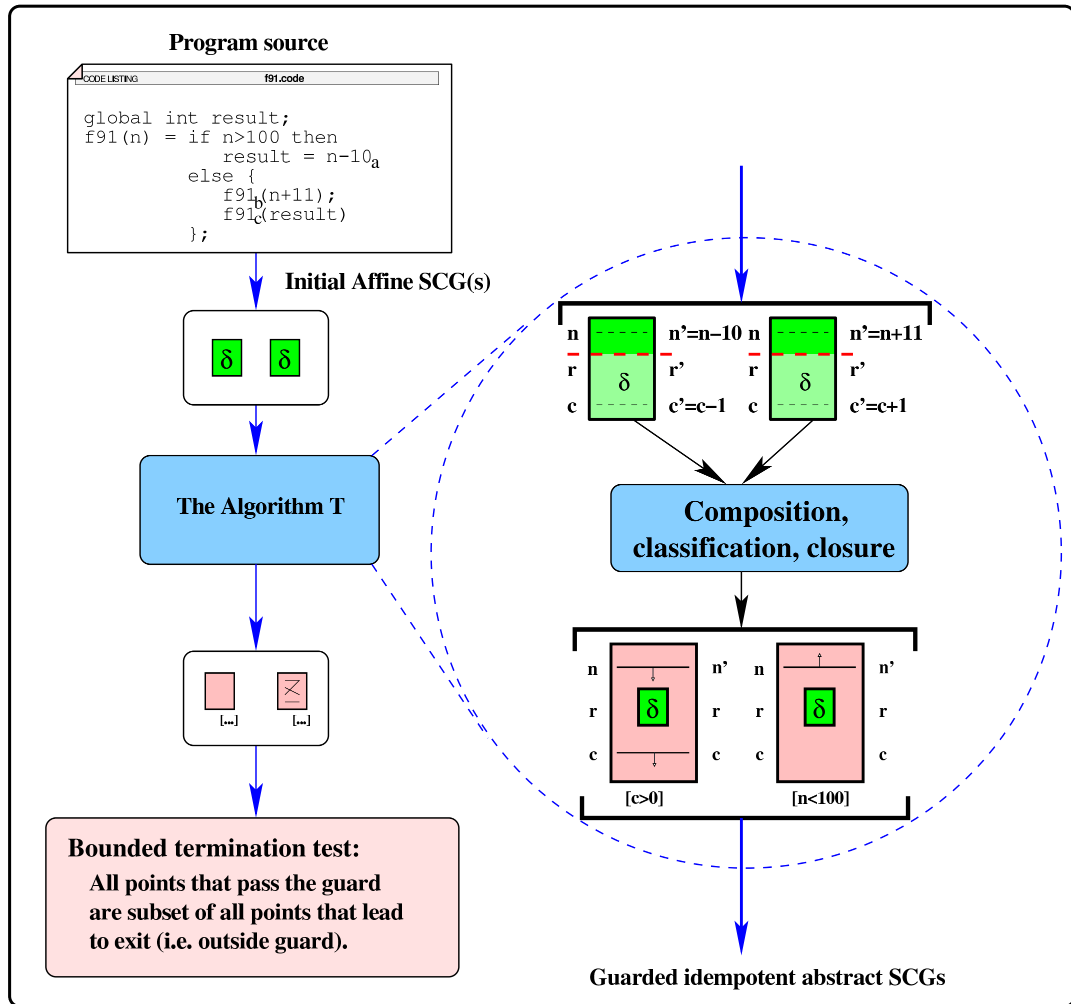


Figure 4.5: Overview of changes to affine SCT for bounded termination

In Figure 4.5 the changes to the affine SCT algorithm are illustrated, with guards associated with the abstract SCGs, and with a new termination test at the end. The test for this is referred to here as *reduction-termination*.

4.4.1 Bounded termination

At this stage a formal treatment of a new property over the guarded graphs is given, the property of *bounded-termination*. In previous work on size-change termination, an infinite reduction of a well-founded type led to an impossibility argument for the graph; as no such infinite reduction can occur, then it can be concluded that the corresponding function call cannot occur infinitely often.

A similar argument can be made over guarded graphs, and Theorem 5 asserts a sufficient condition to establish bounded-termination.

In this treatment, all functions are assumed to be of the form

$$f(\bar{x}) \stackrel{\text{def}}{=} \begin{cases} b_1 & \rightarrow \quad f_1(\bar{y_1}) \\ b_2 & \rightarrow \quad f_2(\bar{y_2}) \\ \dots & \\ \text{else} & \dots \end{cases}$$

where the terms b_1, b_2 represent guards for each recursive function call, limited to conjunctions of simple affine relationships between the elements of $\bar{x}$. This restriction ensures that there are no *holes* in the domains of the functions. The terms $\bar{y_1}$, $\bar{y_2}$ represent the parameters to the corresponding call to the function, where $\bar{y} = \bar{x}[\psi]$, $[\psi] = [y_1 \mapsto \delta_1(x_1, x_2, \dots), y_2 \mapsto \delta_2(x_1, x_2, \dots), \dots]$ and δ_1, δ_2 represent affine relations over the input parameters, as in the affine relationships in Chapter 3.

The graphs g_1, $g_2 \dots$ for such a function can again be given in affine relationship form: $g_1 = \{[\bar{x}] \rightarrow [\bar{y_1}]\}$, and the notation $\{[\bar{x}] \rightarrow [\bar{y_1}]\}_{[b_1]}$ represents the guarded graph. Note the close association between the functions and the graphs. In the following the phrase "recursive graph" is used when it would be more accurate to say "the graph associated with a recursive function call...".

Some preliminary definitions for the domain and range restrictions associated with recursive graphs are needed first.

Definition 17 *The domain restriction function* $\lhd$: $\mathbb{P}X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$ *is defined by*

$$\forall D : \mathbb{P}X;\ G : (X \leftrightarrow Y) \bullet D \lhd G = \{(\bar{x}, \bar{y}) : G \mid \bar{x} \in D\}$$

Definition 18 *The range restriction function* $\rhd$: $(X \leftrightarrow Y) \times \mathbb{P}Y \rightarrow (X \leftrightarrow Y)$ *is defined by*

$$\forall G : (X \leftrightarrow Y);\ R : \mathbb{P}Y \bullet G \rhd R = \{(\bar{x}, \bar{y}) : G \mid \bar{y} \in R\}$$

In addition, a new set is defined, the restricted domain of a graph, which is defined over self recursive monotonically changing guarded graphs. This allows *bounded termination* to be defined.

Definition 19 *A self recursive graph $g = \{[\bar{x}] \rightarrow [\bar{y}]\}$ corresponding to a function call $f(\bar{y})$ is* ***strictly monotonically changing*** *(or* ***order-preserving****) under an ordering $\sqsupset$ and over a continuous domain r, if whenever $\bar{x} \sqsupset \bar{y}$, then $f(\bar{x}) \sqsupset f(\bar{y})$.*

The ordering $\sqsupset$ does not need to be defined here, but could be any ordering captured by a testable condition (given later). This testable condition captures orderings over common linear transformations of program variables, including simple ones (like counting up and down), more complex ones involving linear combinations of variables, and even/odd relations.

Definition 20 *Given a self-recursive graph g defined over a domain R, the* ***restricted domain*** *$\Omega(g^n, R)$ of recursive graph g is defined by*

$$\Omega(g^n, R) \stackrel{\text{def}}{=} \{v \in \text{domain}(g) \mid (\{v\} \lhd g^n \rhd R) \neq \varnothing\}$$

(Note that g^n means g composed with itself n times: $g \circ g \circ g \circ \ldots \circ g$).

Definition 21 *A recursive and monotonically changing graph g over a domain R is* ***bounded-terminating by*** *R if, for all $v \in R$*

$$\exists n > 0 : (\{v\} \lhd g^n \rhd R) = \varnothing$$

The idea here is that the restricted domain in some sense defines the allowable set of input values for the graph g^n, and if this set is empty, then the graph g cannot be applied to itself n times:

$$\forall n \in \mathbb{N}, v \in R \ : \ (\{v\} \lhd g^n \rhd R) = \varnothing \Rightarrow (\{v\} \lhd g^{n+1} \rhd R) = \varnothing$$

During the analysis, the function f is approximated by a (monotonically changing) affine relation. Consequently, the domain R must be restricted such that bounded-termination of an approximation of f inevitably implies boundedness of repeated calls to f during runtime. A sufficient condition that ensures this is the definition of a *continuous* domain R below:

Definition 22 *Given a recursive and monotonically changing affine relation g over a domain R, let $\bar{R}$ be the complement of R. The domain R is said to be a* ***continuous domain*** *if, for all $v \notin R$,*

$$\forall n > 0 : \text{range}(\{v\} \lhd g^n) \subseteq \bar{R}$$

A testable condition may now be established, sufficient to assert that a function f (and its approximation g) is bounded-terminating. Bounded termination is closely associated with function termination in that if a function is bounded-terminating, then it must be terminating. However, bounded termination subsumes the reduction-termination test used for size-change termination (that at least one parameter reduces). Any reduction-terminating graph is bounded-terminating, but a bounded-terminating graph may not be reduction-terminating. As usual, if bounded termination cannot be established, then nothing is known.

Theorem 5 *Given a self recursive graph g which is monotonic under an ordering $\sqsupset$ and over a continuous domain R, then if*

$$R \subseteq \Omega(R \lhd g^+ \rhd \bar{R}, \top)$$

then g is bounded-terminating by R[1].

The intuition here is that the term $R \lhd g^+ \rhd \bar{R}$ specifies all the graphs that must have a range outside the guard of the function call f corresponding to the graph g. If the guard is a subset of the restricted domain Ω, then all initial values for the function must lead out of the restricted domain, leading to termination of the recursive call.

Proof: Suppose g is *not* bounded-terminating, then by the definition of bounded-termination and the continuous property of R, there exists a value $v_1 \in R$ such that $\forall n : (\{v_1\} \lhd g^n \rhd R) \neq \varnothing$, and

$$\begin{array}{ll} & \exists v_2, \ldots, v_i, \ldots : g^i(v_1) = v_{i+1} \wedge \forall j > 1 : v_j \in R \\ st. & v_1 \sqsupset v_2 \sqsupset \ldots \sqsupset v_i \sqsupset \ldots \qquad (\because \text{g is monotone}) \\ \Rightarrow & \forall i > 0 : v_i \notin \Omega(R \lhd g^i \rhd \bar{R}, \top) \\ \Rightarrow & v_i \notin \bigcup_{i>0} \Omega(R \lhd g^i \rhd \bar{R}, \top) \\ \Rightarrow & v_i \notin \Omega(R \lhd g^+ \rhd \bar{R}, \top) \end{array}$$

This contradicts a premise of the theorem, since $R \subseteq \Omega(R \lhd g^+ \rhd \bar{R}, \top)$ and $v_i \in R$, then it must be that $v_i \in \Omega(R \lhd g^+ \rhd \bar{R}, \top)$. Thus it is concluded (by contradiction) that g is bounded-terminating by R. □

This theorem provides a mechanism for testing if a final size-change graph for guarded function calls leads to a terminating program. The test is not computationally expensive, requiring only domain and range restriction and set inclusion to an existing graph in the final step of termination analysis.

In summary, the graphs have been extended with a guard derived from a static analysis of the program. This guard is used in the final step of the size-change termination

[1]The top $\top$ is just $R \cup \bar{R}$.

algorithm, allowing the analysis of both monotonically decreasing and bounded below, as well as monotonically increasing and bounded above, or combinations of these. A testable sufficient condition has also been established, which can test if an arbitrary self-recursive guarded call is bounded-terminating. This test is surprisingly general, capturing many indicators of termination including ones that rely on even/odd tests, and exceeding or reducing bounds.

4.5 Summary of changes to affine-SCT

In summary, affine size-change termination has been changed by improving algorithm $\mathcal{T}$, and by a new test performed on the final set of idempotent affine SCGs. The changes to algorithm $\mathcal{T}$ are as follows:

1. Component size-change graphs derived from the source of a program are extended with extra parameters, to record other information about the program.
2. The component size-change graphs are annotated with a regular expression representing the trace for the graph. For example, composing g_a with g_b, and then taking the transitive closure will result in $g_{(ab)^+}$.
3. The component size-change graphs are also annotated with a *guard* expression, representing the guard for the initial element of the trace. For notational convenience, the affine graph $g_{(ab)^+}$ with the guard $n > 10$ is written as $(ab)^+_{[n>10]}$.
4. *Value* graphs, corresponding to function return values, are added to the component size-change graphs in the first step of the algorithm, but are immediately composed with a matching call graph (corresponding to a following function call). This results in a larger set of *initial* graphs for the algorithm $\mathcal{T}$.
5. A regular expression L_{RE} which approximates the CFG representation of the call structure for the program is constructed. In addition, counters are added to the component size-change graphs.
6. The test for allowable composition of two graphs g_a and g_b is the function $\mathrm{legitimate}(g_a, g_b) \stackrel{\mathrm{def}}{=} ab \subseteq L_{\mathrm{RE}} \wedge C_x \geq 0$, where C_x are the counters associated with the function calls.

The changes above do not change affine-SCT in essence, but make the analysis more precise. The other significant change is in the test performed after the closure algorithm $\mathcal{T}$ is complete. In affine-SCT, the test is that each remaining idempotent SCG has a reducing parameter, termed reduction-termination, and closely following the technique used for LJB SCT. In the extended system, each remaining idempotent SCG $g_{[r]}$ (abstract graph g associated with bound r), is tested using the bounded-termination test:

$$r \subseteq \Omega(r \lhd g^+ \rhd \bar{r}, \top)$$

4.6 Termination analysis examples

The termination analysis for bounded termination mirrors the analysis for affine SCGs. The major step of the termination analysis is the algorithm $\mathcal{T}$, which builds a closure of an initial set of affine size-change graphs, constructing compositions of existing affine size-change graphs until no new affine graphs are created. After the algorithm $\mathcal{T}$ terminates, then the second step is to examine the resultant set of graphs, and establish if each idempotent recursive call graph is bounded-terminating.

The function `generate` in the algorithm $\mathcal{T}$ returns a new set of affine size-change graphs constructed by composing any possible pairs of existing size-change graphs. Intuitively, the legitimate compositions only include those which result in a legitimate call sequence, as is the case for LJB-analysis. Here, the counters are preset to 0 before composition, and remain non-negative at the end of composition.

The second step of the termination analysis is to identify those non-empty idempotent containers, and then check to see if the affine graphs therein contain a varying component that matches the guard for that abstract graph. The test derived from Theorem 5 is computationally inexpensive, and captures many indicators of termination including ones that rely on even/odd tests, and increasing or reducing arguments towards a guard. If all these idempotent graphs pass the test, then it can be concluded that the associated program terminates. If one of these graphs does not pass the test, it can be concluded that the associated program does not belong to the size-change terminating programs. The affine-based analysis described here captures more information from a program, and is still known to terminate.

In summary, at each iteration graph composition is done repetitively until a fixed point is reached. If the result shows that the function terminates, then the algorithm halts. If not, the size-change graphs are abstracted, and the graph-composition process is repeated. In the worst case, the repetitive abstraction of the size-change graphs will result in abstract graphs that mimic LJB size-change graphs, and the composition will degenerate into the usual LJB-graph composition. Thus, the termination property is determined in the same way as for LJB-analysis.

4.6.1 Termination for $f91$

The $f91$ function is repeated here, along with it's annotations. All the steps of the analysis for the $f91$ function are automatic.

```
f91(n) = if n > 100 then
             n - 10_a
           else
             f91_c(f91_b(n + 11));
```

The CFG for the $f91$ function is $\mathcal{P} \to a \mid b\mathcal{P}c\mathcal{P}$. The following table shows the call and value affine graphs for each component of the trace:

Trace	**Component affine graphs**
a_{val}	$\{[n,r,c] \to [n', n-10, c]\}$
b_{call}	$\{[n,r,c] \to [n+11, r'', c+1]\}$
c_{call}	$\{[n,r,c] \to [r, r'', c-1]\}$

The possibly infinite traces of the CFG are approximated by L_{RE}. A counter c is also included to control the frequency of occurrence of c's with respect to b's. From a static analysis of the program, the affine graph a_{val} is restricted in it's domain to $n > 100$ and the affine graph b_{call} is restricted in it's domain to $n \leq 100$.

These restrictions are used to generate the initial extended affine graphs:

Trace	**Initial extended affine graphs**	
ac	$\{[n,r,c] \to [n-10, r'', c-1]$	$: n > 100 \wedge c > 0\}$
b	$\{[n,r,c] \to [n+11, r'', c+1]$	$: n \leq 100\}$

These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs. These lead to a final classification of graphs into containers as follows:

Trace	**Final affine graphs**	**Abstract graph**
$(ac)^+$	$\{[n,r,c] \to [n',r',c'] : n' \leq n-10 \wedge c > 0\}$	$(ac)^+_{[c>0]}$: $n' < n \wedge c' < c$
$b^+ + b^+(ac)^+$	$\{[n,r,c] \to [n',r',c'] : n' \geq n+1 \wedge c \geq 0\}$	$b^+_{[n\leq 100]}$: $n' > n$

The composition of any pairing of these graphs generates no new affine or abstract graphs, and so the algorithm $\mathcal{T}$ terminates, to be followed by the second step of the analysis of the resultant containers.

First, identify monotonicity of those idempotent graphs. This can be easily determined from the respective abstract graphs. Next, test each one using $r_1 = \{c \mid c > 0\}$ and $r_2 = \{n \mid n \leq 100\}$:

$$\begin{aligned} r_1 &\subseteq \Omega(r_1 \lhd (ac)^+ \rhd \bar{r_1}, \top) \\ r_2 &\subseteq \Omega(r_2 \lhd b^+ \rhd \bar{r_2}, \top) \end{aligned}$$

The final result may now be asserted: *the $f91$ function terminates for all traces.*

Note the interesting termination argument for the ac call. It relies on the encoding of the nesting level of the call sequence in the graph. It is not actually necessary to check ac for termination; it is only possible to have an infinite sequence of ac calls if it was preceded by an infinite sequence of b calls.

4.6.2 Termination of Ackermann's function

The termination of Ackermann's function can be derived by LJB analysis, but it is interesting to consider its evaluation within the new framework. The function is:

```
ack(x,y) = if x ≤ 0 then
               y + 1_a
           else if y > 0 then
               ack_c(x − 1, ack_b(x, y − 1))
           else
               ack_d(x − 1, 1);
```

The CFG for the core of the *ack* function is $\mathcal{P} \rightarrow a \mid b\mathcal{P}c\mathcal{P} \mid d\mathcal{P}$, and the (possibly) infinite traces are approximated by $\mathcal{P}_2 \rightarrow a \mid ac\mathcal{P}_2 \mid b\mathcal{P}_2 \mid d\mathcal{P}_2$. The following table shows the call and value affine graphs for each component:

Trace	Component affine graphs
a_{val}	$\{[x,y,r,c] \rightarrow [x',y',y+1,c]\}$
b_{call}	$\{[x,y,r,c] \rightarrow [x,y-1,r',c+1]\}$
c_{call}	$\{[x,y,r,c] \rightarrow [x-1,r,r'',c-1]\}$
d_{call}	$\{[x,y,r,c] \rightarrow [x-1,1,r',c+1]\}$

Observe that in any trace of the function, there must be more b's or d's than there are ac's. From a static analysis of the program, the affine graph a_{val} is restricted in it's domain to $x \leq 0$. The affine graph b_{call} is restricted in it's domain to $x > 0 \land y > 0$. The affine graph d_{call} is restricted in it's domain to $x > 0 \land y \leq 0$. These restrictions are used to generate the initial graphs:

Trace	Initial extended affine graphs
ac	$\{[x,y,r,c] \rightarrow [x'-1,y+1,r'',c-1] \quad : x \leq 0 \land c > 0\}$
b	$\{[x,y,r,c] \rightarrow [x,y-1,r',c+1] \quad : x \geq 1 \land y \geq 1\}$
d	$\{[x,y,r,c] \rightarrow [x-1,1,r',c+1] \quad : x \geq 1 \land y \leq 0\}$

These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs through affine-graph composition. These lead to a final classification of affine graphs into containers as follows:

Trace	Abstract graph
$(ac)^+$	$(ac)^+_{[c>0]}: \quad c' < c$
$b^+ + b^+d^+(ac)^+$	$b^+_{[x>0 \land y>0]}: \quad x' = x \lor y' < y$
$d^+ + d^+(ac)^+$	$d^+_{[x>0 \land y \leq 0]}: \quad x' < x$

The composition of any pairing of these graphs generates no new affine or abstract graphs, and so the algorithm $\mathcal{T}$ terminates, to be followed by the second step of the analysis of the resultant containers. Test each one using $r_1 = \{c \mid c > 0\}$, $r_2 = \{(x,y) \mid x > 0 \wedge y > 0\}$ and $r_3 = \{(x,y) \mid x > 0 \wedge y \leq 0\}$:

$$\begin{aligned} r_1 &\subseteq \Omega(r_1 \lhd (ac)^+ \rhd \bar{r_1}, \top) \\ r_2 &\subseteq \Omega(r_2 \lhd b^+ \rhd \bar{r_2}, \top) \\ r_3 &\subseteq \Omega(r_3 \lhd d^+ \rhd \bar{r_3}, \top) \end{aligned}$$

The final result is: *the ack function terminates for all traces.*

4.6.3 Termination for function $\mathcal{X}$

For contrast, here is a function $\mathcal{X}$ which merges the call structure of Ackermann's function and the $f91$ function:

```
X(x,y) = if x > 0 then
             X_d(x − 1, y + 1)
         else if y ≤ 100 then
             X_c(x − 1, X_b(x, y + 11))
         } else;
             y − 10_a;
```

The CFG for the core of the $\mathcal{X}$ function is $\mathcal{P} \to a \mid b\mathcal{P}c\mathcal{P} \mid d\mathcal{P}$, and the (possibly) infinite traces are approximated by $\mathcal{P}_2 \to a \mid ac\mathcal{P}_2 \mid b\mathcal{P}_2 \mid d\mathcal{P}_2$. The following table shows the call and value affine graphs for each component of the trace:

Trace	Component affine graphs
a_{val}	$\{[x,y,r,c] \to [x',y',y-10,c]\}$
b_{call}	$\{[x,y,r,c] \to [x,y+11,r',c+1]\}$
c_{call}	$\{[x,y,r,c] \to [x-1,r,r'',c-1]\}$
d_{call}	$\{[x,y,r,c] \to [x-1,y+1,r',c+1]\}$

As before, by examination of the CFG, the restrictions generate the initial extended affine graphs. These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs. These lead to a final classification of affine graphs into containers as follows:

Trace	Abstract graph
$(ac)^+$	$(ac)^+_{[c>0]}:\ c' < c$
$b^+ + b^+(ac)^+$	$b^+_{[x \leq 0 \wedge y \leq 100]}:\ y' > y \vee x' < x$
$d^+ + d^+(ac)^+$	$d^+_{[x>0]}:\ x' < x$

The composition of any pairing of these graphs generates no new affine or abstract graphs, and so the algorithm $\mathcal{T}$ terminates, to be followed by the second step of the analysis of the resultant containers. Test each one using $r_1 = \{c \mid c > 0\}$, $r_2 = \{(x, y) \mid x \leq 0 \wedge y \leq 100\}$and $r_3 = \{(x, y) \mid x > 0\}$:

$$\begin{aligned} r_1 &\subseteq \Omega(r_1 \lhd (ac)^+ \rhd \bar{r_1}, \top) \\ r_2 &\subseteq \Omega(r_2 \lhd b^+ \rhd \bar{r_2}, \top) \\ r_3 &\subseteq \Omega(r_3 \lhd d^+ \rhd \bar{r_3}, \top) \end{aligned}$$

The final result is: *the $\mathcal{X}$ function terminates for all traces.*

4.6.4 Functions which cannot be analyzed

There are of course many functions which terminate, but which fail the test. Functions in which the change of *size* of parameters is not relevant to the termination will fail any attempt at size-change termination analysis. In addition, if the size-change is not affine, or the testable condition does not represent a half-space, then the technique is unlikely to return a positive result.

If a function's termination is dependant on two separate values returned by two different function calls, then the technique will not work.

4.7 Related work

Bounded size-change termination is an approach to the analysis of integer programs for termination that is both path and context sensitive. The analysis is guaranteed to terminate, and in addition, it retains accurate information about both the change in values of parameters, and the call structure of the program. The following two sections briefly outline related work in this area.

4.7.1 Avery's SCT

In [Ave06], Avery introduces another approach to SCT, specifically directed at handling non-well-founded data types. This work is thus clearly related to the work on bounded size-change termination introduced in this chapter and Avery refers to my bounded termination work in his paper.

In Avery's approach, an abstract interpretation over the domain of convex polyhedra in the style of [CC77] is combined with size-change graph techniques. Avery has implemented his technique for an imperative C-like language with integers. The specific language used for illustration of the method has no function calls, stack or

dynamic memory. The size-change graphs detail the change in a *store* of program variables from one block of the program to another. Consider the following C code:

```
for (j = 0; j − 1 < N; j++) ;
```

A flow graph is constructed with program points and constrained arcs. Two size-change graphs are constructed as in Figure 4.6.

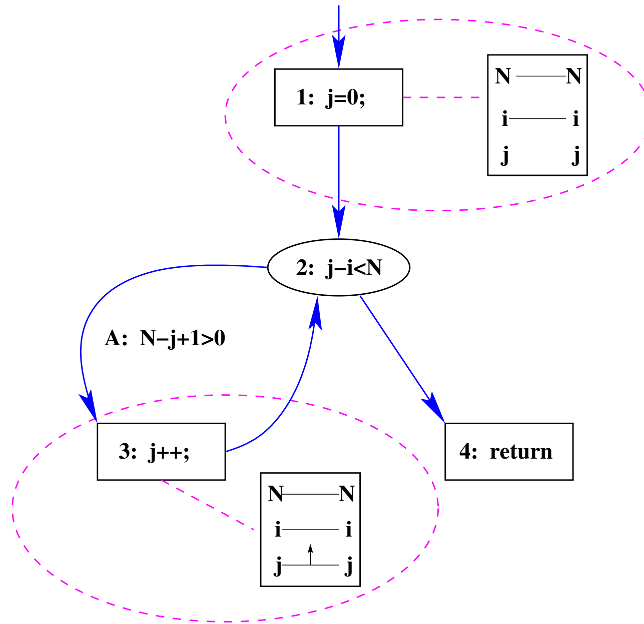


Figure 4.6: Flow graph for sample program, with size-change graphs

The store for this program fragment is just the three variables N, i and j, and the only two program points that modify the store are the points 1 and 3. The SCG for program point 3 has an increasing value for variable j, and (all other values being constant) so $N - j + i$ reduces. This SCG represents the program paths that lead from the (constrained) arc A back to A, and since this is bounded from below (by 0), A can only be visited finitely many times. As a result of this the arc A may be removed, and the remaining flow graph has no cycles, and hence the program terminates. The constraint for the arc A is derived directly from the conditional, but if the loop was nested within others, the constraints are more complex, and are derived using abstract interpretation invariant discovery. In Avery's SCT, the graphs are standard LJB SCT graphs, although including as well the $\uparrow$ relation. Termination is guaranteed in the same way as for LJB SCT.

The restriction placed on arcs in the flow graph corresponds to the approach in this thesis of annotating the graphs with guards, although in Avery's work the restrictions are derived from the abstract interpretation, and include information related to enclosing program paths. In the bounded-termination approach in this chapter, this

information is kept in the affine graphs. Avery's SCGs correspond with the abstract SCGs in this chapter, but during composition of these SCGs, any affine relationships between program variables are lost.

The approach for bounded SCT is different, but it is interesting to note that the bounded-termination test captures not only computations where the guard is a convex polyhedron, but also computations where the guard relies on even/odd properties.

4.7.2 Cook's Terminator

Recent work by Byron Cook [CPR06, CPR07], based on Podelski's *transition* invariants [PR04], has made dramatic headway into the termination analysis of conventional programming languages. The analysis is path and context-sensitive, and is performed on C code (specifically Windows device drivers), using the TERMINATOR tool. It has been applied to *large* program fragments of up to 35,000 lines of code.

TERMINATOR identifies a series of program cutpoints (specifically the beginning of while-loops, and function entry points), and constructs an initial (finite) set of over-approximated linear, well-founded, ranking relations. It then enumerates over all possible paths from each cutpoint, and for each it enumerates all possible pairs of states along that path, checking if the associated transition relation is a subset of one of the well-founded relations. If a state pair along a path is not a subset of one of the relations, then TERMINATOR synthesizes new ranking relations, and restarts the analysis.

Since the ranking relations are generated on demand (rather than all at once), the termination of the process cannot be guaranteed, but even so, the tool has been able to analyse large program fragments successfully. Bounded size-change termination guarantees termination of the algorithm, but as yet is only applied to a simple language. TERMINATOR can analyse C programs with pointers and aliases.

TERMINATOR requires that state pairs along the path are a subset of a well founded relation, and it is easy to construct programs that (by default) cannot be analysed, in part because the path analysis is not exhaustive. For example, if a loop permutes variable values, there may be no well founded relation, and the analysis will falsely accuse a program fragment of not terminating. If the permutation is (say) just a pair of variables swapped, then by unrolling the loop once, a well founded relation may be found. If the permutation was more complex, then the loop may need to be unrolled some more.

By contrast, bounded SCT analysis can handle any such program, because it has an automatic technique for constructing all *idempotent* sequences between cutpoints (function entry points). A concrete example of this is given below, in both while-loop form (on the left) and functional form (on the right). The variables m, n follow a sequence at the while cutpoint (alternatively the entry point of function f) like this: $\langle 3, -1\rangle$, $\langle -4, 1\rangle$, $\langle 5, -1\rangle$, $\langle -6, 1\rangle$, $\langle 7, -1\rangle$ until m reaches 1000. Bounded SCT

analysis automatically shows termination generating a single idempotent guarded abstract graph.

```
if (m > 0) then                      g(m)    = if  m > 0 then
    n = 1;                                         f(m, -1);
while m < 1000 {                     f(m,n)  = if m < 1000 then
    m = n - m;                                     f(n - m, -n);
    n = -n;
}
```

Though TERMINATOR could analyse this code by unrolling the loop once, this seems an ad-hoc solution to the problem. The bounded SCT method of generating all possible idempotent sequences back to a cutpoint may benefit TERMINATOR's overall analysis, by extending its path analysis.

4.8 Commentary

The size-change principle has been extended with new techniques which allow the analysis technique to be applied to functions in which the return values are relevant to termination. In addition, the idea that termination can be derived for a changing argument that is bounded has been formally expressed. The use of guards associated with the abstract graphs enable us to extend this directly into the termination argument, without the limitations implied by the previous technique of constraints over program arguments. A new test for bounded-termination using this approach subsumes the previous test for (only) reducing parameters, is computationally inexpensive, and captures termination properties for a wide range of guarded recursive calls, including ones where the guard depends on even/odd properties, or combinations of increasing and decreasing parameters.

The combination of these approaches significantly extends the set of programs that are size-change terminating.

Chapter 5

Time and stack costs for SCT programs

The preceding chapters have established the usefulness of affine size-change analysis for determining the termination property. The same style of analysis is also capable of compactly recording and calculating other properties of programs, including their runtime, maximum stack depth, and (relative) path time costs. In this chapter precise polynomial bounds on such costs are calculated on programs, by a characterization as a problem in quantifier elimination. The technique relies on a decision procedure, and is complete for a class of size-change terminating programs with limited-degree polynomial costs. An extension to the technique allows the calculation of some classes of exponential-cost programs. The new technique is demonstrated by recording the calculation in numbers-of-function (or procedure) calls for a simple definition language, but it can also be applied to functional and imperative languages. The technique is automated within the `reduce` computer algebra system.

5.1 Introduction

Polynomial runtime properties are considered essential in many applications. The ability to calculate such properties statically and precisely will contribute significantly to the analysis of complex systems. In real-time systems, the time-cost of a function or procedure may be critical for the correct operation of a system, and may need to be calculated for validation of the correct operation of the system. For example, a device-driver may need to respond to some device state change within a specified amount of time.

In other applications, the maximum stack usage may also be critical in (for example) embedded systems. In these systems, the memory available to a process may have severe limitations, and if these limits are exceeded the behaviour of the embedded system may be unpredictable. An analysis which identifies the maximum depth of

nesting of function or procedure calls can solve this problem, as the system developer can make just this amount of stack available.

A third motivation for calculating polynomial runtime properties is to calculate more precise relative costs of the individual calls. For example in a flow analysis of a program it may be interesting to know which calls are used most often, with a view to restructuring a program for efficiency. In this scenario, the relative costs between the individual calls is of interest. In the gcc compiler, a static branch predictor [BL93] uses heuristics to restructure the program code, optimizing the location of code for a branch more likely to occur. The approach described here can calculate more precise relative costs to improve these heuristics.

Wilhelm [Wil04] uses integer linear programming techniques to estimate WCET (Worst Case Execution Times) for programs. The estimates are not precise, as the use of abstract interpretation results in approximations to the WCET. However, the technique has been applied to large systems.

In this chapter the automatic and precise calculation of each of these costs is explored through static analysis of the source of programs which are known to be affine size-change terminating (as in Chapter 3), where the focus is on recording parameter size-changes only. The overall approach has three steps: firstly, assume a (degree k) polynomial bound related to the runtime or space cost, where the polynomial variables are the parameter *sizes*; secondly, derive from the source a set of equations constrained by this bound; thirdly, solve the equations to derive the precise runtime.

If the equations reduce to a vacuous result, then the original assumption of the degree of the polynomial must have been incorrect, and the process is repeated with a degree $k + 1$ assumption. This technique is surprisingly useful, and it is possible to derive precise runtime bounds on non-trivial programs.

It is also possible to calculate the time or space costs for a subclass of exponential costs, in particular those of the form $\phi_1 \cdot K^{\phi_2} + \phi_3$ where ϕ_1, ϕ_2 and ϕ_3 are each a limited-degree polynomial in the parameter sizes, and $K \in \Re$ is a constant.

There has been some research into runtime analysis for functional programs. For example, [San90] explores a technique to evaluate a program's execution costs through the construction of recurrences which compute the time-complexity of expressions in functional languages. It focuses on developing a calculus for costs, and does not provide automated calculations. In [Gro01], Grobauer explores the use of recurrences to evaluate a DML program's execution costs. The focus in this thesis is more with decidability aspects and precise time-costs than either of these approaches. In [AAMK06], ideas of termination and runtime analysis are applied to a-priori verification of the stability of a PID controller, implemented in a real-time system as a succession of timed interrupt response routines. The (functional) control program is reduced to a set of recurrences, which are solved to derive a worst case execution time.

An alternative approach is to limit the language in some way to ensure a certain runtime complexity. For example, in [Hof99], Hofmann proposes a restricted type system which ensures that all definable functions may be computed in polynomial time. The system uses inductive datatypes and recursion operators. In this work, time and stack costs of arbitrary functions or procedures are calculated through analysis of size-change information. A compact summary of a general technique for the calculation of time and space efficiency is found in the book [RH03] by Van Roy and Haridi, where recurrence relations are used to model the costs of the language elements of the programming language. There is unfortunately no general solution for an arbitrary set of recurrence relations, and in practice components of the costs are ignored, capturing at each stage only the most costly recurrence, and leading to big-$\mathcal{O}$ analysis.

Here the technique is improved for a specific class of functions, calculating more precise bounds than those derived from big-$\mathcal{O}$ analysis. By exploiting a-priori knowledge that a particular function terminates, and that the (polynomial) degree of the particular function is bounded, a formula representing the runtime (or stack depth) of the function is derived, with constant unknowns. By using quantifier elimination on this formula, an assignment to the unknowns is discovered, and after substitution back in the original formula, the time (or stack depth) cost of the program is discovered.

In the approach presented here, runtime is measured in terms of the number of calls to each procedure in a simple definition language. This is an appropriate measure, as the language does not support iteration constructs, and recursive application of procedures is the only way to construct iteration. Note that this approach does not restrict the applicability of the technique. Any iteration construct can be expressed as a recursion with some simple source transformation.

5.1.1 Preliminaries

The language used earlier is modified, to specify a *linear* guard expression as seen in Table 5.1.

v	$\in$	Var	⟨Variables⟩
f, g, h	$\in$	PName	⟨Procedure names⟩
n	$\in$	$\mathbb{Z}$	⟨Integer constants⟩
β	$\in$	Guard	⟨Boolean expressions⟩
		β ::= $\alpha \mid \neg\beta \mid \beta_1 \vee \beta_2 \mid \beta_1 \wedge \beta_2$	
		α ::= $\mathrm{T} \mid \mathrm{F} \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2$	
e	$\in$	AExp	⟨Expressions⟩
		e ::= $n \mid v \mid n \star e \mid e_1 + e_2 \mid -e$	
s	$\in$	Stat	⟨Statements⟩
		s ::= `if` β `then` s_1 `else` s_2 | s_1;s_2 | $f(e_1, \ldots, e_n)$ | ~	
d	$\in$	Decl	⟨Definitions⟩
		d ::= $f\ (x_1, \ldots, x_n) = s$;	

Table 5.1: Extended language syntax

This language is still in some sense an *abstract* language, omitting any parts not relevant to the runtime. In addition, the expressions are given as if they were all integer values, when in fact they refer to expressions based on the *size* of the data types of the language. For example, a list may be represented here by a *size* integer representing the length of the list, and list concatenation represented by addition of the size values. The boolean expressions are referred to as *guards* to highlight that they limit (or guard) function calls: a function call only being made if the guard is true. Finally, an important point is that the language only admits affine relations between the program variables and expressions.

5.2 Runtime analysis

In the process of size-change termination analysis described in [LJBA01], arbitrary sets of functions are processed, constructing a finite set of idempotent SCGs (Size-Change Graphs). These SCGs characterize the function, and detail all the ways in which a particular function entry point may be re-entered. In the following description, the functions are all derived from an affine SCT (Size-Change Termination) analysis, and hence are known to terminate. A subclass of these functions in which the conditional tests are linear, termed LA-SCT (Linear-affine SCT programs) define the class of programs analysed here. Limiting the analysis to this class of functions is not a severe restriction, as it is common for conditionals to be linear.

The first step is to formally define the runtime of such functions. The term $\bar{y}$ refers to the vector $(y_1, \ldots, y_n)$. For the sake of notational brevity, a *contextual notation* is used to represent an expression containing *at most* one function call. For an expression containing a function call $f(\bar{y})$, the corresponding contextual notation is $\mathcal{C}[f(\bar{y})]$. For an expression containing no call, the corresponding contextual notation is $\mathcal{C}[]$.

Definition 23 *Given an LA-SCT program p with program parameters $\bar{x}$ and body e_p and input arguments $\bar{n}$, the runtime of p, $B(p)[\bar{n}/\bar{x}]$,*[1] *is defined by the runtime of e_p inductively as follows:*

$$\begin{array}{lcl} B(s_1;\ s_2)[\bar{n}/\bar{x}] & \stackrel{\text{def}}{=} & B(s_1)[\bar{n}/\bar{x}] + B(s_2)[\bar{n}/\bar{x}] \\ B(\texttt{if}\, g\, \texttt{then}\, s_1\, \texttt{else}\, s_2)[\bar{n}/\bar{x}] & \stackrel{\text{def}}{=} & \text{if}\, g[\bar{n}/\bar{x}]\, \text{then}\, B(s_1)[\bar{n}/\bar{x}]\, \text{else}\, B(s_2)[\bar{n}/\bar{x}] \\ B(\mathcal{C}[])[\bar{n}/\bar{x}] & \stackrel{\text{def}}{=} & 0 \\ B(\mathcal{C}[f(\bar{m})])[\bar{n}/\bar{x}] & \stackrel{\text{def}}{=} & B(e_f)[\bar{m}/\bar{y}] + 1 \quad (\text{where}\, e_f\, \text{is the body of}\, f(\bar{y})) \end{array}$$

Note that the last definition above is recursive, and so the above definition may not be efficiently directly used for calculating the runtime $B(p)$. Instead, the implementation of analysis tools to evaluate the runtime generates a symbolic bound using other techniques. In practical terms, the runtime definition indicates that function calls

[1] $[\bar{n}/\bar{x}]$ means that the values for $\bar{n}$ replace the parameters $\bar{x}$ in the body of the function.

are being *counted* as the only measure of runtime. Such calls are the only difficult part of a runtime calculation, as other program constructs add *constant* time delays. To clarify this presentation, the definition of runtime is limited to just recording the function calls.

In the case of a function $f(\bar{x})$ containing only a direct call $h(\bar{y})$, where $\bar{y} = \bar{x}[\psi]$, with substitution $[\psi] = [y_1 \mapsto \delta_1(x_1, x_2, \ldots), y_2 \mapsto \delta_2(x_1, x_2, \ldots), \ldots]$ and δ_1, δ_2 `Aexp` expressions (represented by affine relations) over the input parameters, we have:

$$B(f(\bar{x})) = B(h(\bar{x}[\psi])) + 1$$

The primary interest is in runtimes that can be expressed as a polynomial in the parameter variables.

Definition 24 *The degree-k polynomial runtime $B_k(p)$ of an LA-SCT program p with m parameters $\overline{x} = x_1, \ldots, x_m$ is a multivariate degree-k polynomial expression:*

$$\begin{aligned} B_k(p) &\stackrel{\text{def}}{=} c_1 x_1^k + c_2 x_2^k + \ldots + c_m x_m^k + c_{m+1} x_1^{k-1} x_2 + \ldots + c_n \\ &= \sum c_i x_1^{i_1} \ldots x_m^{i_m} \end{aligned}$$

where $c_i \in \mathbb{Q}$, $i_1 + \ldots + i_m \leq k$, $i_1 \ldots i_m \geq 0$.

An example of such a degree-2 polynomial runtime for a program $p(x, y)$ is

$$B_2(p) = x + \tfrac{1}{2}y^2 + \tfrac{3}{2}y$$

An assumption of the runtime of a program p is denoted by $A(p)$ and the actual runtime by $B(p)$.

Definition 25 *An assumption $A(p)$ of a polynomial runtime of an LA-SCT program p with m parameters $\overline{x} = x_1, \ldots, x_m$ is a multivariate polynomial expression:*

$$\begin{aligned} A(p) &\stackrel{\text{def}}{=} c_1 x_1^k + c_2 x_2^k + \ldots + c_m x_m^k + c_{m+1} x_1^{k-1} x_2 + \ldots + c_n \\ &= \sum c_i x_1^{i_1} \ldots x_m^{i_m} \end{aligned}$$

where $c_i \in \mathbb{Q}$, $i_1 + \ldots + i_m \leq k$, $i_1 \ldots i_m \geq 0$, but the terms $c_1 \ldots c_n$ are unknown.

$A(p)$ contains all possible terms of degree at most k formed by the product of parameters of p. The assumption $A(p)$ and the bound $B(p)$ are related when an assignment $[\theta]$ to the constants $c_1 \ldots c_n$ has been found such that $B(p) = A(p)[\theta]$.

Initially, assume a polynomial upper bound of degree k on the running time of such a program $p(x, y, \ldots)$. This upper bound for the particular program p will be denoted by $A_k(p)$. If a program p had two parameters x and y, then

$$\begin{aligned}
A_1(p) &= c_1x + c_2y + c_3 \\
A_2(p) &= c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6 \\
A_3(p) &= c_1x^3 + c_2y^3 + c_3x^2y + c_4xy^2 + c_5x^2 + c_6y^2 + c_7xy + c_8x + c_9y + c_{10}
\end{aligned}$$

In this presentation, runtime behaviour is captured by deriving sets of equations of the form $A_k(p(\bar{x})) = \sum(A_k(f_i(\bar{x}[\psi_i])) + 1)$ for each of the sets of calls f_i which are calls isolated and identified by the same guard. The substitution $[\psi_i]$ relates the values of the input parameters to p to the values of the input parameters on the call f_i. Note that with this formulation, each substitution is linear, and thus cannot change the degree of the equation.

5.2.1 Characterization as a quantifier-elimination problem

The sets of assumptions and runtimes presented in the previous section are universally quantified over the parameter variables, and this leads to the idea of formulating this problem as a QE (quantifier-elimination) one. Consider the following program p_1 operating over the naturals with parameters $x, y \in \mathbb{N}$:

```
p1(x,y) = if (x = 0 ∧ y ≥ 1) then
              p1a(y,y − 1)
            else
              if (x ≥ 1) then
                 p1b(x − 1,y)
              else
                 ˜ ;            // ... exit ...
```

Once again, for clarity, the different calls are annotated. so it is possible to identify the specific function calls. In the guarded notation used in the previous chapter, this program would correspond to the following, which clarifies the guards over the calls to the function, and provides annotations for each of the recursive calls:

$$p_1(x, y) \stackrel{\text{def}}{=} \begin{cases} (x = 0 \wedge y \geq 1) & \rightarrow \quad p_{1a}(y, y-1) \\ (x \geq 1) & \rightarrow \quad p_{1b}(x-1, y) \\ (x = 0 \wedge y = 0) & \rightarrow \quad \ldots \end{cases}$$

The runtime properties for each path through the program p_1 can be represented with the following three equations, the first corresponding to the call p_{1a}, the second to the call p_{1b}, and the third corresponding to the function return (a runtime of 0):

$$\begin{aligned}
A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 &= 0 \\
A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 &= 0 \\
A_2(p_1) &= 0
\end{aligned}$$

For all values of x and y, the equations reduce to:

$$\begin{aligned}
-c_1x^2 + (c_1 + c_3)y^2 - c_3xy - c_4x + (c_4 - c_3 - 2c_2)y + c_2 - c_5 + 1 &= 0 \\
c_1 - 2c_1x - c_3y - c_4 + 1 &= 0 \\
c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6 &= 0
\end{aligned}$$

To find suitable values for the (real-valued) coefficients $c_1 \ldots c_6$, the process involves *eliminating the universally quantified elements* of the equalities (i.e. x and y) .

There are several advantages of this QE formulation of the problem. Firstly, there is an automatic technique for solving sets of polynomial equalities and inequalities of this form, developed by Alfred Tarski in the 1930's, but first fully described in 1951 [Tar51]. Tarski gives a decision procedure for a theory of elementary algebra of real numbers. Quantifier elimination is part of this theory, and after eliminating the quantifiers x and y in the above expressions, what remains are constraints over the values of the coefficients. However, the algorithm is not particularly efficient, although more recent methods are usable.

Secondly, precise analysis may be performed by including in the *guards* for each of the paths. For example, the QE problem can be expressed as the single formula[2] (universally quantified over x and y):

$$\begin{array}{ll}
\left(\begin{array}{rcl} & x & = 0 \\ \wedge & y & \geq 1 \end{array}\right) \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 = 0 \\
\wedge \quad (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 = 0 \\
\wedge \left(\begin{array}{rcl} & x & = 0 \\ \wedge & y & = 0 \end{array}\right) \Rightarrow & A_2(p_1) = 0
\end{array}$$

In [Kap04], the author clearly shows how quantifier elimination may be used to generate program invariants using either a theory of Presburger arithmetic, a theory involving parametric Gröbner bases, or Tarski's theory of real closed fields. This last theory is the most expressive, and a claim is made that the approach is more widely applicable, and generates stronger invariants than the Gröbner basis approach in [SSM04].

The construction here is different, and in a different field (program running time rather than program invariants). Expressions are constructed which characterize the program run time as a constraint quantified over the program parameters. The constraint constants are then solved by QE, and algebraic simplification (if needed).

5.2.2 Quantifier elimination

In 1973, Tarski's method was improved dramatically by the technique of Cylindrical Algebraic Decomposition (CAD) first described in [Col75]. The book [Ce98] has a

[2]The derivation of this particular form will be explained in the next subsection.

good introduction to the method, which leads to a quantifier free formula for a first order theory of real closed fields. In this theory, atomic formulæ may be of the form $\phi_1 = \phi_2$ or $\phi_1 > \phi_2$, where ϕ_1 and ϕ_2 are arbitrary polynomials with integer coefficients. They may be combined with the boolean connectives $\Rightarrow$, $\wedge$, $\vee$ and $\neg$, and variables may be quantified ($\forall$ and $\exists$).

Definition 26 *A Tarski formula T is any correctly formed sentence in the first order theory of real closed fields. Note that quantifier elimination is decidable in this theory.*

The approach here is to construct a particular subset of Tarski formulæ, $T[A(p)]$, where $A(p)$ is an assumption of the polynomial runtime of an LA-SCT program.

Definition 27 *The Tarski formulæ $T[A(p)]$ are of the form*

$$T[A(p)] = \left(\begin{array}{llll} \forall \bar{x}, \bar{y}, \ldots & g_1 & \Rightarrow & F_1 \\ \wedge & g_2 & \Rightarrow & F_2 \\ \wedge & \ldots & \ldots & \ldots \end{array} \right)$$

where $g_1, g_2, \ldots$ identify different paths from $p(\bar{x})$ to enclosed function calls $f_i(\bar{y})$[3] *(where $\bar{y} = \bar{x}[\psi]$ with $[\psi]$ an affine substitution). $F_1, F_2, \ldots$ are formulæ derived from the program p source such that*

$$\forall \bar{x} : g_j \Rightarrow (F_j \Leftrightarrow (A_k(p(\bar{x})) = \textstyle\sum_i A_k(f_i(\bar{x}[\psi_i])) + 1))$$

The inference rules in Table 5.2 can be used to automatically generate these "Tarski" formulæ from an arbitrary input program. They are presented in a form much like typing rules, where the type for a statement s is replaced by the runtime cost $A(s)$. The context (or environment) Γ is a list which specifies the parameters in the enclosing function.

Note that each application of a rule preserves the runtime of the statement. In addition, a substitution $[\psi]$ is applied in context, and is dependent on both the enclosing functions parameter names, and the (fresh) names for any other parameters.

This set of rules produces a guarded expression form for the assumed runtime $A(p)$. This is then transformed to a normal form, by first flattening the expression (distributing the guards outwards), and then distributing $A(p)$ in.

For example, for the program p_1 the above rules generate

$$A_2(p_1) = \left(\begin{array}{lrcr} & (x = 0 \wedge y = 0) & : & 0 \\ \wedge & (x = 0 \wedge y \geq 1) & : & A_2(p_1)[x \mapsto y, y \mapsto y - 1] + 1 \\ \wedge & (x \geq 1) & : & A_2(p_1)[x \mapsto x - 1] + 1 \end{array} \right)$$

[3] Note that they must cover the parameter space of interest and be distinct.

$$\Gamma \vdash g(\bar{x}[\psi]) \; : \; A(g)[\psi] + 1 \qquad \textbf{B-call}$$

$$\vdash \tilde{} \; : \; 0 \qquad \textbf{B-nocall}$$

$$\frac{\Gamma, \langle \bar{x} \rangle_f \vdash s \; : \; A(s)}{\Gamma \vdash f(\bar{x}) \overset{\text{def}}{=} s \; : \; A(s)} \qquad \textbf{B-def}$$

$$\frac{\Gamma \vdash s_1 \; : \; A(s_1) \qquad \Gamma \vdash s_2 \; : \; A(s_2)}{\Gamma \vdash \texttt{if}\, c\, \texttt{then}\, s_1\, \texttt{else}\, s_2 \; : \; \left\{ \begin{array}{rcl} c & : & A(s_1) \\ \wedge \; \neg c & : & A(s_2) \end{array} \right.} \qquad \textbf{B-if}$$

$$\frac{\Gamma \vdash s_1 \; : \; A(s_1) \qquad \Gamma \vdash s_2 \; : \; A(s_2)}{\Gamma \vdash s_1; \; s_2 \; : \; A(s_1) + A(s_2)} \qquad \textbf{B-seq}$$

$$\frac{\Gamma \vdash p \; : \; A(p) \qquad \Gamma \vdash f(\bar{x}) \overset{\text{def}}{=} s \; : \; A(s)}{\Gamma \vdash p\,; \; f(\bar{x}) \overset{\text{def}}{=} s \; : \; (A(s) \cup A(p))} \qquad \textbf{B-defs}$$

Table 5.2: Inference rules for Tarski formulæ

and the equation $T[A_2(p_1)]$ derived is thus (for all x and y):

$$\left(\begin{array}{lcrcl} & \left(\begin{array}{rcl} x & = & 0 \\ \wedge \; y & = & 0 \end{array} \right) & \Rightarrow & A_2(p_1) & = \; 0 \\ \wedge & \left(\begin{array}{rcl} x & = & 0 \\ \wedge \; y & \geq & 1 \end{array} \right) & \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 & = \; 0 \\ \wedge & (x \geq 1) & \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 & = \; 0 \end{array} \right)$$

and the task now is to reduce this to an expression without the quantifiers x and y, and then find any example of $c_1 \ldots c_6$ satisfying the resultant expression. This example is termed the assignment $[\theta] = [c_1 \mapsto \ldots, c_2 \mapsto \ldots, \ldots]$.

The following theorem asserts that the assignment $[\theta]$ derived from the formula $T[A_k(p)]$, when applied to the assumption $A_k(p)$, correctly represents the runtime $B_k(p)$ of any LA-SCT program p with a degree-k polynomial runtime.

Theorem 6 *If $B_k(p)$ is the degree-k polynomial runtime of affine SCT program p with parameters $\bar{x}$, and $A_k(p)$ is a degree-k polynomial assumption of the runtime of LA-SCT program p, and $[\theta]$ is the assignment derived from $T[A_k(p)]$, then*

$$\forall \bar{n} : \quad A_k(p)[\theta][\bar{n}/\bar{x}] \equiv B_k(p)[\bar{n}/\bar{x}]$$

Proof: By structural induction over the form of $B_k(p)[\bar{n}/\bar{x}]$. □

5.2.3 Tool support

There exist a range of tools capable of solving this sort of reduction. The tool `QEPCAD` [ea] is an implementation of quantifier elimination by partial CAD developed by Hoon Hong and his team over many years.

Another system is the `redlog` package [DS97] which can be added to the computer algebra system `reduce`, and may be used to eliminate quantifiers giving completely automatic results.

The following sequence shows `redlog` commands that *specify* the runtime for program p_1:

```
1: A2p1  := c1*x^2+c2*y^2+c3*x*y+c4*x+c5*y+c6;
2: path1 := sub(x=y,y=y-1,A2p1)-A2p1+1;
3: path2 := sub(x=x-1,A2p1)-A2p1+1;
```

Line 1 of the above sequence defines the `A2p1` assumption of the runtime bounds B_2 of the program. In lines 2 and 3, $A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1$ and $A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1$ (the `sub` command in `reduce` performs a series of substitutions in the expression `A2p1`).

The following sequence shows the `redlog` commands to *solve* the problem:

```
4: TA2p1 := rlqea ex({c1,c2,c3,c4,c5,c6},
                  rlqe all({x,y},
                          ((x=0 and y=0)  impl A2p1=0) and
                          ((x=0 and y>=1) impl path1=0) and
                          ((x>=1)         impl path2=0)));
5: B2p1  := sub( part(part(TA2p1,1),2),A2p1);
```

In line 4 of the above sequence, the inner `rlqe` function performs quantifier elimination on the equation $T[A_2(p_1)]$, returning the following relations between the constants $c_1 \ldots c_6$:

$$c_4 = 1 \wedge 2c_2 - c_4 = 0 \wedge c_2 - c_5 = -1 \wedge c_1, c_3, c_6 = 0$$

In this example, $c_1 \ldots c_6$ are uniquely determined, and can be found easily with a few simple reductions, but in the general case, the constraints over the constants may lead to many solutions. The `redlog` package can also be used to find an instance of a solution to an existentially quantified expression, and hence the outer `rlqea` function above, which returns an instance of a solution to the above relations existentially quantified over $c_1 \ldots c_6$: $\exists c_1 \ldots c_6 : T[A_2(p_1)]$.

The solution returned by `redlog` is

```
TA2p1 := {{true,{c1=0, c2=1/2, c3=0, c4=1, c5=3/2, c6=0}}}
```

This corresponds to the assignment

$$[\theta] = [c_1 \mapsto 0, c_2 \mapsto \tfrac{1}{2}, c_3 \mapsto 0, c_4 \mapsto 1, c_5 \mapsto \tfrac{3}{2}, c_6 \mapsto 0]$$

Finally, in line 5, the solution instance is substituted back in the original assumption $A_2(p_1) = c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6$, giving

$$\begin{aligned} B_2(p_1) &= A_2(p_1)[c_1 \mapsto 0, c_2 \mapsto \frac{1}{2}, c_3 \mapsto 0, c_4 \mapsto 1, c_5 \mapsto \frac{3}{2}, c_6 \mapsto 0] \\ &= x + \frac{1}{2}y^2 + \frac{3}{2}y \end{aligned}$$

The example given above appears to lead more naturally to a constraint logic programming [JL87] based solution to these sort of problems, but most such systems can normally only handle linear equations, not the polynomial ones used here.

There are constraint solving systems, for example `RISC-CLP(Real)` [Hon93], which use (internally) CAD quantifier elimination to solve polynomial constraints. However, here the discussion is restricted to just the underlying techniques, rather than cluttering up the discussion with other, perhaps confusing, properties of constraint solving systems.

5.3 Calculating other program costs

So far the presentation has been limited to examples which calculate polynomial runtimes for programs. However, the technique is also useful for deriving other invariant properties of programs, such as the maximum stack depth and the relative runtime costs.

5.3.1 Stack depth calculation

Consider program p_2:

```
p2(x,y) = if (x = 0 ∧ y ≥ 1) then
              p2a(y,y − 1);
              p2b(0,y − 1)
          else
              if (x ≥ 1) then
                  p2c(x − 1,y)
              else
                  ~ ;            // ... exit ...
```

Note that this program has a sequential composition of two function calls, and an exponential runtime cost. An interesting question for this program is to calculate its maximum stack depth. The maximum stack depth is defined to be the maximum nesting level of a function. The depth D of our class of programs is calculated in precisely the same way as the runtime B, with only a minor change.

In the event of sequential composition of the two functions, the system does not record the sum of the runtimes, but the maximum value:

$$\frac{\Gamma \vdash s_1 \ : A(s_1) \qquad \Gamma \vdash s_2 \ : A(s_2)}{\Gamma \vdash s_1;\ s_2 \ : \max(A(s_1), A(s_2))} \qquad \textbf{B-seq}$$

This corresponds with a Tarski formula for a polynomial solution like this:

$$\left(\begin{array}{rrcl} \forall x, y: & (x = 0 \wedge y \geq 1 \wedge D[\psi_{2a}] \geq D[\psi_{2b}]) & \Rightarrow & (D[\psi_{2a}] - D + 1 = 0) \\ \wedge & (x = 0 \wedge y \geq 1 \wedge D[\psi_{2a}] < D[\psi_{2b}]) & \Rightarrow & (D[\psi_{2b}] - D + 1 = 0) \\ \wedge & (x \geq 1) & \Rightarrow & (D[\psi_{2c}] - D + 1 = 0) \end{array} \right)$$

Given the formula, `redlog` immediately finds the stack *depth* cost[4]:

$$D(p_2) = x + \tfrac{1}{2}y^2 + \tfrac{3}{2}y$$

5.3.2 Relative runtime costs

The third motivation for this approach was to derive relative costs for the different possible paths through a program. Consider program p_1 (again):

```
p1(x,y) = if (x = 0 ∧ y ≥ 1) then
              p1a(y, y − 1)
            else
              if (x ≥ 1) then
                 p1b(x − 1, y)
              else
                 ˜ ;          // ... exit ...
```

Consider the question "In program p_1, which function is called more often, and what are the relative costs for each call?". This could be used in compiler optimization, improving the efficiency of the code by re-ordering and placing more commonly used functions nearby.

[4] Coincidentally, this value is the same as the runtime for program p_1.

The same approach may be used, calculating B for each path. The equation $T[A(p_{1a})]$ for the program choosing the first function call may be written as (for all x and y):

$$\left(\begin{array}{lrcl} & \left(\begin{array}{rcl} x & = & 0 \\ \wedge\ y & = & 0 \end{array}\right) & \Rightarrow & A_2(p_1) = 0 \\ \wedge & \left(\begin{array}{rcl} x & = & 0 \\ \wedge\ y & \geq & 1 \end{array}\right) & \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 = 0 \\ \wedge & (x \geq 1) & \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) = 0 \end{array}\right)$$

$$\Rightarrow\ B_2(p_{1a}) = y$$

The equation $T[A(p_{1b})]$ for the program choosing the second function call may be written as (for all x and y):

$$\left(\begin{array}{lrcl} & \left(\begin{array}{rcl} x & = & 0 \\ \wedge\ y & = & 0 \end{array}\right) & \Rightarrow & A_2(p_1) = 0 \\ \wedge & \left(\begin{array}{rcl} x & = & 0 \\ \wedge\ y & \geq & 1 \end{array}\right) & \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) = 0 \\ \wedge & (x \geq 1) & \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 = 0 \end{array}\right)$$

$$\Rightarrow\ B_2(p_{1b}) = x + \frac{1}{2}(y^2 + y)$$

Note that the sum of $B_2(p_{1a})$ and $B_2(p_{1b})$ is exactly $B_2(p_1)$ derived for the whole program, as expected.

5.4 Exponential program costs

The presentation so far has concentrated on LA-SCT programs with costs that may be expressed as polynomials over the program variables. However many such programs have costs that are exponential rather than polynomial. For example, consider the following program:

```
p3( x, y, n ) = if (x ≠ 0 ∧ n ≥ 1) then
                    p3a( x − 1, y, n )
                else
                    if (x = 0 ∧ n > 1) then
                        p3b( 2y + n, 2y, n − 1 )
                    else
                        ~ ;              // ... exit ...
```

This program has a runtime of $B(p_3) = y2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y - 2$, not immediately apparent by observation. The technique such as just described relies on repeatedly trying ever higher degree *polynomial* time costs, and would never discover this runtime. The runtime of programs of this form may be discovered however. An automated tool can be fed with an input having a generic form like this:

$$B(p_3) = \phi_1 \cdot K^{\phi_2} + \phi_3$$

and the solution (if it exists) will be found. This exponential form is used in a similar manner to the previous polynomial style analysis. An exponential runtime A for a program p is assumed, initially for a base of K, and using polynomials of (say) degree 2. The assumed runtime is thus $A(p) = \phi_1 \cdot K^{\phi_2} + \phi_3$, where ϕ_1, ϕ_2 and ϕ_3 are three polynomials of degree 2.

The three polynomials bear a peculiar relationship to each other due to the linearity of the parameter relationships. For example, for any single recursive call path, since the changes in the parameters are linear, then the runtime for this call path cannot be exponential. As a result of this, for any single recursive call path, $\phi_3[\psi] - \phi_3 + 1 = 0$, and in the case of a base of K, the following (non-exponential) relation holds:

$$\begin{array}{lrcl@{\quad}c@{\quad}rcl} & (\phi_1[\psi] & = & \phi_1 & \wedge & \phi_2[\psi] & = & \phi_2) \\ \vee & (\phi_1[\psi] & = & K\phi_1 & \wedge & \phi_2[\psi] & = & \phi_2 - 1) \\ \vee & (K\phi_1[\psi] & = & \phi_1 & \wedge & \phi_2[\psi] & = & \phi_2 + 1) \end{array}$$

The relation is true if and only if the exponential component of the runtime is $\phi_1 \cdot K^{\phi_2}$. This relation can be seen to be in the same polynomial form used before, and it can be plugged into the same software package, and be reduced in the same way, yielding polynomials ϕ_1, ϕ_2 and ϕ_3 which can define the exponential runtime for the program.

5.4.1 Explanation of the form of the relation

The explanation of the form of this relation has three steps. Firstly, for any single guarded recursive path, the runtime must be linear. Secondly, since the runtime for the whole program is a sum of an exponential and a polynomial, then (for this guarded recursive path) the exponential part of the runtime must be constant. Finally, the relation is true if and only if the exponential component of the runtime is $\phi_1 \cdot K^{\phi_2}$.

Step 1: Linearity for a single guarded recursive path

At this stage it is shown that any terminating single recursive call, with linear integer parameter changes, and with a linear guard over the call, must run in linear (or sub-linear) time. The complexity of simple (tail-recursive) functions like these have been studied in many places [CA69, Cob64, Mac71, MM69, MR67, Rit63, Yam62], summarised in Cook's Turing award lecture [Coo83]. However none of these approaches precisely answers this linearity question.

Consider a simplified situation, where a program is of the form

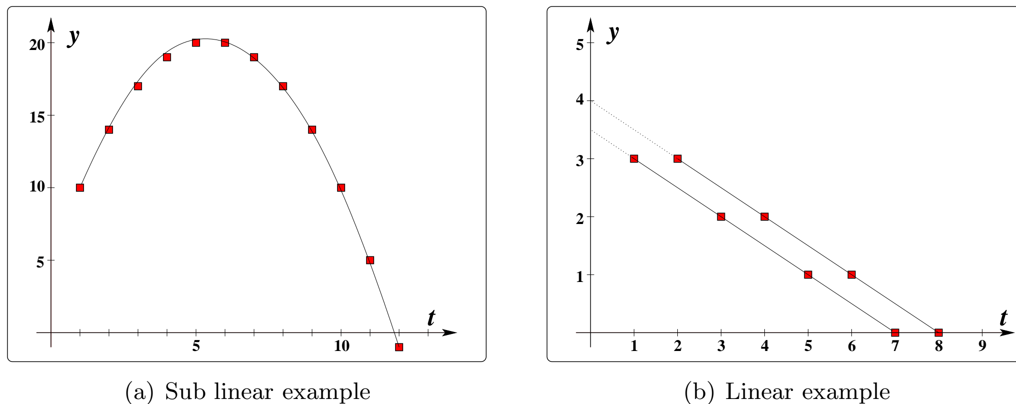
$$p(\bar{x}) \stackrel{\text{def}}{=} \begin{cases} x_i > 0 & \rightarrow & p(\bar{x}[\psi]) \\ x_i \leq 0 & \rightarrow & \sim \end{cases}$$

Once again $[\psi]$ is assumed to be a linear substitution, the guard defines a half-space for the x_i parameter, and this program must be known to terminate. Consider the parameter x_i: the possibilities are that it is constant (in which case the program must terminate immediately), or that it changes at some rate, and eventually reaches the terminating condition.

Ignoring the constant case, a successive series of t linear substitutions would at least result in a linear change in x_i (for example, by adding a constant value c to x_i, we would have the tth value being $x_i + tc$). However, it can also result in more complex changes in x_i (for example, if x_i is multiplied by some constant value k, then the tth value would be $k^t x_i$). The relationship between t and x_i can be summarized using a notion of *domination*, where x_i *dominates* t if $\frac{|x_i|}{t} \rightarrow +\infty$ as $t \rightarrow +\infty$, indicating that (in the limit) the x_i parameter changes at a faster speed than (in a sense) time. As an example of the case where x_i dominates t, consider the following program:

$$p(x,y) \stackrel{\text{def}}{=} \begin{cases} y > 0 & \rightarrow & p(x-1, x+y) \\ y \leq 0 & \rightarrow & \sim \end{cases}$$

For any invocation of this program $p(a,b)$, it is easy to derive a relation between t (the number of function calls, and hence *time*), and y, the parameter of interest. In this case, it is $t^2 - (2a+1)t + 2y - 2b = 0$[5]. Note how the y term dominates the t: y has to change faster than t. Figure 5.1(a) shows a plot of the trajectory of y varying with time t for an invocation of $p(5,5)$. Since the boundary of the halfspace (guard) condition is fixed, and since this program terminates, it is easy to see that this program terminates in sub-linear time.



(a) Sub linear example

(b) Linear example

Figure 5.1: Trajectories of guard variables with time

[5]Imagine an extra program parameter t, with an initial value of zero, and which is increased by 1 on each function call. An initial call $p(a,b,0)$ will have a second call $p(a-1, a+b, 1)$, a third call $p(a-2, 2a+b-1, 2)$. The tth call is $p(a-t, ta+b-\frac{t^2-t}{2}, t)$, giving the relationship between y and t.

As an example of the case where x_i and t do not dominate each other, consider the following program:

$$p(x,y) \stackrel{\text{def}}{=} \begin{cases} y > 0 & \rightarrow \quad p(y, x-1) \\ y \leq 0 & \rightarrow \quad \sim \end{cases}$$

For any invocation of this program $p(a,b)$, again it is easy to derive a relation between t and y, the parameter of interest. In this case it is

$$t^2 + 4ty - (2a + 2b - 1)t + 4y^2 - (4a + 4b - 2)y + 6a + 6b = 0$$

Note how the y and t terms are the same order. Figure 5.1(b) shows a plot of the trajectory of y for this program varying with time t for an invocation of $p(4,4)$ (the roots are $y = \frac{7-t}{2}$ and $y = \frac{8-t}{2}$). Since the boundary of the halfspace (guard) condition is fixed, and since this program terminates, it is easy to see that this program terminates in linear time.

It is now appropriate to consider more complex guard tests, and it is clear that for any program P_{guard} (annotating the program with the guard):

$$\begin{aligned} B(P_{x>0 \wedge y>0}) &= \min(B(P_{x>0}), B(P_{y>0})) \\ B(P_{x>0 \vee y>0}) &< B(P_{x>0}) + B(P_{y>0}) \end{aligned}$$

and so any terminating single recursive call, with linear integer parameter changes, and with a linear guard over the call, must run in linear (or sub-linear) time.

If the conditions are relaxed, it will no longer be the case that the functions run in linear time. For example if rationals were allowed, both in the variables, and in the coefficients of the linear expressions, it becomes possible to construct a function which changes x_i in a sub-linear (perhaps logarithmic) fashion, and as a result, the guard variable x_i will be dominated by t. As an example of this case where t dominates a program variable (not possible with integer variables and coefficients), consider the following program over rational variables x and y:

$$p(x,y) \stackrel{\text{def}}{=} \begin{cases} y > 0 & \rightarrow \quad p(\frac{3}{4}x, y - x) \\ y \leq 0 & \rightarrow \quad \sim \end{cases}$$

For any invocation of this program $p(a,b)$, the relation between t and y, the parameter of interest is:

$$y - b + a(1 + \sum_t \left(\tfrac{3}{4}\right)^t) = 0$$

This has time dominating the guard variable y: t has to change faster than y, and as a result the program runs in more-than linear time.

In summary, when constructing the equations to represent a particular recursive call, it can be assumed that while this call is being made successively, the runtime change will be *no more than* linear.

Step 2: Exponential part is constant

In the analysis of exponential runtimes, a runtime (for the whole program), of the form $\phi_1 \cdot K^{\phi_2} + \phi_3$ is assumed. This runtime applies to each individual recursive call, as well as to the program as a whole. For a program like

$$p(\bar{x}) \stackrel{\text{def}}{=} \begin{cases} b_1 & \rightarrow & p_1(\bar{x}[\psi_1]) \\ b_2 & \rightarrow & p_2(\bar{x}[\psi_2]) \\ & & \\ b_n & \rightarrow & \sim \end{cases}$$

then $A(p)$ is assumed to be $\phi_1 \cdot K^{\phi_2} + \phi_3$, and this runtime applies not only to the whole program, but also to each individual recursive call p_1, p_2 and so on. However, for any successive sequences of calls to p_1, the change in $B(p)$ must be linear, and will be subsumed by the term ϕ_3 in $A(p)$. As a result, during a sequence of calls to p_1, the term $\phi_1 \cdot K^{\phi_2}$ must be constant, and this may be exploited in forming the formula $T[A(p)]$.

Step 3: Form of the relation

Unfortunately, quantifier elimination from exponentials directly is not efficient. However, the particular form here can be re-expressed in polynomial form. Consider a single recursive call, and the effect on $A(p)$. Before the call, $A(p) = \phi_1 \cdot K^{\phi_2} + \phi_3$, and if the call made a linear substitution, then $A(p) = \phi_1[\psi] \cdot K^{\phi_2[\psi]} + \phi_3[\psi] + 1$ (i.e. 1 call has been made).

But during this call, the exponential part must have remained constant, so that $\phi_1 K^{\phi_2} = \phi_1[\psi] \cdot K^{\phi_2[\psi]}$ (and of course $\phi_3 = \phi_3[\psi] + 1$). Consider the exponentials: this can happen if $\phi_1[\psi] = \phi_1 \wedge \phi_2[\psi] = \phi_2$. But it can also happen if $\phi_1[\psi] = K\phi_1 \wedge \phi_2[\psi] = \phi_2 - 1$, and also in the case that $K\phi_1[\psi] = \phi_1 \wedge \phi_2[\psi] = \phi_2 + 1$. In a sense these equations *define* the exponential $\phi_1 K^{\phi_2}$. This explains the use of the following expression in the formula $T[A(p)]$, avoiding direct use of exponential terms, but still keeping the essence of the exponential term. For each single recursive call path:

$$\begin{array}{c} \phi_3[\psi] - \phi_3 + 1 = 0 \\ \wedge \\ \left(\begin{array}{rrcllcl} & (\phi_1[\psi] & = & \phi_1 & \wedge \ \phi_2[\psi] & = & \phi_2) \\ \vee & (\phi_1[\psi] & = & K\phi_1 & \wedge \ \phi_2[\psi] & = & \phi_2 - 1) \\ \vee & (K\phi_1[\psi] & = & \phi_1 & \wedge \ \phi_2[\psi] & = & \phi_2 + 1) \end{array} \right) \end{array}$$

5.4.2 Using the relation for exponential runtimes

This relationship between the polynomials may be exploited by constructing the equations in a similar form to the presentation for polynomial runtime analysis, solving

them in a similar manner, and finally deriving a sample solution. The `redlog` package is used to define

$$\begin{aligned}
\phi_1 &= c_1x^2 + c_2y^2 + c_3n^2 + c_4xy + c_5xn + c_6yn + c_7x + c_8y + c_9n + c_{10} \\
\phi_2 &= c_{11}x^2 + c_{12}y^2 + c_{13}n^2 + c_{14}xy + c_{15}xn + c_{16}yn + c_{17}x + c_{18}y + c_{19}n \\
\phi_3 &= c_{21}x^2 + c_{22}y^2 + c_{23}n^2 + c_{24}xy + c_{25}xn + c_{26}yn + c_{27}x + c_{28}y + c_{29}n + c_{30} \\
A &= \phi_1 \cdot K^{\phi_2} + \phi_3
\end{aligned}$$

The substitutions $[\psi_{3a}] = [x \mapsto x-1]$ and $[\psi_{3b}] = [x \mapsto 2y+n, y \mapsto 2y, n \mapsto n-1]$ for the two paths are applied to ϕ_1, ϕ_2 and ϕ_3, yielding the primed polynomials, and the equation $T[A(p_3)]$ for program p_3 may be written as (for all x, y and n):

$$\begin{array}{cl}
& \left(\begin{array}{lrcl} & x,y & > & 0 \\ \wedge & n & \geq & 0 \end{array}\right) \Rightarrow \left(\begin{array}{c} \phi_3[\psi_{3a}] - \phi_3 + 1 = 0 \\ \wedge \\ \left(\begin{array}{lrclcrcl} & (\phi_1[\psi_{3a}] & = & \phi_1 & \wedge & \phi_2[\psi_{3a}] & = & \phi_2) \\ \vee & (\phi_1[\psi_{3a}] & = & K\phi_1 & \wedge & \phi_2[\psi_{3a}] & = & \phi_2 - 1) \\ \vee & (K\phi_1[\psi_{3a}] & = & \phi_1 & \wedge & \phi_2[\psi_{3a}] & = & \phi_2 + 1) \end{array}\right) \end{array}\right) \\
\wedge & \left(\begin{array}{lrcl} & x & = & 0 \\ \wedge & y & > & 0 \\ \wedge & n & \geq & 0 \end{array}\right) \Rightarrow \left(\begin{array}{c} (\phi_3[\psi_{3b}] - \phi_3 + 1 = 0) \\ \wedge \\ \left(\begin{array}{lrclcrcl} & (\phi_1[\psi_{3b}] & = & \phi_1 & \wedge & \phi_2[\psi_{3b}] & = & \phi_2) \\ \vee & (\phi_1[\psi_{3b}] & = & K\phi_1 & \wedge & \phi_2[\psi_{3b}] & = & \phi_2 - 1) \\ \vee & (K\phi_1[\psi_{3b}] & = & \phi_1 & \wedge & \phi_2[\psi_{3b}] & = & \phi_2 + 1) \end{array}\right) \end{array}\right)
\end{array}$$

$T[A(p_3)]$ is easily manipulated by `redlog`, removing the quantifiers x, y and n, and suggesting an assignment that gives a family of solutions for the bounds:

$$A(p_3) = \alpha y 2^n + \tfrac{1}{2}n^2 + \tfrac{3}{2}n + x - 2y + c_{30}$$

where α indicates that any value here might be a solution, and c_{30} is unknown. To constrain the solution further, boundary cases for the system are added, for example $A(p_3(0,1,1)) = 0$, $A(p_3(0,2,1)) = 0$, giving:

$$B(p_3) = y2^n + \tfrac{1}{2}n^2 + \tfrac{3}{2}n + x - 2y - 2$$

5.4.3 Exponential examples

The program p_2 introduced in subsection 5.3.1, is repeated here:

```
p2(x,y) = if (x = 0 ∧ y ≥ 1) then
                p2a(y, y − 1);
                p2b(0, y − 1)
          else
                if (x ≥ 1) then
                    p2c(x − 1, y)
                else
                    ~ ;            // ... exit ...
```

The QE formulation is automatic and simple, deriving the equation for the runtime cost $T[A(p_2)]$ for all x and y:

$$
\begin{array}{lll}
 & (x = 0 \wedge y \geq 1) & \Rightarrow \quad (\phi_2[\psi_{2a}] + \phi_3[\psi_{2b}] - \phi_3 + 2 = 0) \\
\wedge & (x \geq 10 \wedge y \geq 0) & \Rightarrow \quad \left(\begin{array}{c} (\phi_3[\psi_{2c}] - \phi_3 + 1 = 0) \\ \wedge \\ \left(\begin{array}{lllllll} & (\phi_1[\psi_{2c}] & = & \phi_1 & \wedge & \phi_2[\psi_{2c}] & = & \phi_2) \\ \vee & (\phi_1[\psi_{2c}] & = & K\phi_1 & \wedge & \phi_2[\psi_{2c}] & = & \phi_2 - 1) \\ \vee & (K\phi_1[\psi_{2c}] & = & \phi_1 & \wedge & \phi_2[\psi_{2c}] & = & \phi_2 + 1) \end{array} \right) \end{array} \right)
\end{array}
$$

Given two independent base cases, `redlog` immediately finds the runtime cost:

$$B(p_2) = 4 * 2^y + x - y - 4$$

It is relatively easy to automatically derive exponential runtimes for programs like these, with polynomials of small degree, and the following examples illustrate a range of programs operating over naturals, with their automatically generated runtime costs.

Program 4:

```
p4(x,y)   = if (y ≤ 0) then
                 g_a(x,0)
              else
                 p4a(x+y,y−1);
g(x,y)    = if (x ≤ 0) then
                 ~                    // ... exit ...
              else
                 g_b(x−1,y+1);
```

The solution returned by `redlog` is that $B_2(p_4) = x + \frac{1}{2}y^2 + \frac{3}{2}y + 1$.

Program 5:

```
p5(x,y) = f_a(x,y,y+1);
f(x,y,z)  = if (y = z ∧ x > y − z) then
                 f_b(x−1,y,z)
              else if (x = y − z ∧ y ≠ 0) then
                 f_c(−x,y−1,z)
              else if (x < y ∧ y ≠ 0) then
                 f_d(x+1,y,z)
              else if (y < z ∧ x = y) then
                 f_e(x,y,y)
              else
                 ~  ;               // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_5) = y^2 + 3y - x + 1$.

Program 6:

```
p6(x,y,z)  = if (x ≠ 0 ∧ z ≥ 1) then
                 p6a(x − 1, y + 1, z)
             else if (x = 0 ∧ z ≥ 1) then
                 p6b(2y, 2y, z − 1)
             else
                 ~                  // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_6) = \frac{1}{6}((x+y)4^z + 6z + 2x - 4y - 6)$.

5.5 Commentary

A technique for calculating precise bounds on the runtime of a class of programs which are known to terminate has been explored. The technique begins with an assumption of the form and degree of the runtime, and is complete in the sense that if the program p is LA-SCT, and if the runtime is of the form $B_k(p)$, then a solution will be found.

The technique has application in the areas of precise runtime analysis, stack depth analysis for embedded systems, and in calculations of the relative execution path time (for compiler optimization). A prototype tool has been implemented, which accepts a limited ML-like programming language, and is described in [AKL07].

The technique is safe and complete for the particular class of programs we have considered. A particular form of exponential time-costs can be also relatively easily solved. In the case of the limited class of exponential time-costs, these solutions may still be expressed in terms of some unknowns, but these unknowns are resolved immediately by considering independent boundary cases for the function.

Part III

Conclusion

Chapter 6

Concluding remarks

In this concluding section the current direction of this research is outlined, with a summary of the contribution of the thesis.

6.1 Research directions

The research directions taken so far have been fairly fruitful, and the results are useful. Practical techniques for deciding termination properties of a large class of arbitrary programs have been developed, and also techniques for calculating (at least some of) their *precise* time and space complexities. Some preliminary steps towards tool support have been taken.

However the research has opened up many new questions, and there seem to be many paths to take:

1. Improving the algorithms
   (a) Finding better ways to reduce the complexity explosion problem
   (b) Integrating the techniques with other program analysis methods
2. Other application areas: optimizing compilers and general analysis
3. Improving, extending and changing the formalisms

6.1.1 Improving the algorithms

Each of the algorithms given here has room for improvement. In particular, the SCT algorithm has a complexity exponential in the number of variables, and the affine-SCT algorithm is no different. Improvements can be made by reducing the number of variables, and as well, the techniques for time and space cost calculation may be improved.

Reducing complexity. The affine SCT algorithm given in Section 3 seems to be a good compromise between preciseness and complexity. In the examples given, the translation and abstraction functions result in a complexity similar to that from LJB-analysis. By altering the translation and abstraction functions to give more precise analysis we can do a better analysis, at the expense of efficiency. However, the explosion of affine-graphs may make for intractable solutions.

Approaches to reducing the complexity are needed for the affine SCT algorithm, although even with six or more variables for each function and multiple entry points to multiple functions, the affine-based analysis is still very fast.

However, the cost analysis of (say) functions with 100 parameters, 100 variables and many entry points is not solvable using current systems. To manage such systems a suitable approach is to identify only those parameters and variables and entry points that are relevant to the cost analysis of the functions. The result of the graph composition and closure algorithm clearly identifies those parameters that are still relevant at each stage. A variation of the graph composition and closure algorithm could only carry through those parameters that are still relevant at each stage, giving a possible mechanism for reducing the complexity. In addition, the graph composition and closure algorithm is in some sense compositional: once a graph has been shown to be terminating, it can be used for the analysis of other functions which call it, without having to re-compute the closure.

The general problems of finding solutions for sets of polynomial equalities and inequalities have been studied for many years, and computational algebraic geometry is considered one of the core subjects of the pure mathematics curriculum.

In the work on time and space costs introduced in Chapter 5, the edge of the field has been explored, constructing an obvious (affine-space) geometrical system with known solutions. It is possible that a more cultured look at the geometries generated by this (or a similar) construction may lead to the use of other (more effective) techniques.

Integration with other analysis techniques. The geometric/arithmetic techniques exploited in this work are used largely on their own. The general method is a translation of some analysis method to make use of computationally efficient geometric/arithmetic techniques.

However, by integrating these techniques with other program analysis tools it may be possible to improve the analysis. Towards this end, the development of a “safe”

programming interface to the theorem prover HOL is in progress [Ng04]. The import of this project is that it fixed some inadequacies[1] of the PROSPER toolkit [DCN+03], allowing a relatively simple programming interface to be used between a theorem prover and a program analysis tool.

This API may be used to integrate the computationally efficient geometric/arithmetic program analysis techniques with other ones found in a HOL system for program analysis. This may work both ways, improving the termination, time or space cost analysis, and/or improving the HOL system program analysis.

6.1.2 Other application areas

At this stage, having established some results in a particular area, the general approach may have application in other areas not yet considered in detail. Consider the two possible application areas: optimizing compilers and general invariants.

Most modern compilers perform a flow analysis of the programs they are compiling, with a view to restructuring a program for efficiency. In the gcc compiler, a static branch predictor [BL93] uses heuristics to restructure the program code, optimizing the location of code for a branch more likely to occur.

It should be possible to integrate the precise time-cost calculation from Section 5 with a traditional static branch predictor in a compiler such as gcc, and verify the improvement to the resultant code.

6.1.3 Extending and changing the formalisms

The time (or space) cost calculated can be considered to be a program variable of a kind. It is possible to imagine a variable τ, which is the current value of time, and which increments by some amount for every instruction, and then use techniques to discover the relationship between τ and the other program variables. In a preliminary investigation into this, using the polynomial QE approach explored in Chapter 5, it was possible to calculate some other non-linear relationships (such as `log` or `min`).

For example, the system could discover an invariant polynomial relationship between the program variables, and if it discovered (say) that $\tau^2 - x = 0$, then we interpret this to mean that the runtime is $\tau = \log_2 x$. At this stage there appear to be some problems with the compositionality of the approach. In addition, τ is different from program variables in the sense that it only grows, and in the analysis this property is exploited. For instance, if the solution to a relationship between x and other program variables has two solutions a and b, then if x is a program *variable* it can have either

[1] The original API was unstable, and did not allow for persistent client programs. The new API is more stable, and allows clients to re-connect at any time.

the value a or b. However, if x is the program *runtime*, then the runtime is the minimum value of a and b, as once the program has terminated, it has terminated!

It is of interest to further study the relationship between time and traditional program invariants, with a view to clarify the advantages gained by separating time from traditional program invariants.

The termination results achieved so far apply directly to programs in a simple first-order functional language. The time and space cost results apply to a subset of these programs, where the guards for function calls are linear, and where the costs are restricted to either being a multi-variate polynomial in the program variables, or a restricted class of exponential costs. Weakening these restrictions will widen the applicability of the methods.

The direct calculation of exponential costs is also interesting. The time and space cost analysis can calculate exponential costs having a generic form like this:

$$B(p) = \phi_1 \cdot K^{\phi_2} + \phi_3$$

where ϕ_1, ϕ_2 and ϕ_3 are three polynomials. This allows for a runtime with quite a complex exponential form, but fails to capture many exponential cost programs. For example, the venerable Fibonacci function has exponential costs of the following form:

$$B(\text{fib}) = \phi_1 \cdot K_1^{\phi_2} + \phi_3 \cdot K_2^{\phi_4} + \phi_5$$

A quick check with `reduce` shows that given K_1 and K_2, it is possible to automatically derive $\phi_1 \dots \phi_5$. However it is not clear how to solve the general case of such programs. Some steps towards using QE in real exponential fields for this application have been taken.

Other classes of exponential programs may be amenable to a similar style of automatic calculation.

6.2 Summary of contribution

The thesis has three contributions to termination and size-change analysis, each contribution briefly summarized below.

The first contribution is an approach to improve the analysis of program termination properties based on the size-change termination method. Size-change graphs are encoded using Presburger formulæ representing affine relations, giving more refined size-change graphs. The algorithm for calculating the closure of the affine size-change graphs has been shown to terminate. Consequently, this affine-related analysis improves the effectiveness of the LJB termination analysis by capturing constant changes in parameter sizes, and affine relationships of the sizes of the source parameters.

The way in which closures are found is different from the normal approach of finding closures and fixed points. In this approach, the ultimate closure is expressed in terms of several closures called the idempotent graphs. This is similar to the idea of polyvariant program analysis [VB99, Con93], but differs in that there are also some graphs around during the analysis which cannot be made idempotent, yet are important for closure building. There are two ways to obtain polyvariant information for termination analysis: one way is to make use of the constraint enabling the call. Such constraints are commonly obtained from the conditional tests of the code leading to the call. The use of Presburger formulæ enables such information to be easily included in the analysis (as in Chapter 3), resulting in a context-sensitive analysis [NNH99, Shi88].

The second contribution is that the size-change principle has been extended with new techniques which allow the analysis technique to be applied to functions in which the return values are relevant to termination. In addition, the idea that termination can be derived for a changing argument that is bounded has been formally expressed. The use of guards associated with the abstract graphs enable us to extend this directly into the termination argument, without the limitations implied by the previous technique of constraints over program arguments. A new test for bounded-termination using this approach subsumes the previous test for (only) reducing parameters, is computationally inexpensive, and captures termination properties for a wide range of guarded recursive calls, including ones where the guard depends on even/odd properties, or combinations of increasing and decreasing parameters.

The third contribution is a technique for calculating precise bounds on the runtime of a class of programs, which are known to terminate. The technique begins with an assumption of the form and degree of the runtime, and is complete in the sense that if the program p is LA-SCT, and if the runtime is of the form $B_k(p)$ (a limited-degree polynomial in the program parameters), then a solution will be found. The technique has application in the areas of precise runtime analysis, stack depth analysis for embedded systems, and in calculations of the relative execution path time (for compiler optimization). A restricted class of exponential runtimes may also be derived using the same approach.

The techniques provide useful and practical approaches to the problems of termination and runtime analysis.

Bibliography

[AAMK06] S. Andrei, H. Anderson, G. Manolache, and S.C. Khoo. Runtime and Termination Analysis by Program Inversion. Technical Report TRA8/06, National University of Singapore, August 2006.

[Aik99] A. Aiken. Introduction to Set Constraint-Based Program Analysis. *Science of Computer Programming*, 35(1999):79–111, 1999.

[AK03a] H. Anderson and S.C. Khoo. Affine-based Size-change Termination. In Atsushi Ohori, editor, *APLAS 03: Asian Symposium on Programming Languages and Systems*, pages 122–140, Beijing, 2003. Springer Verlag.

[AK03b] H. Anderson and S.C. Khoo. Affine-Based Size-Change Termination. Technical Report TRA9/03, National University of Singapore, September 2003.

[AKAL05] H. Anderson, S.C. Khoo, S. Andrei, and B. Luca. Calculating Polynomial Runtime Properties. In *APLAS 05: Asian Symposium on Programming Languages and Systems*, pages 230–246, 2005.

[AKL07] H. Anderson, S.C. Khoo, and Y. Liu. A Tool for Calculating Exponential Runtimes. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC-2007)*, 2007.

[AMR04] C. Allauzen, M. Mohri, and B. Roark. A General Weighted Grammar Library. In *CIAA*, pages 23–34, 2004.

[Apt97] K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[Aug98] L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[Ave06] J. Avery. Size-change termination and bound analysis. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[BA02] A.M. Ben-Amram. General Size-Change Termination and Lexicographic Descent. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 2002.

[BA06] A.M. Ben-Amram. Size-Change Termination with Difference Constraints. submitted for publication, 2006.

[Bez93] M. Bezem. Strong Termination of Logic Programs. *Journal of Logic Programming*, 15(1-2):79–97, 1993.

[BL93] T. Ball and J.R. Larus. Branch Prediction For Free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[Blu04] W. Blum. Termination analysis of lambda calculus and a subset of core ML. Master's thesis, University of Oxford, http://william.famille-blum.org/, september 2004.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

[Bro95] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley Professional, August 1995.

[BvW98] R-J. Back and J. von Wright. *Refinement Calculus A Systematic Introduction.* Springer, 1998.

[CA69] S.A. Cook and S.O. Aanderaa. On the Minimum Computation Time of Functions. *Transactions of the AMS*, 142:291–314, 1969.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM Press, New York, NY.

[Ce98] B.F. Caviness and J.R. Johnson (eds.). *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer, 1998.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CK00] W.N. Chin and S.C. Khoo. Calculating Sized Types. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 62–72, 2000.

[Cob64] A. Cobham. The Intrinsic Computational Difficulty of Functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North-Holland, 1964.

[Col75] G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, Berlin, 1975. Springer.

[Con93] C. Consel. Polyvariant Binding-Time Analysis For Applicative Languages. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 66–77, 1993.

[Coo83] S.A. Cook. An overview of computational complexity. *Commun. ACM*, 26(6):400–408, 1983.

[CP90] T. Coquand and C. Paulin. Inductively Defined Types. In P. Martin-Lof and G. Mints, editors, *Proceedings of COLOG'88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. ACM, Springer, 1990.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code, 2006.

[CPR07] B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In J. Ferrante and K.S. McKinley, editors, *PLDI'07 : Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 320–330, San Diego, CA, USA, 2007. ACM.

[CS01] M. Colón and H. Sipma. Synthesis of Linear Ranking Functions. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81, London, UK, 2001. Springer-Verlag.

[CS02] M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer, 2002.

[CT97] M. Codish and C. Taboch. A Semantic Basis for Termination Analysis of Logic Programs and Its Realization Using Symbolic Norm Constraints. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97–HOA '97, Southampton, U.K., September 3–5, 1997*, volume 1298 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 1997.

[CT99] M. Codish and C. Taboch. A Semantic Basis for Termination Analysis of Logic Programs. *The Journal of Logic Programming*, 41(1):103–123, 1999. preliminary (conference) version in LNCS 1298 (1997).

[DCN+03] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer*, 4(2):189–210, 2003.

[DD94] D. De Schreye and S. Decorte. Termination of Logic Programs: the never-ending story. *The Journal of Logic Programming*, 19-20:199–260, 1994.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.

[Der95] N. Dershowitz. 33 Examples of Termination. In H. Comon and J.-P. Jouannaud, editors, *French Spring School of Theoretical Computer Science Advanced Course on Term Rewriting (Font Romeux, France, May 1993)*, volume 909, pages 16–26, Berlin, 1995. Springer-Verlag.

[DH88] J.H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1-2):29–35, 1988.

[Dij70] E.W. Dijkstra. Notes on Structured Programming. EWD249: circulated privately, April 1970.

[Dij82] E.W. Dijkstra. On weak and strong termination. In *Selected Writings on Computing: A Personal Perspective*, pages 355–357. Springer-Verlag, 1982.

[DLSS99] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Automatic Termination Analysis of Programs Containing Arithmetic Predicates. *Electronic Notes in Theoretical Computer Science*, 30(1), 1999.

[DLSS01] N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1–2):117–156, 2001.

[DM79] N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. *Commun. ACM*, 22(8):465–476, 1979.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[DP97] P. Dybjer and A. Pitts, editors. *Syntax and Semantics of Dependent Types*, volume Semantics of Logics of Computation. Cambridge University Press, 1997.

[DS97] A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 31(2):2–9, 1997.

[ea] H. Hong et al. `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–31, 1967.

[FR74] M.J. Fischer and M.O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, 1974.

[Gim95] E. Gimenez. Codifying Guarded Definitions with Recursion Schemes. In P. Dybjer and B. Nordstrom, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. ACM, Springer, 1995.

[GJ05] A.J. Glenstrup and N.D. Jones. Termination Analysis and Specialization-Point Insertion in Offline Partial Evaluation. *ACM Transactions on Programming Languages and Systems*, 27(6):1147–1215, 2005.

[Gri79] D. Gries. Is Sometimes Ever Better Than Always? *ACM Trans. Program. Lang. Syst.*, 1(2):258–265, 1979.

[Gri81] D. Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, 1981. GRI d 81:1 1.P-Ex.

[Gro01] B. Grobauer. Cost Recurrences for DML Programs. In *International Conference on Functional Programming*, pages 253–264, 2001.

[GTSKF04] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD thesis, CMU, 1992.

[Hin69] J.R. Hindley. An Abstract Form of the Church-Rosser Theorem. I. *J. Symb. Log.*, 34(4):545–560, 1969.

[HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 2001.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hof99] M. Hofmann. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science*, pages 464–473, 1999.

[Hon93] H. Hong. RISC-CLP(Real): Constraint Logic Programming over Real Numbers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM Principles of Programming Languages*, St Petersburg, Florida, January 1996.

[JB] N. Jones and N. Bohr. Termination Analysis of the Untyped λ-Calculus.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL*, pages 111–119, 1987.

[Jon03] N. Jones. Private communication. email, June 2003.

[KA05] S.C. Khoo and H. Anderson. Bounded Size-Change Termination. Technical Report TRB6/05, National University of Singapore, June 2005.

[Kap04] D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Proceedings of the 10th International Conference on Applications of Computer Algebra*. ACA and Lamar University, July 2004.

[Kar76] M. Karr. Affine Relationships Among Variables of a Program. *Acta Inf.*, 6:133–151, 1976.

[KMP+96] P. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Version 1.1.0 Interface Guide. Technical report, University of Maryland, College Park, November 1996.

[KS02] S.C. Khoo and K. Shi. Output Constraint Specialization. *ACM SIGPLAN ASIA Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 106–116, September 2002.

[Lee02] C.S. Lee. Program Termination Analysis in Polynomial Time. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 218–235. ACM, Springer, October 2002.

[LJBA01] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.

[LS97] N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs (with detailed experimental results). `http://www.cs.huji.ac.il/~naomil/`, 1997.

[LSS97] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A System for Checking Termination of Queries to Logic Programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.

[Mac71] M. Machtey. Classification of computable functions by primitive recursive classes. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 251–257, New York, NY, USA, 1971. ACM Press.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[MLP79] R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22:271–280, 1979.

[MM69] E. M. McCreight and A. R. Meyer. Classes of computable functions defined by bounds on computation: Preliminary Report. In *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*, pages 79–88, New York, NY, USA, 1969. ACM Press.

[MM70] Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. *Machine Intelligence*, 5:27–37, 1970.

[Mor94] C.C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 1994.

[MR67] A.R. Meyer and D.M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, pages 465–469, New York, NY, USA, 1967. ACM Press.

[Ned97] M-J Nederhof. Regular Approximations of CFLs: A grammatical view. In *International Workshop on Parsing Technologies*, pages 159–170, 1997.

[Ng04] W. Ng. Safe interface to a theorem prover. Honours year thesis, April 2004.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NO78] G. Nelson and D.C. Oppen. A simplifier based on efficient decision algorithms. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 141–150, New York, NY, USA, 1978. ACM Press.

[Nuu94] E. Nuutila. An Efficient Transitive Closure Algorithm for Cyclic Digraphs. *Information Processing Letters*, 52(4):207–213, 1994.

[Opp78] D.C. Oppen. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332, 1978.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PR04] A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.

[Pre27] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die addition als einzige Operation hervorstritt. *Comptes Rendus du Premier Congrès des Mathèmaticiens des Pays Slaves, (Warsaw, Poland)*, pages 92–101, 1927.

[Pre91] M. Presburger. On the Completeness of a Certain System of Arithmetic of Whole Numbers in Which Addition Occurs as the Only Operation. *History of Philosophy and logic*, 12(2):225–233, 1991. Translated from the German and with commentaries by Dale Jacquette.

[PW97] F.C.N. Pereira and R.N. Wright. Finite-State Approximation of Phrase-Structure Grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 149–173. MIT Press, Cambridge, 1997.

[RH03] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2003.

[Ric53] H. Rice. Classes of Recursively Enumerable Sets and their Decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[Rit63] R. W. Ritchie. Classes of Predictably Computable Functions. *Transactions of the AMS*, 106:139–173, 1963.

[RR00] R. Rugina and M.C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[San90] D. Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proceedings of the Third European Symposium on Programming*, number 432 in LNCS, pages 361–376. Springer-Verlag, May 1990.

[Shi88] O. Shivers. Control Flow Analysis in Scheme. *ACM SIGPLAN Notices*, 7(1):164–174, 1988.

[SJ05] D. Sereni and N.D. Jones. Termination Analysis of Higher-Order Functional Programs. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2005.

[SSM04] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 318–329, 2004.

[SSS97] C. Speirs, Z. Somogyi, and H. Sondergaard. Termination Analysis for Mercury. In *Static Analysis Symposium*, pages 160–171, 1997.

[Tar51] A. Tarski. A decision method for elementary algebra and geometry. In *A decision method for elementary algebra and geometry. Prepared for publication by J.C.C. Mac Kinsey*. Berkeley, 1951.

[TG05] R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 16(4):229–270, June 2005.

[Tur37] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. LMS, Series 2*, 42:230–265, 1936-1937.

[VB99] W. Vanhoof and M. Bruynooghe. Binding-time Analysis for Mercury. In *International Conference on Logic Programming*, pages 500–514, 1999.

[Wei99] V. Weispfenning. Mixed Real-Integer Linear Quantifier Elimination. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))*, 1999.

[Wei00] V. Weispfenning. Deciding linear-exponential problems. *SIGSAM Bull.*, 34(1):30–31, 2000.

[Wil04] R. Wilhelm. Timing Analysis and Timing Predictability. In *FMCO*, pages 317–323, 2004.

[Xi98] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[Xi02] H. Xi. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, October 2002.

[XP99] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[Yam62] H. Yamada. Real-Time Computation and Recursive Functions Not Real-Time Computable. *IRE Transactions on Electronic Computers*, 11:753–760, 1962.