Calculating Polynomial Runtime Properties

Hugh Anderson, Siau-Cheng Khoo, Stefan Andrei and Beatrice Luca

Department of Computer Science
School of Computing
National University of Singapore
{hugh,khoosc,andrei,lucabeat}@comp.nus.edu.sg

Abstract. Affine size-change analysis has been used for termination analysis of eager functional programming languages. 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 paper we show how precise polynomial bounds on such costs may be calculated on programs, by a characterization as a problem in quantifier elimination. The technique is decidable, and 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. We demonstrate the new technique 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.

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 we may be interested in 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 [2] 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.

In this paper we explore the automatic calculation of each of these costs through static analysis of the source of programs which are known to be affine size-change terminating [1, 13], where the focus is on recording parameter size-changes only. The overall approach has three steps:

1. Assume a (degree k) polynomial upper bound related to the runtime or space cost. The polynomial variables are the parameter *sizes*.
2. Derive from the source a set of equations constrained by this upper bound.
3. 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 we repeat the process with a degree $k + 1$ assumption. This technique is surprisingly useful, and it is possible to derive precise runtime bounds on non-trivial programs.

We can also 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 run-time analysis for functional programs. For example, [16] 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 [7], Grobauer explores the use of recurrences to evaluate a DML program's execution costs. Our focus is more with decidability aspects and precise time-costs than either of these approaches.

An alternative approach is to limit the language in some way to ensure a certain run-time complexity. For example, in [8], 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 our work, we generate time and stack costs of arbitrary functions or procedures, through analysis of the derived size-change information. A compact summary of a general technique for the calculation of time and space efficiency is found in the book [15] 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.

Our paper improves the technique 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, we can derive a decidable formula, the solution of which gives the time or space cost of the program.

In the approach presented here, we measure runtime 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.

In Section 2, preliminary concepts and definitions are introduced. In Section 3, the framework used for constructing the equations is introduced, along with practical techniques that may be used to solve the equations. In Section 4, we show examples of relative time costs for compiler optimization, and calculation of stack depth. In Section 5, we use recurrence relations to indicate how to classify costs into polynomial or exponential forms. In Section 6, exponential cost calculations are explored. We conclude in Section 7.

2 Preliminaries

v	$\in$ `Var`	⟨Variables⟩
f, g, h	$\in$ `PName`	⟨Procedure names⟩
n	$\in \mathbb{Z}$	⟨Integer constants⟩
β	$\in$ `Guard`	⟨Boolean expressions⟩
	$\beta ::= \delta \mid \neg\beta \mid \beta_1 \vee \beta_2 \mid \beta_1 \wedge \beta_2$	
	$\delta ::=$ True $\mid$ False $\mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2$	
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 1. The language syntax

The language is a simple procedural language, defined in Table 1. This language is 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. Finally, an important point is that the language only admits affine relations between the program variables and expressions.

2.1 Runtime analysis

In the process of performing size-change termination analysis described in [14], 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 [1, 13], and hence are known to terminate. A subclass of these functions in which argument size-changes are linear, termed LA-SCT (Linear-affine SCT programs) define the class of programs analysed here. Limiting our analysis to this class of functions is not a severe restriction, as most useful size-change parameter changes would be linear.

We begin by formally defining the runtime of such functions. The term $\bar{y}$ refers to the vector $(y_1, \ldots, y_n)$. For the sake of notational brevity, we use a *contextual notation* 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 1. *Given an LA-SCT program p with program parameters $\bar{x}$ and body e_p and input arguments $\bar{n}$, the runtime of p, $B(p)[\bar{n}/\bar{x}]$, is defined by the runtime of e_p inductively as follows:*

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

In practical terms, this indicates that we are *counting* function calls as a 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, we choose to limit the definition to the analysis of function calls as a measure of runtime.

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

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

We are primarily interested in runtimes that can be expressed as a polynomial in the parameter variables.

Definition 2. *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:*

$$B_k(p) \stackrel{\text{def}}{=} c_1 x_1^k + c_2 x_2^k + \ldots + c_m x_m^k + c_{m+1} x_1^{k-1} x_2 + \ldots + c_n$$

where $c_1 \ldots c_n \in \mathbb{Q}$, and $B_k(p)$ is the runtime of the program.

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

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

Lastly, we differentiate between an assumption $A(p)$ of the runtime of a program p, and the actual runtime $B(p)$.

Definition 3. *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:*

$$A(p) \stackrel{\text{def}}{=} c_1 x_1^k + c_2 x_2^k + \ldots + c_m x_m^k + c_{m+1} x_1^{k-1} x_2 + \ldots + c_n$$

where $c_1 \ldots c_n$ are unknown. $A(p)$ contains all possible terms of degree at most k formed by the product of parameters of p. Note that in this presentation, we search for an assignment $[\theta]$ to the constants $c_1 \ldots c_n$ 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

$$A_1(p) = c_1 x + c_2 y + c_3$$
$$A_2(p) = c_1 x^2 + c_2 y^2 + c_3 xy + c_4 x + c_5 y + c_6$$
$$A_3(p) = c_1 x^3 + c_2 y^3 + c_3 x^2 y + c_4 x y^2 + c_5 x^2 + c_6 y^2 + c_7 xy + c_8 x + c_9 y + c_{10}$$

In this presentation, we capture runtime behaviour 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 ψ_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.

3 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 ...
```

We can represent the runtime properties for each path through the program p_1 with the three equations:

$$A_2(p_1)[x \mapsto y, y \mapsto y - 1] - A_2(p_1) + 1 = 0$$
$$A_2(p_1)[x \mapsto x - 1] - A_2(p_1) + 1 = 0$$
$$A_2(p_1) = 0$$

which reduce to:

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

We wish to find suitable values for the (real-valued) coefficients $c_1 \ldots c_6$. That is, we want to *eliminate the universally quantified elements* of the equalities.

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 [18]. 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, we can express our QE problem as the single formula[1]:

$$\begin{aligned} \forall x, y: \quad & \begin{pmatrix} x = 0 \\ \wedge\ y \geq 1 \end{pmatrix} \Rightarrow A_2(p_1)[x \mapsto y, y \mapsto y - 1] - A_2(p_1) + 1 = 0 \\ \wedge \quad & (x \geq 1) \Rightarrow \quad A_2(p_1)[x \mapsto x - 1] - A_2(p_1) + 1 = 0 \\ \wedge \quad & \begin{pmatrix} x = 0 \\ \wedge\ y = 0 \end{pmatrix} \Rightarrow \quad A_2(p_1) = 0 \end{aligned}$$

In [11], 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 [17].

Our construction is different, and in a different field (program running time rather than program invariants). We construct expressions characterizing the program run time as a constraint quantified over the program parameters. The constraint constants are then solved by QE, and algebraic reduction.

3.1 Quantifier elimination

In 1973, Tarski's method was improved dramatically by the technique of Cylindrical Algebraic Decomposition (CAD) first described in [4]. The book [3] has a 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

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

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

Our approach 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. This subset is of the form

$$T[A(p)] = \begin{pmatrix} \forall \overline{x}, \overline{y}, \ldots & g_1 & \Rightarrow & F_1 \\ & \wedge\ g_2 & \Rightarrow & F_2 \\ & \wedge\ \ldots\ldots\ldots & & \end{pmatrix}$$

where $g_1, g_2, \ldots$ identify different paths from $p(\overline{a})$ to enclosed function calls $f_i(\overline{b})$[2]. $F_1, F_2, \ldots$ are formulæ derived from the program p source such that

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

The following inference rules 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.

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

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

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

$$\frac{\Gamma \vdash s_1 \ : \ A(s_1) \qquad \Gamma \vdash s_2 \ : \ A(s_2)}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \ : \ \begin{cases} c \ : \ A(s_1) \\ \wedge\ \neg c \ : \ A(s_2) \end{cases}} \quad \textbf{B-if}$$

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

$$\frac{\Gamma, \langle \bar{x} \rangle_f \vdash s \ : \ A(s)}{\Gamma \vdash p_1 \, ; \, f(\bar{x}) \stackrel{\text{def}}{=} s \ : \ A(s)} \quad \textbf{B-defs}$$

Note that each application of a rule preserves the runtime of the statement. In addition, a substitution ψ 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.

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

For example, for the program p_1 the above rules generate

$$A_2(p_1) \;=\; \begin{pmatrix} (x=0 \wedge y=0): & 0 \\ \wedge\,(x=0 \wedge y \geq 1): & A_2(p_1)[x \mapsto y, y \mapsto y-1]+1 \\ \wedge\quad\;\; (x \geq 1): & A_2(p_1)[x \mapsto x-1]+1 \end{pmatrix}$$

and the equation $T[A_2(p_1)]$ derived is thus:

$$T[A_2(p_1)] \;=\; \begin{pmatrix} \forall x,y: & \begin{pmatrix} x=0 \\ \wedge\; y=0 \end{pmatrix} \Rightarrow & A_2(p_1)=0 \\ & \wedge \begin{pmatrix} x=0 \\ \wedge\; y \geq 1 \end{pmatrix} \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1]-A_2(p_1)+1=0 \\ & \wedge \quad (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1]-A_2(p_1)+1=0 \end{pmatrix}$$

and our 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.

The following theorem asserts that the solution of the formula $T[A_k(p)]$ correctly represents the runtime $B_k(p)$ of any LA-SCT program p with a degree-k polynomial runtime.

Theorem 1. *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 the definition for $B_k(p)[\bar{n}/\bar{x}]$.

3.2 Tool support

There exists a range of tools capable of solving this sort of reduction. The tool `QEPCAD` [6] 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 [5] 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;
```

In line 1 of the above sequence, we define 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}}}
```

Finally, in line 5, we substitute the solution instance 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 programming based solution to these sort of problems, but most such systems can only handle linear equations, not the polynomial ones used here.

There are constraint solving systems, for example `RISC-CLP(Real)` [9], which use (internally) CAD quantifier elimination to solve polynomial constraints, however here we prefer to restrict ourselves to just the underlying techniques, and not clutter up the discussion with other, perhaps confusing, properties of constraint solving systems.

4 Calculating other program costs

So far we have limited the presentation 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.

4.1 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 in this program, we have the sequential composition of two function calls, and this program has an exponential runtime cost. An interesting question for this program is to calculate its maximum stack depth. 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, we record not the sum of the two functions composed, but the maximum value of the two functions:

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

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

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

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

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

4.2 Relative runtime costs

The third motivation for this approach was to derive relative costs for the different possible paths through a program. For example 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.

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:

$$T[A(p_{1a})] = \left(\begin{array}{lll} \forall x, y: & \left(\begin{array}{r} x = 0 \\ \wedge\ y = 0 \end{array} \right) \Rightarrow & A_2(p_1) = 0 \\ & \wedge \left(\begin{array}{r} x = 0 \\ \wedge\ y \geq 1 \end{array} \right) \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) + 1 = 0 \\ & \wedge \quad (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) = 0 \end{array} \right)$$

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

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

$$T[A(p_{1b})] = \left(\begin{array}{lll} \forall x, y: & \left(\begin{array}{r} x = 0 \\ \wedge\ y = 0 \end{array} \right) \Rightarrow & A_2(p_1) = 0 \\ & \wedge \left(\begin{array}{r} x = 0 \\ \wedge\ y \geq 1 \end{array} \right) \Rightarrow & A_2(p_1)[x \mapsto y, y \mapsto y-1] - A_2(p_1) = 0 \\ & \wedge \quad (x \geq 1) \Rightarrow & A_2(p_1)[x \mapsto x-1] - A_2(p_1) + 1 = 0 \end{array} \right)$$

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

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.

5 Towards a classification of 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, 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.

We have an approach to solving programs of this form, but it requires us to find some way of classifying program costs into either polynomial or exponential.

In this section we present a characterization of the problem as a recurrence, explaining the choice of the particular class of exponential cost programs that can be solved.

The previous discussion employs a simple translation from program source to a decidable Tarski formula. However this approach gives no indication of the runtime cost for a function. For example, if we started assuming the program was polynomial, the algorithm indicates that we should try a degree-2 assumption, followed by a degree-3 assumption and so on. There is no indication as to when we should give up. Towards this, we consider a *flattened* version of the original program source, in which an arbitrary collection of functions is flattened into a single function which calls itself. This new flattened source can be easily characterized as a recurrence relation, and the solutions to the recurrence relations give indications of the maximum polynomial degree.

In addition this presentation highlights a particular class of exponential programs that can be solved.

A flattened version of an arbitrary program is easily derived in the absence of mutual recursion. However, in the case of mutually recursive functions, it is not as clear how a program may be transformed. The papers [19, 12] contain necessary and sufficient conditions to transform all mutual recursion to direct or self-recursion.

Supposing that our programs are transformed into equivalent programs which are using only self-recursion, we can define a *self-recursive normal form* over a representation of the state of the program variables at any time. Consider an m-dimensional array a, indexed by the values of parameters $n_1 \dots n_m$ to the self-recursive program $p(n_1 \dots n_m)$:

Definition 5. *The array $a_{n_1,\dots,n_m}$ is in linear self-recursive normal form iff it is defined as:*

$$a_{n_1,\dots,n_m} = a_{f_1(n_1,\dots,n_m),\dots,f_m(n_1,\dots,n_m)} + g(n_1,\dots,n_m) \tag{1}$$

where $f_i(n_1,\dots,n_m) = k_{i,1}\cdot n_1 + \dots + k_{i,m}\cdot n_m + k_{i,m+1}$, $\forall k_{i,j} \in \Re$, $\forall i \in \{1,\dots,m\}$, $\forall j \in \{1,\dots,m+1\}$, and $g(n_1,\dots,n_m) = k_1 \cdot n_1 + \dots + k_m \cdot n_m + k_{m+1}$.

The above recurrence (1) is supposed to iterate for an arbitrary finite number of times, say ℓ. We shall explore the expression obtained from (1) after applying the substitution $n_i \to f_i(n_1,\dots,n_m)$, $\forall i \in \{1,\dots,m\}$ for ℓ times.

Theorem 2. *All linear self-recursive normal forms have a solution.*

Proof. (By construction). Denoting by $\overline{n}$ the vector $(n_1,\dots,n_m)$, the first iteration of (1) leads to:

$$a_{f_1(\overline{n}),\dots,f_m(\overline{n})} = a_{f_1(f_1(\overline{n}),\dots,f_m(\overline{n})),\dots,f_m(f_1(\overline{n}),\dots,f_m(\overline{n})))} + g(f_1(\overline{n}),\dots,f_m(\overline{n})) \tag{2}$$

In order to write this more compactly, let us inductively define the notations

$$\left\langle f_{1,m}^{(1)}(\overline{n})\right\rangle \stackrel{\text{def}}{=} (f_1(\overline{n}),...,f_m(\overline{n}))$$
$$\left\langle f_{1,m}^{(\ell)}(\overline{n})\right\rangle \stackrel{\text{def}}{=} \left(f_1\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right),\ldots,f_m\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right)\right) \quad \text{for } \ell \geq 2$$

where $\left\langle f_{1,m}^{(1)}(\overline{n})\right\rangle$ is a compressed form of $\langle f_{1,m} \circ ... \circ f_{1,m}(\overline{n})\rangle$, and "$\circ$" stands for the function composition. In this way, the recurrence (2) can be re-written as:

$$a_{\left\langle f_{1,m}^{(1)}(\overline{n})\right\rangle} = a_{\left\langle f_{1,m}^{(2)}(\overline{n})\right\rangle} + g\left(\left\langle f_{1,m}^{(1)}(\overline{n})\right\rangle\right) \tag{2a}$$

The given substitution can be further applied $\ell - 1$ times, to obtain:

$$a_{\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle} = a_{\left\langle f_{1,m}^{(\ell)}(\overline{n})\right\rangle} + g\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right) \tag{ℓ}$$

By combining the recurrences $(1)\ldots(\ell)$, we obtain an expression for $a_{\overline{n}}$:

$$a_{\overline{n}} = a_{\left\langle f_{1,m}^{(\ell)}(\overline{n})\right\rangle} + g\left(\left\langle f_{1,m}^{(1)}(\overline{n})\right\rangle\right) + \ldots + g\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right) \tag{I}$$

By replacing $f_i(n_1,...,n_m)$ with $k_{i,1}\cdot n_1+\ldots+k_{i,m}\cdot n_m+k_{i,m+1}$, $\forall i \in \{1,\ldots,m\}$, we get the general form:

$$\left\langle f_{1,m}^{(l)}(\overline{n})\right\rangle = (E_{1,\ell},\ldots,E_{m,\ell})$$

where $E_{i,l}$ is:

$$\sum_{i_l=1}^{m}...\sum_{i_1=1}^{m} k_{i,i_1}\cdot\ldots\cdot k_{i_{l-1},i_l}\cdot n_{i_l} + \sum_{i_{l-1}=1}^{m}...\sum_{i_1=1}^{m} k_{i,i_1}\cdot\ldots\cdot k_{i_{l-1},m+1}+\ldots+\sum_{i_1=1}^{m} k_{i,i_1}\cdot\ldots\cdot k_{i_1,m+1}$$

We have established a solution for all recurrences of the self-recursive normal form defined before, and this confirms the completeness for this class of recursive programs. □

For ease of presentation, and in order to see the complexity of $a_{\overline{n}}$ from (I), let us highlight only the last (dominant) term. It is:

$$g\left(\left\langle f_{1,m}^{(\ell-1)}(\overline{n})\right\rangle\right) = k_1\cdot E_{1,l-1}+\ldots+k_m\cdot E_{m,l-1}+k_{m+1}$$

Looking at the general form of the dominant term, namely

$$k_i\cdot\sum_{i_{l-1}=1}^{m}...\sum_{i_1=1}^{m} k_{i,i_1}\cdot\ldots\cdot k_{i_{\ell-1},m+1}+\ldots+\sum_{i_1=1}^{m} k_{i,i_1}\cdot\ldots\cdot k_{i_1,m+1}$$

we observe that very few cases correspond to a polynomial as an expression for $a_{\overline{n}}$. Because of the large number of coefficients in the expression of $a_{\overline{n}}$, it is almost impossible to provide a precise boundary between the cases when $a_{\overline{n}}$ is a polynomial and when it is an exponential. However, the formula does immediately give the following classifications:

1. if $\forall i \in \{1, \ldots, m\}$, we have $k_i = 0$, then $a_{\overline{n}} = \ell \cdot k_{m+1}$ is a polynomial in ℓ of degree 1;
2. if $m = 1$ then
   (a) if $k_{1,1} = 1$ then $a_{\overline{n}}$ is a polynomial of degree 2;
   (b) if $k_{1,1} \neq 1$ then $a_{\overline{n}}$ is an exponential of base $k_{1,1}$.
3. if $\exists i \in \{1, \ldots, m\}$ such that $k_i \neq 0$ and $\exists u, v \in \{1, \ldots, m\}$ such that $k_{u,v} \notin \{0, 1\}$ then $a_{\overline{n}}$ contains at least one exponential of base $k_{u,v}$.

The third classification above covers a considerable number of situations when $a_{\overline{n}}$ is an exponential.

A useful slight generalization of recurrence (1) can be done by taking g as a non-linear polynomial. It is easy to see that if $m = 1$, and $k_{1,1} = 1$, then for a polynomial g of degree k, the solution of $a_{\overline{n}}$ is a polynomial of degree $k + 1$. In this way, we enlarge the class of self-recursive normal form equations.

5.1 A case-study

Let us take a useful example which corresponds to particular values for m, followed by a practical application of its use in computing the runtime of a given program.

When trying to compute the runtime cost of p_3, we get the following identities, formed by a guard and a recurrence relation:

$$\begin{aligned} x \neq 0 \wedge n \geq 1 \quad &\text{implies} \quad B(x, y, n) = B(x - 1, y, n) + 1 \\ x = 0 \wedge n > 1 \quad &\text{implies} \quad B(x, y, n) = B(2y + n, 2y, n - 1) + 1 \end{aligned}$$

By inspection of the first identity, and by iterating $x \to x - 1$ for x times, we get $B(x, y, n) = B(0, y, n) + x$. By applying the second identity, we have $B(0, y, n) = B(2y + n, 2y, n - 1) + 1 = B(0, 2y, n - 1) + 2y + n + 1$. We rewrite this latter identity, omitting the first argument (without loss of generality), to the equivalent recurrence relation:

$$a_{y,n} = a_{2y,n-1} + 2y + n + 1$$

This is a particular instance of recurrence (1), where m is replaced by y, and $f(n, m) = 2m$, $g(n, m) = 2m + n + 1$. Since $k_1 = 2$, the solution of $a_{y,n}$ is an exponential (case 2(b)). This implies that our automated tool should be fed with an input having a generic form like this:

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

This allows for a runtime with quite a complex exponential form.

6 Exponential cost calculations

Having established a classification of program costs, we now revert to the original approach, where we assume an exponential runtime A for the program, initially for a base of K, and using polynomials of (say) degree 2. The assumed runtime is $A(p_3) = \phi_1 \cdot K^{\phi_2} + \phi_3$, 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 relation holds:

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

This relationship between the polynomials may be exploited by constructing the equations in a similar form to the previous presentation, 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:

$$\left(\begin{array}{l} \forall x, y, n : \quad \begin{pmatrix} x, y > 0 \\ \wedge \quad n \geq 0 \end{pmatrix} \Rightarrow \left(\begin{array}{c} \phi_3[\psi_{3a}] - \phi_3 + 1 = 0 \\ \wedge \\ \begin{pmatrix} (\phi_1[\psi_{3a}] = \phi_1 \quad \wedge \phi_2[\psi_{3a}] = \phi_2) \\ \vee \quad (\phi_1[\psi_{3a}] = K\phi_1 \wedge \phi_2[\psi_{3a}] = \phi_2 - 1) \\ \vee \ (K\phi_1[\psi_{3a}] = \phi_1 \quad \wedge \phi_2[\psi_{3a}] = \phi_2 + 1) \end{pmatrix} \end{array}\right) \\ \\ \qquad \wedge \quad \begin{pmatrix} x = 0 \\ \wedge \ y > 0 \\ \wedge \ n \geq 0 \end{pmatrix} \Rightarrow \left(\begin{array}{c} (\phi_3[\psi_{3b}] - \phi_3 + 1 = 0) \\ \wedge \\ \begin{pmatrix} (\phi_1[\psi_{3b}] = \phi_1 \quad \wedge \phi_2[\psi_{3b}] = \phi_2) \\ \vee \quad (\phi_1[\psi_{3b}] = K\phi_1 \wedge \phi_2[\psi_{3b}] = \phi_2 - 1) \\ \vee \ (K\phi_1[\psi_{3b}] = \phi_1 \quad \wedge \phi_2[\psi_{3b}] = \phi_2 + 1) \end{pmatrix} \end{array}\right) \end{array}\right)$$

$T[A(p_3)]$ is easily reduced by `redlog`, giving a family of solutions for the bounds:

$$A(p_3) = \alpha y 2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y + c_{30}$$

where α indicates that any value here might be a solution, and c_{30} is unknown. To constrain the solution further, we add in boundary cases for the system, for example $A(p_3(0,1,1)) = 0$, $A(p_3(0,2,1)) = 0$, giving:

$$B(p_3) = y 2^n + \frac{1}{2}n^2 + \frac{3}{2}n + x - 2y - 2$$

6.1 Another example

Despite the simplicity of program p_2 introduced in subsection 4.1, a translation to a single-term recurrence is not obvious. The function would have to be flattened, generating extra guards and program parameters. However, the QE formulation is still automatic and simple, deriving the equation for the runtime cost $T[A(p_2)]$:

$$\left(\begin{array}{l} \forall x, y: \quad (x = 0 \wedge y \geq 1) \Rightarrow (\phi_2[\psi_{2a}] + \phi_3[\psi_{2b}] - \phi_3 + 2 = 0) \\ \\ \qquad \wedge (x \geq 10 \wedge y \geq 0) \Rightarrow \left(\begin{array}{c} (\phi_3[\psi_{2c}] - \phi_3 + 1 = 0) \\ \wedge \\ \left(\begin{array}{ll} (\phi_1[\psi_{2c}] = \phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2) \\ \vee\ (\phi_1[\psi_{2c}] = K\phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2 - 1) \\ \vee\ (K\phi_1[\psi_{2c}] = \phi_1 & \wedge\ \phi_2[\psi_{2c}] = \phi_2 + 1) \end{array}\right) \end{array}\right) \end{array}\right)$$

Given two independent base cases, `redlog` immediately finds the runtime cost:

$$B(p_2) = 4 * 2^y + x - y - 4$$

We have found it relatively easy to automatically derive exponential runtimes for programs like these, with polynomials of small degree.

7 Conclusion

In this paper, we have shown 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 $B_k(p)$, 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).

We have shown that the technique is safe and complete for the particular class of programs we have considered. In addition, we have presented an approach to classifying the costs into either polynomial or exponential time costs using a recurrence-relation complexity analysis. This outlined a particular form of exponential time-costs that can be 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.

Acknowledgments: We thank Professor Neil Jones for initially proposing the form of this challenging problem [10].

References

1. 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.
2. T. Ball and J.R. Larus. Branch Prediction For Free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.
3. B.F. Caviness and J.R. Johnson (eds.). *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, 1998.
4. G.E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, volume 33, pages 134–183, Berlin, 1975.
5. 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.
6. H. Hong et al. `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.
7. B. Grobauer. Cost Recurrences for DML Programs. In *International Conference on Functional Programming*, pages 253–264, 2001.
8. M. Hofmann. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science*, pages 464–473, 1999.
9. 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.
10. N. Jones. Private communication. June 2003.
11. 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.
12. O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the Conversion of Indirect to Direct Recursion. *LOPLAS*, 2(1-4):151–164, 1993.
13. S.C. Khoo and H. Anderson. Bounded Size-Change Termination. Technical Report TRB6/05, National University of Singapore, June 2005.
14. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Conference Record of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.
15. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2003.
16. 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.
17. 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.
18. A. Tarski. In *A decision method for elementary algebra and geometry. Prepared for publication by J.C.C. Mac Kinsey*. Berkeley, 1951.
19. T. Yu and O. Kaser. A Note on "On the Conversion of Indirect to Direct Recursion". *ACM Trans. Program. Lang. Syst.*, 19(6):1085–1087, 1997.

A More examples

The following extra 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( x, 0 )
               else
                  p4a( x + 1, y − 1 );
g( x, y )    = if (x ≤ 0) then
                  ~                    // ... exit ...
               else
                  g( 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( x, y, y + 1 );
f( x, y, z )  = if (y = z ∧ x > y − z) then
                  fa( x − 1, y, z )
               else if (x = y − z ∧ y ≠ 0) then
                  fb( −x, y − 1, z )
               else if (x < y ∧ y ≠ 0) then
                  fc( x + 1, y, z )
               else if (y < z ∧ x = y) then
                  fd( 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 $B_2(p_6) = \frac{1}{6}((x+y)4^z + 6z + 2x - 4y - 6)$.

Program 7:

With a refinement of our approach not explored in this paper, we can also derive minimum values for costs, not just polynomial or restricted exponential costs. For example:

```
p7( x, y )   = if (x ≥ 1 ∧ y ≥ 1) then
                  p7a( x − 1, y − 1 )
               else
                  ~                    // ... exit ...
```

The solution returned by `redlog` is that $B_2(p_7) = \min(x, y)$.