Classification of recursive functions into polynomial and superpolynomial complexity classes

Carsten Schürmann and Jatin Shah

Department of Computer Science, Yale University, New Haven, CT 06511.

Abstract. We present a decidable and sound criterion for classifying recursive functions over higher-order data structures into polynomial and superpolynomial complexity classes generalizing the seminal results by Bellantoni and Cook [1] and Leivant [4] to complex structural datatypes. The criterion is complete for the special case of binary strings; whether it is also complete for arbitrary higher-order data structures remains an open problem. Logic programming serves as the underlying model of computation and our results apply to the Horn fragment as well to the fragment of hereditary Harrop formulas.

1 Introduction

Cobham [3] gave the first functional characterization of a complexity class by exhibiting an equational schema that generates precisely the poly-time functions over natural numbers. Later, in their seminal papers, Bellantoni and Cook [1], and Leivant [2, 4] have given a recursion-theoretic characterization of polynomial time computable functions. Based on that work, Bellantoni, *et al.* [5] and Hofmann [6] have developed type systems where all well-typed functions correspond to polynomial time computable functions. On the other hand, Ganzinger and McAllester [7], and Givan and McAllester [8] have given various criteria for identifying polynomial time predicates in bottom-up logic programs. In this paper, we chose to represent recursive functions as relations between input and output arguments hereby lifting the underlying model of computation away from Turing and RAM machines into the world of logic programming and the search for uniform derivations and Abstract Logic Programming Languages (ALPL) [9, 10]. We measure the complexity of an algorithm as the size of the resulting derivation (should it exist), in terms of the size of the input arguments. The main contribution of this paper is a decidable criterion that decides if a logic program runs in polynomial time. Our criterion runs in time depending only on the size of the program and is independent of the inputs to the program.

Interpreting functions as relations and logical programs allows us to reason about the run-time of an algorithm in terms of the size of a uniform derivation where each inference is counted as one computation step, and not in terms of the number of execution steps of a RAM or Turing machine. Functions as relations alleviate many restrictions commonly associated with RAM machines. Arguments that are usually elements of a freely generated term algebra need not be artificially encoded as data objects in RAM memory, but can be analyzed and constructed as they are. The technique developed in this paper has many applications as many logical formalisms possess a uniform derivations property, for example, the Horn or hereditary Harrop fragment of first-order logic, and even the logical framework LF. Thus, the complexity of computational counterparts that are commonly associated with representational particularities, for example, dependently typed, linear, or higher-order encodings, can be easily studied in extensions to the framework presented here.

The search for uniform derivations is in general parametrized by a unification algorithm. For the purpose of this work, where functions are encoded as logic programs, all input arguments may be assumed to be ground (i.e. terms that do not contain free variables). Every subgoal in the logic program will have a logic variable in its output position, to bind (but not match) the result of the subcomputation, which will be a ground term again. The running time of the unification algorithm is a substantial part of the overall proof search runtime. Therefore, we require of each function we analyze to trigger only runs of the unification algorithms whose running time can be bounded by a polynomial in the size of the pattern. Different term algebras require domain specific unification algorithms. For example, in our setting, the runtime of first-order unification restricted to linear patterns is bounded by a polynomial in the size of the pattern. Higher-order unification with dependent types is in general not decidable unless it is restricted to Miller patterns [11] (each existential variable is applied only to pairwise distinct parameters). In addition, variables must occur linearly, which can only be achieved if redundant information is removed from the pattern ahead of time.

The central contribution of this paper is a criterion that decides if a logic program can be executed in polynomial time. Informally, it consists of two parts. The first part requires that the sum (of the sizes) of all recursive arguments is not larger than the sum (of the sizes) of all input arguments to the function. In the second part, we require that all auxiliary (non-recursive) functions that take recursively computed arguments as inputs are non-size-increasing. If these conditions are satisfied, we show that the search for uniform derivations will terminate in a number of steps that is bounded by a polynomial in the size of the input arguments. The technique applies to various logical formalisms and handles the case of higher-order encodings correctly.

The paper is organized as follows. First, we give a general theorem for computing complexity of integer-valued recursive functions. The criteria we describe in Section 3 for classifying recursive functions are based on this theorem. We develop these conditions in two stages. In Stage 1 we restrict ourselves to functions where outputs of recursive calls are not provided as input to auxiliary functions used. In Stage 2, we relax this condition, but now require that these auxiliary functions are non-size-increasing. Aehlig, *et al.* [12] and Hofmann [13, 14] have also used the latter condition (originally proposed by Caseiro [15]) to extend Hofmann's polynomial-time type system to include a larger class of functions. Finally, in Section 4, we argue that our results can be extended to higher-order hereditary harrop formulas and illustrate the expressiveness of our results with some examples. The proofs of all theorems in this paper are given in the Appendix.

2 Functions as logic programs

We are primarily interested in studying general recursive functions and classifying their running time into complexity classes using syntactic criteria. We shall represent recursive function by predicates where a function $(z_1, \ldots, z_n) = f(x_1, \ldots, x_l; y_1, \ldots, y_m)$ is denoted by the predicate $P_f(x_1, \ldots, x_l; y_1, \ldots, y_m; z_1, \ldots, z_n)$ where x_i, y_i and z_i correspond to recursive input arguments, non-recursive input arguments and output arguments respectively. We represent function computation by restricting ourselves to a particular subclass of logic programs whose argument positions have a well-defined meaning with respect to input and output behavior of ground terms. A *ground term* is a term not containing any variables. The value of the function is computed by performing a proof search where the input arguments are ground and the output arguments are free logic variables. On successful return, the output arguments are ground. Such logic programs are considered to have a well-defined *mode* behavior. The reader may refer **[[cite]]** for more information on algorithms for identifying *mode correct* logic programs.

2.1 Logic programming model of computation

We shall present our results in the logic programming language shown below. The goals G and clauses D are represented using Horn clauses and the terms L_i, M_i and N_i are simply-typed λ-terms in canonical form. Predicates are given by

$$P_f(L_1, \ldots, L_l; N_1, \ldots, N_n; M_1, \ldots, M_m)$$

where L_i, M_i and N_i are recursive input, non-recursive input and output arguments respectively. These arguments are separated by ;.

$$\begin{array}{lll}
\textit{Goals} & G & ::= \top \mid P \\
\textit{Clauses} & D & ::= G \supset D \mid \forall x : A.D \mid P \\
\textit{Predicates} & P & ::= P_f(L_1, \ldots, L_l; N_1, \ldots, N_n; M_1, \ldots, M_m) \\
\hline
\textit{Programs} & \mathcal{F} & ::= \cdot \mid \mathcal{F}, D
\end{array}$$

Often we find it convenient to reverse the direction of $G \supset D$ and use $D \subset G$ instead. $\subset$ is left-associative.

We represent input and output arguments in simply-typed λ-calculus shown below. Further, we disallow non-canonical terms (terms with β-redexes). Later, in Section 4, we shall show how to extend our results to programs with non-canonical terms.

$$\begin{array}{lll}
\textit{Types} & A & ::= a \mid A \to B \\
\textit{Canonical Terms} & L, M, N & ::= \lambda x : A.N \mid R \\
\textit{Atomic Terms} & R & ::= c \mid x \mid RN
\end{array}$$

Definition 1 (Predicate symbol). *For a clause D or a goal G, we define predicate symbol of D or G as given below:*

$$\begin{aligned}
\mathsf{symbol}(P_f(\cdot;\cdot;\cdot)) &= P_f \\
\mathsf{symbol}(\forall x : A.D) &= \mathsf{symbol}(D) \\
\mathsf{symbol}(G \supset D) &= \mathsf{symbol}(D)
\end{aligned}$$

Definition 2 (Head of a clause). *For a clause D, we define head of a clause as given below:*

$$\begin{aligned} \mathsf{head}(P) &= P \\ \mathsf{head}(\forall x : A.D) &= \mathsf{head}(D) \\ \mathsf{head}(G \supset D) &= \mathsf{head}(D) \end{aligned}$$

2.2 Function computation through proof search

Goals:

$$\frac{}{\mathcal{F} \to \top}\ \mathsf{g\_True} \qquad \frac{D \in \mathcal{F} \quad \mathcal{F} \to D \gg P}{\mathcal{F} \to P}\ \mathsf{g\_Atom}$$

Clauses:

$$\frac{}{\mathcal{F} \to P \gg P}\ \mathsf{c\_Atom}$$

$$\frac{\mathcal{F} \to [\iota/x]D \gg P}{\mathcal{F} \to \forall x : A.D \gg P}\ \mathsf{c\_Exists} \quad \frac{\mathcal{F} \to D \gg P \quad \mathcal{F} \to G}{\mathcal{F} \to G \supset D \gg P}\ \mathsf{c\_Imp}$$

Fig. 1. Proof search semantics for the Horn fragment

We have given the proof search semantics of the Horn fragment in Figure 1. They are based on the notion of *uniform* proofs for Abstract Logic Programming Langauges (ALPLs) [10].

Given a program $\mathcal{F}$ and a goal G with *ground* terms in its input positions, the interpreter constructs a derivation of the judgment $\mathcal{F} \to G$. In the rule g_Atom an appropriate clause D corresponding to the goal G is selected. It is possible to construct a derivation for the judgment $\mathcal{F} \to D \gg P$ if and only if head of D unifies with P.

Whenever a variable x is replaced by a logic variable, we denote it by ι. For the sake of our analysis, we assume that the variable is replaced by the actual proof term which is guessed appropriately. In an actual interpreter, the variable would be replaced by a logic variable. Since, our logic program has a well-defined *mode* behavior, these logic variables which appear in the output positions of clauses will be bound to *ground* terms eventually in the rule c_Atom.

For example, the logic program corresponding to the Fibonacci function is given by $\mathcal{F} = \{\forall N.+(\mathsf{z}; N; N), \forall N_1 N_2 M.+(N_1; M; N_2) \supset +(\mathsf{s}N_1; M; \mathsf{s}N_2), \mathsf{fib}(\mathsf{z};;\mathsf{s\ z}), \mathsf{fib}(\mathsf{s\ z};;\mathsf{s\ z}),$ $\forall M_1 M_2 N. + (M_1; M_2; M) \supset \mathsf{fib}(N;;M_1) \supset \mathsf{fib}(\mathsf{s}N;;M_2) \supset \mathsf{fib}(\mathsf{s\ s}N;;M)\}$, where the constants z, s, and fib are appropriately defined. In the rest of this paper, we shall omit the universal quantifiers whenever there is no confusion.

For a logic program $\mathcal{F}$, we denote a proof search derivation for a goal G by $\mathcal{D} :: \mathcal{F} \to G$ and measure the size of this derivation as the number of inference rules in the derivation. In the next subection, we shall show that every rule can be implemented on a RAM machine in a constant number of steps.

Definition 3 (Size of proof search derivation). *Given a logic program $\mathcal{F}$ and a proof search derivation $\mathcal{D} :: \mathcal{F} \to G$, we define the size of $\mathcal{D}$, $\mathsf{sz}(\mathcal{D})$, as the number of proof search inference rules in $\mathcal{D}$.*

2.3 Translation to a RAM machine

In this section, we shall show that the proof search shown in Figure 1 can be implemented on a RAM machine in time proportional to the number of proof search rules in a proof search derivation. Thus, a proof search corresponding to a function computation can be implemented on a RAM machine without any increase in its asymptotic complexity.

We require that our logic program satisfy the following conditions:

1. *Well-defined mode behavior:* Since, the logic program actually implements a function, every proof search on a goal with its input arguments as *ground* terms either succeeds with its output arguments set to *ground* terms or fails.
2. *Deterministic and non-backtracking:* The proof search is deterministic and has no backtracking behavior as there is a unique clause corresponding to every function computation where the inputs are ground terms.
3. *Complexity of unification:* We will show that every rule can be implemented on a RAM machine in a time depending only on the size of the logic program. Since we are using the logic programming model of computation, we have to account for the cost of unification of variables with *ground terms*. We will require that unification corresponding to every rule can be implemented in time depending on the size of the logic program.
   For our logic programming language based on the Horn fragment, the following two conditions are *sufficient* to ensure that complexity of unification is independent of the inputs to the logic program.
   (a) *No patterns in output positions of subgoals:* We do not allow patterns in the output positions of any subgoals and always store a single copy of a variable in a clause even when it appears multiple times in a clause. Thus, clauses like $P(x;;y) \subset Q(x;;\mathsf{c}\ y)$ are not allowed and in the clause $P(x;;\mathsf{c}\ u_1\ u_2) \subset Q(x;;z) \subset R_1(z;;u_1) \subset R_2(z;;u_2)$ a single copy of the variable z is shared by R_1 and R_2.**[[carsten, why this condition?]]**
   (b) *Linearity:* We restrict ourselves to functions where a variable does not appear more than once in an input position. Thus, clauses such as $P(x,x;;x) \subset \top$ are disallowed. In such cases, the interpreter will have to verify that the two input arguments are identical by unification. This unification process takes time proportional to the size of the *ground* input terms (see Figure 2 for the formal definition) which may be arbitrarily large. However, we can allow such functions, if we make this unification explicit by translating a clause such as $P(x,x;;x) \subset \top$ to $P(x,y;;x) \subset \mathsf{equal}(x;y;) \subset \top$ where equal checks that x and y are identical.

Theorem 1. *Given a logic program $\mathcal{F}$ and a goal G satisfying the conditions given above. If there exists a derivation $\mathcal{D} :: \mathcal{F} \to G$, then*

– *The goal G can be represented on a RAM machine in size proportional to $\mathsf{sz}_\mathsf{i}(G)$.*
– *The corresponding proof search can be implemented in time proportional to $\mathsf{sz}(\mathcal{D})$.*

3 Conditions for polynomial and superpolynomial complexity classes

In this section, we shall describe criteria for classifying recursive functions into polynomial and superpolynomial complexity classes. These criteria are decidable and can be checked in time depending only on the size of the logic program corresponding to the function. We shall only prove *soundness* of our criteria. In the following section, we shall prove completeness for functions over binary strings; whether these criteria are also complete for arbitrary higher-order data structures is an open problem. Thus, a checker implementing these criteria can only have two responses *yes* and *don't know*.

First, we shall present a general theorem on integer valued recursive functions given by

$$\begin{array}{ll} T(x) = \sum_{i=1}^{m} T(x_i) + f(x) & \text{if } x \geq K \\ T(x) = b & \text{if } 1 \leq x \leq K \end{array} \tag{1}$$

where $x, x_i \in \mathbb{Z}^+$ and there exists functions $g_i(\cdot)$ (not defined using $T(\cdot)$) such that $x_i = g_i(x)$ for all $i = 1, \ldots, m$ such that $x_i < x$, each $f(x)$ is an integer valued function defined on $\mathbb{Z}^+$ (not defined using $T(\cdot)$), b and K are positive integers; and m is an positive integer constant.

Theorem 2 ([16]). *Given a recursive function $T(x)$ defined in equation 1. If $f(x)$ is a monotonically increasing function such that $f(x) > 0$ for all $1 \leq x \leq K$, and $x \geq \sum_{i=1}^{m} x_i$, then there exists a constant $c \geq 1$ such that $T(x) \leq cx^2 f(x)$ for all $x \geq 1$.*

For example, if $f(x) = f(\lfloor x/3 \rfloor) + f(\lfloor x/4 \rfloor) + x$, then $f(x) = O(x^3)$ as $x \geq \lfloor x/3 \rfloor + \lfloor x/4 \rfloor$. On the other hand, we know that $f(x) = f(x-1) + f(x-2) + 1$ when $x \geq 2$ and $f(0) = f(1) = 1$ is not a polynomial. In this case, $x \not\geq (x-1) + (x-2)$.

In fact, the theorem can be generalized to a set of functions $\mathcal{T} = \{T_1(\cdot), T_2(\cdot), \ldots, T_k(\cdot)\}$ where each $T_i(\cdot)$ is defined as

$$\begin{array}{ll} T_i(x) = \sum_{j=1}^{m_i} T_{l_j}(x_{ij}) + f_i(x) & \text{if } x \geq K_i \\ T_i(x) = b_i & \text{if } 1 \leq x \leq K_i \end{array} \tag{2}$$

where m_i, K_i and b_i are positive integer constants, each $l_j \in \{1, \ldots, k\}$, every $f_i(x)$ is an integer-valued function defined on $\mathbb{Z}^+$ (not defined using $T(\cdot)$), $x, x_{ij} \in \mathbb{Z}^+$ and there exists functions $g_{ij}(\cdot)$ (not defined using $T(\cdot)$) such that $x_{ij} = g_{ij}(x)$.

Theorem 3. *Given a set of recursive functions $\mathcal{T} = \{T_1(\cdot), T_2(\cdot), \ldots, T_k(\cdot)\}$ such that each function is given by equation 2. If for all $i = 1, \ldots, k$:*

1. *$f_i(\cdot)$ are monotonically increasing functions such that $f_i(x) > 0$ for all $1 \leq x \leq K_i$.*
2. *$x \geq \sum_{j=1}^{m_i} x_{ij}$*

then there exists a constant $c \geq 1$ and a monotonically increasing function $F(\cdot)$ such that $T_i(x) \leq cx^2F(x)$ for all $x \geq 1$.

For the remainder of the paper, we shall assume that $f(x)$ and $f_i(x)$ are polynomials. In this case, the functions $T(x)$ and $T_i(x)$ are bounded by polynomials as well. In general, $f(x)$ or $f_i(x)$ could be any set of monotonically increasing superpolynomial functions closed under composition and all the results in this paper still hold.

We present our result in two stages. In Section 3.1 we present the basic criterion which capture the essence of our solution. The functions identified by this criterion satisfy the following condition: the sum of the sizes of the recursive input arguments to the recursive calls is less than the original recursive input arguments. Thus, we are merely generalizing the results of Theorem 2 and 3 to higher-order data structures by defining an appropriate size function. In Section **??**, we first show that the idea of *safe-recursion* developed by Bellantoni and Cook [1] is special case of this criteria. Later in Section **??**, we will also extend our criteria to include functions which permit *unsafe* recursion (recursion over recursively computed values) as long as functions which perform *unsafe* recursion are non-size-increasing.

$$\begin{aligned}
&\text{Goals:}\\
&\qquad \mathsf{sz_u}(\top) = 0\\
&\text{Clauses:}\\
&\qquad \mathsf{sz_u}(G \supset D) = \mathsf{sz_u}(D)\\
&\qquad \mathsf{sz_u}(\forall x : A.D) = \mathsf{sz_u}(D)\\
&\text{Predicates:}\\
&\qquad \mathsf{sz_i}(P(L_1, \ldots, L_l; \cdot; \cdot)) = \textstyle\sum_{i=1}^{l} \#(L_i)\\
&\qquad \mathsf{sz_o}(P(\cdot; \cdot; N_1, \ldots, N_n)) = \textstyle\sum_{i=1}^{n} \#(M_i)
\end{aligned}$$

Fig. 2. Size function for goals G and clauses D ($\mathsf{u} = \mathsf{i}$ or $\mathsf{u} = \mathsf{o}$)

3.1 Basic criteria

We generalize Theorems 2 and 3 to functions on arbitrary simply-typed λ-terms. First, we shall begin by defining an appropriate size function for terms which ensures that the terms can be represented on a RAM machine in space proportional to the size of the terms. (See subsection 2.3). The size function # for simply typed λ-terms counts the number of variables and constants in the term. Similarly, the size of a LF goal G or a clause D is defined using $\mathsf{sz_i}(\cdot)$ and $\mathsf{sz_o}(\cdot)$ depending on whether we wish to compute the size of *input* or *output* arguments. $\mathsf{sz_i}(G)$ computes the sum of #-sizes of all the *input* arguments in the goal G and $\mathsf{sz_i}(D)$ computes the sum of #-sizes of all the *input* arguments in predicate P in the clause D. It is shown in Figure 2.

$$\begin{aligned}
\#(x) = \#(c) &= 1\\
\#(RN) &= \#(R) + \#(N)
\end{aligned}$$

$$\#(\lambda x.N) = \#(N)$$

Definition 4 (goals). *Given a clause D, we define the set* $\mathsf{goals}(D)$ *as given below.*

$$\begin{aligned}
\mathsf{goals}(P) &= \{\} \\
\mathsf{goals}(G \supset D) &= \{G\} \cup \mathsf{goals}(D) \\
\mathsf{goals}(\forall x : A.D) &= \mathsf{goals}(D)
\end{aligned}$$

Definition 5 (Mutually recursive functions). *Given a logic program $\mathcal{F}$, a set S of predicate symbols is said to be mutually recursive if and only if for any predicate symbols $P_f, P_g \in S$ there exists clauses $D_1, D_2 \in \mathcal{F}$ such that* $\mathsf{symbol}(D_1) = P_f$, $\mathsf{symbol}(D_2) = P_g$ *and there exist a goals $G_1 \in$* $\mathsf{goals}(D_1)$ *and $G_2 \in$* $\mathsf{goals}(D_2)$ *such that* $\mathsf{symbol}(G_1) = P_g$ *and* $\mathsf{symbol}(G_2) = P_f$.

Programs:

$$\frac{}{\vdash_S \cdot\ \mathsf{poly}_1}\ \mathsf{pp\_empty} \qquad \frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{poly}_1}{\vdash_S \mathcal{F}, D\ \mathsf{poly}_1}\ \mathsf{pp\_clause1}$$

$$\frac{\mathsf{symbol}(D) \in S \quad \vdash_S \cdot/D\ \mathsf{poly}_1 \vdash_S \mathcal{F}\ \mathsf{poly}_1}{\vdash_S \mathcal{F}, D\ \mathsf{poly}_1}\ \mathsf{pp\_clause2}$$

Clauses:

$$\frac{}{\vdash_S \varDelta/P\ \mathsf{poly}_1}\ \mathsf{pc\_Atom}\left\langle \sum_{\substack{G\in\varDelta \\ \mathsf{symbol}(G)\in S}} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(P) \right\rangle$$

$$\frac{\vdash_S \varDelta, G/D\ \mathsf{poly}_1 \quad \mathsf{symbol}(G) \in S}{\vdash_S \varDelta/G \supset D\ \mathsf{poly}_1}\ \mathsf{pc\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \varDelta, G/D\ \mathsf{poly}_1 \quad \mathsf{symbol}(G) \notin S \quad \vdash_T \mathcal{F}\ \mathsf{poly}_1}{\vdash_S \varDelta/G \supset D\ \mathsf{poly}_1}\ \mathsf{pc\_Imp2}\langle \mathsf{sz_i}(G) < f_G(\mathsf{sz_i}(D))\rangle$$

(where $\mathsf{symbol}(D) \in T$ and $f_G(\cdot)$ is a polynomial)

$$\frac{\vdash_S \varDelta/D\ \mathsf{poly}_1}{\vdash_S \varDelta/\forall x : A.D\ \mathsf{poly}_1}\ \mathsf{pc\_Forall}$$

Fig. 3. Basic criteria for identifying for polynomial time functions

Theorem 4 (Basic Criteria). *Given a program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$* poly_1*. Given a goal G such that* $\mathsf{symbol}(G) \in S$*, if $\mathcal{D} :: \mathcal{F} \longrightarrow G$, then there exists a monotonically increasing polynomial $p(\cdot)$ (not depending on the ground input terms of G) such that* $\mathsf{sz}(\mathcal{D}) \leq p(\mathsf{sz_i}(G))$.

Example 1 (Combinators). The combinators $c ::= \mathsf{S} \mid \mathsf{K} \mid \mathsf{MP}\ c_1\ c_2$ that are prevalent in programming language theory are represented as constructors of type comb. We study the complexity of the bracket abstraction algorithm ba, which converts a parametric combinator M (a representation-level function of type $\mathsf{comb} \to \mathsf{comb}$) into a combinator with one less parameter (of type comb) to which we refer as M'. The bracket abstraction algorithm is expressed by a predicate relating M and M'. For any N, combinator, it holds that $\mathsf{MP}\ M'\ N$ corresponds to $M[N/x]$ in combinator logic. Let $\mathcal{F}$ be defined as the following program.

$$\begin{array}{l}
\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, x; \mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ \mathsf{K})\ \mathsf{K}) \\
\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{K}; \mathsf{MP}\ \mathsf{K}\ \mathsf{K}) \\
\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{S}; \mathsf{MP}\ \mathsf{K}\ \mathsf{S}) \\
\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{MP}\ (C_1\ x)\ (C_2\ x); \mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ D_1)\ D_2) \\
\qquad \subset\ \mathsf{ba}\ (\lambda x : \mathsf{comb}.\, C_1\ x; D_1) \\
\qquad \subset\ \mathsf{ba}\ (\lambda x : \mathsf{comb}.\, C_2\ x; D_2)
\end{array}$$

It is easy to see that $\sum_{i=1}^{2} \#(\lambda x : \mathsf{comb}.\, C_i\ x) < \#(\lambda x : \mathsf{comb}.\, \mathsf{MP}\ (C_1\ x)\ (C_2\ x))$, and hence $\vdash_{\mathsf{ba}} \mathcal{F}\ \mathsf{poly}$. □

3.2 Safe recursion

Bellantoni and Cook [1] introduced the concept of *safe recursion* to give a characterization of polynomial time recursive functions. They formulated a class of functions closed under composition with two kinds of inputs - *safe* inputs and *normal* inputs. The functions can perform any (polytime) operation on their *normal* inputs, but can only apply a restricted set of operations to their *safe* inputs. In particular, it is not possible to perform recursion over *safe* inputs. They showed that this class of functions is exactly the class of polynomial-time computable recursive functions.

3.3 Unsafe recursion

Aehlig, *et al.* [12] and Hofmann [13, 14] have developed

The second stage of our criterion allows us to reason about the complexity of functions, where recursively computed values may be passed to auxiliary functions as well. We require that such auxiliary functions are non-size-increasing. We say that a predicate P_f is non-size-increasing if and only if, the sum of the sizes of the output arguments is never greater than the sizes of its input arguments (within an additive constant, i.e. $\mathsf{sz_o}(G) \leq \mathsf{sz_i}(G) + C$, where C is independent of the input variables of G). The concept of multiplicity defined below will be used in building a formal deductive system to identify non-size-increasing predicates.

Definition 6 (Multiplicity). *Given a clause D, a goal $G \in \mathsf{goals}(D)$ the α and β_G multiplicities of D are defined as follows.*

1. *$\alpha(D)$ is defined as the maximum number of times any input variable in $\mathsf{head}(D)$ appears in the output positions of $\mathsf{head}(D)$.*
2. *$\beta_G(D)$ is defined as the maximum number of times any output variable in G appears in the output positions of $\mathsf{head}(D)$.*

For example the values of $\alpha(\forall N_1 N_2 M.+(N_1, M; N_2) \supset +(\mathsf{s}N_1, M; \mathsf{s}N_2))$ and $\beta_{+(N_1,M;N_2)}(\forall N_1 N_2 M.+(N_1, M; N_2) \supset +(\mathsf{s}N_1, M; \mathsf{s}N_2))$ corresponding to the second declaration of addition $+$ operation are 0 and 1 respectively. Similarly, for a clause of the form $P(N; \mathsf{c}NN)$, $\alpha(P(N; \mathsf{c}NN))$ is given by 2.

The judgment corresponding to the non-size-increasing property is written as $\vdash_S \mathcal{F}\ \mathsf{nsi}$ and the corresponding deductive system is given in Figure 4. The first three rules examine the non-size increasing property for every clause defining a function in S, and the following four rules for each type constructor. For a clause D, the contribution to the size of the output $\mathsf{sz_o}(D)$ due to the outputs from the subgoals of D (stored in Δ) is given by $\sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G)$ and due to the the original inputs of D is $\alpha(D)\mathsf{sz_i}(D)$. The rule nsi_Atom ensures that the sum of these two contributions is always less than the total original input $\mathsf{sz_i}(D)$. It will be shown in Theorem 5 that this condition is sufficient to ensure that the predicate corresponding to the clause D is non-size-increasing.

Programs:

$$\frac{}{\vdash_S \cdot\ \mathsf{nsi}}\ \mathsf{nsi\_empty} \qquad \frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{nsi}}{\vdash_S \mathcal{F}, D\ \mathsf{nsi}}\ \mathsf{nsi\_clause1} \qquad \frac{\mathsf{symbol}(D) \in S \quad \vdash_S \cdot/D\ \mathsf{nsi} \quad \vdash_S \mathcal{F}\ \mathsf{nsi}}{\vdash_S \mathcal{F}, D\ \mathsf{nsi}}\ \mathsf{nsi\_clause2}$$

Clauses:

$$\frac{}{\vdash_S \Delta/P\ \mathsf{nsi}}\ \mathsf{nsi\_Atom}\left\langle \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G) \le (1-\alpha(P))\mathsf{sz_i}(P)\right\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{nsi} \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{nsi}}\ \mathsf{nsi\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{nsi} \quad \mathsf{symbol}(G) \notin S \quad \vdash_T \mathcal{F}\ \mathsf{nsi}}{\vdash_S \Delta/G \supset D\ \mathsf{nsi}}\ \mathsf{pc\_Imp2}$$

(where T is a set of mutually recursive predicates such that $\mathsf{symbol}(G) \in T$)

$$\frac{\vdash_S \Delta/D\ \mathsf{nsi}}{\vdash_S \Delta/\forall x : A.D\ \mathsf{nsi}}\ \mathsf{nsi\_Forall}$$

Fig. 4. *Sufficient* conditions for non-size-increasing predicates

Definition 7. *Given a clause D, and goals G and H in the clause, $H \leftsquigarrow_m G$ iff variables of G in output positions appear in input positions of H and no variable of G appears more than m times in H.*

Definition 8 (Dependence Path). *Given a clause D and goals $H = G_0, G_1, \ldots, G_n = G \in$ goals(D), a dependence path from G to H of* length *n denoted by $H \leftsquigarrow G$ is a sequence of goal and positive integer pairs $(G_1, m_1), \ldots, (G_n = G, m_n)$ such that for each pair of goals G_i, G_{i+1} for $i = 0, \ldots, n-1$, $G_i \leftsquigarrow_{m_{i+1}} G_{i+1}$. The* width *of this dependence path is defined as $\Pi_{i=1}^{n} m_i$.*

For example, consider the example of Fibonacci numbers shown below. In this case, there are two dependence paths each of length 1 from $\mathsf{fib}(N;X)$ to $+(X,Y;Z)$ and from

$\mathsf{fib}(\mathsf{s}\ N;\ Y)$ to $+(X,Y;Z)$.

$$\begin{aligned}
\mathsf{fib}(\mathsf{z};\ \mathsf{s}\ \mathsf{z}) &\subset \top \\
\mathsf{fib}(\mathsf{s}\ \mathsf{z};\ \mathsf{s}\ \mathsf{z}) &\subset \top \\
\mathsf{fib}(\mathsf{s}\ (\mathsf{s}\ N);\ Z) &\subset \mathsf{fib}(N;X) \\
&\subset \mathsf{fib}(\mathsf{s}\ N;Y) \\
&\subset +(X,Y;Z)
\end{aligned}$$

It is worth noting that dependence paths are structural property of a logic program $\mathcal{F}$ and hence identifying dependence paths is independent of any of the inputs to the program.

Definition 9 (Set of Dependence Paths). *Given a clause D and two goals $G, H \in \mathsf{goals}(D)$, $H \triangleleft^* G$ is the set of all dependence paths from G to H*

For a clause D and a goal H, we define a judgment $\vdash_S\ H \triangleleft D$ which is provable if and only if there exists a goal $G \in \mathsf{goals}(D)$ such that $\mathsf{symbol}(G) \in S$ and there is a dependence path from G to H. Similarly, we define the judgment $\vdash_S\ H \ntriangleleft D$. Figure 5 gives the deductive systems corresponding to these judgments.

$$\frac{\vdash_S\ H \triangleleft D}{\vdash_S\ H \triangleleft \forall x : A.D}\ \mathsf{dp\_Forall} \qquad \frac{\vdash_S\ H \triangleleft D}{\vdash_S\ H \triangleleft G \supset D}\ \mathsf{dp\_Imp1}\langle H \not\leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \in S}{\vdash_S\ H \triangleleft G \supset D}\ \mathsf{dp\_Imp2}\langle H \leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \notin S \quad \vdash_S\ G \triangleleft D}{\vdash_S\ H \triangleleft G \supset D}\ \mathsf{dp\_Imp3/1}\langle H \leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \notin S \quad \vdash_S\ H \triangleleft D}{\vdash_S\ H \triangleleft G \supset D}\ \mathsf{dp\_Imp3/2}\langle H \leftsquigarrow G\rangle$$

$$\frac{\vdash_S\ H \ntriangleleft D}{\vdash_S\ H \ntriangleleft \forall x : A.D}\ \mathsf{ndp\_Forall} \qquad \frac{\vdash_S\ H \ntriangleleft D}{\vdash_S\ H \ntriangleleft G \supset D}\ \mathsf{ndp\_Imp1}\langle H \not\leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \notin S \quad \vdash_S\ G \ntriangleleft D \quad \vdash_S\ H \ntriangleleft D}{\vdash_S\ H \ntriangleleft G \supset D}\ \mathsf{ndp\_Imp2}\langle H \leftsquigarrow G\rangle$$

Fig. 5. Proving existence and non-existence of dependence paths

Now we can define an extended version of the conditions given in Figure 3. These conditions are given in Figure 6 below and they generalize the conditions given earlier.

In this case, $\vdash_S \mathcal{F}\ \mathsf{poly}_{\{1,2\}}$ means that either $\vdash_S \mathcal{F}\ \mathsf{poly}_1$ or $\vdash_S \mathcal{F}\ \mathsf{poly}_2$ is true. According to these conditions, if output of a recursive call (output variables of $G \in \mathsf{goals}(D)$ such that $\mathsf{symbol}(G) \in S$) appear in input positions of an auxiliary function (input positions of $H \in \mathsf{goals}(D)$ such that $\mathsf{symbol}(H) \notin S$) then we require the auxiliary function to be non-size-increasing. This condition is ensured through the rule pp_Imp2/1. As we will show later, these conditions actually ensure that the size of the output of the logic programs which satisfy these criteria is polynomially bounded in their input. Thus, condition given in the rule pp_Atom is similar to the condition pc_Atom of Figure 3. In the rule pc_Atom we require that the sum of all the inputs to the recursive calls is not larger than the original input. In this case, we require a similar condition, except that we count the inputs to those recursive calls whose outputs have been used either as input to other predicates or in the final output (with appropriate multiplicities). Thus, the sum $\sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G)$ accounts for the first case and $\sum_{\substack{H\in\Delta \\ \mathsf{symbol}(H)\notin S}} \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S \\ p\in H\lhd^* G}} \beta_H(P)\mathsf{sz_i}(G)\mathsf{width}(p)$ for the second.

This ensures that the input arguments to goal H are polynomial in the original input arguments of the clause D. Hence, the condition pc_Imp2 given in Figure 3 (stage 1) is satisfied.

Programs:

$$\frac{}{\vdash_S \cdot\ \mathsf{poly}_2}\ \text{pp\_empty} \qquad \frac{\mathsf{symbol}(D) \in S \quad \vdash_S \cdot/D\ \mathsf{poly}_2 \quad \vdash_S \mathcal{F}\ \mathsf{poly}_2}{\vdash_S \mathcal{F}, c : D\ \mathsf{poly}_1}\ \text{pp\_clause1} \qquad \frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{poly}_2}{\vdash_S \mathcal{F}, c : D\ \mathsf{poly}_2}\ \text{pp\_clause2}$$

Clauses:

$$\frac{}{\vdash_S \Delta/P\ \mathsf{poly}_2}\ \text{pp\_Atom}\left\langle \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{H\in\Delta \\ \mathsf{symbol}(H)\notin S}} \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S \\ p\in H\lhd^* G}} \beta_H(P)\mathsf{sz_i}(G)\mathsf{width}(p) \le \mathsf{sz_i}(P) \right\rangle$$

$$\frac{\vdash_S \Delta/D\ \mathsf{poly}_2}{\vdash_S \Delta/\forall x : A.D\ \mathsf{poly}_2}\ \text{pp\_Forall} \qquad \frac{\vdash_S \Delta, G/D\ \mathsf{poly}_2 \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{poly}_2}\ \text{pp\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly}_2 \quad \mathsf{symbol}(G) \notin S \quad \vdash_S G \lhd D \quad \vdash_T \mathcal{F}\ \mathsf{nsi} \quad \vdash_T \mathcal{F}\ \mathsf{poly}_{\{1,2\}}}{\vdash_S \Delta/G \supset D\ \mathsf{poly}_2}\ \text{pp\_Imp2/1}$$

(where T is a set of mutually recursive predicates such that $\mathsf{symbol}(G) \in T$)

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly}_2 \quad \mathsf{symbol}(G) \notin S \quad \vdash_S G \not\lhd D \quad \vdash_T \mathcal{F}\ \mathsf{poly}_{\{1,2\}}}{\vdash_S \Delta/G \supset D\ \mathsf{poly}_2}\ \text{pp\_Imp2/2}$$

(where T is a set of mutually recursive predicates such that $\mathsf{symbol}(G) \in T$)

Fig. 6. *Sufficient* conditions for polynomial time predicate (Stage 2)

Theorem 5 (Non-size-increasing functions). *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ nsi. If $\mathcal{D} :: \mathcal{F} \longrightarrow G$, then $\mathsf{sz_o}(G) \le \mathsf{sz_i}(G) + C$ where C is a constant depending on the logic program $\mathcal{F}$.*

Theorem 6 (Stage 2). *Given a program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ poly_2. Given a goal G such that $\mathsf{symbol}(G) \in S$, if $\mathcal{D} :: \mathcal{F} \rightarrow G$, then there exists monotonically increasing polynomials $p(\cdot)$ and $p'(\cdot)$ (not depending on the ground input terms of G) such that $\mathsf{sz_o}(G) \leq p(\mathsf{sz_i}(G))$ and $\mathsf{sz}(\mathcal{D}) \leq p'(\mathsf{sz_i}(G))$.*

Example 2 (Merge Sort). Consider a representation of a list using the constants nil and cons. The logic program $\mathcal{F}$ corresponding to merge sort is given below.

$$
\begin{array}{l}
\mathsf{mergesort}(\mathsf{nil}; \mathsf{nil})\\
\mathsf{mergesort}(\mathsf{cons}\ x\ xs; w)\\
\quad \subset \mathsf{split}(\mathsf{cons}\ x\ xs; y, z)\\
\quad \subset \mathsf{mergesort}(y; y_1)\\
\quad \subset \mathsf{mergesort}(z; z_1)\\
\quad \subset \mathsf{merge}(y_1, z_1; w)\\
\\
\mathsf{split}(\mathsf{nil}; \mathsf{nil}, \mathsf{nil})\\
\mathsf{split}(\mathsf{cons}\ x\ \mathsf{nil}; \mathsf{cons}\ x\ \mathsf{nil}, \mathsf{nil})\\
\mathsf{split}(\mathsf{cons}\ x\ (\mathsf{cons}\ y\ xs); \mathsf{cons}\ x\ x_1, \mathsf{cons}\ y\ y_1)\\
\quad \subset \mathsf{split}(xs; x_1, y_1)\\
\\
\mathsf{merge}(\mathsf{nil}, w; w)\\
\mathsf{merge}(w, \mathsf{nil}; w)\\
\mathsf{merge}(\mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; \mathsf{cons}\ u\ z)\\
\quad \subset \mathsf{compare}(x, y; t)\\
\quad \subset \mathsf{merge}'(t, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; u, v, w)\\
\quad \subset \mathsf{merge}(v, w; z)\\
\\
\mathsf{merge}'(\mathsf{true}, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; x, xs, \mathsf{cons}\ y\ ys)\\
\mathsf{merge}'(\mathsf{false}, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; y, \mathsf{cons}\ x\ xs, ys)
\end{array}
$$

In the example given above $\mathsf{compare}(x, y; t)$, t is true if $x < y$ and t is false otherwise (clauses omitted for brevity). It is not hard to see that $\vdash_{\mathsf{compare}} \mathcal{F}$ poly_1

It is also clear that $\vdash_{\mathsf{split}} \mathcal{F}$ poly_1 as $\#(xs) \leq \#(\mathsf{cons}\ (\mathsf{cons}\ y\ xs))$ for the third declaration of split. The predicate merge′ is also in polynomial time as it is not recursive. We can also check that $\vdash_{merge'} \mathcal{F}$ nsi. In this case, the side condition of nsi_Atom is satisfied because $\alpha(\cdot) = 1$ and $\beta_G(\cdot) = 0$ for both declarations of merge′. In fact, we can show that $\mathsf{sz_o}(\mathsf{merge}'(G)) = \mathsf{sz_i}(\mathsf{merge}'(G)) - 2$ when given some input through a goal G[1].

We can also show that $\vdash_{\mathsf{merge}} \mathcal{F}$ poly_1. For this we need to show that $\#(v) + \#(w) \leq \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys)$. It is true because merge′ is non-size-increasing and we know that $1 + \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys) - 2 = \#(u) + \#(v) + \#(w)$. We can also show that merge is non-size increasing. Here $\alpha(\mathsf{merge}'(\cdot)) = \alpha(\mathsf{merge}(\cdot)) = 1$ and we need to show that $\#(\mathsf{cons}) + \#(u) + \#(v) + \#(w) \leq \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys)$. This follows from the fact that merge′ is non-size-increasing.

Finally, it needs to be shown that $\vdash_{\mathsf{mergesort}} \mathcal{F}$ poly_2 as the outputs y_1 and z_1 of mergesort are given as inputs to the predicate merge. In this case, $\beta_{\mathsf{mergesort}}(\cdot) = 0$ for both the mergesort subgoals and $\beta_{\mathsf{merge}}(\cdot) = 1$ for the second declaration of mergesort.

[1] This is not shown in the formal system given in Figure 4 for the sake of clarity, but is easy to incorporate in it.

There are also two dependence paths of $\mathsf{length} = 1$ from mergesort to merge. Thus, this conditions in Figure 6 require that merge is non-size-increasing and $\#(y) + \#(z) \leq \#(\mathsf{cons}\ x\ xs)$. This follows from split being non-size-increasing.

4 Extending to Hereditary Harrop Formulas

The results presented so far are quite general and even apply to logic programming languages with dependent types, higher-order terms, and embedded implication. Let us consider Hereditary Harrop formulas [17, 18] which allow embedded implications by extending Horn goals G as shown below.

$$\begin{array}{ll} \mathit{Goals} & G ::= \top \mid P \mid \forall x : A.G \mid D \supset G \\ \mathit{Clauses} & D ::= G \supset D \mid \forall x : A.D \mid P \end{array}$$

The proof search semantics are extended as shown in Figure 7. The embedded implication is operationally interpreted as extending the logic program dynamically during proof-search.

Goals:

$$\frac{}{\mathcal{F} \to \top}\ \mathsf{g\_True} \qquad \frac{D \in \mathcal{F} \quad \mathcal{F} \to D \gg P}{\mathcal{F} \to P}\ \mathsf{g\_Atom}$$

$$\frac{\mathcal{F}, D \to G}{\mathcal{F} \to D \supset G}\ \mathsf{g\_Imp} \qquad \frac{c \text{ new} \quad \mathcal{F} \to [c/x]G}{\mathcal{F} \to \forall x : A.G}\ \mathsf{g\_Forall}$$

Clauses:

$$\frac{}{\mathcal{F} \to P \gg P}\ \mathsf{c\_Atom}$$

$$\frac{\mathcal{F} \to [\iota/x]D \gg P}{\mathcal{F} \to \forall x : A.D \gg P}\ \mathsf{c\_Exists} \qquad \frac{\mathcal{F} \to D \gg P \quad \mathcal{F} \to G}{\mathcal{F} \to G \supset D \gg P}\ \mathsf{c\_Imp}$$

Fig. 7. Proof search semantics for the Hereditary Harrop formulas

Thus, a logic program with Hereditary Harrop formulas is polynomial time if we can ensure that all embedded implications satisfy the polynomial time conditions that we have presented so far.

Example 3 (β-redexes). Since the arguments to predicates P have to be in canonical form, it is not possible to represent functions such as eval which simplify a term in lambda-calculus to its β-normal form.

$$\begin{array}{rl} \mathsf{eval}\ (\mathsf{lam}\ E)\ (\mathsf{lam}\ E) & \subset \top \\ \mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V & \subset \mathsf{eval}\ E_1\ (\mathsf{lam}\ E_1') \\ & \subset \mathsf{eval}\ E_2\ V_2 \\ & \subset \mathsf{eval}\ (E_1'\ V_2)\ V \end{array}$$

However, such predicates can be represented by defining a predicate $\mathsf{subst}^{A,B} : (A \to B) \to A \to B$ which performs the substitution explicitly and computes the canonical form. For example, if $A = B = \mathsf{exp}$ then $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ is given by

$$
\begin{array}{l}
\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.x, V; V) \subset \top. \\
\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.\mathsf{app}(E_1 x)(E_2 x), V; (\mathsf{app}(E_1')(E_2'))) \\
\qquad \subset \mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.(E_1 x); E_1') \\
\qquad \subset \mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.(E_2 x); E_2'). \\
\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.\mathsf{lam}\ (\lambda y.(E\ x\ y))), V; \mathsf{lam}\ (\lambda y.(E' y))) \\
\qquad \subset (\forall y : \mathsf{exp}.\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.y, V; y) \\
\qquad\qquad \supset \mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.(E\ x\ y), V; (E'\ y)))
\end{array}
$$

In this case, we observe that for logic program $\mathcal{F}$ corresponding to $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$, $\vdash_{\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}} \mathcal{F}\ \mathsf{poly}_1$ because the first declaration is non-recursive, $\sum_{i=1}^{2} \#(\lambda x.(E_i x)) < \#(\lambda x.\mathsf{app}\ (E_1 x)\ (E_2 x))$ in the second declaration, and the embedded implication in the third declaration in non-recursive.

On the other hand, when $A = \mathsf{exp} \to \mathsf{exp}$ and $B = \mathsf{exp}$ then $\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}$ is given by

$$
\begin{array}{l}
\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.f, V; V) \\
\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.(\mathsf{app}\ (E_1\ f)\ (E_2\ f)), V; \mathsf{app}\ E_1'\ E_2') \\
\qquad \subset \mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.(E_1 f), V; E_1') \\
\qquad \subset \mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.(E_2 f), V; E_2') \\
\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.\mathsf{lam}\ \lambda y.(E\ f\ y), V; \mathsf{lam}\ \lambda y.(E'\ y)) \\
\qquad \subset (\forall y : \mathsf{exp}.\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.y, V; y) \\
\qquad\qquad \supset \mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.(E\ f\ y), V; (E'\ y)) \\
\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.f\ (E f), V; E'') \\
\qquad \mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}(\lambda f.E\ f, V; E') \\
\qquad \mathsf{subst}^{\mathsf{exp},\mathsf{exp}}(\lambda x.Vx, E'; E'')
\end{array}
$$

In this case, the first three declarations satisfy the polynomial time conditions we have described so far. In the fourth declaration, output term E' from the recursive call $\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}$ is provided as input to $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$. It is easy to see that Stage 1 conditions do not hold for this case because, it is not possible to determine the run time of $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ as we do not know the size of its input E'. Stage 2 conditions do not hold either because, $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ is a size-increasing function.

Now the $\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V$ is changed to

$$
\begin{array}{rl}
\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V & \subset \mathsf{eval}\ E_1\ (\mathsf{lam}\ E_1') \\
 & \subset \mathsf{eval}\ E_2\ V_2 \\
 & \subset \mathsf{subst}^{A,\mathsf{exp}}(E_1', V_2; E_1'') \\
 & \subset \mathsf{eval}\ (E_1''\ V)
\end{array}
$$

where an appropriate $\mathsf{subst}^{A,\mathsf{exp}}$ is chosen.

Therefore, when $A = \mathsf{exp}$ we know that β-reduction is a polynomial time operation, but when A is a higher-order type, our conditions can no longer guarantee that β-reduction is in polynomial time.

Example 4 (Combinators cont'd). Recall the bracket abstraction algorithm from Example 1 that is used in the conversion from λ-expressions into combinators. We follow standard practice and define a new type exp together with the two constructors app of type $\mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ and lam of type $(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$. Using our syntax, extend the program $\mathcal{F}$ from Example 1 to a program $\mathcal{F}'$ by the following new declarations.

$$
\begin{aligned}
&\mathsf{convert}(\mathsf{app}\ E_1\ E_2; \mathsf{MP}\ C_1\ C_2)\\
&\qquad \subset \mathsf{convert}(E_1; C_1)\\
&\qquad \subset \mathsf{convert}(E_2; C_2)\\[4pt]
&\mathsf{convert}(\mathsf{lam}\ E); D)\\
&\qquad \subset\ (\forall x : \mathsf{exp}.\, \forall y : \mathsf{comb}.\, \mathsf{ba}(\lambda z : \mathsf{comb}.\, y; \mathsf{MP}\ \mathsf{K}\ y)\\
&\qquad\quad \supset \mathsf{convert}(y; z) \supset \mathsf{convert}\ (E\ \ x; C\ y))\\
&\qquad \subset \mathsf{ba}\ (\lambda y : \mathsf{comb}.\, C\ y; D)
\end{aligned}
$$

We observe that $\vdash_{\mathsf{convert}} \mathcal{F}'\ \ \mathsf{poly}_2$ because the first declaration satisfies that $\sum_{i=1}^{2} \#(E_i) < \#(\mathsf{app}\ E_1\ E_2)$, and each embedded implication in the second is non-recursive. Furthermore $\#(E\ \ x) < \#(\mathsf{lam}\ \ E)$ because E is applied to a paramter x (and not an arbitrary term). In addition, $\vdash_{\mathsf{ba}} \mathcal{F}'$ nsi by rule nsi_Atom where we choose $\alpha(\cdot) = 0$ and $\beta_{\mathsf{ba}}(\cdot) = 1$ for the two recursive calls, and hence the dynamic extension of the bracket abstraction algorithm ba is non-size increasing. □

5 Conclusions

The polynomial time criteria that we have developed in this paper while not complete are able to identify a sufficiently large class of functions. Further, we are not limited to functions on integers or lists but can apply these criteria to functions which range over first-order, higher-order or even dependently-typed domains.

These criteria are not specific to a particular programming language but can be extended to any logic or functional programming language as long as unification and substitution can be implemented in constant time. Currently, we are working on implementing these criteria for Twelf [19].

In our presentation, we distinguish predicates based on whether they do or do not receive input from a output of a recursive call. By making this distinction at the level of input arguments instead of predicates, we can further refine our criteria to identify a larger class of polynomial time functions.

References

1. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. In: Twenty-fourth Annual ACM Symposium on Theory of Computing. (1992)
2. Leivant, D.: Subrecursion and lambda representation over free algebras. In Buss, S., Scott, P., eds.: Feasible Mathematics. Birkhauser (1990) 281–292
3. Cobham, A.: The intrinsic computational complexity of functions. In Bar-Hellel, Y., ed.: Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science. (1965) 24–30

4. Leivant, D.: A foundational delineation of computational feasibility. In: Sixth Annual IEEE Symposium on Logic in Computer Science, IEEE (1991) 2–11
5. Bellantoni, S., Niggl, K.H., Schwichtenberg, H.: Higher type recursion, ramification and polynomial time. Annals of Pure and Applied Logic **104** (2000)
6. Hofmann, M.: Typed lambda calculi for polynomial-time computation. Habilitation thesis, TU Darmstadt (1998)
7. Ganzinger, H., McAllester, D.: A new meta-complexity theorem for bottom-up logic programs. In: Proc. International Joint Conference on Automated Reasoning. Volume 2083 of Lecture Notes in Computer Science., Springer-Verlag (2001) 5114–5128
8. Givan, R., McAllester, D.: Polynomial-time computation via local inference relations. ACM Transactions on Computational Logic **3** (2002) 521–541
9. Cervesato, I.: Proof-theoretic foundation of compilation in logic programming languages. In Jaffar, J., ed.: Joint International Conference and Symposium on Logic Programming, MIT Press (1998)
10. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic **51** (1991) 125–157
11. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1** (1991) 497–536
12. Aehlig, K., Schwichtenberg, H.: A syntactical analysis of non-size-increasing polynomial time computation. In: Fifteenth Annual IEEE Symposium on Logic in Computer Science. (2000)
13. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. In: Fourteenth Annual IEEE Symposium on Logic in Computer Science, IEEE (1999) 464–473
14. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Information and Computation **183** (2003) 57–85
15. Caseiro, V.H.: Equations for defining poly-time functions. PhD thesis, University of Oslo (1997)
16. Verma, R.M.: General techniques for analyzing recursive algorithms with applications. SIAM Journal of Computing **26** (1997) 568–581
17. Harrop, R.: Concerning formulas of the types $A \to B \vee C, A \to (Ex)(Bx)$. Journal of Symbolic Logic **25** (1960) 27–32
18. Miller, D.: Hereditary harrop formulas and logic programming. In: Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science, Moscow (1987)
19. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In Ganzinger, H., ed.: Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, Springer-Verlag LNAI 1632 (1999) 202–206
20. Walther, C.: Mathematical induction. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 2. Oxford University Press, Oxford (1994) 127–227

Appendix

A Proofs of Recursive Functions

Theorem 7 ([16]). *Given a recursive function $T(x)$ defined in equation 1. If $f(x)$ is a monotonically increasing function such that $f(x) \geq d > 0$ for all $1 \leq x \leq K$, and $x \geq \sum_{i=1}^{m} x_i$, then there exists a constant $c \geq 1$ such that $T(x) \leq cx^2 f(x)$ for all $x \geq 1$.*

Proof:[2] Choose $c = max\{1, b/d\}$. We shall prove by induction[3].
Base case: When $1 \le x \le K$, $T(x) = b = (b/d)d \le cf(x) \le cx^2 f(x)$.
Induction case: When $x > K$,

$$\begin{aligned}
T(x) &= \sum_{i=1}^{m} T(x_i) + f(x) \\
&\le \sum_{i=1}^{m} cx_i^2 f(x_i) + f(x) \\
&\qquad \text{(Using Induction Hypothesis on } x_i \sqsubset x\text{:} \\
&\qquad\qquad T(x_i) \le cx_i^2 f(x_i)) \\
&\le \sum_{i=1}^{m} cx_i^2 f(x) + cf(x) \\
&\qquad (\because x_i \le x \Rightarrow f(x_i) \le f(x) \text{ and } c \ge 1) \\
&= cf(x)(\sum_{i=1}^{m} x_i^2 + 1) \\
&\le cf(x)(\sum_{i=1}^{m} x_i)^2 \\
&\qquad (\because \textstyle\sum_{i=1}^{m} x_i^2 + 1 \le (\sum_{i=1}^{m} x_i)^2) \\
&\le cx^2 f(x) \\
&\qquad (\because x \ge \textstyle\sum_{i=1}^{m} x_i)
\end{aligned}$$

∎

Theorem 8. *Given a set of recursive functions $\mathcal{T} = \{T_1(\cdot), T_2(\cdot), \ldots, T_k(\cdot)\}$ such that each function is given by equation 2. If for all $i = 1, \ldots, k$:*

1. *$f_i(\cdot)$ are monotonically increasing functions such that $f_i(x) \ge d_i > 0$ for all $1 \le x \le K_i$.*
2. *$x \ge \sum_{j=1}^{m_i} x_{ij}$*

then there exists a constant $c \ge 1$ and a monotonically increasing function $F(\cdot)$ such that $T_i(x) \le cx^2 F(x)$ for all $x \ge 1$.

Proof: Choose $c = max\{1, b_1/d_1, \ldots, b_k/d_k\}$ and $F(x) = max\{f_1(x), \ldots, f_k(x)\}$. We shall prove this theorem using the induction hypothesis: $\forall i = 1, \ldots, k. y \sqsubset x \Rightarrow T_i(y) \le cy^2 F(y)$

Base Case: For any $i = 1, \ldots, k$: When $1 \le x \le K_i$, $T_i(x) = b_i = (b_i/d_i)d_i \le cf_i(x) \le cx^2 F(x)$.

[2] This version of the proof is due to Adam Poswolsky and Valery Trifonov.
[3] We shall assume the partial ordering $\sqsubset$ as : $\ldots \sqsubset g_i(g_i(x)) \sqsubset g_i(x) \sqsubset x$ and use the principle of Noetherian Induction [20]

Induction Case: For any $i = 1, \ldots, k$: When $x \geq K$,

$$\begin{aligned}
T_i(x) &= \sum_{j=1}^{m_i} T_{l_i}(x_{ij}) + f_i(x) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x_{ij}) + f_i(x) \\
&\qquad \text{(Using Induction Hypothesis on } x_{ij} \sqsubset x\text{:} \\
&\qquad\quad T_{l_i}(x_{ij}) \leq c x_{ij}^2 F(x_{ij})) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x_{ij}) + F(x) \\
&\qquad (\because \forall i = 1, \ldots, k. F(x) \geq f_i(x)) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x) + cF(x) \\
&\qquad (\because x_i \leq x \Rightarrow F(x_i) \leq F(x) \text{ and } c \geq 1) \\
&\leq cF(x)(\sum_{j=1}^{m_i} x_{ij}^2 + 1) \\
&\leq cF(x)(\sum_{j=1}^{m_i} x_{ij})^2 \\
&\qquad (\because \textstyle\sum_{j=1}^{m} x_{ij}^2 + 1 \leq (\sum_{j=1}^{m} x_{ij})^2) \\
&\leq cx^2 F(x) \\
&\qquad (\because x \geq \textstyle\sum_{j=1}^{m} x_{ij})
\end{aligned}$$

∎

B Proofs of Stage 1

Lemma 1. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$*, if $\mathcal{D} :: \mathcal{F} \rightarrow D \gg P$, then* $\mathsf{sz}(\mathcal{D}) = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})} \mathsf{sz}(\mathcal{D}_G) + C_D$ *where C_D is a constant depending only on the structure of D and not its input terms.*

Proof: We shall prove by induction on the size of derivation $\mathcal{D}$.

(Base) Case: When the derivation $\mathcal{D}$ is given by

$$\frac{}{\mathcal{F} \rightarrow P \gg P}\ \mathsf{c\_Atom},$$

$\mathsf{sz}(\mathcal{D}) = 1$ and hence the theorem is true.

Case: When the derivation $\mathcal{D}$ is given by

$$\dfrac{\overset{\mathcal{D}_1}{\mathcal{F} \to D \gg P} \quad \overset{\mathcal{D}_2}{\mathcal{F} \to H}}{\mathcal{F} \to H \supset D \gg P}\ \mathsf{c\_Imp}$$

By induction hypothesis,

$$\mathsf{sz}(\mathcal{D}_1) = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}_1)} \mathsf{sz}(\mathcal{D}_G) + C_D.$$

Hence,

$$\begin{aligned}
\mathsf{sz}(\mathcal{D}) &= \mathsf{sz}(\mathcal{D}_1) + \mathsf{sz}(\mathcal{D}_2) + 1 \\
\mathsf{sz}(\mathcal{D}) &= \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}_1)} \mathsf{sz}(\mathcal{D}_G) + C_D + \mathsf{sz}(\mathcal{D}_2) + 1 \\
\mathsf{sz}(\mathcal{D}) &= \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})} \mathsf{sz}(\mathcal{D}_G) + C_{H \supset D} \\
&\text{(where } C_{H \supset D} = C_D + 1)
\end{aligned}$$

Case: When the derivation $\mathcal{D}$ is given by

$$\dfrac{\overset{\mathcal{D}'}{\mathcal{F} \to [\iota/x]D \gg P}}{\mathcal{F} \to \forall x : A.D \gg P}\ \mathsf{c\_Exists}$$

The proof of this case is also similar to the above cases, if we define $C_{\forall x:A.D} = C_{[\iota/x]D} + 1$.

∎

Lemma 2 (Stage 1). *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ *and* $\mathcal{E} ::\vdash_S \Delta/D\ \mathsf{poly}_1$, *if* $\mathcal{D} :: \mathcal{F} \to D \gg P$, *then*

1. *For all* $\mathcal{D}_G :: \mathcal{F} \to G \in \mathsf{GOALS}(\mathcal{D})$, *if* $\mathsf{symbol}(G) \in S$ *then* $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$ *and if* $\mathsf{symbol}(G) \in T \neq S$, *then* $\vdash_T \mathcal{F}\ \mathsf{poly}_1$ *and* $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(P))$.
2. $\sum_{\substack{\mathcal{D}_G :: \mathcal{F} \to G \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(G) \in S}} \mathsf{sz_i}(G) + \sum_{G \in \Delta} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(P)$.

Proof:
Since $\mathcal{E} ::\vdash_S \Delta/D\ \mathsf{poly}_1$, it is easy to show by induction that

- For all $G \in \mathsf{goals}(D)$, if $\mathsf{symbol}(G) \in S$ then $\mathsf{sz_i}(G) < \mathsf{sz_i}(D)$ and if $\mathsf{symbol}(G) \in T \neq S$, then $\vdash_T \mathcal{F}\ \mathsf{poly}_1$ and $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(D))$.
- $\sum_{\substack{\mathcal{D}_G :: \mathcal{F} \to G \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(G) \in S}} \mathsf{sz_i}(G) + \sum_{G \in \Delta} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(D)$

These properties are mathematical side conditions that are proved using additional properties of the predicates, if necessary. Thus, if a goal $G \in \mathsf{goals}(D)$ or the clause D contain any free variables then, the mathematical side conditions are true for any substitution of variables by ground terms for the variables in G and D. We only need to ensure that this substitution is generated by a successful proof search. ∎

Lemma 3. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}\ \mathsf{poly}_1$. Given a predicate P and a goal G , if $\mathcal{D} :: \mathcal{F} \longrightarrow G$, then there exists a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(D) = \mathsf{symbol}(P) \in S$ and a sub-derivation $\mathcal{D}' :: \mathcal{F} \longrightarrow D \gg P$ such that $\mathsf{sz}(\mathcal{D}) = \mathsf{sz}(\mathcal{D}') + C_G$ where C_G is a constant depending only on the structure of G and not on its input terms. Also, $\mathsf{sz_i}(P) = \mathsf{sz_i}(G)$ and $\mathsf{sz_o}(P) = \mathsf{sz_o}(G)$.*

Proof: (Sketch) Given a goal G, identifying the right clause D corresponding to that goalis independent of the input arguments to the goals. The proof follows from the proof search semantics of Figure 1. ∎

Theorem 9 (Stage 1). *Given a program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}\ \mathsf{poly}_1$. Given a goal G such that $\mathsf{symbol}(G) \in S$, if $\mathcal{D} :: \mathcal{F} \longrightarrow G$, then there exists a monotonically increasing polynomial $p(\cdot)$ (not depending on the ground input terms of G) such that $\mathsf{sz}(\mathcal{D}) \leq p(\mathsf{sz_i}(G))$.*

Proof: Using Lemma 9, we know that there exists a derivation $\mathcal{D}' :: \mathcal{F} \longrightarrow D \gg P$ such that

$$\mathsf{sz}(\mathcal{D}) = \mathsf{sz}(\mathcal{D}') + C_G.$$

Using Lemma 7, we know that

$$\mathsf{sz}(\mathcal{D}') = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}')} \mathsf{sz}(\mathcal{D}_G) + C_D.$$

Hence,

$$\begin{aligned}
\mathsf{sz}(\mathcal{D}) &= \sum_{\mathcal{D}_H :: \mathcal{F} \rightarrow H \in \mathsf{GOALS}(\mathcal{D}')} \mathsf{sz}(\mathcal{D}_H) + C_D + C_G \\
&= \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \rightarrow H \in \mathsf{GOALS}(\mathcal{D}') \\ \mathsf{symbol}(H) \in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + C_G \\
&\quad + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \rightarrow H \in \mathsf{GOALS}(\mathcal{D}') \\ \mathsf{symbol}(H) \notin S}} \mathsf{sz}(\mathcal{D}_H)
\end{aligned}$$

By Lemma 8, for goals H such that $\mathsf{symbol}(H) \in T_H \neq S$, $\vdash_{T_H} \mathcal{F}\ \mathsf{poly}_1$. Hence, by induction,

$$\mathsf{sz}(\mathcal{D}) \leq \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \rightarrow H \in \mathsf{GOALS}(\mathcal{D}') \\ \mathsf{symbol}(H) \in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + C_G$$

$$
\begin{aligned}
& + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(\mathsf{sz_i}(H)) \\
\leq & \sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + C_G \\
& + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(f_H(\mathsf{sz_i}(G))) \\
& \text{(Using Lemma 8, } \mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P)) \\
& \leq f_H(\mathsf{sz_i}(G)) \text{ when } \mathsf{symbol}(H) \notin S\text{))}
\end{aligned}
$$

Let us define

$$F(\mathsf{sz_i}(H)) = \sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(f_H(\mathsf{sz_i}(H))) + C_G + C_D^{max}$$

where $C_D^{max} = max\{C_D | D \in \mathcal{F}\}$.

We shall prove by induction on $\mathsf{sz_i}(G)$ that the polynomial $p(x) = x^2F(x)$. The theorem follows by applying Theorems 8. ∎

C Proofs of Stage 2

Lemma 4. *Given a program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$*. If $\mathcal{D} :: \mathcal{F} \to D \gg P$, then*

$$\mathsf{sz_o}(P) \leq \alpha(D)\mathsf{sz_i}(P) + \sum_{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})} \beta_G(D)\mathsf{sz_o}(H) + C$$

where C is a constant depending only on the structure of D and not its ground input terms.

Proof: (Sketch) The size of output of a clause D given by $\mathsf{sz_o}(D)$ consists of three kinds of terms: a fixed number of term constants, the input variables of D and the output variables of the subgoals G of D ($G \in \mathsf{goals}(D)$). By taking into account the α and β_G multiplicities of the variables and the fact that the output terms of P are unified with the output variables of D the theorem follows. ∎

Lemma 5 (Stage 2). *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ *and* $\vdash_S \Delta/D$ poly_2.

If $\mathcal{D} :: \mathcal{F} \to D \gg P$, then

– *For all $\mathcal{D}_G :: \mathcal{F} \to G \in \mathsf{GOALS}(\mathcal{D})$, if* $\mathsf{symbol}(G) \in S$*, then* $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$.

– *For all* $\mathcal{D}_G :: \mathcal{F} \rightarrow G \in \mathsf{GOALS}(\mathcal{D})$, *if* $\mathsf{symbol}(G) \in T \neq S$ *and* $\mathcal{E} ::\vdash G \lhd D$, *then there exists a polynomial* $f_G(\cdot)$ *such that*

$$\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(P)) + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \rightarrow H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S \\ p \in G \lhd^* H}} \mathsf{sz_o}(H)\mathsf{width}(p)$$

and $\vdash_T \mathcal{F}$ *nsi.*

– *For all* $\mathcal{D}_G :: \mathcal{F} \rightarrow G \in \mathsf{GOALS}(\mathcal{D})$, *if* $\mathsf{symbol}(G) \in T \neq S$ *and* $\mathcal{E} ::\vdash G \ntriangleleft D$, *then there exists a polynomial* $f_G(\cdot)$ *such that* $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(D))$ *and* $\vdash_S \mathcal{F}$ poly_2.

– $\left(\sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S}} \sum_{\substack{G \in \Delta' \\ \mathsf{symbol}(G) \in S \\ p \in H \lhd^* G}} \beta_H(D)\mathsf{sz_i}(G)\mathsf{width}(p) \right) + \left(\sum_{\substack{G \in \Delta' \\ \mathsf{symbol}(G) \in S}} \beta_G(D)\mathsf{sz_i}(G) \right) \leq \mathsf{sz_i}(P)$ *where*

$$\Delta' = \Delta \cup \{G | \mathcal{D}_G :: \mathcal{F} \rightarrow G \in \mathsf{GOALS}(\mathcal{D})\}.$$

Proof: (Sketch)

Let $\Delta'' = \{G | \mathcal{D}_G :: \mathcal{F} \rightarrow G \in \mathsf{GOALS}(\mathcal{D})\}$.

For $G \in \Delta''$, if $\mathsf{symbol}(G) \in T$ and $\mathcal{E} ::\vdash_S G \ntriangleleft D$, then all terms that appear in input positions in G are either the terms from input positions of D or from output positions of goals H such that $\mathcal{E}' ::\vdash_S H \ntriangleleft D$. We can show by induction that for such goals $\mathsf{sz_o}(H) \leq p(\mathsf{sz_i}(D))$ for some polynomial $p(\cdot)$. Hence there exists a polynomial $f_G(\cdot)$ such that $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(D))$.

For $G \in \Delta''$, if $\mathsf{symbol}(G) \in T$ and $\mathcal{E} ::\vdash_S G \lhd D$, then all terms that appear in input positions in G are either from input positions of D, output positions of goals H such that $\mathcal{E}' ::\vdash_S H \ntriangleleft D$ or form output positions of goals H such that $\mathcal{E}' :\vdash_S H \lhd D$.

For the first two cases, we have already shown that there exists a polynomial $f'_G(\cdot)$ that bounds the total contribution to $\mathsf{sz_i}(G)$ due to the terms that satisfy the conditions of these two cases. Thus,

$$\mathsf{sz_i}(G) \leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \vdash_S H \lhd D \\ G \leftsquigarrow_m H}} m\mathsf{sz_o}(H).$$

We shall bound the contribution due to the third case using induction on the length of dependence paths ending in a goal I such that $\mathsf{symbol}(I) \in S$. For the base case (length of dependence paths is 1), we have,

$$\begin{aligned} \mathsf{sz_i}(G) &\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \vdash_S H \lhd D \\ G \leftsquigarrow_m H}} m\mathsf{sz_o}(H) \\ &\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \in S \\ \vdash_S H \lhd D \wedge G \leftsquigarrow_m H}} m\mathsf{sz_o}(H) + \\ &\sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D \wedge G \leftsquigarrow_m H}} m\mathsf{sz_o}(H) \end{aligned}$$

$$\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\in S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) + 0$$

(As all dependence paths have length 1)

$$\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\in S \\ p\in G\lhd^* H}} \mathsf{width}(p)\mathsf{sz_o}(H)$$

In this case $f_G(\cdot) = f'_G(\cdot)$.

For the induction case,

$$\mathsf{sz_i}(G) \leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \vdash_S H\lhd D \\ G\leftsquigarrow_m H}} m\mathsf{sz_o}(H)$$

$$\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} m(\mathsf{sz_i}(H) + C)$$

$$+ \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\in S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H)$$

(For $\mathsf{symbol}(H) \in U$ and $\vdash_S H \lhd D$, $\vdash_U \mathcal{F}$ nsi
and using Theorem 10.) (3)

By induction hypothesis,

$$\mathsf{sz_i}(H) \leq f'_H(\mathsf{sz_i}(D)) + \sum_{\substack{I\in\Delta'' \\ \mathsf{symbol}(I)\in S \\ q\in H\lhd^* I}} \mathsf{sz_o}(I)\mathsf{width}(q) \tag{4}$$

where $f'_H(\cdot)$ is a polynomial.

By substituting right side of equation 4 for $\mathsf{sz_i}(H)$ in equation 3 we get,

$$\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\in S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H)$$

$$+ \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} \sum_{\substack{I\in\Delta'' \\ \mathsf{symbol}(I)\in S \\ q\in H\lhd^* I}} m\mathsf{width}(q)\mathsf{sz_o}(I)$$

$$\leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta'' \\ \mathsf{symbol}(H)\in S \\ \vdash_S H\lhd D \wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H)$$

$$+ \sum_{\substack{I\in\Delta'' \\ \mathsf{symbol}(I)\in S \\ r\in G\lhd^* I \wedge \mathsf{length}(r)>1}} \mathsf{width}(r)\mathsf{sz_o}(I)$$

$$\leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{I\in\Delta'' \\ \mathsf{symbol}(I)\in S \\ r\in G\lhd^* I}} \mathsf{width}(r)\mathsf{sz_o}(I)$$

where $f_G(\mathsf{sz_i}(D))$ is a polynomial and is given by

$$f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\notin S\\ \vdash_S H\lhd D\wedge G\leadsto_m H}} m\left(f'_H(\mathsf{sz_i}(D)) + C\right).$$

The remaining cases of the proof is by induction on the size of the derivation $\mathcal{D}$ is quite similar to the proof of Lemma 8. ∎

Lemma 6. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ and $\vdash_S \Delta/D$ nsi. If $\mathcal{D} :: \mathcal{F} \to D \gg P$, then*

- *For all $\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})$, if $\mathsf{symbol}(D) \in S$ then $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$.*
- *For all $\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})$, if $\mathsf{symbol}(D) \notin S$ then $\vdash_T \mathcal{F}$ nsi.*
- *$\sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\in S}} \beta_G(D)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\notin S}} \beta_G(D)\mathsf{sz_o}(G) \le (1-\alpha(D))\mathsf{sz_i}(P)$ where $\Delta' = \Delta \cup \{G|\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})\}$.*

Proof: The proof is by induction on the size of the derivation $\mathcal{D}$ and is similar to the proof of Lemma 8. ∎

Theorem 10 (Non-size-increasing functions). *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ nsi. If $\mathcal{D} :: \mathcal{F} \to G$, then $\mathsf{sz_o}(G) \le \mathsf{sz_i}(G) + C$ where C is a constant depending on the logic program $\mathcal{F}$.*

Proof: (Sketch) We shall prove by induction on the size of the derivation.

Using Lemma 9, we know that there exists a derivation $\mathcal{D}' :: \mathcal{F} \to D \gg P$ such that $\mathsf{sz_i}(G) = \mathsf{sz_i}(P)$ and $\mathsf{sz_o}(G) = \mathsf{sz_o}(P)$.

From Lemma 10, we know that $\mathsf{sz_o}(G) \le \alpha(D)\mathsf{sz_i}(G) + \sum_{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})} \beta_H(D)\mathsf{sz_o}(H) + C$.

When $\mathsf{symbol}(H) \in S$, $\mathsf{sz_o}(H) \le \mathsf{sz_i}(H)$ by induction hypothesis. Hence,

$$\begin{aligned}\mathsf{sz_o}(G) \le \alpha(D)\mathsf{sz_i}(G) + &\sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_i}(H)\\ &\sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\notin S}} \beta_H(D)\mathsf{sz_o}(H) + C\\ &\le \mathsf{sz_i}(G) + C\end{aligned}$$

This is because, $\sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_i}(H) + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\to H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_o}(H) \le (1-\alpha(D))\mathsf{sz_i}(G)$ is implied by $\vdash_S \mathcal{F}$ nsi and $\mathsf{sz_i}(G) = \mathsf{sz_i}(P)$.

∎

Theorem 11 (Stage 2). *Given a program $\mathcal{F}$ and a set S of mutually recursive predicates from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ poly_2. Given a goal G such that $\mathsf{symbol}(G) \in S$, if $\mathcal{D} :: \mathcal{F} \to G$, then there exists monotonically increasing polynomials $p(\cdot)$ and $p'(\cdot)$(not depending on the ground input terms of G) such that $\mathsf{sz_o}(G) \le p(\mathsf{sz_i}(G))$ and $\mathsf{sz}(\mathcal{D}) \le p'(\mathsf{sz_i}(G))$.*

Proof: Let the derivation $\mathcal{D}$ be given by

$$\frac{D \in \mathcal{F} \quad \mathcal{F} \to \overset{\mathcal{D}'}{D} \gg P}{\mathcal{F} \to P}$$

and

$$\Delta' = \{H | \mathcal{D}_H :: \mathcal{F} \to H \in \mathsf{GOALS}(\mathcal{D})\}$$

By Lemma 9 and Lemma 10, we know that,

$$\begin{aligned}
\mathsf{sz}_\mathsf{o}(G) &\leq \alpha(D)\mathsf{sz}_\mathsf{i}(G) + \sum_{H \in \Delta'} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) + C \\
&\leq \alpha(D)\mathsf{sz}_\mathsf{i}(G) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S}} \beta_H \mathsf{sz}_\mathsf{o}(H) + C \\
&\leq \alpha(D)\mathsf{sz}_\mathsf{i}(D) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \Gamma' \vdash_S H \triangleleft D}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \Gamma' \vdash_S H \ntriangleleft D}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) \\
&\quad + C_m
\end{aligned}$$

In this case, C is the total size of the term constants appearing in output positions of D. Clearly, it is a constant (depending only on $\mathcal{F}$). Let C_m be the maximum among all such constants.

By Lemma 11, for goals $H \in \Delta'$ such that $\mathsf{symbol}(H) \in T$, $\vdash_T \mathcal{F}$ nsi if $\vdash_S H \triangleleft D$ and $\vdash_T \mathcal{F}$ poly_2 if $\vdash_S H \ntriangleleft D$.

By theorem 10, $\mathsf{sz}_\mathsf{o}(H) \leq \mathsf{sz}_\mathsf{i}(H)$ in the former case. In the latter case, we can show by induction on the call graph of $\mathcal{F}$ rooted at S that $\mathsf{sz}_\mathsf{o}(H) \leq p_T(\mathsf{sz}_\mathsf{i}(H))$ where $p_T(\cdot)$ is a polynomial. Hence,

$$\begin{aligned}
\mathsf{sz}_\mathsf{o}(G) &\leq \alpha(D)\mathsf{sz}_\mathsf{i}(P) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D)\mathsf{sz}_\mathsf{i}(H) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \ntriangleleft D}} \beta_H(D) p_T(\mathsf{sz}_\mathsf{i}(H)) \\
&\quad + C_m \\
&\leq \alpha(D)\mathsf{sz}_\mathsf{i}(G) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz}_\mathsf{o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D)\mathsf{sz}_\mathsf{i}(H)
\end{aligned}$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \ntriangleleft D}} \beta_H(D) p_T(f_H(\mathsf{sz_i}(G))) + C_m$$

$$\text{(By Lemma 11, } \mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P)) \leq f_H(\mathsf{sz_i}(G)))$$

$$\leq F_1(\mathsf{sz_i}(G)) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D) \mathsf{sz_i}(H)$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D) \mathsf{sz_o}(H) \tag{5}$$

$$\text{(where } F_1(\mathsf{sz_i}(G)) = \alpha(D)\mathsf{sz_i}(G) + C_m$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \ntriangleleft D}} \beta_H(D) p_T(f_H(\mathsf{sz_i}(G))))$$

By Lemma 11, we have

$$\mathsf{sz_i}(H) \leq f'_H(\mathsf{sz_i}(P)) + \sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \triangleleft^* I}} \mathsf{sz_o}(I)\mathsf{width}(p)$$

where $f'_H(\cdot)$ is a polynomial.

Substituting in equation 5, we get,

$$\mathsf{sz_o}(G) \leq F_1(\mathsf{sz_i}(G)) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D) \mathsf{sz_o}(H)$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D) f'_H(\mathsf{sz_i}(D))$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D) \left(\sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \triangleleft^* I}} \mathsf{sz_o}(I)\mathsf{width}(p) \right)$$

$$\leq F(\mathsf{sz_i}(G)) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D) \mathsf{sz_o}(H)$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \triangleleft^* I}} \beta_H(D) \mathsf{sz_o}(I)\mathsf{width}(p)$$

$$\text{(where } F(\mathsf{sz_i}(D)) = F_1(\mathsf{sz_i}(D)$$

$$+ \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \triangleleft D}} \beta_H(D) f'_H(\mathsf{sz_i}(D)))$$

Now, by Lemma 11, we know that,

$$\left(\sum_{\substack{G \in \Delta \\ \mathsf{symbol}(G) \in S}} \alpha_G \mathsf{sz_i}(G) \right) + \left(\sum_{\substack{H \in \Delta \\ \mathsf{symbol}(H) \notin S}} \sum_{\substack{G \in \Delta \\ \mathsf{symbol}(G) \in S \\ p \in H \triangleleft^* G}} \alpha_H \mathsf{sz_i}(G)\mathsf{width}(p) \right) \leq \mathsf{sz_i}(P)$$

The polynomial $p(x) = x^2 F(x)$ and the remainder of the proof follows by induction on $\mathsf{sz_o}(G)$. It is similar to the proofs of Theorem 7 and 8.

To prove that $\mathsf{sz}(\mathcal{D}) \leq p'(\mathsf{sz_i}(G))$, we shall first need to show that for all $H \in \Delta'$ such that $\mathsf{symbol}(H) \notin S$, $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$ for some polynomial $f_H(\cdot)$. All terms that appear in input positions of H are either sub-terms of the terms in input positions of D or from output positions of other goals H. When $\mathsf{symbol}(H) \in S$, we have already proved that $\mathsf{sz_o}(H) \leq p_1(\mathsf{sz_i}(G))$ and when $\mathsf{symbol}(H) \in T \neq S$, we know that $\vdash_T \mathcal{F}$ $\mathsf{poly_2}$ and hence $\mathsf{sz_o}(H) \leq p_2(\mathsf{sz_i}(G))$ for some monotonically increasing polynomials $p_1(\cdot)$ and $p_2(\cdot)$. Hence, $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$ for some polynomial $f_H(\cdot)$. Now it is possible to show that the conditions given in Figure 3 are satisfied. The condition pc_Atom is always true if pp_Atom is true and the condition pc_Imp2 is true as $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$. ∎

D Proof of RAM Machine Translation

Theorem 12. *Given a logic program $\mathcal{F}$ satisfying the conditions given above and a goal G. If there exists a derivation $\mathcal{D} :: \mathcal{F} \to G$, then*

- *The goal G can be represented on a RAM machine in size proportional to $\mathsf{sz_i}(G)$.*
- *The corresponding proof search can be implemented in time proportional to $\mathsf{sz}(\mathcal{D})$.*

Proof: The goal G can be represented on RAM machine by simply storing the ground terms in the input positions of the goal G. The total size of this input is bound by $\mathsf{sz_i}(G)$.

We shall now show that every rule in Figure 1 can be implemented in a constant number of steps.

For the rules, g_True and c_Imp, it is clear that the implementation can be done in constant number of steps. Implementing g_Atom involves selecting the correct clause based on the inputs to the goal G. This selection is done by matching the inputs of the goal G with the input patterns in the clauses in $\mathcal{F}$. Since the program $\mathcal{F}$ is fixed, the maximal depth of the patterns is known and it is possible to design a hash function which maps every unique pattern to a hash value[4], thus providing a constant time implementation for pattern matching.

During the implementation of c_Exists, we substitute the universally quantified variables by a logic variables which are unified with the ground terms in the rule c_Atom. Since, the logic program is mode correct, all logic variables are guaranteed to be ground when the proof search completes. Unification is guaranteed to be decidable since we only allow *higher order patterns*. Moreover, since the program is mode correct and no variable appears more than once in an input position, unification is simply a series of pattern matching operations and hence it runs in time polynomial in the size of the pattern (a constant). The number of such operations per inference rule is bounded by the total number of input positions which is a constant depending only on the logic program $\mathcal{F}$. ∎

[4] A simple implementation would assign a unique prime number to every type family. In this case, the hash value of the pattern would be product of the prime numbers corresponding to the constituent type familes in the pattern.